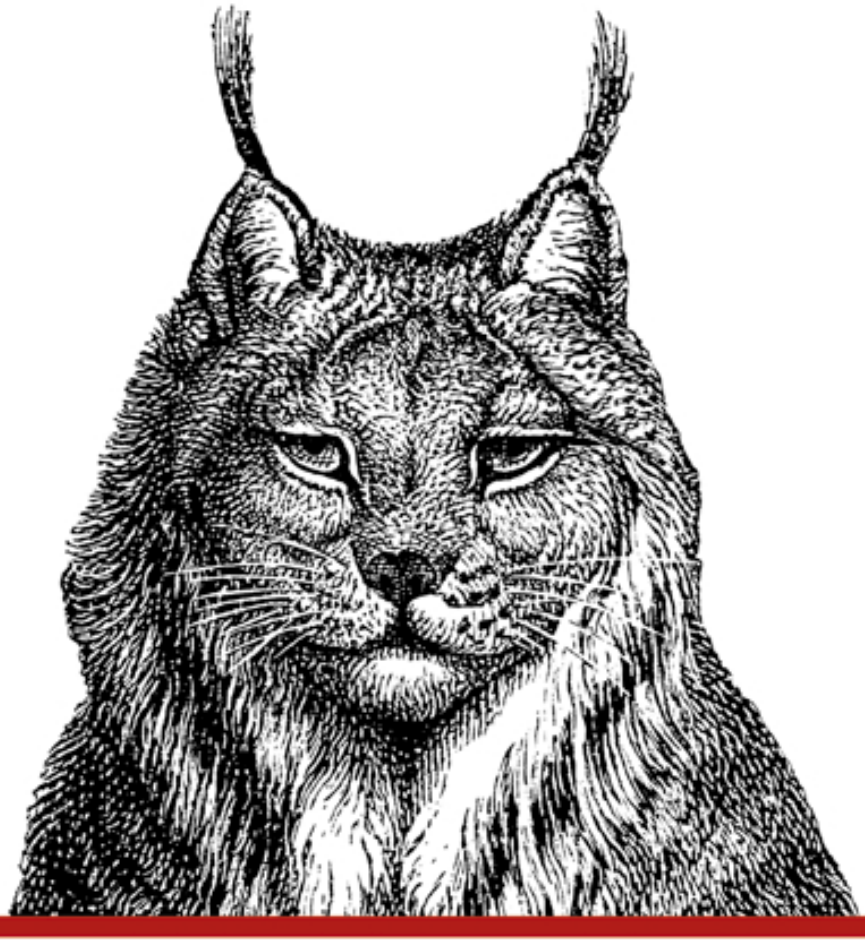


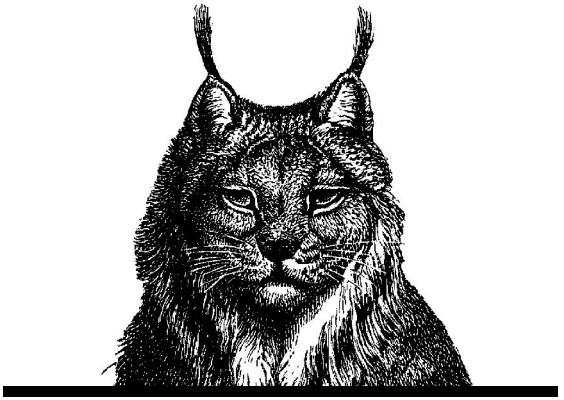
DELPHI

СПРАВОЧНИК

Символ
O'REILLY®

Рэй Лишнер

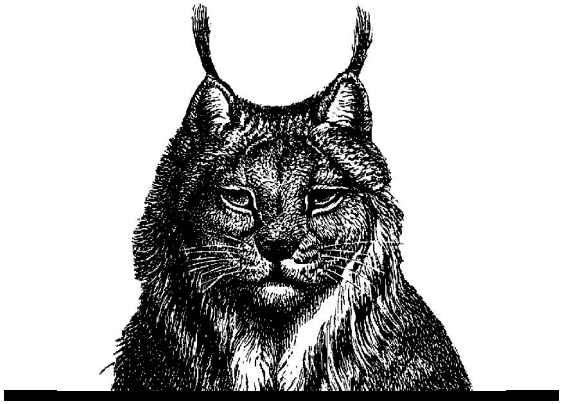
По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-019-7, название «Delphi. Справочник» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.



DELPHI
IN A NUTSHELL

Ray Lischner

O'REILLY®



DELPHI
СПРАВОЧНИК

Рэй Лишнер

Санкт-Петербург
2001

Рэй Лишнер

Delphi. Справочник

Перевод Л. Фрейдина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Г. Шахов*
Редактор	*Н. Японцева*
Корректура	*С. Беляева, С. Журавина*
Верстка	*М. Кривулев, Н. Макарова*

Лишнер Р.
Delphi. Справочник. – Пер. с англ. – СПб: Символ-Плюс, 2001. – 640 с., ил.
ISBN 5-93286-019-7

Популярный справочник по Delphi Pascal издательства O'Reilly содержит подробное описание языка программирования одного из лучших средств быстрой разработки приложений для Windows. Книга открывается введением в язык Delphi Pascal и подробным описанием объектной модели Delphi. Затем следует описание информации о типе времени выполнения (RTTI), являющейся ключом к интегрированной среде разработки (IDE) Delphi. Этот материал плохо документирован в других источниках, в том числе в официальных файлах помощи Delphi. Отдельная глава посвящена параллельному программированию в Delphi и созданию многопоточных приложений.

Основу книги составляет полный упорядоченный справочник по набору средств языка программирования Delphi. Каждая статья справочника включает: синтаксис в соответствии со стандартными соглашениями, описание, список аргументов функции или процедуры, советы и приемы использования данного средства языка в реальных программах, краткий пример и перекрестные ссылки на связанные ключевые слова.

Каким бы опытом работы с Delphi вы ни обладали, эта книга станет вашим постоянным помощником. В ней вы найдете решение многих проблем, а также получите возможность изучить более тонкие вопросы языка.

ISBN 5-93286-019-7
ISBN 1-56592-659-5 (англ)

© Издательство Символ-Плюс, 2001

Authorized translation of the English edition © 2000 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, editorial@books.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 9.04.2001. Формат 70x100$^{1}/_{16}$. Бумага офсетная.
Печать офсетная. Объем 40 печ. л. Тираж 4000 экз. Заказ N

Отпечатано с диапозитивов в ФГУП «Печатный Двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Предисловие *3*
Это не старый добрый Паскаль 3
Delphi для Linux 4
Об этой книге 5
Типографские соглашения 6
Другие источники информации 7
Как с нами связаться 8
Благодарности 8

Глава 1. Delphi Pascal *10*
Модули 10
Программы 14
Библиотеки 16
Пакеты 18
Типы данных 19
Переменные и константы 34
Обработка исключительных ситуаций 36
Файловый ввод/вывод 41
Функции и процедуры 42

Глава 2. Объектная модель Delphi *46*
Классы и объекты 46
Интерфейсы 72
Счетчики ссылок 78
Сообщения 81
Управление распределением памяти 82
Ключевое слово «object» 90

Глава 3. Информация о типе времени выполнения *92*
Таблица виртуальных методов 92
Секция published 94
Модуль TypInfo 101
Виртуальные и динамические методы 111

Инициализация и завершение 112
Автоматические методы 114
Интерфейсы 115
Изучение RTTI 116

Глава 4. Создание многопоточных приложений 120
Потоки и процессы 121
Класс TThread 130
Функции BeginThread и EndThread 136
Локальные данные потока 136
Процессы 137
Фьючерсы 148

Глава 5. Справочник по языку 157

Глава 6. Системные константы 485
Коды типов значений Variant 485
Типы открытых массивов 486
Смещения в таблице виртуальных методов 487
Коды ошибок времени выполнения 488

Глава 7. Операторы 492
Унарные операторы 492
Мультипликативные операторы 495
Аддитивные операторы 496
Операторы сравнения 497

Глава 8. Директивы компилятора 501

Приложение A. Утилиты командной строки 542

Приложение B. Модуль SysUtils 560

Предисловие

Borland Delphi сочетает в себе современный язык программирования, интегрированную среду разработки (IDE) и библиотеку визуальных компонентов (VCL). Среда разработки Delphi понятна любому, кто использовал подобные средства. Например, редактор форм WYSIWYG[1] позволяет визуально создавать окно, перенося на него компоненты мышью. И что более важно, среда Delphi является объектно-ориентированной, легко расширяемой и настраиваемой, что обусловлено мощью и гибкостью языка программирования Delphi.

Ядро Delphi – это язык программирования Delphi Pascal, содержащий ключевые средства для поддержки IDE и VCL. В нем есть все черты современного объектно-ориентированного языка наряду с элегантностью и простотой Паскаля.

«Настольный справочник по Delphi» – это современное справочное руководство по Delphi Pascal, в котором полностью описан язык и рассмотрены способы его эффективного применения. Опытные Delphi-программисты могут использовать эту книгу как алфавитный справочник. Новичкам Delphi стоит потратить время на прочтение первых нескольких глав. Я надеюсь, что каждый найдет для себя на этих страницах что-то ценное.

Это не старый добрый Паскаль

Delphi Pascal – это один из многих объектно-ориентированных вариантов Паскаля. Язык, используемый в Delphi, развивался в течение нескольких лет и больше не похож на Паскаль, который вы когда-то давно изучали в школе. Помимо модульного программирования и мощной модели классов язык Delphi Pascal содержит немало средств, присущих современному языку программирования, в том числе:

- Интерфейсы (сходные с интерфейсами Java™ и COM)

[1] WYSIWYG – What You see is what You get – что видишь, то и получаешь. – *Примеч. науч. ред.*

- Строки Unicode
- Свойства
- Обработка исключительных ситуаций

Среда Delphi с самого начала разрабатывалась как язык и среда программирования для Windows, и многие Delphi-программисты (в том числе и я) считают Delphi лучшей средой разработки для Windows из имеющихся на сегодняшний день. Delphi включает полную поддержку технологий COM и ActiveX, объектно-ориентированную библиотеку компонентов (VCL) и среду быстрой разработки приложений, которая легко расширяется и настраивается.

Delphi для Linux

Как следует из заголовка, Borland активно работает над переносом Delphi на Linux. Возможно сейчас, когда вы читаете эти строки, версия Delphi для Linux уже вышла[1], привнеся в X-Windows интегрированную среду разработки, включая редактор форм WYSIWYG, поддержку многоуровневых баз данных и технологию CORBA.

Пока Borland не закончил работу, и версия Delphi для Linux не вышла, я могу только предполагать, как будет выглядеть конечный продукт. (Нет, никакой секретной информации я не имею). Можно с уверенностью утверждать, что языковое ядро в Delphi для Linux и Delphi для Windows будет одинаковым, включая классы, объекты, интерфейсы, строки, динамические массивы, обработку исключительных ситуаций, а также базовые типы данных. Большинство встроенных процедур будут работать под Linux так же, как и под Windows.

Некоторые средства языка, описанные в этой книге, являются специфичными для Windows, например, переменные `CmdShow` и `DllProc` или функция `FindHInstance`. Если вы хотите писать код, переносимый с Windows на Linux, нужно избегать использования подобных специфических средств.

Среда Delphi для Windows – наилучшее средство разработки приложений и библиотек для Windows. Для достижения такого результата Borland включил в Delphi несколько специфических для Windows средств. Теперь Borland стремится сделать Delphi для Linux лучшим инструментом разработки для Linux. Можно ожидать, что с этой целью в Delphi будут включены некоторые специфические для Linux возможности.

[1] Данный продукт получил кодовое название Kylix. Работа над этим проектом была начата в 1999 году. О выходе официальной версии Kylix фирма Borland объявила 7 марта 2001 года. – *Примеч. науч. ред.*

Я могу только догадываться, но думаю, что вполне реальным будет написание кода, работающего и под Windows, и под Linux. Однако придется пожертвовать средствами, уникальными для каждой из версий. Написание легко переносимых компонентов, особенно интерфейсных элементов, наверное, будет нелегкой задачей. Создание переносимых приложений, скорее всего, будет проще[1].

Об этой книге

Первые четыре главы содержат материал, посвященный эффективному использованию Delphi, последующие главы представляют собой справочник по языку.

В главе 1 «Delphi Pascal» описаны различия между языком Delphi Pascal и стандартным Паскалем. Если вам знаком Turbo Pascal или другие версии Object Pascal, то будет достаточно быстрого просмотра главы 1, чтобы узнать о новых свойствах Delphi Pascal. Если же вы не работали с Паскалем со времени учебы в колледже много лет назад, стоит прочитать главу 1 целиком, чтобы узнать о новых элегантных возможностях Delphi Pascal. Думаю, вы удивитесь, насколько далеко вперед ушел язык программирования Паскаль за эти годы.

Глава 2 «Объектная модель Delphi» содержит более глубокое описание классов и объектов. Если вы использовали другой вариант Object Pascal, стоит прочитать эту главу, т. к. объектная модель Delphi совершенно иная. Если у вас есть опыт применения других объектно-ориентированных языков программирования, прочитайте главу 2, чтобы понять разницу между Delphi и другими языками, например Java и C++.

В главе 3 «Информация о типе времени выполнения» освещается ключевой момент интегрированной среды разработки Delphi. RTTI (Runtime type information) не описана в официальных файлах помощи Borland, но любой разработчик, имеющий дело с компонентами (т. е. каждый Delphi-программист), должен понимать суть RTTI, в том числе имеющиеся ограничения и возможности использования. Глава 3 содержит все, что следует знать о RTTI, и даже немного больше.

Глава 4 «Создание многопоточных приложений» посвящена месту Delphi в современном многопоточном, многопроцессорном мире. Среда Delphi содержит несколько языковых средств для создания многопоточных приложений, но эти средства трудно использовать, не зная тонкостей и проблем многопоточного программирования. Эта глава

[1] По заявлениям разработчиков Kylix, в состав этого продукта будет включена целая библиотека переносимых компонентов, получившая название CLX. Эта библиотека будет включена также в следующие версии Delphi и C++ Builder для Windows. – *Примеч. науч. ред.*

поможет вам начать эффективную разработку современных программ при помощи Delphi.

Глава 5 «Справочник по языку» составляет основной объем книги. Алфавитный справочник содержит все ключевые слова, директивы, процедуры, типы и переменные языка Delphi Pascal и системных модулей. Подробные примеры иллюстрируют способы эффективного и правильного использования средств языка.

Глава 6 «Системные константы» содержит таблицу соответствующих констант. Эти данные вынесены в отдельную главу, чтобы облегчить работу со справочником, поскольку глава 5 и так слишком велика.

Глава 7 «Операторы» описывает все арифметические и прочие операторы языка Delphi Pascal. Символы не так легко расположить в алфавитном порядке, поэтому вынесение их описания в отдельную главу облегчает поиск информации по конкретному оператору.

Глава 8 «Директивы компилятора» содержит список всех специальных комментариев, которые вы можете включить в исходный код для управления процессом компиляции и сборки программы.

Приложение А «Утилиты командной строки» описывает использование и параметры различных утилит командной строки, имеющихся в Delphi. Эти программы не относятся непосредственно к языку Delphi Pascal, но их часто не замечают, хотя они могут быть очень полезны для профессионала в Delphi.

В приложении В «Модуль SysUtils» вы найдете список всех процедур, типов и переменных модуля `SysUtils`. Этот модуль не является ни частью компилятора (как модуль `System`), ни частью языка Delphi Pascal. `SysUtils` – это часть библиотеки времени выполнения Delphi. Несмотря на это, многие Delphi-программисты рассматривают `SysUtils` как часть языка. И действительно, многие процедуры модуля `SysUtils` превосходят аналогичные утилиты модуля `System` (например `AnsiPos` по сравнению с `Pos`).

Типографские соглашения

В книге приняты следующие соглашения по форматированию текста:

`Моноширинный шрифт`

Применяется для идентификаторов и специальных символов Delphi, а также для всех ключевых слов и директив. В справочнике по языку моноширинный шрифт указывает на синтаксические элементы, которые следует использовать точно так, как напечатано. Например, объявление массива требует наличия квадратных скобок и других символов, а примеры использования ключевых слов `type`, `array` и `of` приведены ниже:

```
type Имя = array [Тип индекса, ...] of Базовый тип;
```

Моноширинный курсив

В справочнике по языку обозначает синтаксические элементы, которые должны быть заменены в реальном коде. В предыдущем примере следует подставить Имя типа, Тип индекса и Базовый тип.

Жирный моноширинный шрифт

В больших примерах кода служит для выделения обсуждаемого в данный момент элемента языка.

Курсив

Предназначен для переменных, имен файлов, каталогов, URL и словарных терминов.

Символы примечаний

Изображение совы означает подсказку, совет или общее замечание к обозначенному тексту.

Изображение индюка означает предупреждение, относящееся к данному тексту.

Другие источники информации

Если у вас есть вопрос по Delphi, в первую очередь следует обратиться к файлам помощи. В поставку Delphi также входит много примеров (каталог *Demos*), которые часто оказываются полезнее, чем файлы помощи.

Если вы так и не нашли ответ, попробуйте отправить свой вопрос в одну из многочисленных телеконференций. Существует несколько стандартных телеконференций, а также форумы, созданные фирмой Borland на своем сервере: *forums.borland.com*. В частности, для вопросов по языку Delphi Pascal подходит телеконференция *borland.public.delphi.objectpascal*.

Если вы хотите узнать об интегрированной среде Delphi, библиотеке визуальных компонентов и других темах, связанных с программированием Delphi, прочитайте две наиболее популярные книги: Marco Cantu «Mastering Delphi 5», издательство Sybex, 1999 год и Steve Teixeira и Xavier Pacheco «Delphi 5 Developer's Guide»[1], издательство Sams, 1999 год.

[1] Стив Тейксейра и Ксавье Пачеко «Delphi 5. Руководство разработчика», в 2–х томах, издательство «Вильямс», 2000 г.

Если вы найдете ошибки или пропуски в этой книге, пожалуйста, сообщите о них по адресу *nutshell@tempest-sw.com*. Я получаю очень много писем и не могу ответить на каждое отдельно, но не сомневайтесь, что я прочитаю все (по крайней мере то, что проходит через мои фильтры против почтового мусора).

Как с нами связаться

В этой книге мы постарались дать максимально точную и проверенную информацию, но ошибки и оплошности все равно могут вам встретиться. Сообщения о них, а также предложения для будущих изданий присылайте по адресу:

O'Reilly & Associates, Inc.

101 Morris Street

Sebastopol, CA 95472

(800) 998-9938 (in the U.S. or Canada)

(707) 829-0515 (international or local)

(707) 829-0104 (fax)

Вы также можете связаться с нами по электронной почте. Чтобы подключиться к нашему списку рассылки или заказать каталог, напишите по адресу:

info@oreilly.com

По техническим вопросам или с комментариями к этой книге обращайтесь по адресу:

bookquestions@oreilly.com

Об ошибках и планах по дальнейшим изданиям можно узнать на вебсайте этой книги:

www.oreilly.com/catalog/delphi

Другую информацию об этой и других книгах издательства O'Reilly можно найти на сайте издательства:

www.oreilly.com

Благодарности

Я благодарю Тима О'Рейли за то, что он воспользовался шансом издать свою первую книгу по Delphi. Надеюсь и дальше читать и писать книги по Delphi, издаваемые издательством O'Reilly.

Технические редакторы – Аллен Бауэр и Холвард Вассботн – отлично выполнили работу по поиску моих ошибок. Многочисленные и подробные комментарии Холварда были бесценны. Оставшиеся в книге

ошибки – это те, которые я добавил после того, как редакторы закончили свою тщательную работу.

Я благодарю моего редактора, Саймона Хэйеса, и всю бригаду издательства O'Reilly – Боба Хербстмана, Троя Мотта, сотрудников, занимающихся дизайном и производством, за то, что они превратили мою скромную рукопись в ту элегантную книгу, которую вы теперь видите.

Ни одна из моих книг не была бы возможна без поддержки семьи. Я благодарю мою жену, Черил, и программиста будущего, Артура, ради которых все это делалось.

Глава 1

Delphi Pascal

Delphi Pascal – это объектно-ориентированное расширение традиционного языка программирования Паскаль. Это не простая надстройка над стандартом ISO языка Паскаль, а если вы помните этот язык со школьных времен, то легко заметите расширения, внесенные в Delphi. Delphi – это вовсе не какой-то особый Паскаль. Добавленные в этот язык мощные средства объектно-ориентированного программирования не слишком сильно усложнили его. В Delphi присутствуют классы и объекты, обработка исключительных ситуаций, многопоточное программирование, модульное программирование, динамическое и статическое связывание, OLE-автоматизация и многое, многое другое.

В этой главе описаны расширения Паскаля, имеющиеся в Delphi. Вы наверняка знакомы с традиционным Паскалем или с одним из его популярных расширений, например с Object Pascal. Если вы уже знакомы с Object Pascal по одной из версий Turbo Pascal, вам достаточно изучить новую объектную модель (описанную в главе 2 «Объектная модель Delphi») и несколько новых возможностей.

Borland называет реализованный в Delphi язык программирования «Object Pascal», но многие другие языки называются так же, что создает путаницу. В этой книге для языка программирования среды Delphi авторы используют термин «Delphi Pascal», оставляя, таким образом, термин «Object Pascal» для других объектно-ориентированных вариантов Паскаля.

Модули

Delphi Pascal – это язык модульного программирования, и основной модульный элемент обозначается ключевым словом *unit*. Для компи-

ляции и сборки программы в Delphi требуются исходный текст программы и произвольное количество дополнительных модулей в форме исходного текста или объектного файла. Исходный текст программы обычно называется файлом проекта (*project file*), т. к. проект может быть либо программой, либо динамической библиотекой (DLL).

Когда выполняется сборка программы или библиотеки, все модули программы либо статически объединяются в единый *exe-* или *dll*-файл, либо строятся динамические ссылки на модули, содержащиеся в пакетах (*packages*). *Пакет* – это специальная динамическая библиотека, которая содержит один или более модулей и некоторую дополнительную логику, которая позволяет Delphi скрыть различие между статически присоединенным модулем и модулем из пакета. Более подробную информацию о пакетах можно найти в разделе «Пакеты» этой главы.

Формы и файлы

Некоторые модули содержат в себя формы. *Форма* (*form*) – это термин, принятый в Delphi для обозначения окон, создаваемых в графической среде Delphi. Описание формы (данные о ее виде, содержимом и свойствах) хранится в файле с расширением *.dfm*.

Каждому *dfm*-файлу соответствует *pas*-файл, в котором хранится код данной формы. Формы и *dfm*-файлы являются частью интегрированной среды разработки (IDE) и не относятся к самому языку Delphi Pascal. Однако в языке есть средства, которые предназначены исключительно для поддержки Delphi IDE и работы с данными форм. Об этих средствах читайте в главе 3 «Информация о типе времени выполнения».

Двоичный файл *dfm* фактически является 16-разрядным *res*-файлом (файлом ресурсов Windows), чем обеспечивается совместимость с первой версией Delphi. Версия 2 и все последующие создают только 32-разрядные приложения, поэтому компоновщик Delphi автоматически преобразует *dfm*-файл в 32-разрядный ресурс. Таким образом, двоичный *dfm*-файл фактически совместим со всеми версиями Delphi. Версия Delphi 5 поддерживает также и текстовые *dfm*-файлы, которые представляют собой обычный текст и не совместимы с предыдущими версиями, по крайней мере, без преобразования в двоичный формат. Единственный способ понять, является ли *dfm*-файл текстовым или двоичным, – это открыть файл и проверить его содержимое. Программным способом определения типа *dfm*-файла может быть чтение первых трех байт, которые в двоичном *dfm*-файле всегда равны: $FF $0A $00.

Таблица 1-1 содержит краткое описание и назначение файлов, которые встречаются в Delphi. Файлы, помеченные как (IDE), не относятся к Delphi Pascal, но используются в IDE.

Таблица 1-1. Файлы Delphi

Расширение	Описание
.bpg	Группа проектов (IDE)
.bpl	Компилированный пакет (специальный вид DLL)
.cfg	Настройки компилятора командной строки
.dcp	Информация о компилированном пакете, требуется при динамической сборке проекта, использующего данный пакет
.dcr	Файл bmp-ресурса для визуального компонента (IDE)
.dcu	Объектный код модуля
.dfm	Описание формы (IDE)
.dof	Настройки проекта (IDE)
.dpk	Исходный файл для построения пакета
.dpr	Основной исходный файл программы или библиотеки
.drc	Скрипт ресурсов для `resourcestring`
.dsk	Расположение окон на экране (IDE)
.pas	Исходный файл модуля
.res	Ресурс Windows (каждому *dpr*-файлу соответствует *res*-файл)

Разделение интерфейса и реализации

Каждый модуль состоит из двух частей: интерфейса (interface) и реализации (implementation). В интерфейсной части содержатся объявления типов, переменных, констант и подпрограмм, доступных другим модулям. В разделе реализации находятся «внутренности» процедур, описанных в интерфейсном разделе. В разделе реализации могут присутствовать дополнительные объявления, которые являются внутренними для реализации модуля. Таким образом, модули – это одно из основных средств в Delphi для обеспечения сокрытия информации.

Один модуль может *использовать* (*use*) другой модуль, таким образом получив доступ к данным, объявленным в другом модуле. Изменения в интерфейсной части модуля требуют перекомпиляции всех модулей, которые используют объявления, сделанные в измененном модуле. Компилятор Delphi отслеживает это автоматически, так что вам не придется создавать make-файлы для компиляции проектов.

Использовать другой модуль можно либо в интерфейсном разделе, либо в разделе реализации, и этот выбор достаточно важен при построении проекта:

- Если модуль A использует модуль B в интерфейсном разделе, изменения интерфейсного раздела модуля B воспринимаются как изменения интерфейса модуля A. При этом необходимо перекомпилировать все модули, использующие модуль A.
- Если модуль A использует модуль B в разделе реализации, при изменении объявлений модуля B требуется перекомпилировать только модуль A.

Модули не могут иметь циклических ссылок в их интерфейсных секциях. Иногда встречаются объявления классов, которые содержат взаимозависимые объявления. Простейшим решением является использование единого модуля, но если имеются причины объявить эти классы в отдельных модулях, можно использовать абстрактный базовый класс в одном или обоих модулях с тем, чтобы устранить циклическую зависимость.

Инициализация и завершение

Любой модуль может иметь разделы инициализации (initialization) и завершения (finalization). Код всех разделов инициализации запускается перед началом исполнения основного блока begin-end программы или библиотеки. Код раздела завершения исполняется после окончания работы программ или выгрузки библиотеки. В Delphi принят принцип запуска разделов инициализации, при котором вначале запускаются модули, находящиеся глубже всего по дереву зависимости модулей. Другими словами, до запуска кода инициализации модуля запускаются все разделы инициализации используемых в нем модулей. Каждый модуль инициализируется только один раз. Этот принцип демонстрируется в примере 1-1 .

Пример 1-1. Порядок инициализации модулей

```
program Example1_1;
uses unitA;
{$AppType Console}
begin
  WriteLn('Основная программа модуля 1-1');
end.

unit unitA;
interface
uses unitB;
implementation
initialization
  WriteLn('Инициализация модуля A');
finalization
```

```
  WriteLn('Завершение модуля A');
end.

unit unitB;
interface
implementation
initialization
  WriteLn('Инициализация модуля B');
finalization
  WriteLn('Завершение модуля B');
end.
```

Скомпилируйте пример 1-1 и запустите его из командной строки, а не из IDE, иначе консольное окно появится и исчезнет до того, как вы успеете увидеть результат работы программы. Этот результат приведен ниже в виде примера 1-2.

Пример 1-2. Результат работы примера 1-1

```
W:\nutshell>example1_1
Инициализация модуля B
Инициализация модуля A
Основная программа примера 1-1
Завершение модуля A
Завершение модуля B
```

Модули System и SysInit

Модули `System` и `SysInit` автоматически используются в каждом модуле, таким образом, объявления этих модулей фактически являются частью языка Delphi Pascal, и компилятор специальным образом обрабатывает вызовы многих функций и процедур модулей `System` и `SysInit`. Глава 5 «Справочник по языку» является полным описанием системных процедур, предназначенных для использования при программировании в Delphi.

Программы

Собственно программа в Delphi похожа на традиционную программу языка Паскаль. Она начинается с ключевого слова `program`, а собственно текст основной программы размещается в блоке `begin-end`. Однако в Delphi программы (основной программный блок, содержащийся в *dpr*-файле) обычно короткие, т. к. основную работу выполняют один или более модулей. В графическом приложении, например, основная программа обычно вызывает процедуру инициализации, создает одну или более форм (окон) и вызывает процедуру, запускающую цикл обработки событий Windows.

Для совместимости со стандартным Паскалем Delphi допускает список параметров после имени программы, но, как и большинство современных компиляторов, игнорирует указанные в нем идентификаторы.

В графическом приложении нельзя использовать стандартные для Паскаля процедуры ввода/вывода, так как нет ни устройства ввода, ни устройства вывода. Однако можно создать *консольное приложение* (*console application*), которое может записывать и считывать данные, используя стандартные процедуры ввода/вывода. (Смотрите в главе 8 «Директивы компилятора» о директиве `$AppType`, которая указывает Delphi, какое графическое или консольное приложение нужно создать.)

В предложении `uses` перечисляются все модули, составляющие программу. За каждым именем модуля следует директива `in`, в которой указывается имя файла. IDE и компилятор используют эту директиву для поиска модулей проекта. Модули без директивы `in` – это обычно библиотечные модули, которые не являются частью исходного текста проекта. Если модуль содержит присоединенную к нему форму, IDE, кроме того, сохраняет имя формы в виде комментария. В примере 1-3 показан типичный исходный файл программы.

Пример 1-3. Типичный файл программы

```
program Typical;

uses
  Forms,
  Main in 'Main.pas' {MainForm},
  MoreStuff in 'MoreStuff.pas' {Form2},
  Utils in 'Utils.pas';

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Модуль `Form` является частью стандартной библиотеки Delphi, так что для него не указана директива `in` и исходный файл. Для других модулей указаны имена исходных файлов, таким образом Delphi рассматривает эти файлы как часть проекта. Директива компилятора `$R` рассматривается в главе 8. Объект `Application` – часть библиотеки визуальных компонентов Delphi и в этой книге не описан. По поводу объекта `Application` и других компонентов VCL смотрите файлы справки Delphi.

Библиотеки

Библиотека Delphi компилируется в стандартную DLL-библиотеку Windows. Исходный текст библиотеки похож на текст программы, за исключением того, что в этом случае указывается ключевое слово `library` вместо `program`. Библиотека обычно содержит предложение `exports`, в котором перечислены процедуры, экспортируемые библиотекой. Предложение `exports` не обязательно, и если вы планируете использовать модуль при построении библиотеки, лучше поместить `exports` в текст модуля, ближе к процедурам, которые экспортируются. Если модуль используется вне библиотеки, предложение `exports` игнорируется.

Основное тело библиотеки – блок `begin-end` – выполняется каждый раз, когда приложение загружает эту библиотеку. Поэтому не требуется писать обработчик события `DLL_PROCCESS_ATTACH`. Однако для отделения процесса и потоковых событий обработчик необходим. Delphi автоматически заботится о регистрации процедуры обработки этих событий в Windows, и эта процедура вызывается в случае присоединения процесса, присоединения и отсоединения потока. В примере 1-4 показана простейшая DLL-процедура.

Пример 1-4. Наблюдение за присоединением и отсоединением DLL

```
library Attacher;

uses Windows;

procedure Log(const Msg: string);
begin
  MessageBox(0, PChar(Msg), 'Attacher', Mb_IconInformation + Mb_OK);
end;

procedure AttachDetachProc(Reason: Integer);
begin
  case Reason of
  Dll_Process_Detach: Log('Отсоединение процесса');
  Dll_Thread_Attach:  Log('Присоединение процесса');
  Dll_Thread_Detach:  Log('Отсоединение потока');
  else                Log('Неизвестная проблема!');
  end;
end;

begin
  // Этот код запускается каждый раз при загрузке библиотеки процессом.
  Log('Присоединение процесса');
  DllProc := @AttachDetachProc;
end.
```

Использование динамической памяти

При использовании DLL нужно быть осторожным в работе с динамической памятью. Любая память, зарезервированная в DLL, автоматически освобождается при выгрузке библиотеки. Однако в приложении могут быть указатели на эту память, которые в случае неосторожности приведут к явным или неявным ошибкам. Простейшее решение состоит в том, чтобы сделать модуль `ShareMem` первым модулем вашего приложения и всех библиотек, загружаемых им. Модуль `ShareMem` перенаправляет все запросы к памяти в одну библиотеку DLL (*BorlndMM.dll*), которая загружается сразу после запуска приложения. Тогда вы сможете загружать и выгружать DLL, не беспокоясь о «висящих» указателях.

Разделяемые объекты

`ShareMem` решает одну проблему с памятью, но не решает другую – проблему идентификации классов. Если класс `A` используется и в приложении, и в DLL, Delphi не может определить, что это один и тот же класс. И хотя в обоих случаях имя класса одинаковое, это вовсе не означает, что классы идентичны. Delphi выбирает наиболее безопасный путь и предполагает, что классы различны; и если вы считаете, что вам виднее, простого способа сообщить об этом Delphi все равно нет.

Иногда наличие двух классов с одним именем и не создает проблем, но если программа пытается использовать ссылку на объект через границу DLL, операторы `is` и `as` не будут работать так, как можно было бы ожидать. Так как DLL считает, что ее класс `A` отличается от класса `A` приложения, оператор `is` всегда будет возвращать False.

Единственный способ обойти эту проблему – не передавать объекты через границы DLL. Например, если у вас имеется графический объект, не передавайте сам объект `TBitmap`, а передавайте идентификатор (handle) `HBitmap`. Другое решение – использовать пакеты. Delphi автоматически обеспечивает идентификацию классов в пакетах для решения этой проблемы.

Установка базового адреса

При создании библиотеки обязательно установите базовый адрес (Image Base). В Windows каждый модуль загружается с уникального базового адреса. По умолчанию в Delphi установлен адрес $00400000, но он используется Windows для приложения, поэтому библиотека не может быть загружена по этому адресу. Если Windows будет вынужден переместить DLL на другой адрес, это вызовет падение производительности, т. к. Windows должен переписать таблицу размещения, чтобы отобразить новые адреса. Нельзя дать гарантию, что каждая DLL имеет уникальный адрес, поскольку вы не можете изменить адреса библиотек, написанных другими, но все равно можно сделать лучше, чем

сделано по умолчанию. По крайней мере, вы можете задать для своей библиотеки адрес, отличный от адресов стандартных пакетов Delphi и DLL самой Windows. Воспользуйтесь утилитой Windows Quick View для определения базового адреса библиотек.

Пакеты

Delphi может как статически присоединять модуль к программе или библиотеке, так и выполнять динамическое связывание. Для динамического связывания одного или более модулей эти модули нужно поместить в пакет (package) – специальный вид DLL. Разрабатывая программу или библиотеку, нет необходимости задумываться о том, каким образом будет выполнятся связывание модулей. Если вы решите использовать пакет, модули не будут присоединяться к *exe*- или *dll*-файлу, а вместо этого в файле будет сформирована ссылка на пакетную DLL (имеющую расширение *.bpl*, сокращенно от Borland Package Library).

Пакеты решают многие проблемы, возникающие при использовании DLL, такие как управление памятью и идентификация классов. Delphi отслеживает классы, определенные в каждом модуле, и обеспечивает для приложения и всех связанных с ним пакетов полную идентификацию каждого класса по имени, так что операторы `is` и `as` работают корректно.

Пакеты для разработки и для выполнения

Среда Delphi использует пакеты для загрузки компонентов, форм и других модулей, предназначенных для разработки, таких, например, как редакторы свойств. При написании компонентов выносите код для разработки в отдельный пакет (design-time), а основной код компонента – в пакет для выполнения (runtime). Приложения, использующие ваш компонент, могут либо статически присоединять *dcu*-файл, либо выполнять динамическое связывание с пакетом runtime, содержащим ваш компонент. Сохраняя весь код для разработки в отдельном пакете, вы исключаете необходимость присоединять лишний код к приложению.

Заметим, что пакет для разработки использует пакет для выполнения, т. к. нельзя иметь один и тот же модуль в нескольких пакетах. Приложение или библиотеку можно рассматривать как набор модулей. Один и тот же модуль не может присутствовать в одной программе несколько раз, независимо от того, связываются ли модули статически или динамически. Таким образом, если приложение использует два пакета, один и тот же модуль не может входить в оба пакета. Это было бы эквивалентно двукратному присоединению одного модуля к программе.

Создание пакета

Для создания пакета нужно создать *dpk*-файл, или исходный текст пакета. В *dpk*-файле перечислены все модули, составляющие этот пакет, а также все пакеты, которые он использует. Файл *dpk* можно редактировать как при помощи удобного редактора для пакетов из IDE, так и вручную, используя формат из примера 1-5.

Пример 1-5. Образец исходного файла пакета

```
package Sample;
{$R 'COMP.DCR'}
{$IMAGEBASE $09400000}
{$DESCRIPTION 'Sample Components'}

requires
  vcl50;

contains
  Comp in 'Comp.pas';

end.
```

Как и для любых DLL, используйте для ваших пакетов уникальные базовые адреса. Все остальные команды очевидны. В *dpk*-файлах, как и в других проектах, можно указывать директивы компилятора (см. главу 8) или задавать настройки в редакторе пакетов IDE.

Типы данных

Delphi Pascal поддерживает несколько расширений стандартного набора типов данных Паскаля. Как и любой диалект языка Паскаль, Delphi поддерживает перечисляемые типы, множества, массивы, целые и перечисляемые диапазоны, обычные и вариантные записи. Если вы привыкли к C или C++, разберитесь в этих стандартных для Паскаля типах, это поможет вам сэкономить много времени и избавит от проблем. Разница между C и Паскалем в отношении типов состоит, в частности, в следующем:

- Вместо битовых масок применяйте массивы, такой код обычно легче читать
- Можно использовать указатели вместо массивов, но массивы проще и обеспечивают проверку на выход за границу массива
- Записи Паскаля аналогичны структурам C, вариантные записи подобны объединениям

Целые типы

Основной целый тип – Integer. Тип Integer представляет целое число естественного размера, соответствующее вашей операционной системе и аппаратной платформе. Сейчас Integer – это 32-разрядное целое, но на это не стоит рассчитывать в перспективе. Будущее за 64-разрядными операционными системами, 64-разрядным оборудованием и 64-разрядным типом Integer. Для решения проблем с будущими изменениями разрядности в Delphi введены типы данных, размер которых зависит от естественного размера целого числа, и типы фиксированного для всех настоящих и будущих версий Delphi размера. В табл. 1-2 перечислены стандартные типы целых чисел. Типы, помеченные как «естественные», могут измениться в будущих версиях Delphi, соответственно изменится и диапазон их значений. Другие типы будут всегда иметь указанный в этой таблице размер и диапазон значений.

Таблица 1-2. Стандартные целые типы

Тип	Размер	Диапазон в Delphi 5
Integer	*естественный*	–2 147 483 648 . . 2 147 483 647
Cardinal	*естественный*	0 . . 4 294 967 295
ShortInt	8 бит	–128 . . 127
Byte	8 бит	0 . . 255
SmallInt	16 бит	–32 768 . . 32 767
Word	16 бит	0 . . 65 535
LongInt	32 бита	–2 147 483 648 . . 2 147 483 647
LongWord	32 бита	0 . . 4 294 967 295
Int64	64 бита	–9 223 372 036 854 775 808 . . 9 223 372 036 854 775 807

Вещественные типы

В Delphi имеется несколько типов данных с плавающей точкой. Основные типы это Single, Double и Extended. Single и Double соответствуют стандартным типам по спецификации IEEE-754, на которой основан модуль вычислений с плавающей точкой на платформах Wintel (Windows+Intel). Extended – это формат Intel для вычислений с повышенной точностью, который соответствует минимальным требованиям IEEE-754 для расширенной двойной точности вещественных чисел. В Delphi также определен стандартный тип Паскаля Real, который является синонимом Double. Описания и способ хранения всех этих типов подробно описаны в главе 5.

Аппаратный модуль вычислений с плавающей точкой всегда использует формат Extended для вычислений, но это не значит, что вы должны всегда использовать этот тип для хранения чисел. Extended занимает 10 байт, а Double всего 8 байт, и он более эффективен для пересылки в модуль вещественных операций. В большинстве случаев Double позволяет достичь лучшей производительности и достаточной точности.

Ошибки при операциях с вещественными числами, такие как деление на 0, приводят к ошибкам времени выполнения. Большинство приложений Delphi переводит эти ошибки в исключительные ситуации при помощи модуля SysUtils, и поэтому обычно при выполнении вещественной операции вместо таких ошибок возникает исключительная ситуация. Читайте об обработке ошибок и исключений далее в этой главе в разделе «Обработка исключительных ситуаций».

Вещественные типы могут также содержать бесконечность или нечисловое значение (NaN). Чтобы получить эти специальные значения, необходимо установить управляющее состояние для операций с плавающей точкой. Подробности о бесконечности и NaN читайте в стандарте IEEE-754 (его можно приобрести в IEEE). Информацию об управляющем состоянии смотрите в руководствах по архитектуре Intel-систем, в частности «Pentium Developer's manual», том 3, «Architecture and Programming Manual». Техническая документация Intel доступна в Интернете по адресу *http://developer.intel.com/design/processor*.

В Delphi также имеется тип с фиксированной точкой, Currency (валюта). Этот тип может хранить данные с четырьмя знаками после запятой в диапазоне от −922 337 203 685 477,5808 до 922 337 203 685 477,5807. Этого достаточно для хранения денежного дохода всей планеты с точностью до цента. Тип Currency также реализуется модулем для работы с вещественными числами с помощью двух 64-разрядных чисел. Так как Currency является вещественным типом, с ним нельзя применять целые операции (такие как побитный сдвиг и наложение маски).

Модуль вычислений с плавающей точкой (floating-point unit, FPU) может выполнять расчеты в режимах одинарной, двойной или расширенной точности. Delphi устанавливает FPU в режим расширенной точности и таким образом обеспечивает полную поддержку типов Extended и Currency. Однако некоторые функции Windows API переводят FPU в режим двойной точности. При двойной точности FPU выполняет расчеты, используя только 53 бита вместо 64.

Когда FPU работает в режиме двойной точности, нет смысла использовать значения типа Extended, и это еще одна причина применять Double для большинства вычислений. Значительную проблему создает тип Currency. Вы можете точно отследить, какие функции изменяют управ-

ляющее состояние FPU и переустановить тип точности на расширенную после возврата из проблемной функции. (Смотрите сведения о функции Set8087CM в главе 5.) Другое решение состоит в использовании типа Int64 вместо Currency и реализации собственной обработки чисел с фиксированной точкой, как в примере 1-6.

Пример 1-6. Использование Int64 для хранения значений типа Currency

```
resourcestring
  sInvalidCurrency = 'Неверная строка валюты: ''%s''';
const
  Currency64Decimals = 4;   // количество знаков после запятой
  Currency64Scale = 10000;  // 10**Currency64Decimals
type
  Currency64 = type Int64;

function Currency64ToString(Value: Currency64): string;
begin
  Result := Format('%d%s%.4d',
    [Value div Currency64Scale,
     DecimalSeparator,
     Abs(Value mod Currency64Scale)]);
end;
function StringToCurrency64(const Str: string): Currency64;
var
  Code: Integer;
  Fraction: Integer;
  FractionString: string[Currency64Decimals];
  I: Integer;
begin
  // Преобразование целой части и умножение на Currency64Scale
  Val(Str, Result, Code);
  Result := Result * Currency64Scale;

  if Code = 0 then
    // В строке присутствует только целая часть
    Exit

  else if Str[Code] = DecimalSeparator then
  begin
    // Пользователь может указать большее или меньшее чем 4 количество
    // десятичных знаков, но значимыми являются не более 4.
    FractionString := Copy(Str, Code+1, Currency64Decimals);
    // Заполнение отсутствующих разрядов нулями.
    for I := Length(FractionString)+1 to Currency64Decimals do
      FractionString[I] := '0';
    SetLength(FractionString, Currency64Decimals);

    // Преобразование дробной части и добавление ее к результату.
    Val(FractionString, Fraction, Code);
    if Code = 0 then
```

```
    begin
      if Result < 0 then
        Result := Result - Fraction
      else
        Result := Result + Fraction;
      Exit;
    end;
  end;

  // Строка не является допустимой строкой валюты (числом с фиксированной
точкой со знаком).
  raise EConvertError.CreateFmt(sInvalidCurrency, [Str]);
end;
```

Массивы

В дополнение к стандартным массивам языка Паскаль, в Delphi определено несколько специальных расширений. *Динамические массивы* – это массивы, размер которых может изменяться во время выполнения программы. *Открытые массивы* – это формальные параметры типа «массив», которые в качестве фактических аргументов могут принимать массивы любого размера. Особый случай открытых массивов позволяет передавать массив значений разного типа в качестве аргумента процедуры. Delphi не поддерживает совместимые массивы, описанные в ISO стандарте языка Паскаль, но открытые массивы предоставляют такую же функциональность.

Динамические массивы

Динамический массив – массив, размер которого определяется во время выполнения программы. Это позволяет динамическому массиву увеличиваться и уменьшаться во время работы программы. Динамический массив объявляется без указания типа индекса; тип индекса считается целым, и нумерация элементов начинается с 0. Во время выполнения программы размер динамического массива можно установить процедурой `SetLength`. Присваивание в случае динамического массива обозначает присваивание ссылки на этот массив. В отличие от строк, для динамических массивов в операциях присваивания не используется копирование, поэтому при изменении элемента динамического массива изменяются все ссылки на него. Delphi работает с динамическими массивами при помощи ссылочной арифметики, и поэтому при выходе за границы действия массива занятая им память автоматически освобождается. В примере 1-7 показано, как объявляется и используется динамический массив.

Пример 1-7. Использование динамических массивов

```
var
  I: Integer;
  Data: array of Double;   // Динамический массив, содержащий значения
```

```
                              // типа Double
  F: TextFile;                // Файл, из которого читаются данные
  Value: Double;
begin
  AssignFile(F, 'Stuff.dat');
  Reset(F);
  while not Eof(F) do
  begin
    ReadLn(F, Value);
    // Неэффективный, но простой способ увеличение размера динамического
    // массива. В реальной программе увеличивайте размер массива
    // шагами большего размера.
    SetLength(Data, Length(Data) + 1);
    Data[Length(Data)] := Value;
  end;
  CloseFile(F);
end;
```

Delphi проверяет значение индекса на выход за границы массива (если вы не выключили проверку диапазона; см. описание директивы `$R` в главе 8). Пустые массивы вызывают исключительную ситуацию. В Delphi динамический массив представляется нулевым указателем. При попытке обращения к элементу пустого динамического массива Delphi обращается к пустому указателю, что приводит к ошибке доступа, а не к ошибке выхода за границу массива.

Открытые массивы

Параметр функции или процедуры может быть объявлен как *открытый массив*. При вызове подпрограммы, в качестве аргумента можно использовать массив любого размера (с тем же базовым типом). Границы массива в подпрограмме следует определять при помощи функций `Low` и `High`. (Нижняя граница массива в Delphi всегда равна 0, но функции `Low` и `High` ясно дают понять читателю вашего кода, что именно делает эта часть программы, а назначение константы 0 может быть не так очевидно.) Не забудьте объявить этот параметр как `const`, если подпрограмме не требуется модифицировать массив, и `var`, если вы хотите разрешить изменения.

Объявление открытого массива выглядит так же, как и объявление динамического массива, что иногда может привести к путанице. Если объявляется параметр процедуры или функции, то описание массива без указания типа индекса означает открытый массив. Если объявляется локальная или глобальная переменная, поле класса, новый тип – описание массива без указания индекса означает динамический массив.

Подпрограмме с аргументом, описанным как открытый массив, можно передать динамический массив; в этом случае подпрограмма смо-

жет обращаться к элементам этого массива, но не сможет изменять его размер. Так как открытые массивы и динамические массивы объявляются одинаково, единственный способ объявить в качестве параметра динамический массив – это создать новый идентификатор типа, например:

```
procedure CantGrow(var Data: array of integer);
begin
  // Data - открытый массив, и его размер нельзя изменять.
end;
type
  TArrayOfInteger = array of integer; // тип динамического массива
procedure Grow(var Data: TArrayOfInteger);
begin
  // Data - динамический массив, и его размер можно изменять.
  SetLength(Data, Length(Data) + 1);
end;
```

Процедуре `CantGrow` можно передать динамический массив, но он будет передан ей как открытый массив. Процедура сможет обращаться к элементам массива, но не сможет изменять его размер.

Если функция Delphi вызывается из программы, написанной на другом языке программирования, открытый массив можно передать в виде указателя на первый элемент массива, а длину массива минус единица – в виде отдельного 32-разрядного целого аргумента. Другими словами, нижняя граница массива всегда равна нулю, верхняя граница передается вторым аргументом.

Открытый массив также можно создать путем перечисления набора данных в квадратных скобках. Выражение, формирующее открытый массив, можно использовать только для аргумента подпрограммы, его нельзя присвоить переменной типа «массив». Эта конструкция также неприменима для открытого массива, определенного как `var`-параметр. Создание массива на лету – это значительное удобство, дающее возможность обойтись без объявления массива констант:

```
Avg : = ComputeAverage(1, 5, 7, 42, 10, -13);
```

Еще один способ передачи массива процедуре или функции состоит в использовании функции `Slice`, которая позволяет передать подпрограмме часть массива. Функция `Slice` подробно описана в главе 5.

Открытые массивы вариантного типа

Еще один тип открытых массивов – открытый массив вариантного типа, или `array of const`. Открытый массив вариантного типа позволяет передать в подпрограмму гетерогенный массив, т. е. массив, элементы которого относятся к разным типам данных. Для каждого элемента массива Delphi создает запись типа `TVarRec`, хранящую тип элемента и его значение. Массив записей `TVarRec` передается в подпрограмму как

константный открытый массив. Подпрограмма может определить тип каждого элемента по полю VType соответствующей записи TVarRec. Открытые массивы вариантного типа позволяют передать аргументы переменного размера строго типизированным образом.

TVarRec – это вариантная запись, аналогичная Variant, но реализованная по-другому. В отличие от Variant, с помощью TVarRec можно передать ссылку на объект. В главе 6 «Системные константы» перечислены все типы, которые поддерживаются в TVarRec. В примере 1-8 приведена процедура, преобразующая массив вариантного типа в строку.

Пример 1-8. Преобразование данных вариантного типа в строку

```
function AsString(const Args: array of const): string;
var
  I: Integer;
  S: String;
begin
  Result := '';
  for I := Low(Args) to High(Args) do
  begin
    case Args[I].VType of
    vtAnsiString:
      S := PChar(Args[I].VAnsiString);
    vtBoolean:
      if Args[I].VBoolean then
        S := 'Истина'
      else
        S := 'Ложь';
    vtChar:
      S := Args[I].VChar;
    vtClass:
      S := Args[I].VClass.ClassName;
    vtCurrency:
      S := FloatToStr(Args[I].VCurrency^);
    vtExtended:
      S := FloatToStr(Args[I].VExtended^);
    vtInt64:
      S := IntToStr(Args[I].VInt64^);
    vtInteger:
      S := IntToStr(Args[I].VInteger);
    vtInterface:
      S := Format('%p', [Args[I].VInterface]);
    vtObject:
      S := Args[I].VObject.ClassName;
    vtPChar:
      S := Args[I].VPChar;
    vtPointer:
      S := Format('%p', [Args[I].VPointer]);
    vtPWideChar:
      S := Args[I].VPWideChar;
    vtString:
```

```
        S := Args[I].VString^;
      vtVariant:
        S := Args[I].VVariant^;
      vtWideChar:
        S := Args[I].VWideChar;
      vtWideString:
        S := WideString(Args[I].VWideString);
      else
        raise Exception.CreateFmt('Данный VType=%d не поддерживается',
                                  [Args[I].VType]);
      end;
      Result := Result + S;
    end;
  end;
```

Строки

В Delphi имеется четыре типа строк: короткие (short), длинные (long), «широкие» (wide) и с завершающим нулем (zero-terminated). Короткая строка – это фиксированный массив символов, который может содержать до 255 элементов. Короткие строки редко используются в Delphi-программах, но если вы уверены, что строка будет короче 255 символов, есть смысл использовать переменную типа `ShortString` с целью уменьшения накладных расходов.

Длинные строки (long string) могут быть произвольного размера, и этот размер может изменяться во время выполнения программы. В Delphi используется принцип «копирования для записи», позволяющий минимизировать копирование при присвоении строк переменным или при передаче строк в качестве аргументов в подпрограммы. Delphi поддерживает счетчик ссылок, чтобы обеспечить возможность автоматического освобождения памяти, занятой не используемой больше строкой.

«Широкие» строки (wide string) также создаются и управляются динамически, но они не применяют счетчик ссылок. Когда «широкая» строка присваивается переменной типа `WideString`, Delphi копирует строку целиком.

Delphi проверяет строковые ссылки аналогично ссылкам на динамические массивы, т. е. определяет, попадает ли индекс в границы диапазона. Пустая длинная или «широкая» строка представлена нулевым указателем, таким образом, проверка границ пустой длинной или «широкой» строки приводит к ошибке доступа вместо ошибки выхода за границы диапазона.

Строка с завершающим нулем (zero-terminated string) – это массив символов, пронумерованных целыми значениями, начиная с нуля. В строке не хранится ее размер, но для отметки конца строки использу-

ется символ с нулевым кодом. Такие строки используются в Windows API, и для других целей их использовать не рекомендуется. Без явного указания размера теряется возможность проверок и ухудшается производительность, так как некоторые операции требуют двухпроходного сканирования содержимого строки или осуществляют более медленную его обработку, постоянно проверяя наличие нулевого значения. Подобным образом Delphi работает и с указателями на такие строки.

Для удобства пользователя Delphi хранит нулевое значение и в конце длинных и «широких» строк, поэтому длинную строку легко преобразовать к типу `PAnsiChar`, `PChar` или `PWideChar`, получив указатель на строку с завершающим нулем. Тип `PChar` в Delphi эквивалентен `char*` языков C и C++.

Символьные константы

Символьные константы могут формироваться как в стандартном Паскале, либо при помощи символа фунта (#) с целым значением для задания символа по его коду или знака вставки (^) с буквой для обозначения управляющего символа. При формировании строки можно смешивать любые символьные константы, например:

```
'Normal string: '#13#10'Next line (after CR-LF)'^I'That was a ''TAB'''
```

Символ вставки (^) переключает шестой бит ($40) кода символа[1], таким образом преобразуя заглавную букву в соответствующий управляющий символ. Если буква строчная, символ вставки переключает пятый и шестой биты ($60). Это значит, что символ вставки можно применять и с небуквенными символами. Например, `^2` эквивалентно `'r'`, т. к. код `'2'` равен $32 и, переключив бит $40, мы получим $72, т. е. код буквы `'r'`. Delphi применяет одинаковые правила для всех символов, и вы можете ставить символ «^» перед пробелом, табуляцией, переводом строки, и код будет абсолютно нечитабельным.

Смешение различных строковых типов

Вы можете легко смешивать разные типы строк, и Delphi постарается понять наилучшим образом ваши намерения. Разные типы строк можно объединять в одну, и Delphi по необходимости «сузит» «широкую» строку или «расширит» «узкую». Для передачи строки в функцию, требующую параметра `PChar`, просто воспользуйтесь преобразованием строки к типу `PChar`. Короткие строки автоматически не дополняются нулевым символом, и потому для получения значения типа `PChar` вам придется сделать временную копию такой строки, добавить байт `#0` и определить адрес первого символа.

[1] Здесь и далее автор использует шестнадцатеричный формат записи чисел, принятый в Паскале (с использованием символа $). – *Примеч. науч. ред.*

Строки Unicode и многобайтовые строки

Delphi поддерживает в типах `WideChar`, `WideString` и `PWideChar` стандарт Unicode. Все строковые операции работают как с «широкими» строками, так и с обычными (длинными и короткими). Обычную строку можно присвоить переменной типа `WideString`, и Delphi автоматически преобразует строку в формат Unicode. При присвоении «широкой» строки переменной обычного строкового типа Delphi использует кодовую страницу ANSI для преобразования символов Unicode в однобайтовые символы.

Многобайтовая строка – это строка, в которой один символ занимает более одного байта. (В Windows для многобайтового набора символов принят термин *двухбайтовый набор символов*). В некоторых национальных языках (например, японском и китайском) применяются наборы символов, значительно превосходящие 256-символьный набор стандарта ANSI. Многобайтовые наборы символов представляют символ одним или двумя байтами, таким образом значительно увеличивая возможный символьный набор. В многобайтовой строке байт может быть: одинарным символом, ведущим байтом (т. е. первым байтом многобайтового символа), завершающим байтом (т. е. вторым байтом многобайтового символа). При посимвольной обработке строки следует учитывать многобайтовые символы, т. к. символ, который выглядит как буква «А», может в действительности оказаться завершающим байтом совсем другого символа.

К сожалению, некоторые строковые функции Delphi не обрабатывают правильно многобайтовые строки, но зато несколько функций, корректно работающих с такими строками, есть в модуле `SysUtils`. Работа с многобайтовыми строками особенно важна для имен файлов, и модуль `SysUtils` содержит специальные функции для работы с многобайтовыми символами в именах файлов. Подробности смотрите в приложении В «Модуль SysUtils».

Windows NT и Windows 2000 содержат как обычные, так и Unicode-версии большинства функций API. Delphi по умолчанию применяет обычные версии функций, но вы легко может вызывать и Unicode-версии. Например, для создания файла с именем в формате Unicode служит функция `CreateFileW`, а для файла с именем файла в формате ANSI – `CreateFileA`. `CreateFile` эквивалентна `CreateFileA`. VCL использует «узкие» версии управляющих элементов для обеспечения совместимости со всеми версиями Windows (большинство управляющих элементов Unicode в Windows 95 и 98 не поддерживаются).

Логические типы

В Delphi имеется обычный тип языка Паскаль – `Boolean`, но есть также и несколько других типов, позволяющих облегчить работу с Windows API. Многочисленные функции API и другие функции, написанные

на C и C++, по сути возвращают значения типа Boolean, но документированы как возвращающие целое. В C и C++ любое ненулевое значение считается истинным, и по таким же правилам в Delphi определены значения типа LongBool, WordBool и ByteBool.

Например, если вызывается функция, написанная на C, которая возвращает логический результат в виде короткого целого, тип возвращаемого значения объявляется как WordBool, и к функции можно обратиться так же, как к функции языка Паскаль, возвращающей значение типа Boolean.

```
function SomeCFunc: WordBool; external 'TheCDll.dll'
...
if SomeCFunc then ...
```

Неважно, какое числовое значение на самом деле возвращает функция SomeCFunc; Delphi рассматривает ноль как ложь и любое другое значение – как истину. Все C-подобные логические типы можно использовать так же, как основной логический тип в Delphi – Boolean. Для кода, целиком написанного на Delphi, всегда используйте тип Boolean.

Тип Variant

Delphi поддерживает вариантные типы OLE, что облегчает написание OLE-сервера и OLE-клиента. Вариантные типы применимы в любой ситуации, когда требуется переменная, тип которой может измениться в процессе работы программы. Вариантная переменная может принимать значения массива, строки, числа или даже интерфейса IDispatch. Существуют типы Variant и OleVariant. OleVariant отличается тем, что использует только типы, совместимые с COM, в частности, все строки преобразуются в формат Unicode. Там, где это различие не важно, в книге употребляется термин Variant для обоих этих типов.

Переменная типа Variant всегда инициализируется значением Unassigned. Такой переменной можно присвоить практически любую величину, и она будет содержать также тип и значение последней. Для определения типа данных переменной Variant служит функция VarType. В главе 6 перечислены значения, которые могут возвращаться функцией VarType. Низкоуровневый доступ к переменной типа Variant обеспечивается преобразованием ее к типу TVarData. Тип TVarData подробно описан в главе 5.

Если переменная типа Variant используется в выражении, Delphi автоматически преобразует остальные значения в этом выражении к Variant и возвращает результат также в виде Variant. Этот результат можно присвоить переменной статического типа, если с ней совместимо полученное значение типа Variant.

Самое распространенное применение Variant – это написание клиента OLE. Интерфейс IDispatch присваивается переменной типа Variant, и

затем эта переменная используется для вызова объявленных в интерфейсе функций. Компилятор ничего не знает об этих функциях, и их вызовы не проверяются на правильность при компиляции. Например, вы можете написать OLE-клиент, который печатает номер версии Microsoft Word, установленного в системе (см. код ниже). Delphi ничего не знает о свойстве `Version`, как и о других методах и свойствах OLE-клиента Word. Delphi просто компилирует ссылки на эти свойства и методы в вызовы интерфейса `IDispatch`. При этом теряется преимущество проверки на этапе компиляции, но появляется возможность воспользоваться гибкостью позднего связывания. (Если вы хотите сохранить преимущества строгой типизации, вам потребуется библиотека типов (type library) от поставщика сервера OLE. Для получения COM-интерфейсов библиотеки типов воспользуйтесь редактором библиотеки типов в IDE. Он не относится к языковым средствам Delphi и поэтому в этой книге не рассматривается.)

```
var
  WordApp: Variant;
begin
  try
    WordApp := CreateOleObject('Word.Application');
    WriteLn(WordApp.Version);
  except
    WriteLn('Word не установлен');
  end;
end;
```

Указатели

В Delphi указателям не придается такого значения, как в C и C++. Поскольку в Delphi имеются настоящие массивы, отпадает необходимость в имитации массивов указателями, а наличие специальных конструкций для объектов позволяет отказаться от использования указателей для ссылок на них. В Паскале также имеются настоящие параметры с передачей по ссылке. Чаще всего указатели используются для интерфейса с кодом, написанным на C и C++, в том числе Windows API.

Программисты на C и C++ будут рады увидеть, что правила работы с указателями в Delphi больше напоминают C, чем Pascal. В частности, проверка типов для указателей выполняется значительно менее строго, чем для других типов. (Тем не менее, имеются директивы `$T` и `$TypeAddress`, описанные в главе 8, которые усиливают степень строгости этих проверок.)

Тип `Pointer` – это общий тип указателей, эквивалентный `void*` в C или C++. При присваивании значения переменной типа `Pointer`, а также при присваивании значения типа `Pointer` переменной другого типа указателя, никаких преобразований не требуется. Для получения ад-

реса переменной или подпрограммы используются Addr и @ (эквивалентны & в C или C++). Для доступа к элементу массива или записи с помощью указателя можно пропустить оператор разрешения ссылки (^) – Delphi самостоятельно распознает указатель и применит этот оператор.

Ссылочная арифметика с указателями несколько ограничена по сравнению с C и C++. Для уменьшения или увеличения указателя на определенное количество элементов базового типа служат процедуры Dec и Inc. Фактическое значение указателя изменяется в соответствии с размером базового типа указателя. Например, увеличение указателя на Integer смещает его на 4 байта.

```
var
  IntPtr: ^Integer;
begin
  ...
  Inc(IntPtr);   // Перемещение IntPtr к следующему целому значению,
                 // вперед на 4 байта
  Inc(IntPtr, 3); // Увеличение IntPtr на 12 байтов = 3 * SizeOf(Integer)
```

Программы, работающие непосредственно с Windows API, часто явно используют указатели. Например, для создания логической палитры тип TLogPalette требует выделения динамической памяти и работы с указателями с применением распространенного в C приема – объявления массива из одного элемента. Для того чтобы воспользоваться TLogPalette в Delphi, следует использовать C-подобный подход, как показано в примере 1-9.

Пример 1-9. Использование указателей для создания палитры

```
// Создание серой палитры с NumColors элементов.
type
  TNumColors = 1..256;
function MakeGrayPalette(NumColors: TNumColors): HPalette;
var
  Palette: PLogPalette;          // указатель на запись TLogPalette
  I: TNumColors;
  Gray: Byte;
begin
  // TLogPalette - содержит массив палитр из одного элемента.
  // Чтобы разместить память для всей палитры, укажите размер
  // NumColors-1 элементов палитры.
  GetMem(Palette, SizeOf(TLogPalette) +
                  (NumColors-1)*SizeOf(TPaletteEntry));

  try
    // В стандартном Паскале пришлось бы использовать синтаксис
    // Palette^.palVersion, но Delphi автоматически разыменовывает
    // указатель.
    Palette.palVersion := $300;
```

```
    Palette.palNumEntries := NumColors;

    for I := 1 to NumColors do
    begin
      // Для простоты используем линейную палитру, хотя
      // логарифмическая дает лучшие результаты.
      Gray := I * 255 div NumColors;
// Выключаем проверку на границы диапазона для доступа к элементам
// после первого.
{$R-}
      Palette.palPalEntry[I-1].peRed   := Gray;
      Palette.palPalEntry[I-1].peGreen := Gray;
      Palette.palPalEntry[I-1].peBlue  := Gray;
      Palette.palPalEntry[I-1].peFlags := 0;
{$R+}
    end;
    // Delphi не разыменовывает указатели автоматически в случае
    // автономного использования, как здесь.
    Result := CreatePalette(Palette^);
  finally
    FreeMem(Palette);
  end;
end;
```

Указатели на функции и методы

Delphi позволяет получить адрес функции, процедуры или метода объекта и использовать этот адрес для вызова подпрограммы. Для простоты все три типа указателей называются *процедурными*.

Тип процедурного указателя определяется типом возвращаемого функцией значения, аргументами и тем, является ли он указателем на метод или на обычную процедуру. Исходный код смотрится лучше, если в нем объявляется процедурный тип и затем объявляется переменная этого типа, например:

```
type
  TProcedureType = procedure(Arg: Integer);
  TFunctionType = function(Arg: Integer): string;
var
  Proc: TProcedureType;
  Func: TFunctionType;
begin
  Proc := SomeProcedure;
  Proc(42); // Вызов через указатель Proc, как будто он является
// обычной процедурой
```

Обычно используется непосредственное присваивание значения процедурной переменной. Delphi может распознать по контексту, что требуется не вызов процедуры, а присваивание ее адреса. (Странным следствием этого простого принципа является то, что функцию без ар-

гументов, которая возвращает значение типа «указатель на функцию», нельзя вызвать обычным для Паскаля способом. Когда у функции нет аргументов, Delphi считает, что вы хотите получить адрес функции. Вместо этого вы можете вызвать функцию с пустыми скобками, как обычно вызываются функции без аргументов в C).

Кроме того, для получения адреса подпрограммы можно использовать операторы `@` и `Addr`. Явное применение этих операторов облегчает понимание вашего кода для тех, кто будет его читать и модифицировать.

Для процедурных указателей так же, как и для любых других указателей, можно использовать `nil`. Самый распространенный способ проверки нулевого указателя – с помощью функции `Assigned`.

```
if Assigned(Proc) then
  Proc(42);
```

Объявления типов

Delphi следует основным правилам совместимости типов, принятым в обычном Паскале для арифметических операций, передачи параметров и прочего. В объявлениях типов, однако, есть одно новшество для поддержки IDE. Если объявление типа начинается с ключевого слова `type`, Delphi создает для этого типа отдельный набор данных времени выполнения и рассматривает его как отдельный тип для параметров `var` и `out`. Если объявление является синонимом другого типа, Delphi обычно не создает отдельную таблицу RTTI для него. При наличии дополнительного ключевого слова `type` отдельные RTTI-таблицы позволяют различать два типа данных. Читайте подробности об RTTI в главе 3.

Переменные и константы

В отличие от стандартного Паскаля, Delphi позволяет определить тип константы, а величину константы можно задать, сославшись на глобальную переменную. Delphi также поддерживает многопоточные приложения, позволяя объявлять переменные, имеющие уникальные значения для каждого потока.

Типизированные константы

Когда объявляется тип константы, Delphi отводит для нее дополнительную память и рассматривает эту константу как переменную. Этой «константе» можно назначить значение, сохраняемое, в частности, при последующих вызовах той же подпрограммы. В C и C++ это называется статической переменной.

```
// Возвращает уникальное значение при каждом вызове функции.
function Counter: Integer;
const
```

```
  Count: Integer = 0;
begin
  Inc(Count);
  Result := Count;
end;
```

На уровне модуля переменная сохраняет свое значение аналогичным образом, и объявление константы эквивалентно объявлению глобальной переменной. Ту же функцию можно написать и по другому:

```
var
  Count: Integer = 0;
function Counter: Integer;
begin
  Inc(Count);
  Result := Count;
end;
```

Термин «типизированная константа» явно неудачен, и на уровне модуля следует всегда использовать объявление переменной с инициализацией вместо типизированной константы. Вы можете вынудить себя следовать этому правилу, отключив директиву компилятора `$J` или `$WriteableConst`, которая сообщает Delphi, что константы не могут изменять значение. Установка по умолчанию, однако, обеспечивает обратную совместимость и позволяет изменять значения типизированных констант. Подробнее об этих директивах компилятора смотрите в главе 8.

Локальные переменные процедур и функций нельзя инициализировать, и типизированные константы – это единственный способ иметь постоянные значения при всех вызовах подпрограммы. Вы должны выбрать лучший вариант из двух – объявление типизированной константы или переменной на уровне модуля.

Потоковые переменные

В Delphi имеется уникальный тип переменных, объявляемых с помощью ключевого слова `threadvar`, используемого в данном случае вместо `var`. Разница заключается в том, что переменная, объявленная как `threadvar`, имеет отдельное значение в каждом потоке многопоточного приложения. Обычная переменная имеет одно и то же значение во всех потоках. Переменная `threadvar` определяется на уровне модуля.

Delphi реализует переменную `threadvar` при помощи технологии TLS (thread local storage) из Windows API. Преимущество потоковых переменных по сравнению с непосредственным использованием TLS состоит в том, что в Windows доступно мало ячеек TLS, а потоковых переменных может быть сколько угодно, и размер их не ограничен. Еще важнее то, что потоковые переменные можно применять точно так же, как и любые другие, что значительно проще, чем возиться с TLS. Под-

робнее об использовании потоковых переменных вы можете прочесть в главе 4 «Создание многопоточных приложений».

Обработка исключительных ситуаций

Исключительные ситуации позволяют прерывать нормальную последовательность работы программы. Исключительную ситуацию можно вызвать из любой функции, процедуры или метода. Она возвращает управление к более ранней точке той же подпрограммы или в подпрограмму на значительно более раннем уровне стека вызовов. Где-то в этом стеке должна быть подпрограмма, которая использует блок `try-except-end` для перехвата исключительной ситуации, в противном случае вызывается процедура `ExceptProc`.

В Delphi имеется две родственные конструкции для работы с исключительными ситуациями. Конструкция `try-except` устанавливает обработчик ошибок, который получает управление при возникновении ошибки. Конструкция `try-finally` явным образом не обрабатывает исключительные ситуации, но гарантирует, что код в разделе `finally` всегда выполняется даже при возникновении исключительной ситуации. Для обработки ошибки следует использовать конструкцию `try-except`. Если же есть ресурс (например, динамическая память), который должен быть освобожден независимо от того, что случилось, нужно применять `try-finally`. Конструкция `try-except` аналогична конструкции `try-catch` в C++ и Java. Стандарт C++ не имеет конструкции, аналогичной `finally`, но она есть в Java. Некоторые компиляторы C++, включая компиляторы фирмы Borland, расширяют стандарт C++ добавлением такой функциональности, в частности, ключевым словом `__finally`.

Так же как в C++ и Java, конструкция `try-except` в Delphi может обрабатывать либо все исключительные ситуации, либо только определенный их тип. В блоке `try-except` может быть несколько секций `on`, каждая из которых объявляет определенный класс исключительных ситуаций. Delphi последовательно обходит эти секции в поиске класса, соответствующего классу ситуации, возникшей в программе, или класса, являющегося более общим. В примере 1-10 показано, как использовать конструкцию `try-except`.

Пример 1-10. Использование try-except для обработки исключительной ситуации

```
function ComputeSomething;
begin
  try
    PerformSomeDifficultComputation;
  except
    on Ex: EDivideByZero do
      WriteLn('Деление на ноль');
```

```
    on Ex: EOverflow do
      WriteLn('Ошибка переполнения');
    else
      raise; // повторение исключительной ситуации для возможности
             // дальнейшей обработки
  end;
end;
```

В многопоточном приложении каждый поток поддерживает собственную информацию об исключительной ситуации и может вызывать ее независимо от других потоков. Подробности смотрите в главе 4.

Когда программа генерирует исключительную ситуацию, оператору `raise` должен быть передан объект[1]. Обычно программа создает новый объект типа `exception` как часть работы команды `raise`, однако в редких случаях может потребоваться заново создать уже существующий объект `exception`. Delphi просматривает стек вызовов в поиске предложений `try`. В случае обнаружения конструкции `try-finally`, Delphi вызывает код секции `finally`, а затем продолжает поиск обработчика исключительной ситуации в стеке. Когда в стеке обнаруживается блок `try-except`, Delphi просматривает секции `on` в `except` в поисках соответствующего типа исключительной ситуации. Если секции `on` отсутствуют, Delphi выполняет код раздела `except`, если присутствуют – пытается подобрать соответствие или запускает код секции `else` блока `try-except`.

Переменная, которая объявлена в секции `on`, содержит ссылку на объект типа `exception`. Delphi автоматически освобождает этот объект после завершения обработчика (подробнее об объектах в главе 2).

Если Delphi доходит до вершины стека вызовов, не найдя соответствующего обработчика исключительной ситуации, вызывается процедура `ExceptProc`. `ExceptProc` – это фактически указатель на процедуру с двумя аргументами: объект типа `exception` и адрес, где произошла ошибка. Информацию обо всех необработанных ошибках можно сохранить в специальном файле протокола, как в примере 1-11.

Пример 1-11. Протоколирование необработанных исключительных ситуаций в файле

```
var
  LogFileName: string = 'C:\log.txt';

procedure LogExceptProc(ExceptObject: TObject; ErrorAddr: Pointer);
const
  Size = 1024;
resourcestring
```

[1] Оператор `raise` может применяться без аргументов для повторения возникшей ранее исключительной ситуации, как это показано в примере 1-10. – *Примеч. науч. ред.*

```
  Title = 'Внутренняя ошибка: сообщите в службу технической поддержки';
var
  Buffer: PChar[0..Size-1];
  F: TextFile;
begin
  ExceptionErrorMessage(ExceptObject, ExceptAddr, Buffer, Size);

  AssignFile(F, LogFileName);
  if FileExists(LogFileName) then
    AppendFile(F)
  else
    Rewrite(F);
  WriteLn(F, Buffer);
  CloseFile(F);

  MessageBox(0, Buffer, Title, Mb_IconStop);
end;
...
// Указание Delphi использовать вашу процедуру для исключительных ситуаций.
ExceptProc := @LogExceptProc;
```

Delphi также перехватывает ошибки времени выполнения, например, переполнение стека, и вызывает для каждой из них процедуру ErrorProc. Фактически ErrorProc – это указатель на процедуру. Чтобы задать обработчик ошибки, объявите процедуру и присвойте ее адрес переменной ErrorProc.

Модуль System работает с двумя типами кодов ошибок – внутренними и внешними. Если вы пишете процедуру ErrorProc, придется иметь дело с внутренними кодами, каждый из которых представляет собой целое число, обозначающее определенный тип ошибки. В главе 6 приведен список этих кодов. Процедура ErrorProc, вызываемая в Delphi по умолчанию, ставит в соответствие внутренним кодам внешние коды ошибок. В главе 6 также представлены и внешние коды.

При вызове процедуры ErrorProc, Delphi передает ей два аргумента – код ошибки и адрес инструкции, в которой произошла эта ошибка. Ваш обработчик может выглядеть, например, так:

```
procedure DumbErrorProc(ErrorCode: Integer; ErrorAddr: Pointer);
begin
  ShowMessage(Format('Ошибка времени выполнения %d at %p', [ErrorCode, Erro-
rAddr]));
end;
...
ErrorProc := @DumbErrorProc;
```

В модуле SysUtils имеются дополнительные средства, облегчающие работу с исключительными ситуациями и ошибками времени выполнения. В частности, определяются процедуры ErrorProc и ExceptProc. Процедура ErrorProc преобразует ошибку времени выполнения в исключительную ситуацию, например, ошибка переполнения стека преобразуется в EStackOverflow. Процедура ExceptProc выводит сообщение об исключительной ситуации и прерывает выполнение программы. В консольном приложении сообщение выдается на стандартное устройство вывода, в графических – выводится диалоговое окно.

Модуль SysUtils устанавливает указатели ErrorProc и ExceptProc на свои процедуры в разделе инициализации модуля. Если исключительная ситуация или ошибка времени выполнения возникает в приложении до инициализации модуля SysUtils, вы не сможете воспользоваться преимуществами его обработчиков. Поэтому, если приложение выдает ошибку времени выполнения, а не исключительную ситуацию, проблема наверняка находится в разделах инициализации или завершения.

Для вызова исключительной ситуации служит команда raise с указанием ссылки на объект. Обычно эта команда создает новый экземпляр объекта. Вы можете сами создать объект любого класса для команды raise, однако большинство программ использует класс SysUtils.Exception или один из производных классов.

Delphi хранит информацию о возникшей исключительной ситуации, а именно: где возникла ситуация, программный контекст ее возникновения и прочее. К этой информации имеется доступ через различные переменные модуля System. Полное описание приведено в главе 5, а в табл. 1-3 представлен обзор самых важных переменных.

Таблица 1-3. Переменные, относящиеся к обработке исключительных ситуаций и ошибок

Объявление	Описание
AbstractErrorProc	Обработчик для абстрактного метода
AssertErrorProc	Обработчик отладочных условий (смотрите процедуру Assert в главе 5)
ErrorAddr	Адрес ошибки времени выполнения
ErrorProc	Указатель на процедуру для обработки ошибок времени выполнения
ExceptClsProc	Устанавливает соответствие между исключительной ситуацией Windows и классом Delphi
ExceptionClass	Базовый класс для исключительных ситуаций
ExceptObjProc	Устанавливает соответствие между исключительной ситуацией Windows и объектом Delphi

Объявление	Описание
ExceptProc	Указатель на процедуру, вызываемую для необработанных исключительных ситуаций
SafeCallErrorProc	Обработчик ошибок Safecall

Когда Delphi проходит стек вызовов, вызываются разделы finally всех вложенных блоков try-finally. Delphi также освобождает память всех динамических массивов, длинных строк, «широких» строк, интерфейсов и переменных типа Variant при выходе из области их действия. (Строго говоря, Delphi лишь уменьшает счетчики ссылок, и реальное освобождение памяти происходит, только если других ссылок на строку и массив больше нет.)

Если исключительная ситуация возникает в блоке finally, старый объект exception освобождается, и Delphi обрабатывает новую исключительную ситуацию. Самый распространенный вариант использования конструкции try-finally – это освобождение объектов и других ресурсов. Если в подпрограмме требуется освободить много объектов, самый простой способ заключается в первоначальной установке всех переменных в nil и использовании одного блока try-finally для освобождения всех объектов. Если деструктор объекта может вызвать исключительную ситуацию, следует все-таки использовать вложенные блоки try-finally, однако в большинстве случаев можно обойтись технологией, приведенной в примере 1-12.

Пример 1-12. Использование try-finally для освобождения нескольких объектов

```
// Копируется файл. Если исходный файл не может быть открыт,
// или целевой файл не может быть создан, вызывается исключительная
// ситуация EFileCopyError, и к новому сообщению добавляется исходное
// сообщение об ошибке. Новое сообщение дает чуть больше информации,
// чем исходное.
type
  EFileCopyError = class(EStreamError);

procedure CopyFile(const ToFile, FromFile: string);
var
  FromStream, ToStream: TFileStream;
resourcestring
  sCannotRead = 'Невозможно открыть файл: %s';
  sCannotCreate = 'Невозможно создать файл: %s';
begin
  ToStream := nil;
  FromStream := nil;
  try
    try
      FromStream := TFileStream.Create(FromFile, fmOpenRead);
    except
```

```
        // Обрабатывается только исключительная ситуация EFOpenError.
        on Ex: EFOpenError do
          // Raise a new exception.
          raise EFileCopyError.CreateFmt(sCannotRead, [Ex.Message]);
      end;
      try
        ToStream := TFileStream.Create(ToFile, fmCreate);
      except
        on Ex: EFCreateError do
          raise EFileCopyError.CreateFmt(sCannotCreate, [Ex.Message]);
      end;
      // Копируется файл.
      ToStream.CopyFrom(FromStream, 0);
    Finally
      // Завершение работы программы. Закрытие всех файлов, даже в случае
      // ошибки.
      ToStream.Free;
      FromStream.Free;
    end;
  end;
```

Файловый ввод/вывод

Традиционный файловый ввод/вывод языка Паскаль работает и в Delphi, но в графических приложениях нельзя использовать стандартные файлы `Input` и `Output`. Чтобы присвоить имя файла переменной типа `File` или `TextFile`, применяется процедура `AssignFile`. Процедуры `Reset` и `Rewrite` работают так же, как в стандартном Паскале, кроме того, есть процедура `Append` для добавления данных в конец файла (файл должен уже существовать). Для закрытия файла служит процедура `CloseFile`. В табл. 1-4 приведены процедуры ввода/вывода, имеющиеся в Delphi.

Таблица 1-4. Процедуры файлового ввода/вывода

Подпрограмма	Описание
`Append`	Открывает существующий файл для добавления данных
`AssignFile` или `Assign`	Присваивает имя файла переменной типа `File` или `TextFile`
`BlockRead`	Читает данные из файла
`BlockWrite`	Записывает данные в файл
`CloseFile` или `Close`	Закрывает открытый файл
`Eof`	Возвращает значение True, если текущая позиция находится в конце файла

Подпрограмма	Описание
Erase	Удаляет файл
FilePos	Возвращает текущую позицию в файле
FileSize	Возвращает размер файла в записях
Read	Читает форматированные данные из двоичного или текстового файла
ReadLn	Читает строку данных из текстового файла
Rename	Переименовывает файл
Reset	Открывает файл для чтения
Rewrite	Открывает файл для записи, удаляя прежнее его содержимое
Seek	Изменяет текущую позицию файла
Write	Записывает форматированные данные
WriteLn	Записывает строку текста

Переменная FileMode задает режим открытия файла для процедуры Reset. По умолчанию FileMode равна 2, что разрешает доступ к файлу на запись и чтение. Если вы хотите только читать из файла, следует установить значение FileMode в 0 перед вызовом процедуры Reset. (Установите FileMode в 1 для доступа только на запись.)

Библиотека Delphi представляет и более совершенный способ работы с файлами – с использованием потоков. Потоки – это объектно-ориентированное средство, обеспечивающее значительно большую гибкость и мощность по сравнению с традиционным вводом/выводом языка Паскаль. Потоки неприменимы в единственном случае – когда невозможно использовать библиотеки и есть привязка только к средствам языка Delphi Pascal. В главе 5 подробно описаны все процедуры ввода/вывода. Информацию о классе TStream и родственных классах смотрите в файлах помощи Delphi.

Delphi не поддерживает стандартные процедуры Get и Put языка Паскаль .

Функции и процедуры

Delphi поддерживает несколько расширений синтаксиса функций и процедур стандартного языка Паскаль. Имеется возможность перег-

рузки подпрограмм путем объявления нескольких подпрограмм с одним именем, но разным набором и типом параметров. Возможно также объявление для параметров значений по умолчанию, что делает часть параметров необязательными. Почти все, сказанное в этом разделе, одинаково относится и к процедурам, и к функциям, а потому далее используется термин «подпрограмма».

Перегрузка подпрограмм

В Delphi поддерживается перегрузка (`overload`) путем объявления нескольких подпрограмм с одинаковым именем, но разными параметрами[1]. Для объявления перегружаемых подпрограмм служит директива `overload`, например:

```
function AsString(Int: Integer): string; overload;
function AsString(Float: Extended): string; overload;
function AsString(Float: Extended; MinWidth: Integer):string; overload;
function AsString(Bool: Boolean): string; overload;
```

При вызове перегружаемой подпрограммы компилятор определяет, какую именно версию подпрограммы вы хотите вызвать. Поэтому эти версии должны иметь различное число или тип аргументов. Например, для определения того, какую функцию из приведенных выше следует вызвать, достаточно лишь сравнить тип аргумента:

```
Str := AsString(42);       // вызывается AsString(Integer)
Str := AsString(42.0);     // вызывается AsString(Extended)
Str := AsString(42.0, 8);  // вызывается AsString(Extended, Integer)
```

Может возникнуть следующая ситуация: модуль A объявляет подпрограмму, модуль B использует модуль A, и при этом объявляет подпрограмму с тем же именем. Объявление в модуле B может не иметь директивы `overload`, и вам потребуется при вызове из модуля B использовать имя модуля A для идентификации его версии подпрограммы. Производный класс, перегружающий метод класса-предка, должен объявляться с директивой `overload`.

Параметры по умолчанию

Иногда вместо перегрузки подпрограммы можно использовать параметры по умолчанию[2]. Рассмотрим, например, следующие перегружаемые функции:

```
function AsString(Float: Extended): string; overload;
```

[1] Данная возможность доступна в Delphi 4 и более поздних версиях. – *Примеч. науч. ред.*

[2] Данная возможность доступна в Delphi 4 и более поздних версиях. – *Примеч. науч. ред.*

```
function AsString(Float: Extended; MinWidth: Integer):string; overload;
```

Наверняка, первая из приведенных функций преобразует свой вещественный аргумент в строку с некоторой предопределенной минимальной шириной, равной, например, единице. И эту функцию можно даже написать таким образом, чтобы она вызывала другую, например:

```
function AsString(Float: Extended): string;
begin
  Result := AsString(Float, 1)
end;
```

Вы можете обойтись без лишней головной боли и лишнего кода, написав одну подпрограмму, которая может принимать необязательный параметр. Если при вызове фактический аргумент не подставляется, Delphi использует значение по умолчанию:

```
function AsString(Float: Extended; MinWidth: Integer = 1): string;
```

Разумное применение параметров по умолчанию позволяет уменьшить количество перегружаемых процедур. Тем не менее, будьте осторожны со строковыми параметрами. При компиляции Delphi придется вставлять строку всюду, где вызывается подпрограмма со строковым параметром по умолчанию. Это не проблема, если строка пустая (Delphi представляет пустую строку нулевым указателем), однако если строка не пустая, следует использовать инициализированную переменную (или типизированную константу). Тогда Delphi сможет создать ссылку на эту переменную при необходимости использования параметра по умолчанию. В противном случае вы позволите Delphi тратить место для хранения дополнительных копий строки и время на создание нового ее экземпляра для каждого вызова функции.

Переменная Result

В Delphi позаимствовано одно средство из языка Eiffel, а именно переменная `Result`. Эта переменная неявно объявлена в каждой функции, тип ее соответствует типу возвращаемого функцией значения. Вы можете использовать переменную `Result` как любую другую, и при завершении функция возвратит ей значение. Использовать эту переменную более удобно, чем имя функции, как это принято в стандарте Паскаля для формирования возвращаемого значения. Так как `Result` – это переменная, вы можете сколько угодно раз устанавливать и использовать ее значение. В стандартном Паскале этот результат достигается путем явного объявления результирующей переменной, однако надо не забыть затем присвоить ее значение имени функции. Разница, как видите, не такая большая, но подобные приятные мелочи позволяют в итоге значительно упростить большой проект. Delphi поддерживает и старый метод назначения результата функции, поэтому у вас есть выбор. Будьте последовательны в применении того или иного подхода. В при-

мере 1-13 иллюстрируется два способа расчета факториала: современный вариант для Delphi и старомодный вариант.

Пример 1-13. Использование переменной Result

```
// Расчет факториала в Delphi
function Factorial(Number: Cardinal): Int64;
var
  N: Cardinal;
begin
  Result := 1;
  for N := 2 to Number do
    Result := Result * N;
end;

// Расчет факториала в стандартном Паскале
function Factorial(Number: Integer): Integer;
var
  N, Result: Integer;
begin
  Result := 1;
  for N := 2 to Number do
    Result := Result * N;
  Factorial := Result;
end;
```

В Delphi строки и динамические массивы обычно инициализируются, но с `Result` ситуация иная. Фактически это не локальная переменная, а скрытый параметр `var`. Другими словами, при вызове ее нужно инициализировать. Проблема состоит в том, что Delphi делает это *не всегда*. Если ваша функция возвращает строку, интерфейс, динамический массив или переменную типа `Variant`, на всякий случай инициализируйте переменную `Result` пустой строкой, массивом или константой `Unassigned`.

Глава 2

Объектная модель Delphi

Поддержка объектно-ориентированного программирования в Delphi богата и разнообразна. В дополнение к традиционным классам и объектам, в Delphi также имеются интерфейсы (аналогично имеющимся в Java[1] и в технологии COM), обработка исключительных ситуаций и поддержка многопоточного программирования. В этой главе подробно раскрывается объектная модель Delphi. Вы должны быть уже знакомы со стандартным Паскалем и основными принципами объектно-ориентированного программирования.

Классы и объекты

Классы – это записи на стероидах. Как и запись, класс (`class`) описывает тип, в котором может быть какое угодно количество переменных, называемых *полями*. В отличие от записи, класс может также содержать функции и процедуры, называемые *методами*, и *свойства*. Класс также может быть потомком другого класса, наследуя при этом все поля, методы и свойства объекта-предка.

Объект – это динамический экземпляр класса. Объект всегда создается динамически, в «куче», поэтому ссылка на объект фактически является указателем (но при этом не требует обычного оператора разыменования). Когда вы присваиваете переменной ссылку на объект, Delphi копирует только указатель, а не весь объект. Используемый объект должен быть освобожден явно. Delphi не имеет автоматической

[1] Интерфейсы в Delphi не вполне аналогичны интерфейсам в Java. В частности, любой интерфейс в Delphi является расширением интерфейса IUnknown, в то время как интерфейсы в Java свободны от этого ограничения. – *Примеч. науч. ред.*

системы «сборки мусора» (смотрите раздел «Интерфейсы» далее в этой главе).

Для краткости термин *ссылка на объект* часто сокращается до *объект*, но, если быть точным, объект – это участок памяти, где Delphi хранит значения всех полей объекта. Ссылка на объект – это указатель на объект. В Delphi с объектом можно работать только через ссылку на него. Ссылка на объект обычно представлена в форме переменной, но она может быть также функцией или свойством другого объекта.

Класс – это также вполне конкретное понятие (аналогичное классу в Java, но отличное от C++). В Delphi класс представлен таблицей указателей на виртуальные методы и множеством другой, доступной только для чтения, информации об этом классе. *Ссылка на класс* – это указатель на эту таблицу. (В главе 3 «Информация о типе времени выполнения» подробно описаны структуры этих таблиц.) Чаще всего ссылка на класс используется для создания объектов или для проверки их типов, но ее можно использовать и во многих других ситуациях, в том числе передавать ее в качестве параметра или возвращать как результат функции. Тип ссылки на класс называется *метаклассом*.

В примере 2-1 приведено несколько определений классов. Определение класса – это определение типа, которое начинается с ключевого слова `class`. Определение класса содержит описания полей, методов и свойств, завершающееся ключевым словом `end`. Каждое описание метода аналогично объявлению прототипа процедуры или функции: далее в модуле вы должны привести его код (кроме абстрактных методов, которые обсуждаются в этой главе ниже).

Пример 2-1. Пример объявления классов и объектов

```
type
  TAccount = class
  private
    fCustomer: string;  // имя покупателя
    fNumber: Cardinal;  // номер счета
    fBalance: Currency; // текущий баланс на счете
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer; // фактический процент, умноженный на 1000
  end;
  TCheckingAccount = class(TAccount)
  private
    fReturnChecks: Boolean;
  end;
  TCertificateOfDeposit = class(TSavingsAccount)
  private
    fTerm: Cardinal;  // Срок депозита, в днях
  end;
```

```
var
  CD1, CD2: TAccount;
begin
  CD1 := TCertificateOfDeposit.Create;
  CD2 := TCertificateOfDeposit.Create;
  ...
```

На рис. 2-1 приведено распределение памяти для объектов и классов из примера 2-1. Переменные и относящиеся к ним объекты располагаются в доступной для записи памяти. Классы располагаются в защищенном участке памяти вместе с кодом программы.

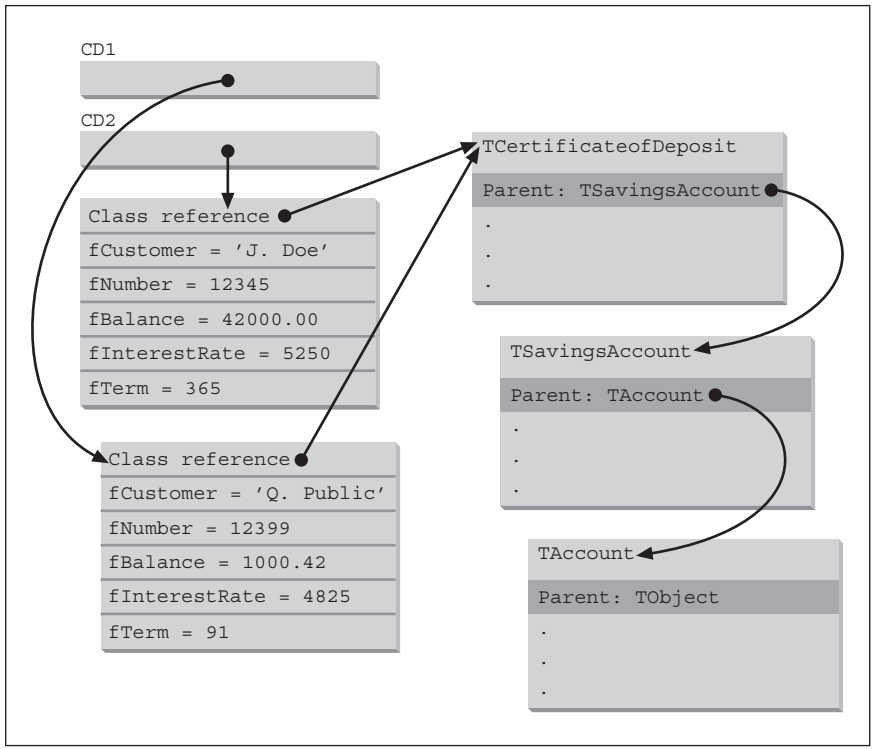


Рис. 2-1. Распределение памяти для объектов и классов

Объектная модель в Delphi аналогична другим языкам, например C++ и Java. В табл. 2-1 приведено краткое сравнение между Delphi и несколькими другими популярными языками программирования. Далее все указанные в таблице языковые возможности описаны подробно[1].

[1] В приведенной таблице автор рассматривает языковые возможности стандартного C++. В ряде широко известных реализаций этого языка (например, в такой популярной, как Microsoft Visual C++) имеется встроенная поддержка многопоточных приложений, OLE-автоматизации, интерфейсов, RTTI и целый ряд других возможностей. – *Примеч. науч. ред.*

Таблица 2-1. Delphi в мире языков программирования

Возможности языка	Delphi	Java	C++	Visual Basic
Наследование	✓	✓	✓	
Множественное наследование			✓	
Интерфейсы	✓	✓	a	✓
Единый корневой класс	✓	✓		
Метаклассы	✓	✓		
Статические поля (поля классов)		✓	✓	
Виртуальные методы	✓	✓	✓	
Абстрактные виртуальные методы	✓	✓	✓	
Статические методы (методы классов)	✓	✓	✓	
Динамические методы	✓			
«Сборка мусора»	b	✓		b
Вариантные типы	✓			✓
OLE-автоматизация	✓			✓
Статическая проверка типов	✓	✓	✓	
Обработка исключительных ситуаций	✓	✓	✓	✓
Перегрузка функций	✓	✓	✓	
Перегрузка операторов			✓	
Неклассовые функции	✓		✓	✓
Необъектные переменные	✓		✓	✓
Свойства	✓			✓
Информация о типах времени выполнения	✓	✓	c	
Шаблоны типов			✓	
Встроенная поддержка многопоточных приложений	✓	✓		
Передача сообщений	✓			
Встроенный ассемблер	✓		d	
Встроенные функции			✓	

[a] C++ может эмулировать интерфейсы с помощью абстрактных классов.

[b] Интерфейсы используют счетчики ссылок для определения своего жизненного цикла.

[c] Использование RTTI в C++ ограничено сравнением и преобразованием типов.

[d] Встроенный ассемблер не является частью стандарта языка C++, но большинство компиляторов, в том числе компиляторы фирмы Borland, поддерживают встроенный ассемблер как расширение языка.

Классы

Определение класса – это особый вид определения типа. В определении класса перечислены поля, методы и свойства класса. Вы можете объявить класс как в разделе интерфейса, так и в разделе реализации модуля, а методы – как и любую процедуру или функцию – только в разделе реализации. Реализация методов класса должна обязательно располагаться в том же модуле, что и определение класса.

Определение класса может иметь одну или более секций для различных уровней доступа (`private`, `public`, `published` или `automated`). Уровни доступа обсуждаются в этой же главе позднее. Вы можете располагать разделы в любом порядке и повторять разделы с одинаковым уровнем доступа.

Однин раздел может содержать любое количество полей, за которыми следуют описания методов и свойств. Описания методов и свойств можно смешивать между собой, но они всегда должны идти после описаний всех полей раздела. В отличие от Java и C++, в Delphi невозможно объявлять какие-либо типы внутри описания класса.

Класс может иметь один базовый класс, от которого наследуются все поля, свойства и методы. Если вы явно не указываете базовый класс, Delphi использует в качество базового класс `TObject`. Кроме того, в классе также может быть реализовано любое количество интерфейсов. Таким образом, объектная модель Delphi больше всего похожа на Java, где класс также может быть расширением только одного класса и реализовывать несколько интерфейсов.

В Delphi принято соглашение, при котором название типа обычно начинается с буквы T, как в `TObject`. Это только соглашение, а не правило языка. IDE, с другой стороны, всегда формирует названия классов форм, начиная с буквы T.

Ссылка на класс – это выражение, ссылающееся на определенный класс. Ссылка на класс не является первым объектом класса, как это принято в Java или Smalltalk, а служит для создания новых объектов, вызовов методов класса, проверки или преобразования типа объекта. Ссылка на класс реализуется как указатель на таблицу, содержащую информацию о классе, в частности, на таблицу виртуальных методов класса (VMT). (Подробности о структуре VMT читайте в главе 3).

Чаще всего ссылка на класс используется для создания экземпляров этого класса с помощью вызова конструктора. Ссылки на класс также

применяются для проверки типа объекта (с помощью оператора `is`) или преобразования объектов к определенному типу (оператор `as`). Обычно ссылка на класс – это имя класса, но это может быть и переменная типа «метакласс», функция или свойство, которое возвращает ссылку на класс. В примере 2-2 представлен образец описания класса.

Пример 2-2. Объявление класса и метакласса

```
type
  TComplexClass = class of TComplex; // тип «метакласс»
  TComplex = class(TPersistent)
  private
    fReal, fImaginary: Double;
  public
    constructor Create(Re: Double = 0.0); overload;
    constructor Create(Re, Im: Double); overload;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    function AsString: string;
  published
    property Real: Double read fReal write fReal;
    property Imaginary: Double read fImaginary write fImaginary;
  end;
```

Объекты

Объект – это динамический экземпляр класса. Объект содержит значения всех полей, объявленных классом и его предками. В объекте также имеется скрытое поле, в котором содержится ссылка на класс этого объекта.

Объекты всегда создаются динамически, в «куче», и потому ссылка на объект – это фактически указатель. Программист сам отвечает за создание объектов и за их своевременное освобождение. Для создания объекта используется ссылка на класс с указанием конструктора, например:

```
Obj := TSomeClass.Create;
```

Большинство конструкторов называются `Create`, но это лишь соглашение, а не требование Delphi. Вы можете иногда встретить конструкторы и с другими именами, обычно у старых классов, написанных еще до появления в Delphi поддержки перегрузки подпрограмм. Для максимальной совместимости с C++ Builder, в котором запрещается самостоятельное именование конструкторов, рекомендуется ограничиваться названием `Create` для всех конструкторов ваших объектов. Чтобы избавиться от объекта, который больше не требуется вашей программе, вызовите метод `Free`. Для гарантии правильного освобождения объекта, даже в случае возникновения исключительной ситуации, используйте обработчик `try-finally`. (Подробную информацию о конструкции `try-finally` смотрите в главе 1 «Delphi Pascal».) Например:

```
Obj := TSomeOtherClass.Create;
try
  Obj.DoSomethingThatMightRaiseAnException;
  Obj.DoSomethingElse;
finally
  Obj.Free;
end;
```

При освобождении глобальной переменной или поля всегда устанавливайте переменную в `nil` до освобождения объекта, чтобы в переменной не оставалось неверного указателя. Если деструктор или метод, вызываемый из деструктора, ссылается на эту переменную, ее обычно обнуляют, чтобы исключить потенциальные проблемы. Самый простой способ это сделать – вызвать процедуру `FreeAndNil` (из модуля `SysUtils`):

```
GlobalVar := TFruitWigglies.Create;
try
  GlobalVar.EatEmUp;
finally
  FreeAndNil(GlobalVar);
end;
```

Каждый объект содержит отдельную копию всех своих полей. Поля не могут разделяться между несколькими объектами. Если вы хотите, чтобы несколько объектов использовали одну переменную, объявите ее на уровне модуля или обращайтесь к ней неявно: многие объекты могут иметь разные указатели или ссылки на объекты, ссылающиеся на общие данных.

Наследование

Класс может быть наследником другого класса. Производный класс наследует все поля, методы и свойства базового класса. Delphi поддерживает только однократное наследование, т.е. класс имеет только один базовый. Базовый класс может иметь свой базовый класс и так далее, и класс наследует поля, свойства и методы всех родительских классов. Класс также может реализовывать любое количество интерфейсов (смотрите далее в этой главе). Так же, как в Java, но в отличие от C++, каждый класс является наследником одного корневого класса – `TObject`, который в Delphi является базовым по умолчанию.

Базовый класс – это непосредственный предок класса, который описан в его объявлении. Родительский класс – это либо базовый класс, либо любой другой класс в цепочке наследования вплоть до `TObject`. Так, в примере 2-1 для класса `TCertificateOfDeposit` базовым является `TSavingsAccount`, а родителями – классы `TObject`, `TAccount` и `TSavingsAccount`.

В классе TObject объявлено несколько методов и одно специальное, скрытое поле для хранения ссылки на класс объекта. Это скрытое поле указывает на таблицу виртуальных методов (VMT). Каждый класс имеет уникальную VMT, а все объекты используют одну и ту же VMT. В главе 5 «Справочник по языку» подробно описан класс TObject и его методы.

Вы можете присвоить ссылку на объект переменной, тип которой либо совпадает с типом класса, либо является одним из его родительских классов. Другими словами, объявленный тип ссылки не всегда совпадает с фактическим типом объекта. Присваивания в обратном направлении – присваивание ссылки на базовый класс переменной производного класса – не допускаются.

В Delphi сохраняется строгая типизация Паскаля, и компилятор выполняет проверку на основе объявленного типа объекта. Таким образом, все методы должны существовать в объявленном классе, и компилятором выполняется обычная проверка аргументов процедур и функций. Компилятор не обязательно привязывает вызов метода к конкретной его реализации. Если метод виртуальный, Delphi откладывает эту работу до времени выполнения и использует в этом случае истинный тип объекта для определения метода, который следует вызвать. Смотрите далее в этой главе раздел «Методы».

Для определения истинного типа объекта используется оператор is. Он возвращает истину, если указанный класс совпадает с классом объекта или с одним из его родительских классов. Если это не так или ссылка на объект нулевая, возвращается «ложь». Например:

```
if Account is TCheckingAccount then ... // Проверяет класс для Account
if Account is TObject then ...          // Истина, если Account не равно nil
```

Кроме того, получить ссылку на объект другого класса можно с помощью преобразования типа. Преобразование типа не изменяет объект; оно лишь дает вам иную ссылку на объект. Обычно для преобразования типов объектов используется оператор as. Этот оператор автоматически проверяет тип объекта и вызывает ошибку в том случае, если класс объекта не является потомком целевого класса. (Модуль SysUtils преобразует ошибку времени выполнения в исключительную ситуацию EinvalidCast.)

Другой способ преобразования ссылки на объект – указание имени целевого класса в стандартном формате преобразования типа, как при вызове функции. Этот вариант преобразования не выполняет проверку на соответствие типов, поэтому используйте его, только если вы уверены в безопасности преобразования, как в примере 2-3.

Пример 2-3. Использование статических преобразований типов

```
var
  Account: TAccount;
```

```
  Checking: TCheckingAccount;
begin
  Account  := Checking;                    // Допустимо
  Checking := Account;                     // Ошибка компиляции
  Checking := Account as TCheckingAccount; // Допустимо
  Account as TForm;                        // Ошибка при выполнении
  Checking := TCheckingAccount(Account);   // Допустимо, но не рекомендуется
  if Account is TCheckingAccount then      // Лучше
    Checking := TCheckingAccount(Account)
  else
    Checking := nil;
```

Поля

Поле – это переменная, являющаяся частью объекта. Класс может объявлять любое количество полей, и в каждом объекте содержится собственная копия всех полей, объявленных в его классе и во всех родительских классах. В других языках поля называются членами (data member), переменными экземпляра (instance variable) или атрибутами (attribute). В Delphi отсутствуют переменные класса, переменные экземпляра, статические члены или их эквиваленты (т. е. переменные, разделяемые между всеми объектами одного класса). Вместо них обычно используются переменные, объявленные на уровне модуля, обеспечивающие аналогичную функциональность.

Поле может быть любого типа, кроме полей из секции `published`. Эти поля обязательно должны относиться к какому-либо классу, который имеет информацию времени выполнения (т. е. компилированному с директивой `$M+`). Подробности смотрите в главе 3.

Когда Delphi впервые создает объект, все его поля пусты, т. е. указатели равны `nil`, строки и динамические массивы пусты, числа равны нулю, логические поля имеют значение «ложь», поля типа `Variant` – `Unassigned`. (Подробности смотрите в описании процедур `NewInstance` и `InitInstance` в главе 5.)

Производный класс может объявить поле с тем же именем, что и поле в родительском классе, в этом случае поле с тем же именем в родительском классе скрывается. Методы в производном классе будут ссылаться на новое поле, методы в классе предке – на старое.

Методы

Методы – это функции и процедуры, применяемые только к объектам определенного класса и его потомкам. В языке C++ методы называются «функции-члены» (member functions). Методы отличаются от обычных процедур и функций тем, что каждый метод содержит неявный параметр `Self`, ссылающийся на объект, вызвавший этот метод. `Self` аналогичен `this` в C++ и Java. Вызов метода выполняется аналогично

вызову функции или процедуры, но ему предшествует ссылка на объект, например:

```
Object.Method(Argument);
```

Метод класса относится к классу в целом и его потомкам. В методе класса `Self` ссылается не на отдельный объект, а на класс. В C++ для методов класса используется термин «статическая функция-член» (static member function).

Вы можете вызвать метод, объявленный в собственном классе объекта или в любом из его классов-предков. Если один и тот же метод объявлен в исходном классе и в производном, Delphi вызывает метод производного класса, как в примере 2-4.

Пример 2-4. Связывание статических методов

```
type
  TAccount = class
  public
    procedure Withdraw(Amount: Currency);
  end;
  TSavingsAccount = class(TAccount)
  public
    procedure Withdraw(Amount: Currency);
  end;
var
  Savings: TSavingsAccount;
  Account: TAccount;
begin
  ...
  Savings.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
  Account.Withdraw(1000.00);     // Вызывается TAccount.Withdraw
```

Обычный метод еще называют *статическим*[1], так как Delphi напрямую связывает вызов метода с его реализацией на этапе компиляции. Другими словами, выполняется статическое связывание. В C++ это называется «обычной функцией-членом» (ordinary member function), в Java – «окончательный метод» (final method). Большинство Delphi-программистов избегают термина «статический метод», предпочитая более простой – *метод* или даже *невиртуальный метод*.

Виртуальный метод – это метод, связывание которого выполняется во время выполнения, а не во время компиляции. При компиляции Delphi определяет по объявленному типу объекта, какие методы допустимы для использования с этим объектом. Вместо формирования прямой ссылки на определенный метод компилятор вставляет неяв-

[1] Не следует путать статические методы в Delphi со статическими функциями (методами) в C++ и Java. Близким аналогом статических функций (методов) C++ и Java в Delphi являются методы класса. – *Примеч. науч. ред.*

ную ссылку на метод, которая зависит от фактического класса объекта. Во время выполнения программы Delphi находит метод в таблицах класса (конкретно, в VMT) и вызывает метод фактического класса объекта. Истинный класс объекта может как совпадать с объявленным, так и быть производным от него, – это значения не имеет, т. к. ссылка на нужный метод будет взята из VMT.

Для объявления виртуального метода предназначена директива `virtual` в базовом классе, и директива `override` в описании нового метода в производном классе. В отличие от Java, все методы по умолчанию являются статическими, и для объявления виртуального метода требуется указание директивы `virtual`. В отличие от C++, для переопределения виртуального метода в производном классе требуется директива `override`. В примере 2-5 применяются виртуальные методы.

Пример 2-5. Связывание виртуальных методов

```
type
  TAccount = class
  public
    procedure Withdraw(Amount: Currency); virtual;
  end;
  TSavingsAccount = class(TAccount)
  public
    procedure Withdraw(Amount: Currency); override;
  end;
var
  Savings: TSavingsAccount;
  Account: TAccount;
begin
  ...
  Savings.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
  Account := Savings;
  Account.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
```

Директива `virtual` может быть заменена директивой `dynamic`. Семантики этих директив одинаковы, но реализации различны. Поиск виртуального метода в VMT выполняется быстрее, так как компилятор генерирует ссылку непосредственно на VMT. Поиск динамического метода происходит дольше, так как его вызов требует линейного поиска в таблице динамических методов класса (DMT). Если класс не переопределяет такой метод, поиск продолжается в DMT базового класса. Поиск продолжается во всех родительских классах либо до `TObject`, либо до того, как метод не будет найден. Динамические методы могут занимать меньше памяти, чем виртуальные, однако, если только вы не пишете заменитель VCL, используйте виртуальные методы, а не динамические. В главе 3 приведено полное описание реализации динамических и виртуальных методов.

Виртуальный или динамический метод может быть объявлен с директивой `abstract`, в этом случае класс не предоставляет реализации этого

метода. Вместо него метод *должен* быть переопределен в производных классах. В C++ для абстрактного метода принят термин «чисто виртуальный метод» (pure virtual method). При вызове конструктора для класса, который содержит абстрактный метод, компилятор выдает предупреждение, сообщающее о возможной ошибке. Возможно, вы хотите создать экземпляр одного из производных классов, в котором абстрактный метод переопределен. Класс, содержащий один и более абстрактных методов, часто называют *абстрактным*, хотя этот термин, строго говоря, обозначает классы, в котором все методы – абстрактные.

Если вы создаете абстрактный класс, который является наследником другого абстрактного класса, вам следует переопределить все абстрактные методы с указанием директив `override` и `abstract`. Delphi этого не требует, но этого требует здравый смысл. Новые определения ясно сообщают, что эти методы являются абстрактными. Иначе читатель вашего кода может не понять, следует ли реализовывать эти методы или оставить их абстрактными. Например:

```
type
  TBaseAbstract = class
    procedure Method; virtual; abstract;
  end;
  TDerivedAbstract = class(TBaseAbsract)
    procedure Method; override; abstract;
  end;
  TConcrete = class(TDerivedAbstract)
    procedure Method; override;
  end;
```

Метод класса или конструктор также могут быть виртуальными. В Delphi ссылка на класс – вполне реальная сущность, которая может быть присвоена переменной, передана как параметр или использована для вызова методов класса. Если конструктор является виртуальным, вы можете присвоить производный класс ссылке на базовый. Delphi найдет виртуальный конструктор в VMT и вызовет конструктор производного класса.

Методы (как и другие функции и процедуры) могут быть перегружены, т. е. несколько подпрограмм могут иметь одно и то же имя, если они имеют разные аргументы. Перегружаемые методы объявляются с директивой `overload`. Класс может перегрузить метод, унаследованный из базового класса. В этом случае директива `overload` требуется только в производном классе. Действительно, создатель базового класса не может предвидеть будущего и знать, что другие программисты захотят перегрузить унаследованный метод. Без директивы `overload` в производном классе новый метод «спрячет» метод базового класса, как в примере 2-6.

Пример 2-6. Перегрузка методов

```
type
  TAuditKind = (auInternal, auExternal, auIRS, auNasty);
  TAccount = class
  public
    procedure Audit;
  end;
  TCheckingAccount = class(TAccount)
  public
    procedure Audit(Kind: TAuditKind); // Скрывает TAccount.Audit
  end;
  TSavingsAccount = class(TAccount)
  public
    // Позволяет вызвать как TSavingsAccount.Audit, так и TAccount.Audit
    procedure Audit(Kind: TAuditKind); overload;
  end;
var
  Checking: TCheckingAccount;
  Savings: TSavingsAccount;
begin
  Checking := TCheckingAccount.Create;
  Savings := TSavingsAccount.Create;
  Checking.Audit;              // Ошибка, так как TAccount.Audit спрятан
  Savings.Audit;               // Правильно, так как Audit перегружен
  Savings.Audit(auNasty);      // Правильно
  Checking.Audit(auInternal); // Правильно
```

Конструкторы

Каждый класс может иметь один и более *конструкторов* (constructor), которые могут быть унаследованы от базового класса. Согласно общепринятой практике, конструкторы обычно называются `Create`, хотя вы можете использовать любое имя. Имена некоторых конструкторов начинаются с префикса `Create`, за которым следует дополнительная информация, например `CreateFromFile` или `CreateFromStream`. Однако обычно достаточно имени `Create`, а несколько конструкторов можно определить при помощи перегрузки методов. Другая причина использования перегрузки конструктора с именем `Create` заключается в необходимости обеспечить совместимость с C++ Builder, поскольку в C++ не разрешаются конструкторы с разными именами.

Вызов конструктора

Конструктор – это гибрид метода объекта и метода класса. Вы можете вызывать его как по ссылке на объект, так и по ссылке на класс. Delphi передает конструктору дополнительный, скрытый параметр для указания способа, при помощи которого он был вызван. Если конструктор был вызван с помощью ссылки на класс, Delphi вызывает метод класса `NewInstance` для создания нового экземпляра класса. После вызова `NewInstance` конструктор продолжает выполнение и инициализирует

объект. Delphi автоматически формирует блок `try-except`, и если при работе конструктора возникает исключительная ситуация, Delphi вызывает деструктор.

При вызове конструктора по ссылке на объект Delphi не формирует блок `try-except` и не вызывает метод `NewInstance`. Вместо этого конструктор вызывается как обычный метод. Это позволяет вызвать унаследованный конструктор без дополнительных накладных расходов.

Обычной ошибкой является попытка создать объект, вызвав конструктор со ссылкой на объект вместо использования ссылки на класс и присваивания результата переменной объектного типа:

```
var
  Account: TSavingsAccount;
begin
  Account.Create;                    // неправильно
  Account := TSavingsAccount.Create; // правильно
```

Одной из возможностей Delphi является предоставление программисту полного контроля над тем, когда, где и как вызывать унаследованный конструктор. Это позволяет создавать мощные и интересные классы, но также является источником частых ошибок.

Delphi всегда вначале исполняет конструктор производного класса, и только в том случае, если в этом классе вызывается конструктор базового класса, исполняет и его. C++ создает классы в обратном порядке, начиная с родительского класса и заканчивая производным классом. Таким образом, если класс C является наследником B, который является наследником A, Delphi сначала вызывает конструктор класса C, потом B и в конце A. В отличие от этого, C++ сначала запускает конструктор класса A, потом B и завершает C.

Виртуальные методы и конструкторы

Другой существенной разницей между C++ и Delphi является то, что в C++ конструктор всегда исполняется с виртуальной таблицей того класса, конструктор которого работает в данный момент, а в Delphi работают виртуальные методы производного класса, даже если исполняется конструктор базового класса. Поэтому следует быть осторожным при написании виртуальных методов, которые могут быть вызваны из конструктора. Возможно, что при вызове этого метода объект еще не полностью сформирован. Чтобы избежать подобных проблем, следует переопределить метод `AfterConstruction` и использовать его для кода, который требует полностью сформированного объекта. Если вы переопределяете `AfterConstruction`, не забудьте также вызвать унаследованный метод.

Один конструктор может вызывать другой конструктор. Delphi может определить, что вызов был сделан с помощью ссылки на объект (а имен-

но Self), и исполнить его как обычный метод. Наиболее частая причина вызова одного конструктора из другого – это желание поместить весь код инициализации в один конструктор. В примере 2-7 показаны несколько различных способов описания и вызова конструкторов.

Пример 2-7. Объявление и вызов конструкторов

```
type
  TCustomer = class ... end;
  TAccount = class
  private
    fBalance: Currency;
    fNumber: Cardinal;
    fCustomer: TCustomer;
  public
    constructor Create(Customer: TCustomer); virtual;
    destructor Destroy; override;
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer; // Умноженное на 1000
  public
    constructor Create(Customer: TCustomer); override; overload;
    constructor Create(Customer: TCustomer; InterestRate: Integer);
        overload;
    // Заметьте, что TSavingsAccount не требуется деструктор. Он только
    // наследует деструктор класса TAccount.
  end;

var
  AccountNumber: Cardinal = 1;

constructor TAccount.Create(Customer: TCustomer);
begin
  inherited Create;             // Вызов TObject.Create.
  fNumber := AccountNumber;     // Присваивание уникального
  // номера счета.
  Inc(AccountNumber);
  fCustomer := Customer;
  Customer.AttachAccount(Self); // Сообщает клиенту о создании счета.
end;

destructor TAccount.Destroy;
begin
  // Если конструктор прерывается до инициализации fCustomer, это
  // поле остается нулевым. Освобождение счета требуется только
  // если fCustomer не равно nil.
  if Customer <> nil then
    Customer.ReleaseAccount(Self);
  // Вызов TObject.Destroy.
  inherited Destroy;
end;
```

```
const
  DefaultInterestRate = 5000;  // 5%, умноженное на 1000

constructor TSavingsAccount.Create(Customer: TCustomer);
begin
  // Вызов «братского» конструктора.
  Create(Customer, DefaultInterestRate);
end;

constructor TSavingsAccount(Customer: TCustomer; InterestRate:Integer);
begin
  // Вызов TAccount.Create.
  inherited Create(Customer);
  fInterestRate := InterestRate;
end;
```

Деструкторы

Деструкторы, так же как и конструкторы, получают дополнительный скрытый параметр. Первый вызов деструктора передает в дополнительном параметре значение True, по которому Delphi «понимает», что необходимо вызвать `FreeInstance` для освобождения объекта. Если деструктор вызывает унаследованный деструктор, Delphi передает в скрытом параметре значение False, чтобы предотвратить попытку повторного освобождения того же объекта.

В классе обычно имеется один деструктор с названием `Destroy`. Delphi позволяет объявить и дополнительные деструкторы, но это не принесет вам никакой пользы, лишь создаст путаницу.

Перед запуском тела деструктора вызывается виртуальный метод `BeforeDestruction`. Вы можете переопределить `BeforeDestruction`, чтобы изменить состояние программы или выполнить другие действия, которые должны завершиться до начала работы конструктора. Это позволяет написать код класса более корректным образом, не беспокоясь о том, как деструктор базового класса будет вызываться производными классами.

Иногда при написании класса вам может потребоваться перекрыть деструктор `Destroy`, но не переопределяйте метод `Free`. При освобождении объекта следует вызывать именно метод `Free`, а не деструктор. Разница имеет значение, т. к. `Free` проверяет, не является ли ссылка на объект нулевой, и вызывает `Destroy` только для ненулевой ссылки. В особых случаях класс может переопределять метод `Free` (как, например, в классе `TInterface` в редко используемом модуле `VirtIntf`), что делает вызов `Free` вместо `Destroy` еще более актуальным.

Если конструктор или метод AfterConstruction создает исключительную ситуацию, Delphi автоматически вызывает деструктор объекта. Создавая деструктор, следует помнить, что уничтожаемый объект может быть не полностью проинициализирован. Delphi может гарантировать, что все поля до начала работы конструктора равны нулю, но если исключительная ситуация возникает в середине его работы, некоторые поля могут быть проинициализированы, а некоторые – оставаться нулевыми. Если деструктор лишь освобождает объекты и указатели, можете не беспокоиться, т. к. и метод Free, и процедура FreeMem проверяют указатели на nil. Если же ваш деструктор вызывает другие методы, всегда выполняйте проверку на нулевые указатели.

Жизненный цикл объекта

Для большинства объектов сначала вызывается деструктор, затем объект используется и в конце освобождается методом Free. Delphi выполняет за вас все мелочи. Однако иногда полезно знать больше о внутренних механизмах объектной модели Delphi. В примере 2-8 приведены методы, которые Delphi вызывает или эмулирует при создании и освобождении объекта.

Пример 2-8. Жизненный цикл объекта

```
type
  TSomething = class
    procedure DoSomething;
  end;
var
  Ref: TSomething;
begin
  Ref := TSomething.Create;
  Ref.DoSomething;
  Ref.Free;
end;

// Скрытый код конструктора выглядит примерно так:
function TSomething.Create(IsClassRef: Boolean): TSomething;
begin
  if IsClassRef then
  try
    // Allocate the new object.
    Self := TSomething.NewInstance;

    // Метод NewInstance инициализирует объект так же, как это
    // делает InitInstance. Если вы переопределите NewInstance
    // и не вызовете унаследованный NewInstance, обязательно
    // вызовите InitInstance. Этот вызов показан ниже, чтобы
    // вы знали, что происходит, но фактически Delphi не вызывает
     // InitInstance.
    InitInstance(Self);
```

```
    // Выполнение реальной работы конструктора, но без
    // накладных расходов, связанных со ссылкой на класс. Delphi
    // фактически не выполняет рекурсивный вызов конструктора.
    Self.Create(False);

    Self.AfterConstruction;
  except
    // Если происходит исключительная ситуация, Delphi автоматически
    // вызывает деструктор объекта.
    Self.Destroy;
  end
  else
    Self.Create(False);
  Result := Self;
end;

// Скрытый код деструктора выглядит примерно так:
procedure TSomething.Destroy(Deallocate: Boolean);
begin
  if Deallocate then
    Self.BeforeDestruction;

  // Delphi фактически не вызывает деструктор рекурсивно, но именно
  // здесь выполняется реальная работа деструктора.
  Self.Destroy(False);

  if Deallocate then
  begin
    // Delphi фактически не вызывает CleanupInstance. Вместо этого
    // очистку выполняет метод FreeInstance. Если вы переопределяете
    // FreeInstance и не вызываете унаследованный FreeInstance, то
    // для очистки строк, динамических массивов и полей типа Variant
    // нужно вызвать CleanupInstance.
    Self.CleanupInstance;
    // Вызов FreeInstance для освобождения памяти объекта.
    Self.FreeInstance;
  end;
end;
```

Уровни доступа

Как в C++ и Java, в Delphi имеются различные уровни доступа, которые определяют, какие объекты могут иметь доступ к полям, методам и свойствам другого объекта. Существуют следующие уровни доступа:

private

Объявления, обозначенные директивой `private` («закрытые»), доступны только собственным методам класса, а также всем методам, процедурам и функциям, определенным в секции реализации того же модуля. В Delphi отсутствуют аналогичные C++ «дружественные» объявления или аналогичный Java доступ на уровне класса. В Delphi для этого можно объявить такие классы в том же модуле,

тем самым дав им доступ к секциям private и protected определенных в модуле классов.

protected

Объявленные в секции protected («защищенные») поля, методы и свойства доступны из любого метода данного класса и его потомков. Производные классы могут находиться в разных модулях.

public

Public («открытые» или «общедоступные») методы предоставляют неограниченный уровень доступа. Все методы, функции и процедуры имеют доступ к полям и методам, объявленным в секции public. Если не используется директива компилятора $M+ (смотрите подробнее в главе 8 «Директивы компиляторов»), по умолчанию принимается уровень доступа public.

published

Секции published («публикуемые») аналогичны секциям public, за исключением того, что для объявлений секции published создается информация времени выполнения. Не все объявления могут быть помещены в секцию published (смотрите подробности в главе 3). Если данный класс или его базовый класс компилировались с директивой $M+, уровень доступа по умолчанию – published.

IDE Delphi помещает объявления полей и методов в первую неименованную секцию описания формы. Поскольку класс TForm является наследником TPersistent, который использует директиву $M+, для первой секции установлен уровень доступа published. Когда Delphi загружает описание формы из *dfm*-файла, информация для построения объекта формы берется из секции published. IDE использует начальную, неименованную секцию класса. Если вы внесете изменения в эту секцию, есть риск нарушить работу редактора форм с этим классом.

automated

Объявления automated («автоматические») аналогичны public, за исключением того, что для них Delphi формирует дополнительную информацию времени выполнения для поддержки работы серверов OLE-автоматизации. Объявления automated – устаревшая технология; вместо этого следует использовать редактор библиотеки типов, но пока директива automated сохранена в языке для обратной совместимости. Из будущих версий Delphi она может быть исключена. Более подробно секции automated описаны в главе 3.

Производный класс позволяет увеличить уровень доступа свойства путем его повторного объявления с другим уровнем доступа (например, можно изменить protected на public). Уменьшить уровень доступа свойства нельзя, также нельзя изменить уровень доступа для поля или метода. Можно переопределить виртуальный метод и объявить новый

метод как с тем же, так и с более высоким уровнем доступа, но уменьшить уровень доступа нельзя.

Свойства

Свойство выглядит как поле, но работает как метод. Свойства заменяют методы для доступа и изменения полей класса (иногда называемые считыватели (getters) и установщики (setters)), но обладают значительно большей гибкостью и мощностью. Свойства являются жизненно важной частью IDE Delphi и используются во многих других ситуациях.

Для каждого свойства определены считыватели и установщики, позволяющие получать и устанавливать их значения. Считыватель может быть именем поля, некоторой частью сложного поля или методом, возвращающим значение свойства. Вы можете создать свойство только для чтения, не указывая установщика, и свойство только для записи, не указывая считывателя, хотя применение такой конструкции несколько ограничено. Отсутствие и считывателя, и установщика не имеет смысла, и Delphi не позволяет это сделать.

Чаще всего в качестве считывателя и установщика указываются поля и имена методов, но вы также можете ссылаться на часть сложного поля (записи или массива). Если считыватель или установщик ссылаются на элемент массива, индекс массива должен быть константой и поле не может быть динамическим массивом. Записи и массивы могут быть вложенными, и вы даже можете использовать вариантные записи. В примере 2-9 приведен расширенный тип прямоугольника, аналогичный типу `TRect` из Windows, но являющийся классом и имеющий свойства и методы.

Пример 2-9. Считыватели и установщики свойств

```
TRectEx = class(TPersistent)
  private
    R: TRect;
    function GetHeight: Integer;
    function GetWidth: Integer;
    procedure SetHeight(const Value: Integer);
    procedure SetWidth(const Value: Integer);
  public
    constructor Create(const R: TRect); overload;
    constructor Create(Left, Top, Right, Bottom: Integer); overload;
    constructor Create(const TopLeft, BottomRight: TPoint); overload;

    procedure Assign(Source: TPersistent); override;

    procedure Inflate(X, Y: Integer);
    procedure Intersect(const R: TRectEx);
    function IsEmpty: Boolean;
```

Скрытый конструктор

Иногда класс может быть не предназначен для общего пользования, а являться вспомогательным, полностью подчиненным другому классу. В этом случае вы, возможно, захотите сделать конструктор вспомогательного класса закрытым или защищенным, что сделать не так просто. В `TObject` объявляется открытый конструктор `Create`. Даже если конструкторы вспомогательного класса являются закрытыми или защищенными, остается возможность вызвать открытый конструктор, унаследованный от `TObject`.

И хотя вы не можете изменить уровень доступа унаследованного конструктора `Create`, можно скрыть его другим открытым конструктором. Поскольку производный конструктор не предназначен для запуска, он может вызывать исключительную ситуацию. Например:

```
type
  TPublic = class;
  TPrivateHelper = class
  private
    // TPublic - это единственный класс, которому
    // разрешен вызов реального конструктора:
    constructor Create(Owner: TPublic);
      overload;
  public
    // Скрывает TObject.Create на тот случай, если кто-то
    // попытается создать экземпляр класса TPrivateHelper.
    constructor Create;
      reintroduce; overload;
  end;
  TPublic = class
  private
    fHelper: TPrivateHelper;
  public
    constructor Create;
    destructor Destroy;
  end;

constructor TPrivateHelper.Create;
begin
  raise Exception.Create('Ошибка программирования')
end;

constructor TPublic.Create;
begin
  // This is the only place where
  // TPrivateHelper is created.
  // Единственное место, где создается TPrivateHelper
  fHelper := TPrivateHelper.Create(Self);
end;
```

```
    function IsEqual(const R: TRectEx): Boolean;
    procedure Offset(X, Y: Integer);
    procedure Union(const R: TRectEx);

    property TopLeft: TPoint read R.TopLeft write R.TopLeft;
    property BottomRight: TPoint read R.BottomRight write R.BottomRight;
    property Rect: TRect read R write R;
    property Height: Integer read GetHeight write SetHeight;
    property Width: Integer read GetWidth write SetWidth;
  published
    property Left: Integer read R.Left write R.Left default 0;
    property Right: Integer read R.Right write R.Right default 0;
    property Top: Integer read R.Top write R.Top default 0;
    property Bottom: Integer read R.Bottom write R.Bottom default 0;
  end;
```

Свойства-массивы

Свойства могут быть либо скалярами, либо массивами. Свойство-массив не может быть публикуемым (published), но может использоваться во многих других ситуациях. Индекс массива может иметь любой тип, допускаются также многомерные свойства-массивы. Для свойств-массивов следует применять методы-считыватели и методы-установщики – вы не можете установить прямое соответствие между свойством типа «массив» и полем типа «массив».

Одно из свойств-массивов можно назначить как свойство по умолчанию. На свойство по умолчанию можно ссылаться с помощью указания объекта и индекса массива без указания имени свойства, как в примере 2-10.

Пример 2-10. Использование свойства-массива по умолчанию

```
type
  TExample = class
    ...
    property Items[I: Integer]: Integer read GetItem write SetItem;
    property Chars[C: Char]: Char read GetChar write SetChar; default;
  end;
var
  Example: TExample;
  I: Integer;
  C: Char;
begin
  Example := TExample.Create;
  I := Example.Items[4];     // Имя свойства должно быть указано явно
  C := Example['X'];         // Свойство-массив по умолчанию
  C := Example.Chars['X'];   // Аналогично предыдущей строке
```

Индексированные свойства

Вы можете связать несколько свойств с одним считывателем или установщиком путем указания номера индекса для каждого из таких

свойств. Значение индекса передается в считыватель и установщик для указания различия между свойствами.

Допускается совместное использование индексов массивов и индексов свойств. Считыватели и установщики принимают в качестве первого аргумента индекс массива, а в качестве второго – индекс свойства.

Значения по умолчанию

Свойства могут также иметь директивы `stored` и `default`. Эта информация не имеет никакого семантического значения в языке Delphi Pascal, но используется IDE для хранения описания форм. Значения директивы `stored` – это логическая константа, поле типа `Boolean` или метод, не имеющий аргументов и возвращающий логическое значение. Значение директивы `default` – константа того же типа, что и тип свойства. Значение по умолчанию могут иметь только свойства перечисляемого, целого или множественного типов. Директивы `stored` и `default` имеют смысл только для публикуемых свойств.

Чтобы отличить массив по умолчанию от значения по умолчанию, директиву `default` для массива располагают после точки с запятой, завершающей объявление свойства. Значение по умолчанию записывается как часть объявления свойства. Обратитесь к главе 5 за более подробной информацией о директиве `default`.

Использование свойств

Обычный подход при написании классов в Delphi – это объявление всех полей закрытыми и создание для доступа к полям открытых свойств. В Delphi применение свойств для непосредственного обращения к полям не приводит к уменьшению производительности. Используя свойства, вы получаете возможность в будущем изменить внутреннюю реализацию класса, например добавить проверку на правильность устанавливаемых значений поля. При помощи свойств также можно устанавливать ограничения доступа, например, использовать свойство только для чтения для доступа к полю, значение которого нельзя изменять. В примере 2-11 показано несколько способов объявления и применения свойств.

Пример 2-11. Объявление и использование свойств

```
type
  TCustomer = record
    Name: string;
    TaxIDNumber: string[9];
  end;
  TAccount = class
  private
    fCustomer: TCustomer;
    fBalance: Currency;
    fNumber: Cardinal;
    procedure SetBalance(NewBalance: Currency);
```

```
  published
    property Balance: Currency read fBalance write SetBalance;
    property Number: Cardinal read fNumber;   // Номер счета нельзя менять
    property CustName: string read fCustomer.Name;
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer;
  published
    property InterestRate: Integer read fInterestRate
        write fInterestRate default DefaultInterestRate;
  end;
  TLinkedAccount = class(TObject)
  private
    fAccounts: array[0..1] of TAccount;
    function GetAccount(Index: Integer): TAccount;
  public
    // Два метода доступа свойств к массиву: с помощью индекса
    // и ссылки на элемент массива.
    property Checking: TAccount index 0 read GetAccount;
    property Savings:  TAccount read fAccounts[1];
  end;
  TAccountList = class
  private
    fList: TList;
    function GetAccount(Index: Integer): TAccount;
    procedure SetAccount(Index: Integer; Account: TAccount);
    function GetCount: Integer;
  protected
    property List: TList read fList;
  public
    property Count: Integer read GetCount;
    property Accounts[Index: Integer]: TAccount read GetAccount
        write SetAccount; default;
  end;

procedure TAccount.SetBalance(NewBalance: Currency);
begin
  if NewBalance < 0 then
    raise EOverdrawnException.Create;
  fBalance := NewBalance;
end;

function TLinkedAccount.GetAccount(Index: Integer): TAccount;
begin
  Result := fAccounts[Index]
end;

function TAccountList.GetCount: Integer;
begin
  Result := List.Count
end;
```

```
function TAccountList.GetAccount(Index: Integer): TAccount;
begin
  Result := List[Index]
end;

procedure TAccountList.SetAccount(Index: Integer; Account: TAccount);
begin
  fList[Index] := Account
end;
```

Объекты-свойства

Свойства, содержащие значение типа «объект», требуют дополнительного рассмотрения. Лучше всего работать со объектами-свойствами таким образом, чтобы объект-владелец сам занимался их управлением. Другими словами, не сохраняйте ссылки на другие объекты, а держите собственную копию объекта-свойства. Используйте установщик для сохранения объекта путем его копирования. IDE Delphi требует такого поведения от публикуемых свойств, то же самое имеет смысл и для всех остальных объектов-свойств.

Единственное исключение из правила работы с объектами-свойствами – это использование их для хранения ссылки на компонент из формы. В этом случае свойство должно содержать ссылку на объект, а не копию компонента.

IDE Delphi хранит ссылки на компоненты в *dfm*-файле в виде имени компонента. Когда загружается *dfm*-файл, Delphi ищет имя компонента для восстановления ссылки на объект. Если вы должны хранить весь компонент внутри другого компонента, придется перечислить все свойства внутреннего компонента.

Наследуйте класс объекта-свойства от `TPersistent` и переопределите метод `Assign`. Реализуйте установщик вашего свойства с помощью вызова метода `Assign`. (Использование класса `TPersistent` из модуля `Classes` – это не обязательный, но наиболее простой способ копирования объекта. Иначе вам придется продублировать метод `Assign` во всех используемых классах.) Считыватель может обеспечить непосредственный доступ к полю. Если класс объекта-свойства имеет событие `OnChange`, есть смысл переписать его так, чтобы объект-хозяин узнавал обо всех изменениях. В примере 2-12 показана типичная схема работы с объектом-свойством. В примере определяется графический элемент, который периодически выводит картинку внутри себя, при необходимости дублируя ее в виде мозаики. В свойстве `Bitmap` хранится объект `TBitmap`.

Пример 2-12. Объявление и использование объекта-свойства

```
unit Tile;

interface
```

```
uses SysUtils, Classes, Controls, Graphics;

type
  // Создание мозаики
  TTile = class(TGraphicControl)
  private
    fBitmap: TBitmap;
    procedure SetBitmap(NewBitmap: TBitmap);
    procedure BitmapChanged(Sender: TObject);
  protected
    procedure Paint; override;
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
  published
    property Align;
    property Bitmap: TBitmap read fBitmap write SetBitmap;
    property OnClick;
    property OnDblClick;
    // Многие другие свойства также могут быть полезны, но здесь пропущены
    // для экономии места. Полный список можно найти в описании TControl.
  end;

implementation

{ TTile }

// Создание TBitmap при формировании управляющего элемента.
constructor TTile.Create(Owner: TComponent);
begin
  inherited;
  fBitmap := TBitmap.Create;
  fBitmap.OnChange := BitmapChanged;
end;

// Освобождение TBitmap при уничтожении управляющего элемента.
destructor TTile.Destroy;
begin
  FreeAndNil(fBitmap);
  inherited;
end;

// При изменении картинки управляющий элемент перерисовывается.
procedure TTile.BitmapChanged(Sender: TObject);
begin
  Invalidate;
end;

// Рисование управляющего элемента путем создания мозаики из картинки.
// Если картинки нет, ничего не рисуется.
procedure TTile.Paint;
```

```
var
  X, Y: Integer;
begin
  if (Bitmap.Width = 0) or (Bitmap.Height = 0) then
    Exit;

  Y := 0;
  while Y < ClientHeight do
  begin
    X := 0;
    while X < ClientWidth do
    begin
      Canvas.Draw(X, Y, Bitmap);
      Inc(X, Bitmap.Width);
    end;
    Inc(Y, Bitmap.Height);
  end;
end;

// Создание новой картинки копированием объекта TBitmap.
procedure TTile.SetBitmap(NewBitmap: TBitmap);
begin
  fBitmap.Assign(NewBitmap);
end;

end.
```

Интерфейсы

Интерфейс определяет тип, состоящий из абстрактных виртуальных методов[1]. И хотя класс может быть наследником только одного базового класса, он может реализовывать любое количество интерфейсов. Интерфейс имеет некоторое сходство с абстрактным классом (т. е. классом без полей, все методы которого абстрактные), но в Delphi есть особые приемы для работы с интерфейсами. Интерфейсы Delphi в чем-то напоминают интерфейсы COM (Component Object Model, объектная модель компонентов), но для использования интерфейсов Delphi не требуется знания COM, и их можно применять для многих других целей. Вы можете объявить новый интерфейс как наследник существующего. Объявление интерфейса может содержать описания методов и свойств, но не может содержать описания полей. Как все классы в Delphi являются потомками `TObject`, так и все интерфейсы в Delphi явля-

[1] Помимо методов, интерфейс может содержать также и свойства (интерфейс не может содержать поля). Следует заметить, что поскольку интерфейс лишь объявляет, но не реализует методы, применение директивы `abstract` в объявлении методов интерфейса лишено смысла и не допускается. – *Примеч. науч. ред.*

ются расширениями IUnknown. В интерфейсе IUnknown объявлено три метода: _AddRef, _Release и QueryInterface. Если вы знаете COM, они вам знакомы. Первые два метода управляют счетчиком ссылок для определения времени жизни объекта, реализующего интерфейс. Третий метод обеспечивает доступ к другим интерфейсам, которые реализует объект.

Если вы объявляете класс, реализующий один и более интерфейсов, вы должны обеспечить реализацию всех методов, объявленных во всех интерфейсах. Класс может либо реализовать методы интерфейса, либо делегировать реализацию свойству, значение которого является интерфейсом. Простейший способ реализации методов _AddRef, _Release и QueryInterface – это унаследовать их от TInterfacedObject или одного из его производных классов, но вы также можете использовать другие базовые классы, если хотите создать эти методы самостоятельно.

Класс реализует каждый из методов интерфейса путем объявления метода с тем же именем, аргументами и соглашением о вызове. Delphi автоматически ставит методы класса в соответствие с методами интерфейса. Если вы хотите дать методу другое имя, вы можете перенаправить на него метод интерфейса. Он должен иметь те же аргументы и соглашение о вызове, что и метод интерфейса. Эта возможность особенно важна, когда класс реализует несколько интерфейсов, содержащих методы с одинаковыми именами. Подробнее о перенаправлении методов смотрите описание ключевого слова class в главе 5 .

Класс может делегировать реализацию интерфейса свойству, содержащему директиву implements. Тип значения свойства должен соответствовать интерфейсу, который класс собирается реализовать. Когда объект преобразуется к этому типу интерфейса, Delphi автоматически извлекает значение свойства и возвращает соответствующий интерфейс. Смотрите описание директивы implements в главе 5.

Для каждого неделегированного интерфейса компилятор создает скрытое поле, в котором хранится указатель на VMT интерфейса. Поле или поля интерфейса следуют сразу за скрытыми VMT-полями объекта. Так же как ссылка на объект является указателем на скрытое VMT-поле объекта, так и ссылка на интерфейс является указателем на скрытое VMT-поле интерфейса. Delphi автоматически инициализирует скрытые поля при создании объекта. В главе 3 обсуждается, как компилятор использует RTTI для отслеживания VMT и скрытого поля.

Счетчик ссылок

Компилятор генерирует для управления временем жизни интерфейсных объектов вызовы методов _AddRef и _Release. Для использования автоматического счетчика ссылок объявите переменную интерфейсного типа. Когда происходит присваивание интерфейсной переменной, Delphi автоматически вызывает метод _AddRef, а когда переменная выходит из области действия, – метод _Release.

Поведение _AddRef и _Release полностью зависит от вас. Если вы наследуете объект от TInterfacedObject, они реализуют счетчик ссылок, причем _AddRef увеличивает счетчик ссылок, а _Release – уменьшает его. Когда счетчик уменьшается до 0, _Release освобождает объект. Если вы используете другой класс в качестве базового, то эти методы можно применять для любых целей. Однако вы должны корректно реализовать метод QueryInterface, так как Delphi использует его для реализации работы оператора as.

Преобразования типов

Delphi вызывает метод QueryInterface как часть реализации оператора as для интерфейса. При помощи оператора as можно преобразовать один интерфейс к любому другому интерфейсу. Для получения ссылки на новый интерфейс Delphi вызывает QueryInterface. Если в ходе выполнения последнего возникает ошибка, оператор as генерирует ошибку времени выполнения. (Модуль SysUtils преобразует ошибку времени выполнения в исключительную ситуацию EIntfCastError.)

Вы можете реализовать QueryInterface по собственному усмотрению, но вероятно последуете тому подходу, который использован в TInterfacedObject. В примере 2-13 приведен класс, который правильно реализует метод QueryInterface, но для _AddRef и _Release в нем оставлены «заглушки». Далее в этой главе вы убедитесь, насколько полезен может быть такой класс.

Пример 2-13. Базовый интерфейсный класс без счетчика ссылок

```
type
  TNoRefCount = class(TObject, IUnknown)
  protected
    function QueryInterface(const IID:TGUID; out Obj):HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;

function TNoRefCount.QueryInterface(const IID:TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := 0
  else
    Result := Windows.E_NoInterface;
end;

function TNoRefCount._AddRef: Integer;
begin
  Result := -1
end;

function TNoRefCount._Release: Integer;
begin
  Result := -1
end;
```

Интерфейсы и объектно-ориентированное программирование

Самый полезный вариант применения интерфейсов – это отделение наследования типов от наследования классов. Наследование классов – это эффективное средство повторного использования кода. Производный класс может наследовать поля, методы и свойства базового класса и, следовательно, не реализовывать общие методы заново. В строго типизированном языке, таком как Delphi Pascal, компилятор рассматривает класс как тип, и, таким образом, наследование классов становится синонимом наследования типов. Но в лучшем из возможных миров, однако, типы и классы должны быть полностью разделены.

В учебниках по объектно-ориентированному программированию наследование часто описывается как отношение «is» («это»), например, `TSavingsAccount` «это» `TAccount`. Та же идея прослеживается и в операторе `is` (это), с помощью которого вы можете проверить, что переменная `Account – is TSavingsAccount`.

Но за пределами учебников, простая идея отношения «это» не всегда работает. Квадрат «это» прямоугольник, но это не значит, что вы захотите создать `TSquare` как класс, производный от `TRectangle`. Прямоугольник «это» многоугольник, но вы не обязательно создадите класс `TRectangle` на основе `TPolygon`. Наследование классов требует, чтобы в производном классе были все поля, объявленные в базовом, но в описанных случаях производному классу это не требуется. Объект `TSquare` может обойтись без хранения значений длины каждой из своих сторон. Однако объект `TRectangle` должен иметь две длины. Объекту `TPolygon` требуется хранить информацию о многих сторонах и вершинах. Решение состоит в отделении наследования типов (квадрат «это» прямоугольник, прямоугольник «это» многоугольник) от наследования классов (класс C наследует поля и методы класса B, который наследует поля и методы класса A). Для наследования типов можно использовать интерфейсы и оставить наследованию классов то, что у него получается лучше всего – наследование полей и методов.

Другими словами, `ISquare` является наследником `IRectangle`, который является наследником `IPolygon`. Для наследования интерфейсов выполняется отношение «это». Независимо от наследования интерфейсов, класс `TSquare` реализует `ISquare`, `IRectangle` и `IPolygon`. `TRectangle` реализует `IRectangle` и `IPolygon`.

В программировании технологии COM принято соглашение, что названия интерфейсов начинаются с буквы `I`. Delphi следует этому соглашению для всех интерфейсов. Заметьте, что это лишь полезное соглашение, а не требование языка.

Что касается реализации повторного использования кода, то вы можете объявить для этого дополнительные классы. Например, класс `TBaseShape` реализует методы и поля, общие для всех фигур. `TRectangle` является наследником `TBaseShape` и реализует методы тем способом, кото-

рый имеет смысл для прямоугольников. TPolygon также является наследником TBaseShape и реализует свои методы способом, подходящим для всех остальных видов многоугольников. Программа рисования может использовать эти классы, работая через интерфейс IPolygon. В примере 2-14 показаны упрощенные классы и интерфейсы, описанные этой схемой. Заметьте, что каждый интерфейс имеет в своем описании GUID (Globally Unique Identifier – глобально уникальный идентификатор). GUID необходим для использования метода QueryInterface. Если вам потребуется GUID (например, для непосредственного вызова QueryInterface), можно использовать имя интерфейса. Delphi автоматически преобразует имя интерфейса в его GUID.

Пример 2-14. Разделение иерархий типов и классов

```
type
  IShape = interface
  ['{5OF6D851-F4EB-11D2-88AC-00104BCAC44B}']
    procedure Draw(Canvas: TCanvas);
    function GetPosition: TPoint;
    procedure SetPosition(Value: TPoint);
    property Position: TPoint read GetPosition write SetPosition;
  end;

  IPolygon = interface(IShape)
  ['{5OF6D852-F4EB-11D2-88AC-00104BCAC44B}']
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  IRectangle = interface(IPolygon)
  ['{5OF6D853-F4EB-11D2-88AC-00104BCAC44B}']
  end;
  ISquare = interface(IRectangle)
  ['{5OF6D854-F4EB-11D2-88AC-00104BCAC44B}']
    function Side: Integer;
  end;

  TBaseShape = class(TNoRefCount, IShape)
  private
    fPosition: TPoint;
    function GetPosition: TPoint;
    procedure SetPosition(Value: TPoint);
  public
    constructor Create; virtual;
    procedure Draw(Canvas: TCanvas); virtual; abstract;
    property Position: TPoint read fPosition write SetPosition;
  end;
  TPolygon = class(TBaseShape, IPolygon)
  private
    fVertices: array of TPoint;
```

```
  public
    procedure Draw(Canvas: TCanvas); override;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  TRectangle = class(TBaseShape, IPolygon, IRectangle)
  private
    fRect: TRect;
  public
    procedure Draw(Canvas: TCanvas); override;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  TSquare = class(TBaseShape, IPolygon, IRectangle, ISquare)
  private
    fSide: Integer;
  public
    procedure Draw(Canvas: TCanvas); override;
    function Side: Integer;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
```

Производный класс наследует интерфейсы, реализованные родительскими классами. То есть, если `TRectangle` является наследником `TBaseShape`, и `TBaseShape` реализует `IShape`, то `TRectangle` реализует `IShape`. Наследование интерфейсов работает несколько по-другому, это лишь средство для сокращения набора текста, чтобы не приходилось заново перепечатывать длинные описания методов. Если класс реализует интерфейс, это не значит, что он автоматически реализует все его родительские интерфейсы. Класс реализует только те интерфейсы, которые перечислены в его описании (и описаниях классов-предков). Поэтому хотя `IRectangle` — наследник `IPolygon`, если вы хотите, чтобы класс `TRectangle` реализовывал интерфейсы `IRectangle` и `IPolygon`, в объявлении класса `TRectangle` должны быть явно указаны и `IRectangle`, и `IPolygon`.

При создании иерархии типов вы, возможно, не захотите использовать счетчик ссылок. Вместо этого можно положиться на явное управление памятью, как для обычных объектов Delphi. В этом случае лучше всего реализовать методы `_AddRef` и `_Release` как заглушки, как это сделано в классе `TNoRefCount` в примере 2-13. Только не храните в других переменных устаревшие ссылки. Переменная, которая ссылается на уже уничтоженный объект, наверняка создаст вам проблемы, когда

Delphi автоматически вызовет для нее метод _Release. Другими словами, в программе не должно быть переменных, содержащих неверные указатели, а при работе с интерфейсами, не использующими средства подсчета ссылок, вы просто вынуждены так поступать.

COM и CORBA

Интерфейсы Delphi также полезны для использования и реализации объектов COM и CORBA. Вы можете создать сервер COM, реализующий много интерфейсов, и Delphi автоматически будет управлять агрегированием. Библиотека Delphi содержит много классов, облегчающих создание серверов COM, «фабрик классов» и прочего. Поскольку эти классы не являются частью языка Delphi Pascal, они не описываются в этой книге. Смотрите фирменную документацию по Delphi об информации по этим классам.

Счетчики ссылок

В предыдущем разделе мы обсуждали управление жизненным циклом интерфейсов при помощи счетчика ссылок. Строки и динамические массивы также используют для этого счетчики ссылок. Компилятор генерирует необходимый код, отслеживающий создание ссылок на интерфейсы, строки и динамические массивы и выход их из области действия, когда они должны быть уничтожены.

Обычно компилятор управляет счетчиками ссылок автоматически, и все работает именно так, как можно было бы ожидать. Иногда, однако, вы должны дать компилятору указание. Например, когда вы объявляете запись, которая содержит поле, имеющее счетчик ссылок, и используете процедуру GetMem для создания экземпляра записи, следует вызвать процедуру Initialize, передав ей данную запись в качестве аргумента. Перед вызовом FreeMem следует вызвать Finalize.

Возможно, вы захотите сохранить ссылку на строку или интерфейс после выхода за границы ее действия, т. е. вне блока, где была объявлена переменная. Например, связать интерфейс с каждым элементом TListView. Это можно сделать, явно управляя счетчиком ссылок. При сохранении интерфейса преобразуйте его в тип IUnknown, вызовите _AddRef и преобразуйте ссылку на IUnknown в простой тип указателя. При извлечении данных преобразуйте указатель в IUnknown. Таким образом, вы сможете использовать оператор as для преобразования интерфейса к любому типу, а также дать возможность Delphi освободить интерфейс. Для удобства создайте пару подпрограмм, выполняющих всю «грязную» работу, и пользуйтесь ими всегда, когда вам потребуется восстановить ссылку на интерфейс. В примере 2-15 приведен пример сохранения ссылки на интерфейс как информации, привязанной к элементу списка.

Пример 2-15. Сохранение интерфейсов вместе со списком

```
// Преобразование интерфейса в Pointer, чтобы увеличить
// счетчик ссылок и не освобождать интерфейс до вызова
// метода ReleaseIUnknown.
function RefIUnknown(const Intf: IUnknown): Pointer;
begin
  Intf._AddRef;               // Увеличение счетчика ссылок.
  Result := Pointer(Intf);    // Сохранение указателя на интерфейс.
end;

// Освобождение интерфейса, указатель на который хранится в P.
procedure ReleaseIUnknown(P: Pointer);
var
  Intf: IUnknown;
begin
  Pointer(Intf) := P;
  // Delphi освобождает интерфейс, когда Intf выходит из области действия
end;

// Когда пользователь нажимает на кнопку, интерфейс добавляется в список.
procedure TForm1.Button1Click(Sender: TObject);
var
  Item: TListItem;
begin
  Item := ListView1.Items.Add;
  Item.Caption := 'Stuff';
  Item.Data := RefIUnknown(GetIntf as IUnknown);
end;

// Когда по какой-либо причине уничтожается список или один из его элементов,
// интерфейс также освобождается.
procedure TForm1.ListView1Deletion(Sender: TObject; Item: TListItem);
begin
   ReleaseIUnknown(Item.Data);
end;

// Когда пользователь выбирает элемент из списка, выполняется некоторое
// действие со связанным с этим элементом интерфейсом.
procedure TForm1.ListView1Click(Sender: TObject);
var
  Intf: IMyInterface;
begin
  Intf := IUnknown(ListView1.Selected.Data) as IMyInterface;
  Intf.DoSomethingUseful;
end;
```

Вы можете также сохранять и строки. Вместо вызова `_AddRef` **преобразуйте строку к типу «указатель», чтобы сохранить ссылку на строку, затем заставьте переменную «забыть» об этой строке. Когда переменная выйдет из своей области действия, Delphi не освободит строку,**

поскольку переменная о ней «не помнит». После получения указателя присвойте его строковой переменной, преобразованной к типу «указатель». При возврате из подпрограммы Delphi автоматически освободит память этой строки. Убедитесь, что в программе не осталось указателей на освобождаемую память. Кроме того, служебные процедуры упростят вашу задачу. В примере 2-16 показан один из способов хранения строк.

Пример 2-16. Хранение строк вместе со списком

```
// Сохранение ссылки на строку и возвращение простого указателя
// на строку.
function RefString(const S: string): Pointer;
var
  Local: string;
begin
  Local := S;                // Увеличение счетчика ссылок.
  Result := Pointer(Local);  // Сохранение указателя на строку.
  Pointer(Local) := nil;     // Предотвращение уменьшения значения
     // счетчика.
end;

// Уничтожение строки, на которую ссылается RefString.
procedure ReleaseString(P: Pointer);
var
  Local: string;
begin
  Pointer(Local) := P;
  // Delphi уничтожает строку, когда заканчивается область действия Local.
end;

// При нажатии пользователем кнопки добавляем элемент в список
// и сохраняем дополнительную скрытую строку.
procedure TForm1.Button1Click(Sender: TObject);
var
  Item: TListItem;
begin
  Item := ListView1.Items.Add;
  Item.Caption := Edit1.Text;
  Item.Data := RefString(Edit2.Text);
end;

// Уничтожает строку при уничтожении элемента списка.
procedure TForm1.ListView1Deletion(Sender: TObject; Item: TListItem);
begin
  ReleaseString(Item.Data);
end;

// Извлечение строки при выборе пользователем элемента списка.
procedure TForm1.ListView1Click(Sender: TObject);
var
```

```
  Str: string;
begin
  if ListView1.Selected <> nil then
  begin
    Str := string(ListView1.Selected.Data);
    ShowMessage(Str);
  end;
end;
```

Сообщения

Вы, должно быть, знакомы с сообщениями Windows: действия пользователя и другие события генерируют сообщения, которые Windows посылает приложению. Приложение по очереди обрабатывает сообщения, реагируя на происходящие события. Каждый тип сообщения имеет уникальный номер и два целых параметра. Иногда параметр фактически является указателем на строку или структуру данных, содержащую более подробную информацию о событии. Сообщения формируют ядро архитектуры Windows, и в Delphi имеется уникальный способ работы с сообщениями Windows. В Delphi любой объект, а не только управляющие элементы, может реагировать на сообщения. Каждое сообщение имеет уникальный целый идентификатор и может содержать любое количество дополнительной информации. В VCL (Visual Component Library) сообщения Windows получает объект `Application`, который преобразует их в эквивалентные сообщения Delphi. Другими словами, сообщения Windows являются особым случаем более общих сообщений Delphi.

Сообщенит Delphi – это запись, первые два байта которой содержат целый идентификатор сообщения, а остаток записи определяется программистом. Диспетчер сообщений Delphi никогда не обращается к оставшейся части сообщения, идущей после номера, поэтому вы можете хранить в этой записи любое количество и тип информации. По соглашению, VCL всегда использует аналогичный Windows формат сообщения (`TMessage`), но если вы сможете использовать сообщения Delphi и для других целей, не ограничивайте себя.

Чтобы послать сообщение объекту, заполните поле идентификатора сообщения и оставшуюся часть записи, а затем вызовите метод `Dispatch` этого объекта. Delphi ищет номер сообщения в таблице сообщений объекта. Эта таблица содержит указатели на все обработчики событий, которые определены в классе. Если для этого сообщения обработчик отсутствует, Delphi продолжает поиск в таблице родительского класса. Поиск продолжается до тех пор, пока не будет найден обработчик, либо поиск не дойдет до класса `TObject`. Если в классе объекта и в его родительских классах не определены обработчики для данного типа сообщений, Delphi вызывает метод `DefaultHandler`. Управляющие элементы из VCL переопределяют метод `DefaultHandler` для передачи

сообщений процедуре окна; другие классы обычно игнорируют неизвестные им сообщения. Вы можете переопределить DefaultHandler для выполнения любых действий, которые вы хотите, например, для вызова исключительной ситуации.

Для объявления обработчика сообщений используется директива message. Подробнее о директиве message смотрите в главе 5.

Обработчики сообщений используют ту же таблицу сообщений и диспетчер, что и динамические сообщения. Каждому методу, который определен с директивой dynamic, присваивается 16-разрядное отрицательное число, которое фактически является номером сообщения. При вызове динамического метода используется тот же алгоритм поиска, что и для обработчиков событий, но если динамический метод не найден, это значит, что он является абстрактным, и Delphi вызывает процедуру AbstractErrorProc для выдачи сообщения о вызове абстрактного метода.

Поскольку динамические методы используют отрицательные индексы, вы не можете написать обработчик сообщения с отрицательным номером, т. е. для номеров сообщений, у которых старший бит равен 1. Это ограничение не должно создать проблем для обычных приложений. Если вам потребуется определить собственные сообщения, для этого имеется пространство от WM_USER ($0F00) до $7FF. Delphi ищет динамические методы и сообщения в одной таблице, используя линейный поиск, и потому ваше приложение будет тратить много времени на нахождение метода в больших таблицах.

Система сообщений Delphi абсолютно универсальна, и можно творчески подойти к ее применению. Обычно интерфейсы предоставляют вам те же возможности, но при лучшей производительности и повышенной безопасности типов.

Управление распределением памяти

Delphi автоматически управляет распределением памяти и жизненным циклом строк, переменных типа Variant, динамических массивов и интерфейсов. Для всех остальных динамических структур данных за это отвечает программист. Здесь легко запутаться, т. к. может показаться, что Delphi также автоматически распределяет память и для компонентов, но это лишь прием VCL.

Управление распределением памяти работает и в многопоточных приложениях в том случае, если вы используете для создания потоков классы или функции Delphi. Если вы будете непосредственно пользоваться средствами Windows API и функцией CreateThread, установите значение переменной IsMultiThread в True. Дополнительную информацию смотрите в главе 4 «Создание многопоточных приложений».

Компоненты и объекты

Класс TComponent из VCL имеет два интересных механизма для управления жизненным циклом объектов, и это может запутать новичка в Delphi, заставляя думать, что Delphi делает это автоматически. Важно понимать, как именно работают компоненты, чтобы не оказаться введенным в заблуждение.

У каждого компонента есть владелец. Когда владелец уничтожается, вместе с ним уничтожаются все имеющиеся у него компоненты. Форма является владельцем всех компонентов, находящихся в ней, и при освобождении она автоматически освобождает все свои компоненты. Поэтому вам обычно не надо беспокоиться об управлении жизненным циклом форм и компонентов.

Когда форма освобождает компонент, которым она владеет, выполняется проверка, нет ли публикуемых полей с тем же именем, что и этот компонент. Если таковые есть, значение поля устанавливается в nil. Поэтому, если форма динамически добавляет и удаляет компоненты, ее поля всегда либо содержат корректную ссылку на объект, либо равны nil. Не думайте, что Delphi делает это для любого другого поля или ссылки на объект. Это делается только для публикуемых полей (например, для тех, которые автоматически создаются при переносе компонента на форму в редакторе форм IDE), и только когда имя поля соответствует имени компонента.

Обычно при создании объекта Delphi вызывает для его распределения и инициализации метод NewInstance. Вы можете переопределить NewInstance для изменения алгоритма выделения памяти для объекта. Предположим, например, что ваша программа часто использует двусвязные списки. Вместо использования универсального менеджера памяти для каждого узла списка значительно эффективнее иметь цепочку доступных для использования узлов и применять менеджер памяти Delphi только тогда, когда список узлов будет исчерпан. Если ваша программа часто выделяет и освобождает узлы, этот специальный менеджер памяти может работать быстрее универсального. В примере 2-17 показана простая реализация этой схемы. (В главе 4 имеется версия этого класса для многопоточного приложения.)

Пример 2-17. Специальный менеджер памяти для связанных списков

```
type
  TNode = class
  private
    fNext, fPrevious: TNode;
  protected
    // Узлы управляются объектом типа TLinkedList.
```

```
    procedure Relink(NewNext, NewPrevious: TNode);
    constructor Create(Next: TNode = nil; Previous: TNode = nil);
    procedure RealFree;

public
    destructor Destroy; override;
    class function NewInstance: TObject; override;
    procedure FreeInstance; override;
    property Next: TNode read fNext;
    property Previous: TNode read fPrevious;
end;

// Односвязный список узлов, доступных для использования.
// Для работы со списком используются только поля Next.
var
  NodeList: TNode;

// Новый узел создается из начального узла NodeList.
// Не забудьте вызвать InitInstance для инициализации узла,
// взятого из Nodelist.
// Если NodeList пустой, создаем узел обычным образом.
class function TNode.NewInstance: TObject;
begin
  if NodeList = nil then
    Result := inherited NewInstance
  else
  begin
    Result := NodeList;
    NodeList := NodeList.Next;
    InitInstance(Result);
  end;
end;

// Так как NodeList использует только поле Next, установите поле
// Previous равным некоторому специальному значению. Если программа
// по ошибке обратится к полю Previous свободного узла, вы, проверив
// это специальное значение, найдете причину ошибки.
const
  BadPointerValueToFlagErrors = Pointer($F0EE0BAD);

// Освобождаемый узел добавляется в начало NodeList.
// Это ЗНАЧИТЕЛЬНО быстрее, чем использовать универсальный
// менеджер памяти.
procedure TNode.FreeInstance;
begin
  fPrevious := BadPointerValueToFlagErrors;
  fNext := NodeList;
  NodeList := Self;
end;

// Если вы хотите корректно очистить список по завершении работы
```

```
// программы, вызовите RealFree для каждого элемента списка.
// Унаследованный метод FreeInstance освобождает и очищает
// узел фактически.
procedure TNode.RealFree;
begin
  inherited FreeInstance;
end;
```

Вы также можете целиком заменить систему управления распределением памяти, используемую Delphi. Новый менеджер памяти устанавливается вызовом процедуры `SetMemoryManager`. Например, вы можете заменить упрощенную версию процедуры выделения памяти на полную версию, осуществляющую дополнительные проверки. В примере 2-18 показан специальный менеджер памяти, который поддерживает список выделенных программе указателей и явно проверяет все попытки их освобождения по этому списку. Любая попытка освободить неверный указатель отклоняется, и Delphi выдает ошибку времени выполнения (которую `SysUtils` заменяет на исключительную ситуацию). Заодно менеджер памяти проверяет, является ли этот список пустым по окончании работы приложения. Если список не пустой, происходит утечка памяти.

Пример 2-18. Установка специального менеджера памяти

```
unit CheckMemMgr;
interface

uses Windows;

function CheckGet(Size: Integer): Pointer;
function CheckFree(Mem: Pointer): Integer;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;

var
  HeapFlags: DWord;       // В однопоточном приложении можно установить
                        // эту переменную в Heap_No_Serialize.
implementation

const
  MaxSize = MaxInt div 4;
type
  TPointerArray = array[1..MaxSize] of Pointer;
  PPointerArray = ^TPointerArray;
var
  Heap: THandle;             // Идентификатор «кучи» Windows для
                             // списка указателей
  List: PPointerArray;       // Список распределенных указателей
  ListSize: Integer;         // Количество указателей в списке
  ListAlloc: Integer;        // Емкость списка указателей

// Если список распределенных указателей не пустой после завершения
```

```
// работы программы, это означает утечку памяти. Работа с утечкой
// памяти оставляется читателю в качестве упражнения.
procedure MemoryLeak;
begin
  // Выдача пользователю сообщения об утечке памяти. Так как программа
  // завершает свою работу, вы ограничены интерфейсом Windows API
  // и не можете использовать VCL.
end;

// Добавление указателя в список.
procedure AddMem(Mem: Pointer);
begin
  if List = nil then
  begin
    // Создание списка.
    ListAlloc := 8;
    List := HeapAlloc(Heap, HeapFlags, ListAlloc * SizeOf(Pointer));
  end
  else if ListSize >= ListAlloc then
  begin
    // Увеличение размера списка. Попытайтесь подойти к этому разумно.
    if ListAlloc < 256 then
      ListAlloc := ListAlloc * 2
    else
      ListAlloc := ListAlloc + 256;
    List := HeapRealloc(Heap, HeapFlags, List,
                        ListAlloc * SizeOf(Pointer));
  end;
  // Добавление указателя в список.
  Inc(ListSize);
  List[ListSize] := Mem;
end;

// Поиск указателя в списке и его удаление. Возвращает истину
// в случае успеха и ложь в случае отсутствия указателя в списке.
function RemoveMem(Mem: Pointer): Boolean;
var
  I: Integer;
begin
  for I := 1 to ListSize do
    if List[I] = Mem then
    begin
      MoveMemory(@List[I], @List[I+1], (ListSize-I) * SizeOf(Pointer));
      Dec(ListSize);
      Result := True;
      Exit;
    end;

  Result := False;
end;
```

```
// Замена процедуры выделения памяти.
function CheckGet(Size: Integer): Pointer;
begin
  Result := SysGetMem(Size);
  AddMem(Result);
end;

// Если указателя нет в списке, реальная функция Free не
// вызывается. Возвращает 0 в случае успеха и ненулевое значение
// в случае ошибки.
function CheckFree(Mem: Pointer): Integer;
begin
  if not RemoveMem(Mem) then
    Result := 1
  else
    Result := SysFreeMem(Mem);
end;

// Удаление старого указателя и добавление нового, который может
// либо быть тем же, либо отличаться. В случае ошибки возвращает nil,
// и Delphi вызывает исключительную ситуацию.
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;
begin
  if not RemoveMem(Mem) then
    Result := nil
  else
  begin
    Result :=SysReallocMem(Mem, Size);
    AddMem(Result);
  end;
end;

procedure SetNewManager;
var
  Mgr: TMemoryManager;
begin
  Mgr.GetMem := CheckGet;
  Mgr.FreeMem := CheckFree;
  Mgr.ReallocMem := CheckRealloc;
  SetMemoryManager(Mgr);
end;

initialization
  Heap := HeapCreate(0, HeapFlags, 0);
  SetNewManager;
finalization
  if ListSize <> 0 then
    MemoryLeak;
  HeapDestroy(Heap);
end.
```

Если вы определяете собственный менеджер памяти, вы должны быть уверены, что он используется для всех операций по распределению памяти. Простейший вариант состоит в установке менеджера памяти в разделе инициализации модуля, как показано в примере 2-18. Модуль управления памятью должен быть первым из перечисленных в разделе `uses` проекта.

Обычно, если модуль выполняет какие-то глобальные изменения в разделе инициализации, он должен вернуть изменения в исходное состояние в разделе завершения. Модуль, находящийся в пакете, может загружаться и выгружаться приложением несколько раз, и поэтому возвращение в исходное состояние имеет большое значение. Но менеджер памяти – это другое дело. Память, распределенная одним менеджером памяти, не может быть освобождена другим, и вы должны гарантировать, что в приложении работает только один менеджер, и он действует в течение всей работы приложения. Это значит, что менеджер памяти нельзя поместить в пакет, хотя, как описывается в следующем разделе, вы можете использовать DLL.

Память и DLL

Если вы используете DLL и хотите передавать объекты между двумя DLL или DLL и приложением, возникает несколько проблем. Во-первых, каждая DLL и исполняемый модуль имеют собственные копии таблиц классов. Операторы `is` и `as` не работают корректно для объектов, передаваемых между DLL и исполняемым модулем. Для решения этой проблемы используйте пакеты (смотрите главу 1). Другая проблема состоит в том, что любая память, выделенная DLL, принадлежит этой библиотеке. Когда Windows выгружает DLL, вся память, выделенная из этой библиотеки, освобождается, даже если в исполняемом модуле или другой DLL имеется указатель на эту память. Это может вызвать серьезные проблемы при использовании строк, динамических массивов и переменных типа `Variant`, так как вы никогда не знаете, когда Delphi автоматически выделит им память.

Решение состоит в использовании модуля `ShareMem` в качестве первого модуля проекта и всех DLL. Модуль устанавливает специальный менеджер памяти, который перенаправляет все запросы на выделение памяти в специальную DLL, *BorlndMM.dll*. Приложение не выгрузит *BorlndMM* до своего завершения. Работа с DLL происходит прозрачным образом, и вам не придется заботиться о деталях. Только не забудьте использовать модуль `ShareMem` и поместить его первым в списке `uses` всех ваших модулей и библиотек. Когда вы будете передавать ваше приложение клиентам или заказчикам, включите в поставку библиотеку *BorlndMM.dll*.

Если вы создаете собственный менеджер памяти и используете DLL, вам придется повторить трюки, выполняемые модулем `ShareMem`. Вы можете также заменить этот модуль на собственный, перенаправля-

ющий запросы к DLL, которая работает с вашим менеджером памяти. В примере 2-19 показан один из способов создания замены модулю ShareMem.

Пример 2-19. Создание менеджера общей памяти

```
unit CheckShareMem;

// Этот модуль должен быть первым, чтобы все запросы к памяти поступали
// к менеджеру общей памяти. Этот модуль должно использовать приложение и
// все DLL. Применение пакетов не допускается, так как эти DLL используют
// стандартный менеджер общей памяти Borland.

interface

function CheckGet(Size: Integer): Pointer;
function CheckFree(Mem: Pointer): Integer;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;

implementation

const
  DLL = 'CheckMM.dll';

function CheckGet(Size: Integer): Pointer; external DLL;
function CheckFree(Mem: Pointer): Integer; external DLL;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;
    external DLL;

procedure SetNewManager;
var
  Mgr: TMemoryManager;
begin
  Mgr.GetMem := CheckGet;
  Mgr.FreeMem := CheckFree;
  Mgr.ReallocMem := CheckRealloc;
  SetMemoryManager(Mgr);
end;

initialization
  SetNewManager;
end.
```

Библиотека CheckMM использует ваш менеджер памяти и экспортирует свои функции так, что с ними может работать модуль CheckShareMem. В примере 2-20 приведен исходный текст библиотеки CheckMM.

Пример 2-20. Создание DLL-библиотеки менеджера общей памяти

```
library CheckMM;

// Замена для BorlndMM.dll, использующая собственный менеджер памяти.
```

```
uses
  CheckMemMgr;

exports
  CheckGet, CheckFree, CheckRealloc;

begin
end.
```

Ваши программы и библиотеки начинаются модулем `CheckShareMem`, и все запросы к памяти направляются в библиотеку *CheckMM.dll*, которая использует менеджер памяти с проверкой ошибок. Вам вряд ли потребуется часто менять менеджер памяти Delphi, но, как видите, это не трудно сделать.

Менеджер памяти, имеющийся в Delphi, работает хорошо для большинства приложений, но он может не слишком подходить для вашей задачи. Среднее приложение выделяет и освобождает память участками переменного размера. Если к вашей программе это не относится, и она выделяет память во все увеличивающемся объеме (например, если у вас имеется динамический массив, который вырастает небольшими этапами до очень большого размера), производительность может упасть. Менеджер памяти Delphi будет запрашивать больше памяти, чем требуется приложению. Одно из решений состоит в переработке программы с тем, чтобы она по-другому использовала память (например, заранее выделяла большой динамический массив). Другое решение – написать собственный менеджер памяти, который будет соответствовать специфическим потребностям вашего приложения. Например, новый менеджер памяти может использовать Windows API (`HeapAllocate` и другие подобные процедуры).

Ключевое слово «object»

Кроме классов, Delphi также поддерживает устаревший тип, использующий ключевое слово `object`. Эти объекты сохранены для обратной совместимости с Turbo Pascal, но в последующих версиях Delphi они могут быть исключены.

«Старые» объекты больше похожи на записи, чем на новые объекты, принятые в Delphi Pascal. Поля старых объектов хранятся так же, как в записях. Если объект не имеет виртуальных методов, то он и не имеет скрытого поля для указателя на VMT. В отличие от записей, «старые» объекты также поддерживают наследование. Новые поля следуют за полями, унаследованными от родительских объектов. Если в объекте объявляется виртуальный метод, первым его полем становится указатель на VMT, который идет сразу за унаследованными объектами. (В отличие от современных классов, в которых указатель на

VMT всегда идет в начале, так как в TObject объявлены виртуальные методы.)

«Старые» объекты могут иметь секции private, protected и public, но не имеют секций published и automated. Так как в них нет секции published, «старые» типы объектов не могут иметь информации о типе времени выполнения. «Старые» объектные типы не могут реализовывать интерфейсы.

Конструкторы и деструкторы в «старых» объектных типах работают не так, как новые классы. Чтобы создать экземпляр «старого» объектного типа, вызывается процедура New. Поля только что созданного объекта инициализируются нулями. Если вы объявили конструктор, вы можете вызвать его как часть вызова процедуры New. Укажите имя конструктора и его аргументы в качестве второго аргумента процедуры New. Аналогично, вы можете использовать деструктор при вызове Dispose для освобождения экземпляра объекта. Имя деструктора и его аргументы указываются вторыми аргументами процедуры Dispose.

«Старые» объекты не обязательно создавать динамически. Вы можете рассматривать тип object как записи и объявлять объектные переменные как на уровне модуля, так и локально. Delphi автоматически инициализирует поля строкового типа, поля-динамические массивы и поля типа Variant, но не инициализирует другие поля экземпляра объектного типа.

В отличие от новых классов, исключения, возникающие в конструкторе «старого» объекта, не приводят к автоматическому освобождению динамически созданного объекта или вызову деструктора.

Глава 3

Информация о типе времени выполнения

Интегрированная среда разработки Delphi (IDE) зависит от информации, предоставляемой компилятором. Эта информация, называемая информацией о типе времени выполнения (Runtime Type Information, RTTI), описывает некоторые аспекты классов и других типов. Данная система не является полным отражением классов, как в Java, но она значительно полнее, чем идентификаторы типов в C++. В обычной, каждодневной работе можно игнорировать детали RTTI и предоставить Delphi делать свое дело. Однако иногда возникает необходимость заглянуть глубже и понять, как именно работает RTTI.

Единственное, чем отличаются объявления `published` и `public`, — это RTTI. Delphi формирует RTTI для публикуемых (`published`) полей, методов и свойств, но не формирует для секций `public`, `protected` и `private`. И хотя RTTI главным образом предназначена для передачи информации IDE, а также сохранения и загрузки *dfm*-файлов, таблицы RTTI содержат и другую информацию. Например, частью RTTI для класса также являются виртуальные и динамические методы, интерфейсы и секции `automated`. Большинство типов также имеют RTTI, называемую *информацией о типе*. В этой главе RTTI описана во всех подробностях.

Таблица виртуальных методов

В таблице виртуальных методов (VMT) содержатся указатели на все виртуальные методы, объявленные данным классом и его родительскими классами. Формат VMT аналогичен формату большинства реализаций C++ (в том числе Borland C++ и C++ Builder) и формату,

принятому в COM, и представляет собой список указателей на методы. Каждый виртуальный метод класса или его предков имеет запись в VMT.

Каждый класс имеет свою уникальную VMT. Даже если в классе не определен ни один собственный виртуальный метод, а лишь наследуются методы базового класса, у него есть собственная VMT, в которой перечислены все виртуальные методы, наследуемые этим классом. Поскольку в каждой VMT содержатся все виртуальные методы, Delphi может компилировать вызовы виртуальных методов в код быстрого поиска по VMT. Так как каждый класс имеет собственную VMT, Delphi использует VMT для идентификации класса. Фактически ссылка на класс – это указатель на его VMT, и метод `ClassType` возвращает указатель на VMT.

Помимо таблицы виртуальных методов VMT содержит и другую информацию о классе, такую как имя класса, указатель на VMT базового класса и указатели на другие таблицы RTTI. Другие указатели на RTTI расположены в VMT перед первым виртуальным методом. В примере 3-1 приведен формат записи, эквивалентной структуре VMT. Фактический список виртуальных методов начинается после записи `TVmt`. Другими словами, ссылку на класс `TClass` можно преобразовать в указатель на запись `TVmt` путем вычитания размера записи, как показано в примере 3-1.

Пример 3-1. Структура VMT

```
type
  PVmt = ^TVmt;
  TVmt = record
    SelfPtr:            TClass;          // Указатель на начало VMT
    // Указатели на другие таблицы RTTI. Если класс не содержит
    // данной таблицы, указатель равен nil. Поэтому, например,
    // большинство классов имеют IntfTable и AutoTable, равные nil.
    IntfTable:          PInterfaceTable; // Таблица интерфейсов
    AutoTable:          PAutoTable;      // Таблица автоматических методов
    InitTable:          PInitTable;      // Поля, требующие завершения
                                         // (finalization)
    TypeInfo:           PTypeInfo;       // Свойства и другая информация
    FieldTable:         PFieldTable;     // Published поля
    MethodTable:        PMethodTable;    // Published методы
    DynMethodTable:     PDynMethodTable; // Список динамических методов

    ClassName:          PShortString;  // Указатель на имя класса
    InstanceSize:       LongInt;       // Размер экземпляра объекта, в байтах
    ClassParent:        ^TClass;       // Базовый класс

    // Следующие поля указывают на специальные виртуальные методы,
    // унаследованные от TObject.
    SafeCallException:  Pointer;
    AfterConstruction:  Pointer;
```

```
    BeforeDestruction:  Pointer;
    Dispatch:           Pointer;
    DefaultHandler:     Pointer;
    NewInstance:        Pointer;
    FreeInstance:       Pointer;
    Destroy:            Pointer;

    // Здесь начинаются указатели на виртуальные методы.
    // Каждый виртуальный метод хранится как указатель на свой код,
    // т.е. VirtualMethodTable: array [1..Count] of Pointer;
    // Но компилятор не сохраняет количество указателей в таблице.
  end;
var
  Vmt: PVmt;
begin
  // Чтобы получить указатель PVmt из ссылки на класс, преобразуйте
  // ссылку на класс в тип PVmt и вычтите из него размер записи TVmt.
  // Это легко сделать процедурой Dec.
  Vmt := PVmt(SomeObject.ClassType);
  Dec(Vmt);
```

Как видите, в VMT имеются указатели на многие другие таблицы. Эти таблицы подробно описаны в последующих разделах.

Секция published

Единственная разница между секциями `published` и `public` заключается в том, что для секции `published` формируются записи в таблице виртуальных методов (VMT). В VMT может храниться информация только определенного вида, поэтому на объявления в секции `published` накладывается ряд ограничений:

- Для того чтобы объявить поля, методы или свойства в секции `published`, для класса должна быть включена поддержка RTTI с помощью директивы `$M+` или путем наследования от класса, поддерживающего RTTI. (Смотрите подробнее в главе 8 «Директивы компилятора».)
- Поля должны иметь объектный тип (другие типы не разрешаются)[1]. Для этого типа необходимо включить поддержку RTTI.
- Свойства-массивы не могут быть опубликованными. Свойство с типом `published` не может быть указателем, записью или массивом. Если это множество, то его размер не должен быть больше размера `integer`. В текущей версии Delphi это означает, что множество не может иметь более 32 элементов.

[1] Опубликованное (`published`) поле может быть также ссылкой на интерфейс. Обратите внимание, что речь идет именно о полях; к свойствам это ограничение не относится. – *Примеч. науч. ред.*

- Секция published не может содержать более одного перегруженного метода с одним именем. Если объявляются перегруженные методы, только один из них может быть published.

В модуле Classes объявлен класс TPersistent с директивой $M+. Обычно этот класс используется в качестве базового для всех классов Delphi, которым требуются объявления published. Заметьте, что TComponent является наследником TPersistent.

Публикуемые методы

Delphi хранит имена и адреса публикуемых (published) методов в RTTI данного класса. IDE использует эту информацию для сохранения значений свойств-событий в *dfm*-файле. В IDE каждое свойство-событие либо принимает значение nil, либо содержит ссылку на метод. Ссылка на метод – это указатель на его точку входа. (На этапе разработки приложения IDE не имеет верного значения точки входа, и поэтому подставляет некоторое произвольное значение. На этапе выполнения приложение использует реальное значение точки входа метода.) Для того чтобы сохранить значение свойства-события, приложение Delphi ищет адрес метода в RTTI класса, находит соответствующее имя метода, которое и записывает в *dfm*-файл. При загрузке *dfm*-файла Delphi считывает имя метода и ищет адрес соответствующего метода в RTTI класса.

В RTTI класса содержатся только публикуемые методы данного класса, а информация о родительских классах там не хранится. Поэтому если поиск имени или адреса метода для производного класса закончился неудачей, следует продолжить поиск в базовом классе. Методы MethodName и MethodAddress класса TObject занимаются поиском сначала в RTTI текущего класса, затем в RTTI базового класса и далее по цепочке наследования. (Смотрите подробнее об этих методах в описании типа TObject в главе 5 «Справочник по языку».) Таблица публикуемых методов содержит только имя и адрес метода.

В секции published описания класса может быть объявлен любой метод. Однако обычно в этом разделе располагаются методы, которые создаются IDE Delphi автоматически. Например, при двойном щелчке по свойству-событию IDE создает метод в начальной неименованной секции класса формы. Поскольку класс формы поддерживает RTTI, эта секция является published. (Класс формы поддерживает RTTI, т. к. он является наследником TPersistent.)

Таблица методов начинается с двухбайтового значения, определяющего количество публикуемых методов, затем следуют записи для каждого метода. Запись метода начинается с двухбайтового значения размера записи, за которым следует адрес метода (4 байта) и имя метода в виде short string, состоящее из 1 байта длины и собственно имени.

Дополнительная RTTI-информация о методах

Размер записи метода обычно равен 2 + 4 + 1 + Length(Имя), но для некоторых методов имеется и дополнительная информация. Она не является официальной для RTTI класса, и будущие версии компилятора могут не генерировать эту информацию, так что не следует писать код с ее использованием. Однако это интересная информация, поэтому давайте посмотрим, что Delphi прячет в записях методов.

Delphi формирует дополнительную информацию для методов, использующих соглашение `stdcall` и имеющих параметры и возвращаемые значения, для которых Delphi обычно хранит информацию о типах. Дополнительная информация хранится после имени метода и включает имя и тип параметров. Об ее присутствии можно судить по размеру записи метода; в этом случае он должен быть больше, чем требуется для хранения размера записи, адреса и имени метода. Если этот размер на 4 байта больше, чем требуется, значит у метода нет параметров, поскольку эти 4 байта всегда содержат нули.

Если же размер записи превышает стандартный более чем на 6 байт, то за шестым байтом содержится информация о параметрах метода. (Два дополнительных байта вряд ли могут принести пользу, но они не всегда нулевые.) Каждый параметр имеет указатель на запись `TTypeInfo`, определяющую тип параметра, за которым следуют имя параметра (в виде `short string`) и завершающий байт `#0`. (Смотрите материал о записях типа `TTypeInfo` далее в этой главе в разделе «Модуль TypInfo».

В примере 3-2 описывается логическая структура таблицы методов. Заметьте, что Delphi не может использовать эти описания буквально, поскольку размер записи изменяется в зависимости от длины строк.

Пример 3-2. Структура таблицы публикуемых методов

```
type
  TMethodParam = packed record
    TypeInfo: PPTypeInfo;
    Name: ShortString;
    // Имя метода, за которым следует байт #0.
  end;
  TMethod = packed record
    Size: Word;                  // Размер записи TVmtMethod.
    Address: Pointer;            // Указатель на точку входа метода.
    Name: packed ShortString;    // Имя публикуемого метода.
    // Некоторые методы могут содержать дополнительные 4 нулевые байта;
    // это означает, что метод не требует параметров.
```

```
    // Некоторые методы имеют дополнительные 6 байт, на которыми следуют
    // записи TMethodParam для каждого параметра.
    // Судя по всему, только методы stdcall имеют этот дополнительный
    // параметр. Дополнительная информация идентифицируется по значению
    // TMethod.Size, которое в этом случае превышает размеры полей Size,
    // Address и Name. Единственный способ определить количество
    // параметров - это проверять каждый параметр до тех пор, пока
    // вы не выйдите за указанный размер записи.
    ExtraStuff: array[1..FourOrSix] of Byte;
    Params: array[1..ParamCount] of TMethodParam;
  end;
  { Таблица published методов }
  TMethodTable = packed record
    Count: Word;
    Methods: array[1..Count] of TMethod;
  end;
```

Публикуемые поля и типы полей

У каждого публикуемого (`published`) поля имеется имя, тип и смещение. Тип – это ссылка на класс данного поля. (Публикуемые поля должны иметь объектный тип.) Смещение – это величина в байтах, определяющая смещение, по которому в объекте располагается данное поле.

Таблица публикуемых полей начинается с двухбайтового значения, содержащего количество полей, за которым следует четырехбайтовый указатель на таблицу класса, а затем – описание полей. Каждое описание поля – это запись, содержащая четырехбайтовое смещение, двухбайтовый индекс для таблицы классов и имя поля в виде `short string`.

В таблице классов перечислены все классы, используемые публикуемыми полями. Каждое поле содержит индекс для этой таблицы. Таблица класса начинается с двухбайтового счетчика, за которым следует список ссылок на классы, занимающих по 4 байта каждая. Ссылка на класс – это указатель на VMT класса.

В примере 3-3 показана логическая структура таблицы полей. Поскольку записи имеют переменную длину, эти описания нельзя использовать в программе.

Пример 3-3. Структура таблицы публикуемых полей

```
type
  { Таблица классов полей }
  PFieldClassTable = ^TFieldClassTable;
  TFieldClassTable = packed record
    Count: Word;
    Classes: packed array[1..Count] of ^TClass;
  end;

  { Запись для публикуемого поля }
```

```
TField = packed record
  Offset: LongWord;         // Смещение в байтах для поля объекта.
  ClassIndex: Word;        // Индекс FieldClassTable, определяющий тип поля.
  Name: packed ShortString;  // Имя публикуемого поля.
end;

{ Таблица публикуемых полей }
TFieldTable = packed record
  Count: Word;
  FieldClassTable: PFieldClassTable;
  Fields: packed array [1..Count] of TField;
end;
```

Публикуемые свойства

Для публикуемых (published) свойств требуется хранить очень много информации: имя, тип, считыватель (reader), установщик (writer), значение по умолчанию, индекс и флаг хранения (stored flag)[1]. Тип – это указатель на запись TTypeInfo (обсуждается в следующем разделе). Считыватель и установщик могут быть полями, методами или отсутствовать вовсе. Значение по умолчанию является порядковым значением; непорядковые свойства не могут иметь значения по умолчанию. Флаг stored может быть константой, полем или ссылкой на метод. Инспектор объектов IDE Delphi использует публикуемые свойства, а компилятор Delphi – информацию по умолчанию и флаг stored при записи и чтении *dfm*-файлов.

Считыватель, установщик и флаг stored могут быть указателями на статические методы, байтовыми смещениями виртуальных методов или байтовыми смещениями полей. Динамические методы не разрешены, статические и виртуальные методы должны использовать передачу параметров через регистры (это принято по умолчанию). Кроме того, флаг stored может быть константой True или False. Delphi хранит эти разнообразные виды значений следующим образом:

- Константы True и False хранятся в виде символа 0 или 1. Только директива stored хранится в виде констант True или False. Равенство нулю поля считывателя или установщика означает, что свойство не имеет данных директив, т. е. оно предназначено только для чтения или только для записи, соответственно.
- В самом старшем байте смещения поля содержится $FF. Например, поле, хранящееся по смещению 42 ($2A), имеет значение смещения $FF00002A. Заметим, что публикуемые поля редко используются для хранения значений свойств, и поэтому вряд ли вам придется искать имя поля в таблице публикуемых полей.

[1] Смотрите описание директивы stored в главе 5. – *Примеч. науч. ред.*

Публикуемые поля и компоненты

При переносе компонента на форму в IDE Delphi для него создается описание публикуемого поля. Delphi пользуется преимуществами публикуемых полей класса формы для записи и чтения *dfm*-файлов, но механизмы Delphi являются общими для всех компонентов, поскольку они реализованы как методы класса `TComponent`. Такие приемы не являются частью языка Delphi, но они существенны для большинства Delphi-программ.

Если компонент (назовем его `Owner`) становится владельцем другого компонента (назовем его `Child`), то дочерний компонент ищет свое имя (т. е. значение свойства `Name`) у владельца. Если `Owner` имеет публикуемое поле с тем же именем, что и `Child`, то владелец устанавливает значение этого поля равным ссылке на дочерний объект. При уничтожении `Child` владелец выполняет повторную проверку на наличие публикуемого поля с тем же именем и, если оно есть, устанавливает значение поля в `nil`.

Обычно Delphi-программист сталкивается с этой технологией только при работе с публикуемыми полями формы. При загрузке *dfm*-файла Delphi создает дочерние компоненты формы. Класс формы (которая является владельцем) отмечает, что имя компонента соответствует имени публикуемого поля, и автоматически устанавливает значение поля в ссылку на вновь созданный компонент.

Другими словами, язык Delphi не рассматривает компоненты или формы специальным образом. Вместо этого класс `TComponent` использует известную ему информацию `o` публикуемых полях для работы со своими дочерними компонентами. Не впадайте в заблуждение, что Delphi автоматически управляет жизненным циклом всех компонент только потому, что управляет некоторыми из них.

Таблица публикуемых полей особенно важна для загрузки *dfm*-файлов. Загружая компонент из *dfm*-файла, Delphi читает имя типа компонента в виде строки. Необходимую ссылку на класс для этого имени Delphi может найти в таблице полей класса. Если разрабатываемый компонент содержит дочерний компонент, не объявленный как публикуемое поле, следует зарегистрировать этот класс явно, вызвав процедуру `RegisterClass` или `RegisterClasses` (обе процедуры содержатся в модуле `Classes`). Регистрация классов не является средством языка Delphi.

- Адрес виртуального метода содержит $FE в самом старшем байте, а байтовое смещение метода хранится в виде `SmallInt` в двух младших байтах. Например, третий виртуальный метод хранится как

$FE00008. (Первый виртуальный метод имеет смещение, равное нулю.)

- Статический метод хранится в виде адреса, например $00401E42. Архитектура памяти Windows не позволяет методу иметь адрес со старшим байтом, равным $FF или $FE, поэтому нет опасности конфликтов или двусмысленностей.

Значение по умолчанию может быть сохранено только для целых, символьных, перечисляемых или множественных типов. При объявлении программистом свойства с директивой `nodefault` (что соответствует отсутствию директивы `default`) Delphi сохраняет в качестве значения по умолчанию наименьшее целое значение ($80000000 или −2 147 483 648). Другими словами, нельзя иметь целое свойство со значением по умолчанию, равным −2 147 483 648, т. к. Delphi будет интерпретировать это свойство как `nodefault`.

Строковые и вещественные свойства, свойства типа `Int64` и `Variant`, а также объектные свойства не могут иметь значения по умолчанию, и директива `nodefault` для них не работает. (При чтении и записи *dfm*-файлов Delphi всегда использует для этих типов в качестве значения по умолчанию пустую строку, ноль, `Unassigned` или `nil`.) Если требуется получить для этих типов эффект значения по умолчанию, можно воспользоваться конструктором класса: устанавливать значение свойства только тогда, когда пользователь помещает компонент на форму, но не тогда, когда Delphi загружает компонент из *dfm*-файла. Прием заключается в том, что до окончания работы конструктора свойство `ComponentSate` не установлено. (Смотрите описание `ComponentState` в файлах помощи Delphi.) Поэтому приходится проверять `ComponentState` владельца компонента следующим образом:

```
constructor TStringDefault.Create(Owner:
   TComponent);
begin
  inherited;
  if (Owner = nil) or
     (([csReading, csDesigning] *
       Owner.ComponentState) = [csDesigning])
  then
    StringProperty := 'Значение по умполчанию';
end;
```

Этот прием не экономит место в *dfm*-файле, но дает возможность установить значение по умолчанию для свойства, которое не может его иметь обычным образом.

Основная задача значений по умолчанию – это экономия места в *dfm*-файле. Если значение свойства совпадает со значением по умолчанию, Delphi не записывает его в *dfm*-файл. Если свойство не имеет значения по умолчанию, Delphi всегда запоминает его значение в *dfm*-файле. Заметим, что производные формы получают значения по умолчанию из

родительской формы, в которой они установлены директивой default. За инициализацию свойства в конструкторе класса отвечает программист – Delphi не делает это автоматически. (Смотрите далее в этой главе пример 3-5, в котором объясняется установка значения по умолчанию.)

Директива index устанавливает значение индекса для данного свойства. Если свойство не индексировано, Delphi сохраняет в качестве значения индекса наименьшее отрицательное целое число.

В информации по публикуемым свойствам также содержится именованный индекс, т. е. порядковый номер свойства в описании класса. Инспектор объектов может сортировать свойства в алфавитном порядке, не соответствующем порядку объявления свойств класса. Значение этого индекса соответствует порядковому номеру свойства.

Delphi обеспечивает легкий доступ к информации о публикуемых свойствах класса с помощью модуля TypInfo, который является предметом обсуждения следующего раздела.

Модуль TypInfo

В модуле TypInfo описано несколько типов и функций, обеспечивающих легкий доступ к публикуемым свойствам объекта и другой информации. Инспектор объектов использует эти данные для выполнения своих функций. Можно получить список публикуемых свойств класса и узнать имя и тип каждого из этих свойств. При наличии ссылки на объект можно получить или установить значение любого публикуемого свойства.

Функция TypeInfo возвращает указатель на запись, содержащую информацию о типе, но без помощи модуля TypInfo извлечь что-нибудь из этой записи невозможно, и приходится рассматривать результат в виде нетипизированного указателя (untyped Pointer). В модуле TypInfo описан фактический тип PTypeInfo, который является указателем на запись TTypeInfo. В этой записи содержится категория типа и его имя. Категория типа – это перечисляемое значение, указывающее на то, к какой категории относится этот тип: целое число, вещественное число, строка и т. д.

Дополнительная информация о типе

Некоторые типы содержат дополнительную информацию, возвращаемую функцией GetTypeData в виде указателя PTypeData. С помощью этой дополнительной информации о типе можно получить имена значений перечисляемого типа, пределы поддиапазона и т. д. В табл. 3-1 приведены данные для каждой категории типов.

Таблица 3-1. Категории типов и соответствующие им данные

Значение TTypeKind	**Соответствующие данные**
tkArray	Данные отсутствуют
tkChar	Пределы символьного поддиапазона
tkClass	Ссылка на класс, базовый класс, модуль, в котором класс объявлен, и публикуемые свойства класса
tkDynArray	Данные для динамических массивов отсутствуют
tkEnumeration	Если тип является поддиапазоном другого типа, данные содержат указатель на базовый тип и пределы поддиапазона; в противном случае данные содержат пределы поддиапазона и упакованный список строк с именами перечисляемых значений
tkFloat	Тип вещественного числа: currency, comp, single, double или extended (но не Real48)
tkInt64	Пределы целого поддиапазона
tkInteger	Пределы целого поддиапазона
tkInterface	Базовый интерфейс, модуль, в котором объявлен интерфейс, и GUID
tkLString	Данные для длинных строк отсутствуют (AnsiString)
tkMethod	Тип возвращаемого значения, тип метода, имена параметров и имена типов параметров
tkRecord	Данные отсутствуют
tkSet	Указатель на перечисляемый тип элементов множества
tkString	Максимальная длина короткой строки
tkUnknown	Данные отсутствуют
tkVariant	Данные отсутствуют
tkWChar	Пределы поддиапазона Unicode-символа
tkWString	Данные для WideString отсутствуют

Заметьте, что основным назначением информации о типах является поддержка IDE Delphi и чтение/запись *dfm*-файлов. Кроме того, эти данные используются для инициализации и завершения работы с этими типами. Эта система не является такой универсальной, как в Java, поэтому информация довольно ограничена, например, информация о записях и массивах.

Публикуемые свойства

Многие функции модуля TypInfo облегчают доступ к публикуемым свойствам объекта. Вместо прямого обращения к информации о типе можно вызвать некоторые функции, чтобы получить или установить значения свойства, определить, является ли свойство stored, получить имя и тип свойства и т. д.

Для того чтобы получить или установить значение свойства, нужно знать категорию типа свойства, с которым вы имеете дело: порядковое, вещественное, строковое, Variant, метод или Int64. Каждая категория типа имеет пару подпрограмм для получения и установки значений свойства. Если для свойства определены методы чтения и записи, подпрограммы модуля TypInfo вызывают эти методы, как если бы вы работали со свойством обычным образом.

Целые, символьные, перечисляемые, множественные и объектные свойства являются порядковыми. В их значениях хранится целое, и поэтому вы должны использовать целые числа для получения или установки значения свойства. Значение получаемого свойства следует привести к нужному типу, а для установки значения свойства необходимо преобразовать значение соответствующего типа к целому. В примере 3-4 приведена процедура, которая принимает в качестве аргумента любой компонент и проверяет его. Если этот компонент имеет публикуемое свойство по имени Font объектного типа, процедура устанавливает для компонента шрифт Arial, 10 пунктов.

Пример 3-4. Установка значения свойства порядкового типа

```
procedure SetFontToArial10pt(Component: TComponent);
var
  Font: TFont;
  PropInfo: PPropInfo;
begin
  // Определение наличия у компонента свойства Font
  PropInfo := GetPropInfo(Component, 'Font');
  if PropInfo = nil then
    Exit;
  // Далее проверяем, является ли тип свойства классом
  if PropInfo.PropType^.Kind <> tkClass then
    Exit;
  Font := TFont.Create;
  try
    Font.Name := 'Arial';
    Font.Size := 10;
    // Установка значения свойства Font.
    SetOrdProp(Component, PropInfo, Integer(Font));
    // SetOrdProp - это то же самое, что и Component.Font = Font,
    // за исключением того, что компилятору не требуется знать о
    // существовании свойства Font.
    // Метод-установщик копирует объект TFont, поэтому данная процедура
```

```
    // должна освободить объект Font, чтобы исключить утечку памяти.
  finally
    Font.Free;
  end;
end;
```

Можно получить список указателей PPropInfo, если требуется узнать обо всех свойствах объекта, или можно вызвать GetPropInfo для получения информации об одном свойстве. Функция GetPropList возвращает только те свойства, категория типа которых соответствует множеству указанных вами категорий. С помощью этой функции можно получить информацию о событиях (tkMethod), информацию только о строковых свойствах (tkString, tkLString, tkWideString) и т. д. В примере 3-5 приведена процедура, которая получает в качестве аргумента объект и ищет все свойства порядковых типов, затем извлекает их значения по умолчанию и устанавливает свойства равными этим значениям. Эту функцию можно вызвать из конструктора, чтобы обеспечить корректную инициализацию свойств и избежать возможной ошибки в том случае, если в описании свойство имеет одно значение по умолчанию, а в конструкторе – другое.

Пример 3-5. Установка значений свойств по умолчанию

```
// Установка значений по умолчанию для всех публикуемых свойств.
procedure SetDefaultValues(Obj: TObject);
const
  tkOrdinal = [tkEnumeration, tkInteger, tkChar, tkSet, tkWChar];
  NoDefault = Low(Integer);
var
  PropList: PPropList;
  Count, I: Integer;
  Value: Integer;
begin
  // Подсчитывает количество свойств порядковых типов, которые
  // могут иметь значения по умолчанию.
  Count := GetPropList(Obj, tkOrdinal, nil);
  // Выделение памяти для хранения PropInfo и получение фактического
  // списка свойств.
  GetMem(PropList, Count * SizeOf(PPropInfo));
  try
    GetPropList(Obj, tkOrdinal, PropList);
    // Цикл по всем свойствам порядковых типов.
    for I := 0 to Count-1 do
      // Если свойство имеет значение по умолчанию, используем его для
      // инициализации.
      if PropList[I].Default <> NoDefault then
        SetOrdProp(Obj, PropList[I], PropList[I].Default)
  finally
    FreeMem(PropList);
  end;
end;
```

Подпрограммы модуля TypInfo не документированы, однако очень понятны и легки в использовании. Ниже приведен список, в котором представлены все подпрограммы модуля TypInfo. Подробности можно найти в исходном файле *TypInfo.pas* (для этого требуется как минимум профессиональная версия Delphi или C++ Builder). Обратите внимание, что эти функции не выполняют или почти не выполняют проверку на ошибки. Ответственность за вызов правильной для данного свойства и его типа функции лежит на разработчике. Например, изменение значения свойства, доступного только для чтения, или извлечение значения свойства, доступного только для записи, приводит к ошибке доступа.

GetEnumName, функция

```
function GetEnumName(TypeInfo: PTypeInfo; Value: Integer): string;
```

Возвращает имя перечисляемого литерала или пустую строку, если Value выходит за пределы диапазона.

GetEnumProp, функция

```
function GetEnumProp(Instance: TObject; PropInfo: PPropInfo):
    string; overload;
function GetEnumProp(Instance: TObject; const PropName: string):
    string; overload;
```

Возвращает имя перечисляемого литерала, представляющего текущее значение свойства.

GetEnumValue, функция

```
function GetEnumValue(TypeInfo: PTypeInfo; const Name: string):Integer;
```

Возвращает порядковое значение перечисляемого литерала или −1, если в типе отсутствует литерал с данным именем.

GetFloatProp, функция

```
function GetFloatProp(Instance: TObject; PropInfo: PPropInfo):
    Extended; overload;
function GetFloatProp(Instance: TObject; const PropName: string):
    Extended; overload;
```

Извлекает значение свойства вещественного типа.

GetInt64Prop, функция

```
function GetInt64Prop(Instance: TObject; PropInfo: PPropInfo):
    Int64; overload;
function GetInt64Prop(Instance: TObject; const PropName: string):
    Int64; overload;
```

Извлекает значение свойства типа Int64 или любого поддиапазона, требующего более 32 бит.

GetMethodProp, функция

```
function GetMethodProp(Instance: TObject; PropInfo: PPropInfo):
    TMethod; overload;
function GetMethodProp(Instance: TObject; const PropName: string):
    TMethod; overload;
```

Извлекает значение свойства-события.

GetObjectProp, функция

```
function GetObjectProp(Instance: TObject; PropInfo: PPropInfo;
    MinClass: TClass = nil): TObject; overload;
function GetObjectProp(Instance: TObject; const PropName: string;
    MinClass: TClass = nil): TObject; overload;
```

Извлекает значение свойства объектного типа. `MinClass` **— это базовый класс, который вам требуется для получения значения свойства; если результат не относится к типу** `MinClass` **или его потомкам,** `GetObjectProp` **возвращает** `nil`**. По умолчанию объект может относиться к любому классу.**

GetObjectPropClass, функция

```
function GetObjectPropClass(Instance: TObject; PropInfo: PPropInfo):
     TClass; overload;
function GetObjectPropClass(Instance: TObject; const PropName: string):
     TClass; overload;
```

Извлекает тип класса из информации о типе свойства. `Instance` **служит только для получения информации о свойстве, поэтому первая версия данной функции (которая уже получила эту информацию в параметре** `PropInfo`**) не использует** `Instance`**.**

GetOrdProp, функция

```
function GetOrdProp(Instance: TObject; PropInfo: PPropInfo):
    Longint; overload;
function GetOrdProp(Instance: TObject; const PropName: string):
    Longint; overload;
```

Извлекает значение любого свойства порядкового типа или любого свойства, значение которого помещается в 32-разрядное целое, например, объекта, множества, символа, перечисления или поддиапазона целого.

GetPropInfo, функция

```
function GetPropInfo(TypeInfo: PTypeInfo; const PropName: string):
    PPropInfo; overload;
function GetPropInfo(TypeInfo: PTypeInfo; const PropName: string;
    AKinds: TTypeKinds): PPropInfo; overload;
function GetPropInfo(Instance: TObject; const PropName: string;
    AKinds: TTypeKinds = []): PPropInfo; overload;
function GetPropInfo(AClass: TClass; const PropName: string;
    AKinds: TTypeKinds = []): PPropInfo; overload;
```

Возвращает указатель PPropInfo для публикуемого свойства или nil, если класс не имеет такого публикуемого свойства или тип указанного свойства не соответствует требуемому типу. Первый аргумент может быть ссылкой на объект, ссылкой на класс или информацией о типе класса (из функции TypeInfo или метода ClassInfo).

GetPropInfos, процедура

```
procedure GetPropInfos(TypeInfo: PTypeInfo; PropList: PPropList);
```

Извлекает список всех указателей PPropInfo для объекта в порядке их объявления. Используйте данные о типе класса для определения количества публикуемых свойств в нем и создания массива PropList соответствующего размера.

GetPropList, функция

```
function GetPropList(TypeInfo: PTypeInfo; TypeKinds: TTypeKinds;
    PropList: PPropList): Integer;
```

Извлекает соответствующие свойства объекта в алфавитном порядке в виде указателей PPropInfo и возвращает количество записанных в PPropList свойств. Для получения только количества соответствующих свойств передайте nil в параметре PropList.

GetPropValue, функция

```
function GetPropValue(Instance: TObject; const PropName: string;
    PreferStrings: Boolean = True): Variant;
```

Извлекает значение публикуемого свойства, имеющее тип Variant. GetPropValue приводит к большему количеству накладных расходов по сравнению с другими Get... функциями, но проще в использовании. Если PreferStrings – истина, GetPropValue возвращает значение свойства в виде строки, если это возможно.

GetSetProp, функция

```
function GetSetProp(Instance: TObject; PropInfo: PPropInfo;
    Brackets: Boolean = False): string; overload;
function GetSetProp(Instance: TObject; const PropName: string;
    Brackets: Boolean = False): string; overload;
```

Извлекает значение свойства множественного типа и возвращает его в виде строки. Формат строки – это список перечисляемых литералов, разделяемых запятыми и пробелами. Дополнительно можно использовать квадратные скобки. Формат аналогичен применяемому в инспекторе объектов.

GetStrProp, функция

```
function GetStrProp(Instance: TObject; PropInfo: PPropInfo): string;
    overload;
function GetStrProp(Instance: TObject; const PropName: string): string;
    overload;
```

Извлекает значение свойства строкового типа. Тип свойства может быть `tkString`, `tkLString` или `tkWString`. Во всех случаях значение автоматически преобразуется в строку.

GetTypeData, функция

```
function GetTypeData(TypeInfo: PTypeInfo): PTypeData;
```

Возвращает указатель на запись `TTypeData` для типа по указателю `PTypeInfo`.

GetVariantProp, функция

```
function GetVariantProp(Instance: TObject; PropInfo: PPropInfo):
    Variant; overload;
function GetVariantProp(Instance: TObject; const PropName: string):
    Variant; overload;
```

Извлекает значение свойства типа `Variant`.

IsPublishedProp, функция

```
function IsPublishedProp(Instance: TObject; const PropName: string):
    Boolean; overload;
function IsPublishedProp(AClass: TClass; const PropName: string):
    Boolean; overload;
```

Возвращает True, если класс имеет публикуемое свойство с данным именем.

IsStoredProp, функция

```
function IsStoredProp(Instance: TObject; PropInfo: PPropInfo):
    Boolean; overload;
function IsStoredProp(Instance: TObject; const PropName: string):
    Boolean; overload;
```

Возвращает значение директивы `stored`. Если в директиве `stored` указан метод, `IsStoredProp` вызывает этот метод; если указано поле – возвращается значение поля.

PropIsType, функция

```
function PropIsType(Instance: TObject; const PropName: string;
    TypeKind: TTypeKind): Boolean; overload;
function PropIsType(AClass: TClass; const PropName: string;
    TypeKind: TTypeKind): Boolean; overload;
```

Возвращает True, если указанное свойство существует и соответствует данному типу.

PropType, функция

```
function PropType(Instance: TObject; const PropName: string):
    TTypeKind; overload;
function PropType(AClass: TClass; const PropName: string):
    TTypeKind; overload;
```

Возвращает категорию типа для публикуемого свойства или вызывает исключительную ситуацию EPropertyError, если класс не имеет свойства указанного типа.

SetEnumProp, процедура

```
procedure SetEnumProp(Instance: TObject; PropInfo: PPropInfo;
  const Value: string); overload;
procedure SetEnumProp(Instance: TObject; const PropName: string;
  const Value: string); overload;
```

Устанавливает значение свойства перечисляемого типа по имени перечисляемого литерала. Если Value не является именем перечисляемого литерала, SetEnumProp вызывает исключительную ситуацию EPropertyConvertError. Если у вас имеется порядковое значение вместо имени литерала, вызывайте SetOrdProp.

SetFloatProp, функция

```
procedure SetFloatProp(Instance: TObject; PropInfo: PPropInfo;
  Value: Extended); overload;
procedure SetFloatProp(Instance: TObject; const PropName: string;
  Value: Extended); overload;
```

Устанавливает значение свойства вещественного типа.

SetInt64Prop, процедура

```
procedure SetInt64Prop(Instance: TObject; PropInfo: PPropInfo;
  const Value: Int64); overload;
procedure SetInt64Prop(Instance: TObject; const PropName: string;
  const Value: Int64); overload;
```

Устанавливает значение свойства, имеющего тип Int64, или поддиапазона, который больше 32 бит.

SetMethodProp, процедура

```
procedure SetMethodProp(Instance: TObject; PropInfo: PPropInfo;
  const Value: TMethod); overload;
procedure SetMethodProp(Instance: TObject; const PropName: string;
  const Value: TMethod); overload;
```

Устанавливает значение свойства-события.

SetObjectProp, процедура

```
procedure SetObjectProp(Instance: TObject; PropInfo: PPropInfo;
  Value: TObject); overload;
procedure SetObjectProp(Instance: TObject; const PropName: string;
  Value: TObject); overload;
```

Устанавливает значение свойства объектного типа. Если Value не соответствует типу свойства, SetObjectProp игнорирует попытку изменить значение свойства.

SetOrdProp, процедура

```
procedure SetOrdProp(Instance: TObject; PropInfo: PPropInfo;
  Value: Longint); overload;
procedure SetOrdProp(Instance: TObject; const PropName: string;
  Value: Longint); overload;
```

Устанавливает значение любого свойства порядкового типа, в том числе множеств, объектов, символов, перечислений или целых.

SetPropValue, процедура

```
procedure SetPropValue(Instance: TObject; const PropName: string;
  const Value: Variant);
```

Устанавливает значение свойства из значения типа `Variant`**.** `SetPropValue` **должна иметь возможность преобразовать значение** `Variant` **к соответствующему типу свойства, иначе будет вызвана исключительная ситуация** `EPropertyConvertError`**.**

SetSetProp, процедура

```
procedure SetSetProp(Instance: TObject; PropInfo: PPropInfo;
  const Value: string); overload;
procedure SetSetProp(Instance: TObject; const PropName: string;
  const Value: string); overload;
```

Устанавливает значение свойства типа «множество» разбором строки со списком перечисляемых литералов. `SetSetProp` **понимает формат, возвращаемый** `GetSetProp`**. Если** `Value` **имеет недопустимый формат,** `SetSetProp` **вызывает исключительную ситуацию** `EPropertyConvertError`**.**

SetStrProp, процедура

```
procedure SetStrProp(Instance: TObject; PropInfo: PPropInfo;
  const Value: string); overload;
procedure SetStrProp(Instance: TObject; const PropName: string;
  const Value: string); overload;
```

Устанавливает значение свойства строкового типа. Тип свойства может быть `tkString`**,** `tkLString` **или** `tkWString`**.**

SetVariantProp, процедура

```
procedure SetVariantProp(Instance: TObject; PropInfo: PPropInfo;
  const Value: Variant); overload;
procedure SetVariantProp(Instance: TObject; const PropName: string;
  const Value: Variant); overload;
```

Устанавливает значение свойства типа `Variant`**.**

Виртуальные и динамические методы

В VMT содержится список указателей на виртуальные методы, а также отдельная таблица, которая в этом разделе называется таблицей динамических методов, хотя в ней перечисляются и динамические методы, и обработчики событий.

Компилятор генерирует небольшое отрицательное число для каждого динамического метода. Это отрицательное число аналогично номеру сообщения для обработчика. Во избежание конфликтов с обработчиками сообщений, компилятор не позволяет создавать обработчики с номерами сообщений, попадающими в диапазон, отведенный динамическим методам. Но как только компилятор завершит эту работу, различие между динамическими методами и обработчиками сообщений теряется. И те и другие располагаются в одной таблице, и больше ничто не указывает на то, что один элемент таблицы относится к динамическому методу, а другой – к обработчику сообщений.

В таблице динамических методов перечислены только динамические методы и обработчики сообщений, объявленные в данном классе; в ней отсутствуют методы, унаследованные от родительских классов. Таблица динамических методов начинается с двухбайтового значения количества динамических методов и обработчиков сообщений, за которым следует список двухбайтовых номеров методов, а затем – четырехбайтовые указатели методов. Таблица динамических методов организована именно таким образом (вместо списка записей, каждая из которых содержит номер метода и его указатель) с целью ускорения поиска метода по его номеру. В примере 3-6 показана логическая структура таблицы динамических методов. Как и другие таблицы, этот код нельзя откомпилировать, т. к. это не настоящий язык Паскаль, а лишь описание того, как выглядит таблица динамических методов.

Пример 3-6. Структура таблицы динамических методов

```
type
  TDynMethodTable = packed record
    Count: Word;
    Indexes: packed array[1..Count] of SmallInt;
    Addresses: packed array[1..Count] of Pointer;
  end;
```

Для обработки сообщения или вызова динамического метода необходимо найти в массиве `Indexes` номер сообщения или метода. Эта таблица не отсортирована, и поиск выполняется линейно. Как только соответствие найдено, по указанному в массиве `Addresses` адресу вызывается искомый метод. Если номер метода не найден, поиск продолжается в ближайшем базовом классе.

Применение динамических методов имеет смысл только в том случае, если выполняются все указанные ниже условия:

- создается большая схема, содержащая сотни классов;
- требуется много виртуальных методов, объявленных вблизи корня дерева наследования;
- динамические методы редко переопределяются в производных классах;
- эти методы никогда не вызываются в тех случаях, когда важна скорость их вызова.

Различие между виртуальными и динамическими методами заключается в том, что таблицы виртуальных методов содержат все методы родительских классов, и поэтому они могут иметь большой размер. Таблицы динамических методов не включают унаследованные методы, и поэтому могут быть меньше. С другой стороны, вызов виртуального метода осуществляется с помощью непосредственного индекса одной таблицы, а для вызова динамического метода требуется поиск в одной и более таблицах.

В VCL динамические методы используются только там, где они вызываются в ответ на действия пользователя, т. е. когда медленный поиск метода не влияет на общую производительность. Кроме того, динамические методы обычно объявлены в корневых классах, таких как `TControl`.

При небольшой иерархии классов виртуальные методы обычно дают более компактный и быстрый код, чем динамические методы, поскольку для динамического метода, помимо адреса метода, требуется хранить и его номер. При недостаточном количестве классов, не переопределяющих динамические методы, таблица динамических методов будет требовать больше памяти, чем таблица виртуальных методов.

Инициализация и завершение

При создании объекта Delphi автоматически инициализирует строки, динамические массивы, интерфейсы и значения типа `Variant`. При уничтожении объекта Delphi должна уменьшить счетчики ссылок для строк, интерфейсов, динамических массивов, освободить значения `Variant` и «широкие» строки. Delphi отслеживает эту информацию с помощью инициализационных записей, являющихся частью RTTI класса. Фактически каждая запись или массив, которые требуют кода завершения, имеют соответствующую запись инициализации, но компилятор прячет эти записи. Программист имеет доступ только к данным, относящимся к полям объектных типов.

В VMT есть указатель на таблицу инициализации, которая содержит список инициализационных записей. Поскольку массивы и записи могут быть вложенными, каждая инициализационная запись содер-

жит указатель на другую таблицу инициализации, которая может содержать инициализационные записи и т. д. Таблица инициализации использует поле TTypeKind для определения того, что требуется инициализировать – строку, запись, массив и т. д.

Таблица инициализации начинается с категории типа (один байт), затем следуют имя типа в виде short string, размер инициализируемых данных (четыре байта), четырехбайтовый счетчик инициализационных записей и массив инициализационных записей, содержащий ноль или более записей. Инициализационная запись – это просто указатель на вложенную инициализационную таблицу, за которым следует четырехбайтовое смещение инициализируемого поля. В примере 3-7 показана логическая структура таблицы и записи инициализации, но объявления лишь иллюстрируют эту схему, не являясь истинным кодом языка Паскаль.

Пример 3-7. Структура таблицы и записи инициализации

```
type
  { Запись инициализации/завершения }
  PInitTable = ^TInitTable;
  TInitRecord = packed record
    InitTable: ^PInitTable;
    Offset: LongWord;        // Смещение поля в объекте
  end;
  { Таблица инициализации/завершения }
  TInitTable = packed record
  {$MinEnumSize 1} // Убедитесь, что TypeKind занимает один байт.
    TypeKind: TTypeKind;
    TypeName: packed ShortString;
    DataSize: LongWord;
    Count: LongWord;
    // Если TypeKind=tkArray, то Count - это размер массива, но
    // InitRecords содержит только один элемент; если TypeKind -
    // tkRecord, Count - это количество элементов записи, и в
    // InitRecords[] содержатся записи для каждого элемента.
    // Для всех остальных типов Count=0.
    InitRecords: array[1..Count] of TInitRecord;
  end;
```

Основная запись TInitRecord класса содержит пустое имя типа и нулевой размер данных. Категория типа – это всегда tkRecord. Count – это количество полей, которые требуют инициализации; массив InitRecords содержит TInitRecord для каждого такого поля. Каждая инициализационная запись указывает на инициализационную таблицу, содержащую категорию типа и имя типа для соответствующего поля. Подобная организация может показаться странной, однако вы к ней быстро привыкнете.

Большинство типов не требуют инициализации или завершения, но для следующих типов это необходимо:

`tkArray`

`DataSize` – это размер каждого элемента массива, а `Count` – количество элементов массива. Все элементы массива одинаковы, поэтому массив `InitRecords` содержит только одну запись `TInitRecord` для всех элементов массива. Значение поля `Offset` записи `TInitRecord` не используется.

`tkDynArray`

`DataSize` и `Count` не используются. Delphi уменьшает счетчик ссылок на массив и освобождает занимаемую им память, если счетчик ссылок становится равным нулю.

`tkInterface`

`DataSize` и `Count` не используются. Delphi вызывает метод `_Release`, который освобождает память интерфейсного объекта, если счетчик ссылок становится равным нулю.

`tkLString`

`DataSize` и `Count` не используются. Delphi уменьшает счетчик ссылок на строку и освобождает отведенную для нее память, если счетчик ссылок становится равным нулю.

`tkRecord`

`DataSize` – это размер записи, `Count` – количество полей записи, которые требуют инициализации. Массив `InitRecords` содержит `TInitRecord` для каждого поля, требующего инициализации.

`tkVariant`

`DataSize` и `Count` не используются. Delphi освобождает всю память, связанную с данными `Variant`.

`tkWString`

`DataSize` и `Count` не используются. Delphi освобождает память, занятую строкой.

Автоматические методы

Секция `automated` в объявлении класса является устаревшей, поскольку сервер COM-автоматизации проще создавать с помощью редактора библиотеки типов Delphi, используя интерфейсы. Однако с целью обратной совместимости компилятор пока поддерживает объявления `automated`. Из будущих версий компилятора поддержка `automated` методов может быть исключена.

В модуле `OleAuto` подробно описана таблица автоматических методов: она начинается с двухбайтового счетчика, за которым следует список автоматических записей. В каждой записи содержится четырехбайто-

вый dispid (идентификатор диспетчера), указатель на имя метода в формате short string, четыре байта флагов, указатель на список параметров и указатель на код метода. Список параметров начинается с однобайтового типа возвращаемого значения, затем идет один байт для количества параметров и в завершение – список однобайтовых типов параметров. Имя параметра не сохраняется. В примере 3-8 приведено описание таблицы автоматических методов.

Пример 3-8. Структура таблицы автоматических методов

```
const
  { Маски для типа параметров }
  atTypeMask = $7F;
  varStrArg  = $48;
  atByRef    = $80;
  MaxAutoEntries = 4095;
  MaxAutoParams = 255;

type
  TVmtAutoType = Byte;
  { Элемент списка параметров }
  PAutoParamList = ^TAutoParamList;
  TAutoParamList = packed record
    ReturnType: TVmtAutoType;
    Count: Byte;
    Types: array[1..Count] of TVmtAutoType;
  end;
  { Элемент таблицы автоматических методов }
  PAutoEntry = ^TAutoEntry;
  TAutoEntry = packed record
    DispID: LongInt;
    Name: PShortString;
    Flags: LongInt; { В младшем байте содержатся флаги }
    Params: PAutoParamList;
    Address: Pointer;
  end;

  { Структура таблицы автоматических методов }
  PAutoTable = ^TAutoTable;
  TAutoTable = packed record
    Count: LongInt;
    Entries: array[1..Count] of TAutoEntry;
  end;
```

Интерфейсы

Любой класс может реализовывать любое количество интерфейсов. Компилятор формирует таблицу интерфейсов как часть RTTI класса. В VMT есть указатель на таблицу интерфейсов, которая начинается с четырехбайтового счетчика, за которым следует список интерфейсных

записей. Каждая интерфейсная запись содержит GUID, указатель на VMT интерфейса, смещение для скрытого поля интерфейса и указатель на свойство, реализующее интерфейс с помощью директивы `implements`. При смещении, равном нулю, интерфейсное свойство (называемое `ImplGetter`) должно быть ненулевым; если смещение не равно нулю, `ImplGetter` должно быть равно `nil`. Интерфейсное свойство может быть ссылкой на поле, виртуальным методом или статическим методом, в соответствии с соглашением, принятым для считывателя свойства (которое описано ранее в этой главе в разделе «Публикуемые свойства»). При создании объекта Delphi автоматически проверяет все интерфейсы, и для каждого интерфейса с ненулевым полем `IOffset` инициализирует поле, расположенное по этому смещению, указателем на `VTable` интерфейса (указателем на его VMT). В отличие от других таблиц RTTI, Delphi определяет типы для таблицы интерфейсов в модуле `System`. Эти типы приведены в примере 3-9.

Пример 3-9. Объявления для таблицы интерфейсов

```
type
  PInterfaceEntry = ^TInterfaceEntry;
  TInterfaceEntry = record
    IID: TGUID;
    VTable: Pointer;
    IOffset: Integer;
    ImplGetter: Integer;
  end;

  PInterfaceTable = ^TInterfaceTable;
  TInterfaceTable = record
    EntryCount: Integer;
    // Тип объявляется максимально возможного размера,
    // однако истинный размер массива равен EntryCount.
    Entries: array[0..9999] of TInterfaceEntry;
  end;
```

В `TObject` реализовано несколько методов для доступа к таблице интерфейсов. Смотрите в главе 5 подробное описание методов `GetInterface`, `GetInterfaceEntry` и `GetInterfaceTable`.

Изучение RTTI

Эта глава дает представление о таблице виртуальных методов класса и информации о типе времени выполнения. Для того чтобы лучше понять, как Delphi формирует и использует RTTI, необходимо самостоятельно изучить таблицы. Код, дополняющий эту книгу и содержащийся на веб-сайте O'Reilly, включает и программу *Vmt.exe*. В модуле `VmtInfo` определен набор интерфейсов, которые раскрывают структуру всех таблиц RTTI. В модуле `VmtImpl` определены классы, реализующие эти интерфейсы. Можно ознакомиться с исходным кодом модуля

VmtImpl или ограничиться программой *Vmt*. Посмотрите модуль VmtForm при желании изменить объявления типов или добавить типы с целью их изучения.

Можно использовать интерфейсы модуля VmtInfo и в ваших собственных программах при необходимости доступа к таблицам RTTI. Например, можно написать собственную библиотеку перманентных объектов, позволяющую получить доступ к таблице полей класса для связывания имен классов со ссылками на классы.

Эти интерфейсы не требуют пояснений. Поскольку они используют автоматические счетчики ссылок Delphi, вам не придется беспокоиться об управлении памятью. Для того чтобы создать интерфейс, вызовите одну из следующих функций:

```
function GetVmtInfo(ClassRef: TClass): IVmtInfo; overload;
function GetVmtInfo(ObjectRef: TObject): IVmtInfo; overload;
function GetTypeInfo(TypeInfo: PTypeInfo): ITypeInfo;
```

Используйте интерфейс VmtInfo и связанные с ним интерфейсы для просмотра и изучения богатого мира информации о типе времени выполнения Delphi. Например, посмотрите на класс TFont, показанный в примере 3-10.

Пример 3-10. Объявление класса TFont

```
type
  TFont = class(TGraphicsObject)
  private
    FColor: TColor;
    FPixelsPerInch: Integer;
    FNotify: IChangeNotifier;
    procedure GetData(var FontData: TFontData);
    procedure SetData(const FontData: TFontData);
  protected
    procedure Changed; override;
    function GetHandle: HFont;
    function GetHeight: Integer;
    function GetName: TFontName;
    function GetPitch: TFontPitch;
    function GetSize: Integer;
    function GetStyle: TFontStyles;
    function GetCharset: TFontCharset;
    procedure SetColor(Value: TColor);
    procedure SetHandle(Value: HFont);
    procedure SetHeight(Value: Integer);
    procedure SetName(const Value: TFontName);
    procedure SetPitch(Value: TFontPitch);
    procedure SetSize(Value: Integer);
    procedure SetStyle(Value: TFontStyles);
    procedure SetCharset(Value: TFontCharset);
  public
```

```
    constructor Create;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property FontAdapter: IChangeNotifier read FNotify write FNotify;
    property Handle: HFont read GetHandle write SetHandle;
    property PixelsPerInch: Integer read FPixelsPerInch
        write FPixelsPerInch;
  published
    property Charset: TFontCharset read GetCharset write SetCharset;
    property Color: TColor read FColor write SetColor;
    property Height: Integer read GetHeight write SetHeight;
    property Name: TFontName read GetName write SetName;
    property Pitch: TFontPitch read GetPitch write SetPitch
        default fpDefault;
    property Size: Integer read GetSize write SetSize stored False;
    property Style: TFontStyles read GetStyle write SetStyle;
  end;
```

Заметьте, что одно из полей имеет тип `IChangeNotifier`. Метод `Changed` объявлен как динамический в базовом классе `TGraphicsObject`. В `TFont` нет публикуемых полей или методов, но есть несколько публикуемых свойств. В примере 3-11 приведена VMT и информация о типе для класса `TFont`. В таблице динамических методов имеется один элемент для `Changed`. Свойство `Size` не сохраняется, но присутствуют другие публикуемые свойства. Программа *Vmt.exe* может показать вам подобную информацию практически для любого класса или типа.

Пример 3-11. Информация о типе времени выполнения

```
Vmt: 40030E78
  Destroy: 4003282C
  FreeInstance: 400039D8
  NewInstance: 400039C4
  DefaultHandler: 40003CAC
  Dispatch: 40003CB8
  BeforeDestruction: 40003CB4
  AfterConstruction: 40003CB0
  SafeCallException: 40003CA4
  Parent: 40030DA4 (TGraphicsObject)
  InstanceSize: 32
  ClassName: 'TFont'
  Dynamic Method Table: 40030EE2
    Count: 1
    40032854 (-3)
  Method Table: 00000000
  Field Table: 00000000
  TypeInfo: 40030EF4
  InitTable: 40030ED0
  TypeName:
  TypeKind: tkRecord
  DataOffset: 0
```

```
  Count: 1
  RecordSize: 0
    [1]
    InitTable: 40030E44
    TypeName: IChangeNotifier
    TypeKind: tkInterface
    DataOffset: 28
  AutoTable: 00000000
  IntfTable: 00000000

type TFontCharset = 0..255; // otUByte
type TColor = -2147483648..2147483647; // otSLong
type Integer = -2147483648..2147483647; // otSLong
type TFontName; // tkLString
type TFontPitch = (fpDefault, fpVariable, fpFixed); // otUByte
type TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
type TFontStyles = set of TFontStyle; // otUByte
type TObject = class // unit 'System'
end;
type TPersistent = class(TObject) // модуль 'Classes'
end;
type TGraphicsObject = class(TPersistent) // модуль 'Graphics'
end;
type TFont = class(TGraphicsObject) // модуль 'Graphics'
published
  property Charset: TFontCharset read (static method 40032CD4)
    write (static method 40032CDC) nodefault stored True; // index 0
  property Color: TColor read (field 20) write (static method 400329AC)
    nodefault stored True; // index 1
  property Height: Integer read (static method 40032B8C)
    write (static method 40032B94) nodefault stored True; // index 2
  property Name: TFontName read (static method 40032BBC)
    write (static method 40032BD4) nodefault stored True; // index 3
  property Pitch: TFontPitch read (static method 40032CA4)
    write (static method 40032CAC) default 0 stored True; // index 4
  property Size: Integer read (static method 40032C30)
    write (static method 40032C4C) nodefault stored False; // index 5
  property Style: TFontStyles read (static method 40032C6C)
    write (static method 40032C78) nodefault stored True; // index 6
end;
```

Глава 4

Создание многопоточных приложений

Не так давно последовательное программирование с текстовым интерфейсом сменилось графическим, событийным программированием, а теперь однопоточное программирование вытесняется программированием многопоточным. Будущее – за параллельным программированием.

Пишете ли вы веб-сервер, который должен обрабатывать одновременно несколько клиентов, или пользовательское приложение, такое как текстовый процессор, параллельное программирование – это то, что вам нужно. Текстовый процессор может проверять правописание, в то время как пользователь набирает текст, и печатать файл в фоновом режиме, пока пользователь продолжает редактирование. На сегодняшний день пользователи ожидают большего от современных приложений, и только параллельное программирование может обеспечить необходимую мощность и гибкость.

В Delphi Pascal включены средства для поддержки параллельного программирования – их не так много, как в языках, подобных Ada, но больше, чем в традиционных языках программирования. В дополнение к языковым средствам можно использовать Windows API и его семафоры, потоки, процессы, конвейеры, общую память и т. д. В этой главе описаны средства, уникальные для Delphi Pascal, а также приемы эффективного написания параллельных программ в Delphi. Дополнительная информация о Windows API и деталях работы с потоками, процессами, семафорами и т. п. в Windows, содержится в книгах по программированию Windows, например, Дэвид Соломон (David Solomon) «Inside Windows NT», 2 издание (Microsoft Press, 1998 год).

Потоки и процессы

В этом разделе приведен обзор многопоточного программирования в Windows. Читатели, уже знакомые с потоками и процессами Windows, могут пропустить этот раздел и продолжить со следующего – «Класс TThreads».

Поток (*thread*) – это единица непрерывного выполнения кода программы. Программа может иметь несколько потоков, каждый со своим стеком, собственной копией регистров процессора и другой подобной информацией. В многопроцессорной системе каждый процессор исполняет отдельный поток[1]. В однопроцессорной системе Windows создает иллюзию параллельного выполнения потоков, хотя в каждую единицу времени процессор занят только одним потоком.

Процесс – это набор потоков, выполняемых в едином адресном пространстве. Каждый процесс должен содержать по крайней мере один поток, который называется *основным*. Потоки одного и того же процесса могут использовать одни ресурсы, например открытые файлы, и обращаться к любой доступной памяти в адресном пространстве процесса. Поток можно рассматривать как экземпляр приложения (вместе со всеми DLL, которые им загружены).

Потоки одного процесса могут легко общаться между собой посредством общих переменных, при использовании которых критические секции защищают потоки от возможных конфликтов. (Подробности о критических секциях смотрите далее в этой главе в разделе «Синхронизация потоков».)

Можно послать Windows-сообщение конкретному потоку, который в этом случае должен иметь цикл обработки этого сообщения. Обычно проще оставить обработку всех Windows-сообщений основному потоку, однако не бойтесь написать собственный цикл обработки сообщений для любого потока, которому это требуется.

Существуют различные способы взаимодействия между отдельными процессами, такие как сообщения, взаимные исключения (mutexes), семафоры, события, файлы в памяти, сокеты (sockets), конвейеры, DCOM, CORBA и т. д., комбинация которых обычно используется для этой цели. Отдельные процессы не могут разделять обычную память, и нельзя вызывать функцию или процедуру одного процесса из другого, хотя существует несколько механизмов удаленного вызова процедур,

[1] Строго говоря, решением вопроса о распределении потоков между процессорами многопроцессорной среды занимается операционная система. Именно она решает, выполнять ли отдельный поток приложения на отдельном процессоре или нет. При принятии такого решения операционная система обычно руководствуется соображениями наилучшей производительности с учетом приоритетов всех работающих процессов и потоков. – *Примеч. науч. ред.*

такие как DCOM и CORBA. Подробнее процессы и их взаимодействие рассмотрены в разделе «Процессы» далее в этой главе.

В Delphi имеется встроенная поддержка многопоточного программирования – написания приложений и DLL, работающих с несколькими потоками в одном процессе. В распоряжении программиста имеется полный набор функций Windows API, независимо от того, работает ли он с потоками или процессами.

Вы должны быть уверены, что в многопоточном приложении или библиотеке глобальная переменная `IsMultiThread` установлена в истину (true). Большинство приложений делает это автоматически, вызывая метод `BeginThread` или используя класс `TThread`. Однако если вы пишете библиотеку (DLL), которая может быть вызвана из многопоточного приложения, вам может потребоваться вручную присвоить переменной `IsMultiThread` значение `True`.

Состояния и распределение потоков

Windows распределяет потоки в соответствии с их приоритетами. Потоки с более высоким приоритетом выполняются в первую очередь. При одинаковом приоритете Windows распределяет потоки так, чтобы у каждого из них был одинаковый шанс выполняться. Windows может приостановить работающий поток (это называется *вытеснением* потока) для того, чтобы дать возможность работать другому. В Windows определено несколько состояний потоков, но все они относятся к одной из трех категорий:

Работает

Поток работает, когда он занимает процессор. Система может иметь столько одновременно работающих потоков, сколько в ней процессоров – по одному потоку на процессор. Поток находится в работающем состоянии до тех пор, пока не будет заблокирован ожиданием какой-либо операции (например, завершения ввода/вывода). Тогда Windows вытесняет поток, чтобы дать другим потокам возможность работать. Кроме того, поток может приостановить свою работу самостоятельно.

Готов

Поток готов к работе, если он не работает и не заблокирован. Такой поток может вытеснить работающий поток с тем же (но не с большим) приоритетом.

Заблокирован

Поток заблокирован, если он чего-то ожидает: завершения ввода/вывода или другой подобной операции, доступа к общему ресурсу и т. д. Программист может явно заблокировать поток, приостановив его работу, и поток будет ждать запуска сколько угодно долго.

Главное, что следует понимать при написании многопоточных приложений, это когда заблокировать поток и когда его разблокировать, с тем чтобы потоки как можно меньше времени находились в заблокированном состоянии и как можно больше – в работающем.

Пребывание одновременно большого числа потоков в состоянии готовности (но не работающих), говорит о проблемах с производительностью – процессор не может обеспечить ресурсами потоки, готовые к работе. Возможно, приложение создает слишком много активных потоков или процессор слишком медленный, и необходимо перейти на многопроцессорную систему. Решение проблемы с ресурсами выходит за рамки этой книги, узнать больше об анализе и решении вопросов производительности можно из книг по администрированию Windows NT.

Синхронизация потоков

Самая большая проблема при многопоточном программировании – это сохранение целостности данных. Потоки, которые имеют доступ к общей переменной, могут «наступать друг другу на пятки». В примере 4-1 показан простой класс, содержащий глобальный счетчик. При попытке двух потоков одновременно увеличить счетчик, последний может принять неверное значение. Рис. 4-1 иллюстрирует эту проблему: `Counter` инициализируется нулем и должен стать равным 3 после создания трех объектов `TCounter`. Три потока борются за доступ к общей переменной, и по окончании работы приложения в `Counter` оказывается неверный результат, а именно 1. Этот эффект называется *состоянием гонок*, т. к. каждый поток стремится завершить свою работу до того, как к финишу придет другой поток, и тем самым путает все карты.

Пример 4-1. Простой класс-счетчик

```
var
  Counter: Integer;
type
  TCounter = class
  public
    constructor Create;
    function GetCounter: Integer;
  end;

constructor TCounter.Create;
begin
  inherited;
  Counter := Counter + 1;
end;

function TCounter.GetCounter: Integer;
begin
  Result := Counter;
end;
```

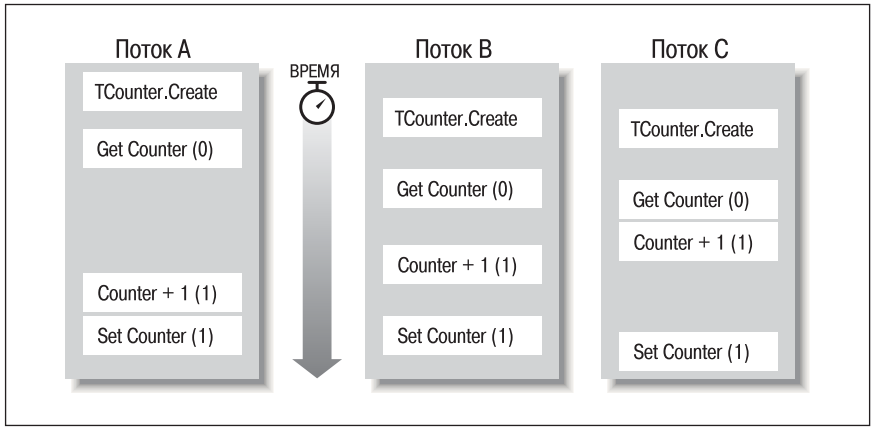


Рис. 4-1. «Состояние гонок» приводит к неверному значению Counter

В этом примере приведен целочисленный счетчик, так как работа с целыми проще всего. Чтение целого значения – это *элементарная* операция, т. е. операция, которая не может быть прервана другим потоком. Если бы счетчик был, например, переменной типа `Variant`, чтение его значения требовало бы нескольких инструкций процессора и могло бы быть прервано другим потоком в любой момент. Таким образом, даже чтение счетчика типа `Variant` в многопоточном приложении не является безопасным без обеспечения одновременного доступа к нему только одного потока.

Критические секции

Для сохранения целостности переменной `Counter` все потоки должны сотрудничать и договориться о том, что в один и тот же момент времени только один из них может изменять значение переменной. Другие потоки могут читать эту переменную и получать ее значение, но когда поток хочет изменить это значение, он должен предотвратить попытки других потоков сделать то же самое. Стандартный прием обеспечения однопоточного доступа – это критическая секция.

Критическая секция – это часть кода, которая резервируется для однопоточного доступа. Когда один из потоков входит в критическую секцию, все остальные потоки могут продолжать свою работу только до тех пор, пока они также не захотят войти в критическую секцию. При попытке сделать это, любой другой поток блокируется и ожидает, пока первый поток завершит исполнение критической секции. Тогда в критическую секцию входит следующий поток. В Windows API определено несколько функций для создания и использования критической секции, как показано в примере 4-2.

Пример 4-2. Счетчик с использованием критической секции

```
var
  Counter: Integer;
```

```
  CriticalSection: TRtlCriticalSection;

type
  TCounter = class
  public
    constructor Create;
    function GetCounter: Integer;
  end;

constructor TCounter.Create;
begin
  inherited;
  EnterCriticalSection(CriticalSection);
  try
    Counter := Counter + 1;
  finally
    LeaveCriticalSection(CriticalSection);
  end;
end;

function TCounter.GetCounter: Integer;
begin
  // Не требует критической секции, так как операции с
  // целыми - элементарные.
  Result := Counter;
end;

initialization
  InitializeCriticalSection(CriticalSection);
finalization
  DeleteCriticalSection(CriticalSection);
end.
```

Новый класс `TCounter` поддерживает многопоточность, т. е. позволяет использовать один объект этого типа одновременно в нескольких потоках. Большинство классов Delphi не поддерживают многопоточность, и нельзя использовать один объект в нескольких потоках, по крайней мере без применения критических секций для защиты целостности внутреннего состояния объекта.

Одним из преимуществ объектно-ориентированного программирования является то, что часто можно обойтись и без класса, поддерживающего многопоточность, – потоки не смогут испортить данные друг друга, если каждый поток создает и использует собственный экземпляр класса. Это позволяет создавать и использовать `TList` и другие объекты внутри потока. Однако делить один объект `TList` между несколькими потоками следует с осторожностью.

Следует минимизировать размер критической секции, т. к. потоки ожидают ее высвобождения. В противном случае потоки будут без необходимости долго ждать окончания выполнения критической секции.

Одновременное чтение

В классе TCounter каждый поток может проверить значение счетчика в любое время, т. к. переменная Counter – элементарная. Критическая секция имеет отношение только к потокам, изменяющим значение счетчика. Если переменная, к которой происходит обращение, не является элементарной, необходимо защищать и запись и чтение – критическая секция в данном случае не подходит. Вместо этого используется класс TMultiReadExclusiveWriteSynchronizer, который объявлен в модуле SysUtils. Громоздкое имя говорит само за себя: этот класс подобен критической секции, но позволяет нескольким потокам иметь к критической области памяти доступ только для чтения. Поток, которому требуется доступ на запись, должен дождаться окончания операций чтения других потоков. Полное описание этого класса приведено в приложении В «Модуль SysUtils».

Исключительные ситуации

Исключительные ситуации в потоках приводят к прекращению работы приложения, поэтому необходимо перехватывать исключительную ситуацию и находить способ передачи информации о ней основному потоку. Существует возможность просто выдать сообщение об ошибке. Для этого необходимо перехватить исключение, извлечь сообщение об ошибке и передать его основному потоку, который покажет его, например, в диалоговом окне.

Для получения основным потоком реального объекта исключительной ситуации потребуется некоторая дополнительная работа. Проблема состоит в том, что по окончании обработки исключительной ситуации в блоке try-except Delphi автоматически уничтожает объект Exception. Однако можно изменить схему обработки исключительной ситуации таким образом, чтобы «обмануть» Delphi и предотвратить уничтожение объекта Exception. Для этого необходимо написать обработчик исключительной ситуации, который бы перехватывал объект, передавал его основному потоку и предотвращал его преждевременное уничтожение.

В примере 4-3 показано одно из возможных решений, при котором блок try-except включает в себя основной блок процедуры потока. В потоковую функцию в качестве параметра передается указатель, по которому функция может записать объект Exception или nil, если функция завершила работу успешно.

Пример 4-3. Перехват исключительной ситуации в потоке

```
type
  PObject = ^TObject;
  PRaiseFrame = ^TRaiseFrame;
  TRaiseFrame = record
    NextFrame: PRaiseFrame;
```

```
    ExceptAddr: Pointer;
    ExceptObject: TObject;
    ExceptionRecord: PExceptionRecord;
  end;

// ThreadFunc перехватывает исключительную ситуацию и
// записывает в Param^ объект Exception или nil,
// если поток не вызвал исключительных ситуаций.
function ThreadFunc(Param: Pointer): Integer;
var
  RaiseFrame: PRaiseFrame;
begin
  Result := 0;
  PObject(Param)^ := nil;
  try
    DoTheThreadsRealWorkHere;
  except
    // RaiseList равен nil, если исключительных ситуаций не
    // возникло, иначе указывает на запись TExceptionFrame.
    RaiseFrame := RaiseList;
    if RaiseFrame <> nil then
    begin
      // Когда в потоке возникает исключительная ситуация, объект
      // Exception записывается в объектный параметр функции.
      // Затем обнуляет ссылку на Exception в локальной переменной
      // потока, чтобы программа не уничтожила объект раньше времени.
      PObject(Param)^ := RaiseFrame.ExceptObject;
      RaiseFrame.ExceptObject := nil;
    end;
  end;
end;
```

Взаимная блокировка

Взаимная блокировка (deadlock) – это ситуация, при которой потоки бесконечно находятся в состоянии ожидания друг друга. Поток A ожидает поток B, поток B ожидает поток A, и ни один из них не выполняет никаких действий. Наличие нескольких потоков приводит к возможности взаимной блокировки. В сложной программе бывает затруднительно найти место, где это может случиться, и тестирование – это не самый надежный способ обнаружения таких мест. Наилучший вариант – исключить возможность взаимных блокировок профилактическими мерами при разработке и написании программы.

Обычный источник взаимных блокировок – ожидание потоком высвобождения нескольких различных ресурсов. Например, приложение может иметь две глобальные переменные, и для каждой из них есть собственная критическая секция. В случае, если потоку потребуется изменить значения обеих переменных, он попытается войти в обе критичес-

кие секции, и если другой поток попробует сделать то же самое, это может привести к взаимной блокировке, что иллюстрирует рис. 4-2.

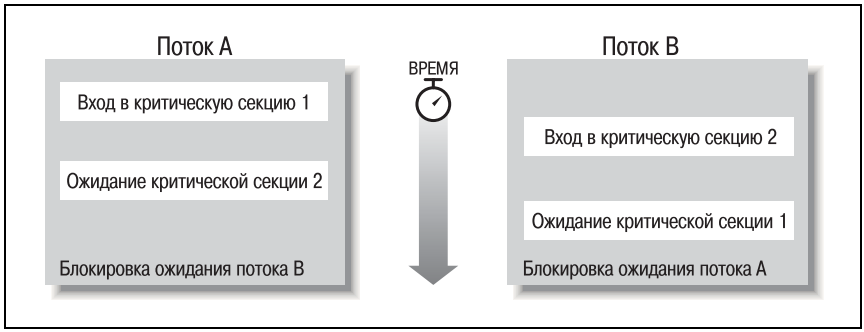


Рис. 4-2. Взаимная блокировка останавливает выполнение обоих потоков

Можно легко избежать взаимной блокировки в такой ситуации, если гарантировать ожидание критических секций обоими потоками в одном порядке. Когда поток A входит в критическую секцию 1, он не дает войти в нее потоку B, затем поток A переходит к критической секции 2. Как только поток A заканчивает выполнение обеих критических секций, поток B может войти в критическую секцию 1 и затем – в критическую секцию 2. На рис. 4-3 показано, как два потока могут работать вместе, избегая взаимной блокировки.

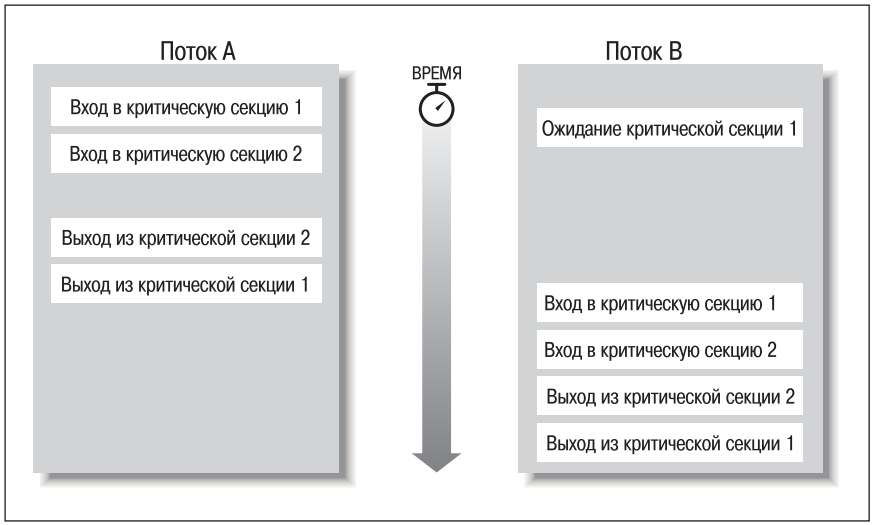


Рис. 4-3. Предотвращение взаимных блокировок изменением порядка выполнения критических секций

Другой способ избежать взаимных блокировок при работе с общими ресурсами – это гарантировать, что поток либо получит все требуемые ресурсы сразу, либо не получит ни один из них. Например, если пото-

ку требуется монопольный доступ к двум файлам, он должен иметь возможность открыть оба. Если поток A открывает файл 1, но не может открыть файл 2 из-за того, что с ним работает поток B, поток A должен закрыть файл 1 и ждать, пока не освободятся оба файла.

Взаимные блокировки, их причины и предотвращение – стандартная тема учебных курсов программирования, и классические стратегии борьбы с ними описаны во многих учебниках. В Delphi нет специальных средств обнаружения и предотвращения взаимных блокировок, и поэтому программисту приходится прикладывать собственные усилия для решения этой задачи.

Многопоточное программирование

Ключ к эффективному многопоточному программированию – в понимании, когда требуется использовать критические секции, а когда – нет. Любая переменная, значение которой может изменяться несколькими потоками, должна быть защищена, однако не всегда известно, когда эти изменения произойдут. Доступ на чтение и запись любых составных структур данных должен быть защищен. Например, иногда лишь чтение значения свойства приводит к изменению глобальных переменных другого модуля. Сама система Windows имеет данные, нуждающиеся в защите. Ниже приведен список объектов, о которых следует позаботиться:

- Любой вызов функции Windows GDI (Graphics Device Interface, интерфейс графического устройства) должен быть защищен.
- Некоторые ссылки на компоненты и управляющие элементы VCL должны быть защищены. В каждой версии Delphi поддержка многопоточного программирования в VCL увеличивается, и в Delphi 5 большая часть VCL уже поддерживает многопоточность. Изменять объект VCL из нескольких потоков, конечно, нельзя, и любое свойство или метод, ссылающийся на Windows GDI, не поддерживает многопоточность. Как правило, все визуальные компоненты не поддерживают многопоточность, а те, что выполняют работу «за сценой» – поддерживают. Если нет уверенности насчет какого-либо свойства или компонента, следует считать, что они не поддерживают многопоточность.
- Чтение длинных строк и динамических массивов поддерживает многопоточность, но запись – нет. Ссылка на строку или динамический массив может изменить счетчик ссылок, но Delphi защищает этот счетчик для обеспечения многопоточности. Однако для изменения значения строки или динамического массива следует использовать критическую секцию, как и при изменении любой другой переменной. (Заметьте, что Delphi 4 и более ранние версии не защищали строки и динамические массивы подобным образом.)

- Выделение памяти (GetMem и New) и освобождение памяти (FreeMem и Dispose) поддерживают многопоточность. Delphi автоматически защищает свой менеджер памяти для использования в многопоточных приложениях (если IsMultiThread равно истине). Это одна из причин отдавать предпочтение функции BeginThread, а не функции CreateThread из Windows API.
- Создание или уничтожение объекта работает корректно в многопоточном приложении, только если конструктор и деструктор поддерживают многопоточность. С точки зрения работы с памятью, создание и уничтожение объектов поддерживает многопоточность, но остальная работа в конструкторе и деструкторе – за программистом. Если нет уверенности, что класс поддерживает многопоточность, нужно считать, что он ее не поддерживает. Например, создание управляющих элементов VCL не поддерживает многопоточность, а создание TList и TStringList – поддерживает.

Для вызовов функций Windows GDI и работы с VCL Delphi предоставляет вместо критических секций другой механизм, работающий лучше, а именно класс TThread и его метод Synchronize, которые рассматриваются в следующем разделе.

Класс TThread

Простейший способ создать многопоточное приложение в Delphi – это создать класс потока, унаследовав его от TThread. TThread не является частью языка Delphi Pascal, а объявляется в модуле Classes. Описание этого класса приводится здесь потому, что он очень важен для программирования в Delphi.

Для программирования функциональности потока переопределите метод Execute. По завершении метода Execute поток также завершает свою работу. Любой поток может создавать другие потоки путем создания экземпляра вашего потокового класса. Каждый экземпляр класса работает в отдельном потоке и с собственным стеком.

Защищать доступ к VCL или вызов Windows GDI следует с помощью метода Synchronize, принимающего в качестве аргумента процедуру и вызывающего ее с учетом многопоточного выполнения. Процедура не может иметь аргументов. Synchronize приостанавливает выполнение текущего потока и заставляет основной поток исполнить эту процедуру. По завершении процедуры управление возвращается в текущий поток. Поскольку все вызовы метода Synchronize обрабатываются основным потоком, они защищены от «состояния гонки». При вызове Synchronize несколькими потоками одновременно, они ожидают в очереди, и только один поток в текущий момент времени получает доступ к основному потоку. Этот процесс также называется *преобразованием в последовательную форму (serialize)*, т. к. параллельный метод вызова преобразуется в последовательный.

При написании синхронизируемой процедуры следует не забывать о том, что она вызывается из основного потока, и если требуется узнать идентификатор вызвавшего потока, нужно проверить свойство `ThreadID` вместо вызова функции `GetCurrentThreadID` из Windows API.

Предположим, что программисту требуется написать текстовый редактор, который может выполнять печать в фоновом режиме. То есть пользователь просит напечатать файл, программа копирует содержимое файла (чтобы исключить «состояние гонки», которое может возникнуть, если пользователь будет печатать файл, когда операция печати еще не закончится) и запускает фоновый поток, форматирующий файл и отправляющий его в очередь на принтер.

Печать требует работы с VCL, и потому необходимо позаботиться о синхронизации всех операций с VCL. На первый взгляд это кажется проблематичным, так как почти все операции требуют обращения к объекту `Printer`. Есть несколько способов сокращения взаимодействия с VCL.

Первый шаг – копирование некоторой информации из объекта `Printer` в локальные поля потока. В потоке, например, содержится копия величины разрешения принтера, и ему не потребуется обращаться к объекту `Printer` за этой информацией. Следующий шаг – разбиение процесса печати файла на несколько этапов и выделение тех операций, которые требуют работы с VCL. Печать файла состоит из следующих этапов:

1. Начало процесса печати.
2. Инициализация полей, номеров страниц и положения текста на странице.
3. Если это новая страница, печать верхнего и нижнего колонтитула.
4. Печать строки текста и увеличение счетчика положения на странице.
5. Если текст на странице выходит за ее край, начало новой страницы.
6. Если текста больше нет, завершение процесса печати, в противном случае – возврат к этапу 2.

Основные операции, которые требуют синхронизации с VCL:

- начало процесса печати;
- печать верхнего колонтитула;
- печать нижнего колонтитула;
- печать строки текста;
- завершение процесса печати.

Все синхронизированные операции должны быть процедурами без параметров, и поэтому информация, необходимая этим процедурам, должна храниться в полях потокового класса. Полученный класс приведен в примере 4-4.

Пример 4-4. Объявление класса TPrintThread

```
type
  TPrintThread = class(TThread)
  private
    fText: TStrings;              // поля для поддержки свойств
    fHeader: string;
    fExceptionMessage: string;
    fPrinter: TPrinter;

    PixelsPerInch: Integer;   // локальное хранилище для
    LineHeight: Integer;      // синхронизируемых операций

    YPos: Integer;
    Line: string;
    LeftMargin, TopMargin: Integer;
    PageNumber: Integer;
  protected
    procedure Execute; override;
    procedure PrintText;

    procedure EndPrinting;      // процедуры для Synchronize
    procedure PrintLine;
    procedure PrintHeader;
    procedure PrintFooter;
    procedure StartPrinting;
    procedure StartNewPage;

    property Header: string read fHeader;
    property Printer: TPrinter read fPrinter;
    property Text: TStrings read fText;
  public
    constructor Create(const Text, Header: string;
                       OnTerminate: TNotifyEvent);
    destructor Destroy; override;
    property ExceptionMessage: string read fExceptionMessage;
  end;
```

При возникновении в процессе печати исключительной ситуации, сообщение об ошибке сохраняется в свойстве `ExceptionMessage`. Основной поток может проверить значение этого свойства после завершения `TPrintThread`.

Конструктор `TThread.Create` требует параметр `Boolean`: если он равен истине, поток создается в приостановленном состоянии и находится в нем до тех пор, пока не будет запущен явно. Если аргумент равен `False`, поток запускается сразу. При переопределении конструктора наследуемому конструктору обычно передается в качестве аргумента `True`. После выполнения всех требуемых операций по инициализации в конструкторе следует вызвать метод `Resume`. Создание потока в приостановленном состоянии позволяет избежать «состояния гонок», когда

поток начинает работу до того, как конструктор проинициализирует необходимые поля. В примере 4-5 приведен конструктор TPrintThread.Create.

Пример 4-5. Создание объекта TPrintThread

```
constructor TPrintThread.Create(const Text, Header: string;
                                OnTerminate: TNotifyEvent);
begin
  inherited Create(True);
  fHeader := Header;

  // Сохранение текста в виде строк для упрощения
  // процесса печати.
  fText := TStringList.Create;
  fText.Text := Text;

  // Сохранение ссылки на текущий принтер на тот случай,
  // если пользователь станет печатать другой файл на
  // другой принтер.
  fPrinter := Printers.Printer;

  // Сохранение обработчика события «прерывание».
  Self.OnTerminate := OnTerminate;

  // Поток освобождается по завершении его работы.
  FreeOnTerminate := True;

  // Запуск потока.
  Resume;
end;
```

Переопределенный метод Execute для выполнения фактического процесса печати вызывает PrintText, но заключает его вызов в обработчик исключительной ситуации. Если в процессе печати возникает исключительная ситуация, сообщение об ошибке сохраняется для использования в основном потоке. Метод Execute приведен в примере 4-6.

Пример 4-6. Метод Execute для TPrintThread

```
// Запуск потока.
procedure TPrintThread.Execute;
begin
  try
    PrintText;
  except
    on Ex: Exception do
      fExceptionMessage := Ex.Message;
  end;
end;
```

Метод `PrintText` управляет основным циклом печати, вызывая при необходимости синхронизированные процедуры. Поток занимается «бухгалтерией», которая не требует синхронизации. Метод `PrintText` приведен в примере 4-7.

Пример 4-7. Основной цикл печати в методе PrintText

```
// Печать всего текста с использованием шрифта по умолчанию.
procedure TPrintThread.PrintText;
const
  Leading = 120;  // 120% номинальной высоты шрифта
var
  I: Integer;
  NewPage: Boolean;
begin
  Synchronize(StartPrinting);
  try
    LeftMargin := PixelsPerInch;
    TopMargin  := PixelsPerInch;
    YPos := TopMargin;
    NewPage := True;
    PageNumber := 1;

    for I := 0 to Text.Count-1 do
    begin
      if NewPage then
      begin
        Synchronize(PrintHeader);
        Synchronize(PrintFooter);
        NewPage := False;
      end;

      // Печать текущей строки.
      Line := Text[I];
      Synchronize(PrintLine);
      YPos := YPos + LineHeight * Leading div 100;

      // Дошел ли принтер до конца страницы?
      if YPos > Printer.PageHeight - TopMargin then
      begin
        if Terminated then
          Printer.Abort;
        if Printer.Aborted then
          Break;
        Synchronize(StartNewPage);
        YPos := TopMargin;
        NewPage := True;
        PageNumber := PageNumber + 1;
      end;
    end;
  finally
```

```
      Synchronize(EndPrinting);
    end;
  end;
```

Кажется, что PrintText тратит очень много времени в синхронизируемых методах, и это на самом деле так, но все-таки часть времени выполняется и собственный поток, позволяя Windows нормально распределить время между основным потоком и потоком печати. Все синхронизированные методы – небольшие и простые, насколько это возможно, чтобы минимизировать время, передаваемое основному потоку. В примере 4-8 приведены образцы таких методов: PrintLine, StartNewPage и StartPrinting. Остальные методы аналогичны.

Пример 4-8. Синхронизированные процедуры печати

```
// Сохранение разрешения принтера для того, чтобы поток печати
// мог использовать эту информацию без обращения к объекту Printer.
procedure TPrintThread.StartPrinting;
begin
  Printer.BeginDoc;
  PixelsPerInch := Printer.Canvas.Font.PixelsPerInch;
end;

// Печать текущей строки текста и определение высоты строки, чтобы
// рабочий поток мог изменить Y-координату на странице.
procedure TPrintThread.PrintLine;
begin
  Printer.Canvas.TextOut(LeftMargin, YPos, Line);
  LineHeight := Printer.Canvas.TextHeight(Line);
end;

// Начало печати новой страницы. При вызове сбрасывается Y-координата.
procedure TPrintThread.StartNewPage;
begin
  Printer.NewPage;
end;
```

Для начала процесса печати нужно просто создать экземпляр класса TPrintThread. В качестве последнего аргумента передается обработчик события, который вызывается потоком в момент завершения его работы. Приложение может показать информацию в строке состояния или не дать пользователю выйти из приложения до завершения процесса печати. Один из возможных вариантов показан в примере 4-9.

Пример 4-9. Использование класса TPrintThread

```
// Пользователь выбрал пункт «Печать» из меню, печатаем файл.
procedure TMDIChild.Print;
begin
  if PrintThread <> nil then
    MessageDlg('Файл уже печатается.', mtWarning, [mbOK], 0)
  else
```

```
    begin
      PrintThread := TPrintThread.Create(Editor.Text, FileName,
                                         DonePrinting);
      MainForm.SetPrinting(True);
    end;
  end;

  // Когда файл напечатан, проверяем на наличие ошибок и
  // очищаем состояние печати.
  procedure TMDIChild.DonePrinting(Sender: TObject);
  begin
    if PrintThread.ExceptionMessage <> '' then
      MessageDlg(PrintThread.ExceptionMessage, mtError, [mbOK], 0);
    PrintThread := nil;
    MainForm.SetPrinting(False);
  end;
```

Функции BeginThread и EndThread

Функции BeginThread и EndThread позволяют избежать создания класса. Они являются «оберткой» для функций CreateThread и ExitThread из Win32 API, но необходимо использовать именно функции Delphi, а не вызывать Win32 API напрямую. В Delphi имеется глобальная переменная-флаг IsMultThread, которая устанавливается в True при вызове BeginThread или запуске потока при помощи TThread. Delphi использует этот флаг для правильного выделения памяти в многопоточном приложении. При вызове CreateThread напрямую следует установить значение IsMultiThread равным истине.

Следует иметь в виду, что при работе с BeginThread и EndThread невозможно воспользоваться преимуществом метода Synchronize. Применяя эти функции, программист должен самостоятельно обеспечивать последовательный доступ к VCL.

Функция BeginThread практически аналогична функции CreateThread, за исключением того, что использует параметры с собственными типами Delphi. Функция потока принимает параметр типа «указатель» и возвращает целое значение, которое является кодом возврата потока. Функция EndThread аналогична функции ExitThread Windows: она прерывает текущий поток. Подробно об этих функциях читайте в главе 5, «Справочник по языку». Пример использования BeginThread приведен в разделе «Фьючерсы» в конце этой главы.

Локальные данные потока

В Windows есть средство, позволяющее каждому потоку иметь собственную локальную память ограниченного объема (thread local storage). Delphi облегчает использование этого средства, давая воз-

можность не заботиться об ограничениях, накладываемых Windows. Просто переменная объявляется с помощью ключевого слова `threadvar` вместо `var`. Обычно Delphi создает один экземпляр переменной на уровне модуля и делит этот экземпляр между всеми потоками. Если же используется `threadvar`, Delphi создает уникальные экземпляры переменной для каждого потока.

Потоковая переменная (`threadvar`) должна быть объявлена на уровне модуля, а не внутри функции или процедуры. Каждый поток имеет собственный стек, и локальные переменные в любом случае являются локальными для потока. Поскольку потоковые переменные локальны для потока, и каждый поток монопольно пользуется своими экземплярами переменных, нет необходимости применять критические секции для их защиты.

Поля класса `TThread` следует использовать для локальных переменных потока, т. к. они требуют меньше накладных расходов, чем потоковые переменные. Получить значения локальных переменных потока вне потокового объекта или при вызове функции `BeginThread` можно с помощью потоковых переменных.

Применять переменные `threadvar` в DLL следует с осторожностью. Когда DLL выгружается, Delphi освобождает все потоковые переменные до вызова `DllProc` и разделов завершения библиотеки.

Процессы

В Delphi имеется поддержка многопоточных приложений, но если требуется создать систему взаимодействующих между собой программ, приходится прибегать к помощи Windows API. Каждый процесс работает в собственном адресном пространстве, но есть несколько вариантов обеспечения взаимодействия между процессами:

Сообщения

Любой поток может послать сообщение (message) другому потоку в том же или в другом процессе. Типичный вариант обмена сообщениями между процессами имеет место, когда одно приложение управляет пользовательским интерфейсом другого приложения.

События

Событие (event) – это триггер, который один поток посылает другому. Потоки могут принадлежать одному или разным процессам. Один поток генерирует событие и пробуждает другой поток, находившийся в ожидании этого события. Несколько потоков могут ожидать одного события, и программист может определить, будет ли его сообщение запускать только один поток или все потоки, его ожидающие.

Мьютексы

Мьютекс (mutex, *сокр.* от mutual exlusion (взаимные исключения)) – это критическая секция, которая может быть общей для нескольких процессов.

Семафоры

Семафоры (semaphore) позволяют разделять счетчик между несколькими процессами. Поток в процессе ожидает семафор, и когда семафор становится доступным, уменьшает значение счетчика. Когда счетчик доходит до нуля, поток должен ждать увеличения значения счетчика. Поток может освободить семафор и тем самым увеличить счетчик. В то время как мьютекс позволяет обратиться к общему ресурсу одновременно только одному потоку, семафор дает доступ сразу нескольким потокам, и можно контролировать количество потоков, установив максимальное значение счетчика.

Конвейеры

Конвейер (pipe) – это файл специального вида, содержимое которого рассматривается как очередь. Один процесс записывает данные с одного конца очереди, другой читает с другого конца. Конвейеры – это простой и мощный способ отсылки потока информации от одного процесса другому, как в одной системе, так и в сети.

Файлы, отображаемые в память

Самый распространенный способ разделения данных между процессами – это использование файлов, отображаемых в память (memory-mapped file). Как следует из названия, файл, отображаемый в память, – это файл, содержимое которого отображается в виртуальное адресное пространство процесса. После отображения файла эта область памяти функционирует так же, как и обычная память (кроме того, что все изменения в памяти сохраняются в файле, и все изменения в файле видны в памяти процесса). Несколько процессов могут отобразить в свою память один и тот же файл и таким образом разделить общие данные. Следует иметь в виду, что каждый процесс отображает файл в разные области своего индивидуального адресного пространства, и поэтому хранить в общей памяти данные можно, а указатели – нельзя.

Эти темы описаны во многих книгах по программированию Windows, но вам придется переводить на Delphi примеры, написанные на C и C++. В этом разделе в помощь читателю приводится пример, который использует многие из перечисленных средств.

Предположим, программист пишет текстовый редактор, позволяющий редактировать несколько файлов. При запуске программы (текстового редактора) сначала выполняется проверка, не запущена ли уже эта программа, и если да, то ей передается управление. При этом должны быть переданы аргументы командной строки, чтобы существующий процесс мог открыть запрошенные файлы. Если по какой-ли-

бо причине существующий процесс реагирует слишком медленно, пользователь может запустить программу многократно, каждый раз добавляя новый запрос к существующему процессу. Эта проблема явно требует надежной системы взаимодействия между процессами.

Одно и то же приложение работает и как клиент, и как сервер. Первая запущенная программа становится сервером. Если сервер уже существует, все последующие запуски программы становятся клиентами, и потому структура приложения должна обеспечивать простой и ясный способ определения, кто будет сервером. Для этого программа использует мьютекс.

Мьютекс (*mutex*) – это критическая секция, разделяемая между процессами. Программа всегда пытается создать мьютекс с определенным именем. Первый процесс успешно становится сервером. Если мьютекс уже существует, процесс становится клиентом. При возникновении ошибки программа создает исключительную ситуацию, что видно в примере 4-10.

Пример 4-10. Создание мьютекса для определения того, какой процесс будет сервером

```
const
  MutexName = 'Tempest Software.Threaditor mutex';
var
  SharedMutex: THandle;

// Создание мьютекса, общего для всех процессов. Первый процесс,
// создавший мьютекс – это сервер. Возвращает истину, если
// процесс становится сервером, ложь – если становится клиентом.
// Сервер сразу становится владельцем мьютекса, и вы
// должны освободить его для того, чтобы другой клиент смог
// завладеть им.
function CreateSharedMutex: Boolean;
begin
  SharedMutex := CreateMutex(nil, True, MutexName);
  Win32Check(SharedMutex <> 0);
  Result := GetLastError <> Error_Already_Exists;
end;
```

Мьютекс также защищает файл, отображаемый в память, который клиенты используют для отправки списка файлов серверу. Применению файла, отображаемого в память, должны предшествовать две операции.

1. *Создание файла.* Отображаемый файл может быть либо файлом на диске, либо находиться в системном файле подкачки.
2. *Отображение файла в адресное пространство процесса.* Возможно отображение либо всего файла, либо некоторой его непрерывной части.

После отображения файла в память процесс получает указатель на общую область памяти. Каждому процессу, отображающему один и тот же файл, отведена отдельная область памяти, и все изменения, которые делает один из процессов, сразу становятся видимы другим процессам. Разделяемый файл может иметь в каждом процессе свой стартовый адрес, и потому нельзя хранить в нем указатели (в том числе длинные строки, «широкие» строки, динамические массивы и сложные вариантные записи).

Программа хранит и извлекает данные с помощью записи TSharedData. Заметьте, что в TSharedData используются короткие строки, во избежание проблем с хранением указателей в разделяемом файле. Пример 4-11 показывает, как создается и отображается в память общий файл.

Пример 4-11. Создание файла, отображаемого в память

```
const
  MaxFileSize = 32768; // 32K - это более, чем достаточно.
  MaxFileCount = MaxFileSize div SizeOf(ShortString);
  SharedFileName = 'Tempest Software.Threaditor shared file';
type
  PSharedData = ^TSharedData;
  TSharedData = record
    Count: Word;
    FileNames: array[1..MaxFileCount] of ShortString;
  end;
var
  IsServer: Boolean;
  SharedFile: THandle;
  SharedData: PSharedData;

// Создание разделяемого, отображаемого в память файла и отображение
// всего его содержимого в адресное пространство процесса. Сохранение
// указателя в SharedData.
procedure CreateSharedFile;
begin
  SharedFile := CreateFileMapping($FFFFFFFF, nil, Page_ReadWrite,
                  0, SizeOf(TSharedData), SharedFileName);
  Win32Check(SharedFile <> 0);

  // Отображение всего файла в адресное пространство процесса.
  SharedData := MapViewOfFile(SharedFile, File_Map_All_Access, 0, 0, 0);
  Win32Check(SharedData <> nil);

  // Сервер создает общие данные, поэтому они должны быть
  // корректно проинициализированы. Не нужно очищать все,
  // но обязательно установите счетчик в ноль.
  if IsServer then
    SharedData.Count := 0;
end;
```

Клиент записывает аргументы своей командной строки в файл, отображаемый в память, и сообщает серверу о необходимости прочесть оттуда имена файлов. Запись в файл могут попытаться выполнить одновременно несколько клиентов, или клиент может попробовать записать данные в файл в то время, когда сервер хочет его прочитать. Мьютекс защищает целостность разделяемых данных.

Клиент вызывает процедуру EnterMutex, которая ожидает момента доступности мьютекса и затем захватывает его. Когда клиент владеет мьютексом, ни один другой поток любого процесса не может его захватить. Клиент может записывать в разделяемую память, не боясь нарушения целостности данных. Он копирует аргументы командной строки в область разделяемых данных и затем освобождает мьютекс, вызвав LeaveMutex. В примере 4-12 приведены эти функции.

Пример 4-12. Вход и выход из мьютекса

```
// Вход в критическую секцию путем захвата мьютекса.
// Эта процедура ожидает возможности получить мьютекс
// или вызывает исключительную ситуацию.
procedure EnterMutex;
resourcestring
  sNotResponding = 'Серверный процесс Threaditor не отвечает';
  sNoServer = 'Серверный процесс Threaditor, видимо, закрыт';
const
  TimeOut = 10000; // 10 seconds
begin
  case WaitForSingleObject(SharedMutex, TimeOut) of
  Wait_Timeout:
    raise Exception.Create(sNotResponding);
  Wait_Abandoned:
    raise Exception.Create(sNoServer);
  Wait_Failed:
    RaiseLastWin32Error;
  end;
end;

// Выход из критической секции путем освобождения мьютекса, чтобы
// его мог захватить другой процесс.
procedure LeaveMutex;
begin
  Win32Check(ReleaseMutex(SharedMutex));
end;
```

Клиент пробуждает сервер, генерируя событие. Событие – это способ для одного процесса послать уведомление другому процессу, не передавая какой-либо дополнительной информации. Сервер ожидает событие, а клиент, осуществив копирование списка файлов в разделяемый файл, генерирует событие, которое пробуждает сервер. В примере 4-13 показан код для создания события.

Пример 4-13. Создание разделяемого события

```
const
  EventName = 'Tempest Software.Threaditor event';
var
  SharedEvent: THandle;

// Создание события, которое клиенты используют для подачи
// сигнала серверу.
procedure CreateSharedEvent;
begin
  SharedEvent := CreateEvent(nil, False, False, EventName);
  Win32Check(SharedEvent <> 0);
end;
```

Сервер захватывает мьютекс для того, чтобы скопировать имена файлов из разделяемого файла. Чтобы избежать слишком долгого периода захвата мьютекса, сервер копирует имена файлов в список строк и сразу же освобождает мьютекс. Затем он открывает каждый из файлов, перечисленных в списке строк. Сетевые задержки и другие проблемы могут замедлить сервер при открытии файлов, и потому он только копирует имена файлов и как можно быстрее освобождает мьютекс.

Каждый когда-то встречался с проблемой бесконечного ожидания. Пользователь нажимает на кнопку и ждет, но курсор в виде песочных часов так и не исчезает. Что-то внутри программы ожидает события, которое так и не происходит. И что же он тогда делает? Нажимает <Ctrl>+<Alt>+<Delete> и пытается закрыть программу? Но программа, возможно, не сохранила еще свои данные. Может быть, она ожидает отклика от сервера. Если сейчас закрыть программу, все результаты работы могут быть потеряны. Очень грустно глядеть на песочные часы и знать, что ничего нельзя исправить.

Такое могло бы никогда не происходить, и это зависит от программистов.

Windows позволяет потоку ждать сколько угодно долго, если в качестве аргумента функций ожидания, таких как `WaitForSingleObject`, используется `Infinite`. Однако единственный момент, где стоит использовать бесконечное время ожидания, – это когда серверный поток ожидает соединения от клиента. В этом случае неограниченное время ожидания действительно оправдано. В большинстве других случаев необходимо явно указать время ожидания. Определите, как долго пользователь может ждать сообщения до того, как программа сообщит ему, что что-то не так. Если, например, поток оказался заблокирован, пользователь имеет возможности узнать об этом или остановить поток, только прервав процесс в Task Manager. Если указать время ожидания явно, как в примере 4-12, программа может выдать вразумительное сообщение, а не оставлять смущенного пользователя в неведении о том, что происходит.

Сервер открывает каждый из этих файлов, как будто пользователь выбрал пункт меню «Открыть» в текстовом редакторе. Поскольку файл открывается с помощью VCL, сервер использует `Synchronize` для каждого файла. После того как файлы открыты, сервер снова ждет события. Он тратит большую часть времени, ожидая событий от клиента, и ожидание не должно мешать его нормальной работе. Поэтому сервер делает свою работу в отдельном потоке с непрерывным циклом: ожидание событий от клиента, чтение имен файлов из разделяемого файла, открытие файлов и снова ожидание. В примере 4-14 приведен код серверного потока.

Пример 4-14. Класс TServerThread

```
type
  TServerThread = class(TThread)
  private
    fFileName: string;
    fFileNames: TStringList;
  public
    constructor Create;
    destructor Destroy; override;
    procedure OpenFile;
    procedure RestoreWindow;
    procedure Execute; override;
    property FileName: string read fFileName;
    property FileNames: TStringList read fFileNames;
  end;

var
  ServerThread: TServerThread;

{ TServerThread }

constructor TServerThread.Create;
begin
  inherited Create(True);
  fFileNames := TStringList.Create;
  FreeOnTerminate := True;
  Resume;
end;

destructor TServerThread.Destroy;
begin
  FreeAndNil(fFileNames);
  inherited;
end;

procedure TServerThread.Execute;
var
  I: Integer;
begin
```

```
  while not Terminated do
  begin
    // Ждет, пока клиент не разбудит сервер. Это один из тех немногих
    // случаев, когда бесконечное время ожидания – правильное
    // решение.
    WaitForSingleObject(SharedEvent, INFINITE);

    EnterMutex;
    try
      for I := 1 to SharedData.Count do
        FileNames.Add(SharedData.FileNames[I]);
       SharedData.Count := 0;
    finally
      LeaveMutex;
    end;

    for I := 0 to FileNames.Count-1 do
    begin
      fFileName := FileNames[I];
      Synchronize(OpenFile);
    end;
    Synchronize(RestoreWindow);
    FileNames.Clear;
  end;
end;

procedure TServerThread.OpenFile;
begin
  // Создание нового дочернего MDI-окна.
  TMDIChild.Create(Application).OpenFile(FileName);
end;

// Вывод главной формы на передний план и восстановление ее из
// минимизированнного состояния.
procedure TServerThread.RestoreWindow;
begin
  if FileNames.Count > 0 then
  begin
    Application.Restore;
    Application.BringToFront;
  end;
end;
```

Клиент очень прост – он захватывает мьютекс и копирует аргументы своей командной строки в разделяемый файл. Если несколько клиентов пытаются запуститься одновременно, первый захвативший мьютекс выполняет свою задачу, а другие должны подождать. До того как сервер откроет файлы, может быть запущено несколько клиентов, и потому каждый из клиентов дописывает свои имена файлов в конец списка, как показано в примере 4-15.

Пример 4-15. Клиент Threaditor

```
// Клиент захватывает мьютекс и дописывает файлы, перечисленные в
// командной строке, в разделяемый файл. После этого он сообщает
// серверу о том, что файлы готовы.
procedure SendFilesToServer;
var
  I: Integer;
begin
  if ParamCount > 0 then
  begin
    EnterMutex;
    try
      for I := 1 to ParamCount do
        SharedData.FileNames[SharedData.Count + I] := ParamStr(I);
      SharedData.Count := SharedData.Count + ParamCount;
    finally
      LeaveMutex;
    end;
    // Wake up the server
    Win32Check(SetEvent(SharedEvent));
  end;
end;
```

При запуске приложение вызывает процедуру `StartServerOrClient`**. Эта процедура создает или открывает мьютекс и таким образом узнает о том, является программа сервером или клиентом. Если это сервер, запускается серверный поток и** `IsServer` **устанавливается в** `True`**. Сервер в начале является владельцем общего мьютекса и может свободно создать и проинициализировать разделяемый файл. Затем он должен освободить мьютекс. Если программа является клиентом, она посылает аргументы командной строки серверу и устанавливает** `IsServer` **в** `False`**. Клиент не владеет мьютексом в начале своей работы, и ему не нужно ничего освобождать. В примере 4-16 приведена процедура** `StartServerOrClient`**.**

Пример 4-16. Запуск приложения в качестве сервера или клиента

```
procedure StartServerOrClient;
begin
  IsServer := CreateSharedMutex;
  try
    CreateSharedFile;
    CreateSharedEvent;
  finally
    if IsServer then
      LeaveMutex;
  end;

  if IsServer then
    StartServer
  else
```

```
    SendFilesToServer;
end;
```

Для запуска сервера надо просто создать экземпляр TServerThread. Чтобы остановить сервер, установите флаг Terminated в True и создайте событие для пробуждения потока. Поток запускается, проходит основной цикл и прерывает работу, так как флаг Terminated равен истине. Чтобы убедиться, что поток завершился корректно, основной поток ожидает окончания работы серверного, но это не занимает много времени. Если что-то не так, основной поток просто игнорирует серверный таким образом, чтобы приложение могло закрыться. Windows уничтожает поток при завершении программы. В примере 4-17 показаны процедуры StartServer и StopServer.

Пример 4-17. Запуск и остановка серверного потока

```
// Создание серверного потока, ожидающего клиентских
// соединений.
procedure StartServer;
begin
  ServerThread := TServerThread.Create;
end;

procedure StopServer;
begin
  ServerThread.Terminate;
  // Wake up the server so it can die cleanly.
  Win32Check(SetEvent(SharedEvent));
  // Ожидание завершения серверного потока, но если он быстро не
  // завершается, не беспокойтесь и дайте Windows самому его
  // уничтожить.
  WaitForSingleObject(ServerThread.Handle, 1000);
end;
```

Для окончательного наведения порядка программа должна снять отображение файла, закрыть мьютекс и закрыть событие. Windows сохраняет разделяемые дескрипторы до тех пор, пока они открыты хотя бы одним потоком. Как только последний поток закрывает дескриптор, Windows избавляется от мьютекса, события и любого другого объекта. Когда приложение закрывается, Windows закрывает все открытые дескрипторы, но всегда полезно сделать это явно. Это помогает пользователю, который читает и обслуживает ваш код, понять, какие дескрипторы должны быть открыты на момент окончания работы программы. Если произошла серьезная ошибка Windows, приложение может прерваться до того, как будут созданы все разделяемые объекты, и код завершения проверяет корректность каждого объекта перед его закрытием.

Пример 4-18. Закрытие всех разделяемых объектов при завершении модуля

```
finalization
  if SharedMutex <> 0 then
```

```
    CloseHandle(SharedMutex);
  if SharedEvent <> 0 then
    CloseHandle(SharedEvent);
  if SharedData <> nil then
    UnmapViewOfFile(SharedData);
  if SharedFile <> 0 then
    CloseHandle(SharedFile);
end.
```

Последний этап – это редактирование исходного файла проекта. Первое, что делает программа, – это вызывает процедуру `StartServerOrClient`. Если программа является клиентом, она завершается без запуска обычного приложения Delphi. Если это сервер, приложение запускается нормальным образом (и запускает в фоновом режиме серверный поток). В примере 4-19 приводится исходный файл нового проекта.

Пример 4-19. Исходный файл проекта для запуска сервера или клиента

```
program Threaditor;

uses
  Forms,
  Main in 'Main.pas' {MainForm},
  Childwin in 'ChildWin.pas' {MDIChild},
  About in 'About.pas' {AboutBox},
  Process in 'Process.pas';

{$R *.RES}

begin
  StartServerOrClient;
  if IsServer then
  begin
    Application.Initialize;
    Application.Title := 'Threaditor';
    Application.CreateForm(TMainForm, MainForm);
    Application.Run;
  end;
end.
```

Оставшаяся часть приложения – это стандартный MDI-проект из репозитария Delphi с небольшими модификациями. Все, что вам потребуется сделать, – это добавить модуль `Process` и немного изменить исходный файл проекта. Разным приложениям могут потребоваться разные действия при работе клиентов, но этот пример обеспечит вам хорошую базу для самостоятельной работы.

Фьючерсы

Создание параллельного приложения может оказаться сложнее, чем последовательно выполняемого. Приходится думать о «состоянии гонки», синхронизации, разделяемых переменных и прочих тонкостях. Фьючерсы (futures) позволяют уменьшить интеллектуальную сложность использования потоков. Фьючерс – это объект, который обещает предоставить значение когда-нибудь в будущем. Приложение делает свою работу в основном потоке и обращается к фьючерсам для осуществления параллельной выборки данных или расчетов. Фьючерс выполняет свою работу в отдельном потоке, и когда главному потоку требуется информация, он получает ее из фьючерсного объекта. Если информация еще не готова, главный поток ожидает окончания работы фьючерса. Программирование с использованием фьючерсов значительно уменьшает сложность многопоточного программирования.

Создайте фьючерсный класс путем наследования от класса `TFuture` и переопределите метод `Compute`. Этот метод делает всю необходимую работу и возвращает результат в виде значения типа `Variant`. Следует попытаться обойтись без синхронизации и доступа к общим переменным из метода `Compute`. Пусть лучше основной поток занимается взаимодействием с другими потоками или фьючерсами. В примере 4-20 приведено объявление класса `TFuture`.

Пример 4-20. Объявление класса TFuture

```
type
  TFuture = class
  private
    fExceptObject: TObject;
    fExceptAddr: Pointer;
    fHandle: THandle;
    fTerminated: Boolean;
    fThreadID: LongWord;
    fTimeOut: DWORD;
    fValue: Variant;
    function GetIsReady: Boolean;
    function GetValue: Variant;
  protected
    procedure RaiseException;
  public
    constructor Create;
    destructor Destroy; override;
    procedure AfterConstruction; override;

    function Compute: Variant; virtual; abstract;
    function HasException: Boolean;
    procedure Terminate;

    property Handle: THandle read fHandle;
    property IsReady: Boolean read GetIsReady;
```

```
    property Terminated: Boolean read fTerminated write fTerminated;
    property ThreadID: LongWord read fThreadID;
    property TimeOut: DWORD read FTimeOut write fTimeOut;
    property Value: Variant read GetValue;
  end;
```

Конструктор инициализирует фьючерсный объект, но воздерживается от запуска потока. Вместо этого `TFuture` переопределяет метод `AfterConstruction` и запускает поток после того, как конструкторы завершат работу. Это позволяет производным классам проинициализировать свои поля до запуска потока.

Приложение читает свойство `Value`, когда ему требуется значение фьючерса. Метод `GetValue` ожидает окончания работы потока. Если поток уже завершился, Windows сразу возвращает управление в `GetValue`. Если поток вызвал исключительную ситуацию, фьючерс вызывает ее снова с исходным адресом ошибки. Это позволяет вызывающему потоку обработать исключительную ситуацию, как будто фьючерс выполнял свою работу в вызывающем потоке, а не в своем собственном. Если все проходит по плану, значение фьючерса возвращается в виде `Variant`. В примере 4-21 приведена реализация класса `TFuture`.

Пример 4-21. Класс TFuture

```
// Все фьючерсы вычисляют свое значение в ThreadFunc. Одновременно
// может работать любое число экземпляров ThreadFunc. Windows
// требует от потока перехвата исключительных ситуаций, иначе
// Windows прервет работу программы. ThreadFunc перехватывает
// исключительные ситуации и сохраняет Exception в объекте Future.
function ThreadFunc(Param: Pointer): Integer;
var
  Future: TFuture;
  RaiseFrame: PRaiseFrame;
begin
  Result := 0;
  Future := TFuture(Param);
  // Поток должен перехватывать все исключительные ситуации внутри
  // потока. Сохраняет объект Exception и адрес в объекте Future,
  // для повторного вызова исключительной ситуации при запросе
  // значения фьючерса.
  try
    Future.fValue := Null;
    Future.fValue := Future.Compute;
  except
    RaiseFrame := RaiseList;
    if RaiseFrame <> nil then
    begin
      Future.fExceptObject := RaiseFrame.ExceptObject;
      Future.fExceptAddr   := RaiseFrame.ExceptAddr;
      RaiseFrame.ExceptObject := nil;
    end;
```

```
  end;
end;

{ TFuture }
// Создание фьючерса и начало его работы.
constructor TFuture.Create;
begin
  inherited;
  // По умолчанию время ожидания бесконечно (Infinite), так как
  // универсальный фьючерс не может знать, сколько времени
  // потребуется конкретному фьючерсу для выполнения своей работы.
  // Производные классы должны установить другое значение,
  // соответствующее конкретной ситуации.
  fTimeOut := Infinite;
end;

// Поток запускается после AfterConstruction для того, чтобы производный
// класс завершил свой конструктор в основном потоке и полностью
// проинициализировал объект TFuture до запуска потока. Это предотвращает
// возникновение «состояния гонок» при инициализации объекта,
// производного от TFuture.
procedure TFuture.AfterConstruction;
begin
  inherited;
  // Запуск фьючерсного потока.
  fHandle := BeginThread(nil, 0, ThreadFunc, Self, 0, fThreadID);
  Win32Check(Handle <> 0);
end;

// Если вызывающий поток разрушает фьючерсный объект до завершения
// работы потока, красивого способа завершить фьючерсный объект нет.
// TerminateThread оставляет распределенный стек и может привести к
// разным неприятностям, особенно если поток находится в процессе
// вызова системной функции ядра. Менее насильственное решение состоит
// в том, чтобы дать потоку завершить свою работу естественным путем.
// Функция Compute должна периодически проверять флаг Terminated и
// завершать работу немедленно, если он равен истине.
destructor TFuture.Destroy;
begin
  if Handle <> 0 then
  begin
    if not IsReady then
    begin
      Terminate; // Tell the thread to stop.
      try
        GetValue;  // Ожидание остановки.
      except
        // Отменяет все исключительные ситуации, возникшие здесь,
        // так как вычисления логически прерываются и фьючерсный
        // объект уничтожается. Вызывающий поток освобождает
        // объект TFuture и не ожидает никаких исключительных
```

```
        // ситуаций.
      end;
    end;
    Win32Check(CloseHandle(Handle));
  end;
  inherited;
end;
// Возвращает истину, если поток завершен.
function TFuture.GetIsReady: Boolean;
begin
  Result := WaitForSingleObject(Handle, 0) = Wait_Object_0;
end;

// Ожидает окончания работы потока и возвращает его значение.
// Если поток вызывает исключительную ситуацию, вызывает ее
// заново, но уже в контексте вызывающего потока.
function TFuture.GetValue: Variant;
resourcestring
  sAbandoned = 'Фьючерсный поток неожиданно прерван';
  sTimeOut   = 'Фьючерсный поток не был завершен до окончания
времени ожидания';
begin
  case WaitForSingleObject(Handle, TimeOut) of
  Wait_Abandoned: raise Exception.Create(sAbandoned);
  Wait_TimeOut:   raise Exception.Create(sTimeOut);
  else
    if HasException then
      RaiseException;
  end;
  Result := fValue;
end;

function TFuture.HasException: Boolean;
begin
  Result := fExceptObject <> nil;
end;

procedure TFuture.RaiseException;
begin
  raise fExceptObject at fExceptAddr;
end;

// Устанавливает флаг Terminate, чтобы функция Compute узнала о
// необходимости завершения.
procedure TFuture.Terminate;
begin
  Terminated := True;
end;
```

Для использования фьючерса следует унаследовать класс от `TFuture` **и переопределить метод** `Compute`**. Нужно создать экземпляр производного**

фьючерсного класса и прочитать свойство Value, когда программе потребуется значение фьючерса.

Предположим, требуется добавить новую возможность в многопоточный текстовый редактор: в фоновом режиме продолжать поиск следующего вхождения нужного текста. Например, пользователь открывает диалоговое окно поиска, вводит искомую строку и нажимает кнопку «Поиск». Редактор находит текст и выделяет его. Пока пользователь просматривает результат, редактор в фоновом режиме ищет следующее вхождение. Это отличная задача для фьючерса. Фьючерс ищет следующее вхождение и возвращает его начальную позицию. При очередном нажатии кнопки «Поиск» редактор получает данные от фьючерса. Если фьючерс еще не завершил работу, редактор ожидает. Если завершил – фьючерс сразу возвращает управление, и, к радости пользователя, поиск даже в большом файле происходит быстро.

Во фьючерсе находится копия содержимого файла, так что не приходится беспокоиться об одновременном доступе к файлу нескольких потоков. Возможно, это не лучшая архитектура для текстового редактора, однако она демонстрирует эффективное использование фьючерса. В примере 4-22 приводится класс TSearchFuture, выполняющий поиск.

Пример 4-22. Поиск текста

```
type
  TSearchFuture = class(TFuture)
  private
    fEditorText: string;
    fFindPos: LongInt;
    fFindText: string;
    fOptions: TFindOptions;
    procedure FindDown(out Result: Variant);
    procedure FindUp(out Result: Variant);
  public
    constructor Create(Editor: TRichEdit; Options: TFindOptions;
      const Text: string);
    function Compute: Variant; override;

    property EditorText: string read fEditorText;
    property FindPos: LongInt read FFindPos write fFindPos;
    property FindText: string read fFindText;
    property Options: TFindOptions read fOptions;
  end;

{ TSearchFuture }

constructor TSearchFuture.Create(Editor: TRichEdit;
  Options: TFindOptions; const Text: string);
begin
  inherited Create;
  TimeOut := 30000; // Ожидаем от фьючерса завершения его работы
                    // не более, чем за 30 секунд.
```

```
  // Сохранение основных параметров поиска.
  fEditorText := Editor.Text;
  fOptions := Options;
  fFindText := Text;
  // Начало поиска с позиции окончания текущего выделения,
  // чтобы избежать нахождения одного и того же места в тексте.
  fFindPos := Editor.SelStart + Editor.SelLength;
end;

// Простейший поиск.
function TSearchFuture.Compute: Variant;
begin
  if not (frMatchCase in Options) then
  begin
    fFindText := AnsiLowerCase(FindText);
    fEditorText := AnsiLowerCase(EditorText);
  end;

  if frDown in Options then
    FindDown(Result)
  else
    FindUp(Result);
end;

procedure TSearchFuture.FindDown(out Result: Variant);
var
  Next: PChar;
begin
  // Поиск следующего вхождения.
  Next := AnsiStrPos(PChar(EditorText) + FindPos, PChar(FindText));
  if Next = nil then
    // Не найдено.
    Result := -1
  else
  begin
    // Найдено, возвращается начальное положение искомого текста.
    FindPos := Next - PChar(EditorText);
    Result := FindPos;
  end;
end;
```

Если пользователь редактирует файл, изменяет область выделенного текста или каким-либо другим образом делает результаты поиска неактуальными, редактор освобождает фьючерс, поскольку тот уже не может предоставить корректные результаты. Когда в следующий раз пользователь начинает поиск, редактор должен запустить новый фьючерс. В примере 4-23 приведен подходящий метод для формы `MDIChild`.

Пример 4-23. Управление фьючерсом для редактора

```
procedure TMDIChild.EditorSelectionChange(Sender: TObject);
begin
  if Future <> nil then
    RestartSearch;
end;

// Когда пользователь закрывает окно диалога, фоновый поток
// прерывается, так как он, скорее всего, больше не потребуется.
procedure TMDIChild.FindDialogClose(Sender: TObject);
begin
  FreeAndNil(fFuture);
end;

// Перезапуск поискового потока после изменения параметров поиска.
procedure TMDIChild.RestartSearch;
begin
  FreeAndNil(fFuture);
  fFuture := TSearchFuture.Create(Editor, FindDialog.Options,
                                  FindDialog.FindText);
end;
```

При нажатии пользователем кнопки «Поиск» объект `TFindDialog` **вызывает свой обработчик события** `OnFind`**. Обработчик события сначала проверяет, нет ли корректного фьючерса, и, если его нет, запускает новый. Таким образом, весь поиск выполняется в фоновом потоке. Объект** `TSearchFuture` **возвращает позицию следующего вхождения или −1, если текст не найден. Большая часть действий обработчика состоит в прокрутке окна редактора, чтобы выбранный текст попал в видимую область. Собственно управление фьючерсом выполняется сравнительно просто, как видно из примера 4-24.**

Пример 4-24. Выполнение поиска текста

```
// Пользователь нажал кнопку «Поиск» в диалоговом окне.
// Получение следующего вхождения, которое, возможно, уже
// найдено поисковым фьючерсом.
procedure TMDIChild.FindDialogFind(Sender: TObject);
var
  FindPos: LongInt;
  SaveEvent: TNotifyEvent;
  TopLeft, BottomRight: LongWord;
  Top, Left, Bottom, Right: LongInt;
  Pos: TPoint;
  SelLine, SelChar: LongInt;
  ScrollLine, ScrollChar: LongInt;
begin
  // Если поиск еще не начат или если пользователь изменил
  // параметры поиска, перезапускаем поиск. Иначе фьючерс, видимо,
  // уже нашел следующее вхождение. В любом случае вызов Future.Value
  // приводит к ожиданию завершения поиска.
```

```
if (Future = nil) or
   (Future.FindText <> FindDialog.FindText) or
   (Future.Options <> FindDialog.Options)
then
  RestartSearch;

FindPos := Future.Value;

if FindPos < 0 then
  MessageBeep(Mb_IconWarning)
else
begin
  // Возвращение фокуса окну редактора от диалогового окна поиска.
  Application.MainForm.SetFocus;
  // Временное отключение возможности изменения выделения
  // текста, чтобы не дать возможность перезапуститься до тех
  // пор, пока не будут установлены и начало и длина выделенного
  // фрагмента.
  SaveEvent := Editor.OnSelectionChange;
  try
    Editor.OnSelectionChange := nil;
    Editor.SelStart := FindPos;
    Editor.SelLength := Length(FindDialog.FindText);
  finally
    Editor.OnSelectionChange := SaveEvent;
  end;

  // Начало поиска следующего вхождения.
  RestartSearch;

  // Прокрутка окна редактора для вывода на экран выделенного текста.
  // Начинается с получения индекса столбца и строки верхнего левого
  // угла управляющего элемента RichText.
  Pos.X := 0;
  Pos.Y := 0;
  TopLeft := Editor.Perform(Em_CharFromPos, 0, LParam(@Pos));
  Top := Editor.Perform(Em_LineFromChar, Word(TopLeft), 0);
  Left := Word(TopLeft) - Editor.Perform(Em_LineIndex, Top, 0);

  // Затем получаем строку и столбец нижнего правого угла.
  Pos.X := Editor.ClientWidth;
  Pos.Y := Editor.ClientHeight;
  BottomRight := Editor.Perform(Em_CharFromPos, 0, LParam(@Pos));
  Bottom := Editor.Perform(Em_LineFromChar, Word(BottomRight), 0);
  Right := Word(BottomRight) -
           Editor.Perform(Em_LineIndex, Bottom, 0);

  // Начало выделенного текста видимо?
  // Если строка не видна, выполняем вертикальную прокрутку.
  SelLine := Editor.Perform(Em_ExLineFromChar, 0, FindPos);
  if (SelLine < Top) or (SelLine > Bottom) then
```

```
        ScrollLine := SelLine - Top
      else
        ScrollLine := 0;
      // Если столбец не виден, выполняем горизонтальную прокрутку.
      SelChar := FindPos - Editor.Perform(Em_LineIndex, SelLine, 0);
      if (SelChar < Left) or (SelChar > Right) then
        ScrollChar := SelChar - Left
      else
        ScrollChar := 0;
        Editor.Perform(Em_LineScroll, ScrollChar, ScrollLine);
    end;
  end;
```

Основное преимущество фьючерсов – в их простоте. Часто можно реализовать метод `Compute` в виде простой линейной подпрограммы (которая, однако, должна периодически проверять значение поля `Terminated`). Использовать фьючерс не сложнее, чем свойство. Вся синхронизация выполняется в `TFuture` автоматически. Параллельное программирование может быть сложным, но если подойти к нему внимательно и осторожно, можно создавать приложения, использующие потоки и процессы, корректно, быстро и эффективно.

Глава 5

Справочник по языку

Справочник по языку – большая глава. Здесь вы найдете все ключевые слова, директивы, функции, процедуры, переменные, классы, методы и свойства, являющиеся частью языка Delphi Pascal. Большинство этих элементов объявлены в модуле `System`, но часть из них описана в `SysInit`. Оба модуля автоматически включаются в каждый модуль Delphi. Помните, что Delphi Pascal нечувствителен к регистру символов, за единственным исключением – процедура `Register` (для совместимости с C++ Builder). Для удобства читателя в этой главе после номеров ошибок времени выполнения указываются соответствующие имена классов исключительных ситуаций. Модуль `SysUtils` преобразует ошибки в исключительные ситуации. Исключительные ситуации, строго говоря, не являются частью языка Delphi, но модуль `SysUtils` используется почти в каждом проекте Delphi, и поэтому программистам, работающим на Delphi, классы исключительных ситуаций знакомы больше, чем номера ошибок.

Каждый из элементов языка относится к одной из приведенных ниже категорий:

Директива

Директива – это идентификатор, который имеет для компилятора специальное значение, но только в определенном контексте. Вне этого контекста имя директивы можно применять как обычный идентификатор. Редактор исходных текстов Delphi пытается помочь программисту, выделяя директивы полужирным шрифтом, когда они используются в собственном контексте, и простым шрифтом – когда они являются обычными идентификаторами. Однако редактор не всегда оказывается прав, т. к. некоторые языковые правила, принятые для директив, сложнее, чем он может «понять».

Функция

Не все функции действительно являются функциями; некоторые из них встроены в компилятор. Разница обычно значения не имеет, поскольку встроенные функции выглядят и работают так же, как обычные, за тем исключением, что для них нельзя получить адрес. В приведенных ниже описаниях функций указано, какие из них являются встроенными, а какие – обычными.

Интерфейс

Описание стандартного интерфейса.

Ключевое слово

Ключевое слово – это зарезервированный идентификатор, значение которого определяется компилятором Delphi. Ключевое слово нельзя использовать в качестве переменной, метода или имени типа.

Процедура

Как и функции, некоторые процедуры встроены в компилятор, не являются обычными процедурами, и их адрес определить нельзя. Некоторые процедуры (например, `Exit`) ведут себя так, как будто являются языковыми конструкциями, но они не являются зарезервированными ключевыми словами, и вы можете использовать их так же, как и любые другие процедуры.

Тип

Что такое тип, читатель уже знает. Некоторые типы встроены в компилятор, а некоторые явно описаны в модуле `System`.

Переменная

Большинство переменных языка Delphi являются обычными переменными модулей `System` и `SysInit`. Разница между этими модулями состоит в том, что переменные модуля `System` разделяются между всеми пакетами, загруженными приложением, а для переменных модуля `SysInit` создается собственная копия для каждого пакета. Значения этих переменных можно изменить, но это требует хорошего понимания их особенностей. Если не соблюдать осторожность, можно нарушить нормальную работу Delphi. Другие переменные (`Self` и `Result`) встроены в компилятор и предназначены для специальных целей.

Abs, функция

Синтаксис

```
function Abs(Number: Числовой тип): Числовой тип;
```

Описание

Функция Abs вычисляет абсолютное значение. Функция встроена в компилятор.

Возвращаемое значение

- Если аргумент относится к целому типу, Abs проверяет, не является ли его значение отрицательным, и, если является, вычисляет противоположное ему значение. Тип возвращаемого значения – Integer или Int64 в зависимости от типа аргумента.
- Для вещественных аргументов Abs очищает знаковый бит без изменения всех остальных. Другими словами, минус ноль и минус бесконечность становятся плюс нулем и плюс бесконечностью. Даже если аргумент равен NaN, результат равен исходному значению, знаковый бит которого равнен нулю.

Аргумент	Возвращаемое значение
–бесконечность	+бесконечность
< 0	–number
–0.0	+0.0
+0.0	+0.0
> 0.0	number
+бесконечность	+бесконечность
«спокойный» NaN	исходное число со знаковым битом, равным 0
сигнализирующий NaN	исходное значение со знаковым битом, равным нулю

- Если аргумент относится к типу Variant, Delphi преобразует его к вещественному типу и затем вычисляет абсолютное значение, возвращая результат вещественного типа (даже если значение вариантного аргумента было целым).

Смотрите также

Типы Double, Extended, Int64, Integer, Single.

Absolute, директива

Синтаксис

```
var Описание absolute Константное выражение;
var Описание absolute Переменная;
```

Описание

Директива absolute указывает Delphi, что переменная должна храниться по определенному адресу в памяти. Адрес может быть либо числовым значением, либо именем переменной. В последнем случае область памяти будет той же, что и для Переменной. Директиву absolute можно применять с локальными или глобальными переменными.

Советы и приемы

- Не используйте директиву absolute, если не уверены, что это действительно нужно. Вместо этого обычно можно использовать вариантные записи, которые лучше защищены от ошибок и проще в чтении и понимании.
- Задание числового значения адреса в директиве absolute – это устаревший прием Delphi 1, который не имеет смысла в новых 32-разрядных операционных системах Windows.

Пример

Смотрите описание типа Extended, где приведен пример использования директивы absolute.

Смотрите также

Ключевые слова Record, Var.

Abstract, директива

Синтаксис

```
Описание виртуального метода; abstract;
```

Описание

Директива abstract применяется для виртуальных или динамических методов и означает, что метод не имеет реализации. Компилятор резервирует для него место в таблице виртуальных методов или присваивает номер динамического метода. Производный класс может предоставить реализацию абстрактного метода. Директива abstract должна следовать за одной из директив virtual, dynamic или override.

Советы и приемы

- Если производный класс не переопределяет абстрактный метод, то можно опустить его описание в объявлении класса или объявить метод с директивами override и abstract (именно в этом порядке). Последнее предпочтительнее, т. к. явно документирует намерение программиста не реализовывать метод и не оставляет читателя в неведении, случайно пропущен метод или намеренно.
- Если в программе создается объект, а компилятор обнаруживает, что в классе имеются абстрактные методы, он генерирует предупреждение, обычно свидетельствующее об одной из двух ошибок:

1) программист забыл реализовать абстрактный метод в производном классе; 2) предпринята попытка создать экземпляр базового класса вместо производного.

- Если разработчик создает экземпляр базового класса и вызывает один из его абстрактных методов, Delphi генерирует вызов процедуры AbstractErrorProc и вызывает ошибку времени выполнения 210 (EAbstractError).

Смотрите также

Переменная AbstractErrorProc, ключевое слово Class, директивы Dynamic и Override.

AbstractErrorProc, переменная

Синтаксис

```
var AbstractErrorProc: Pointer;
procedure ВашаПроцедура;
begin ... end;
AbstractErrorProc := @ВашаПроцедура;
```

Описание

Когда вызывается абстрактный метод, и ссылка на объект указывает на объект базового класса, в котором не реализован метод, Delphi вызывает процедуру, на которую указывает AbstractErrorProc. Если значение этой переменной равно nil, Delphi вызывает ошибку времени выполнения 210 (EAbstractError). Если указатель не равен nil, он должен соответствовать точке входа в процедуру без аргументов. Delphi вызывает эту процедуру для обработки ошибки.

Советы и приемы

Модуль SysUtils устанавливает AbstractErrorProc на процедуру, которая вызывает исключительную ситуацию EAbstractError, и большинству приложений не требуется менять значение этой переменной.

Смотрите также

Директива Abstract, переменные AssertErrorProc, ErrorProc, ExceptProc, процедура Halt

AddModuleUnloadProc, процедура

Синтаксис

```
procedure AddModuleUnloadProc(Proc: TModuleUnloadProc);

procedure ВашаПроцедура(HInstance: THandle);
begin ... end;
AddModuleUnloadProc(ВашаПроцедура);
```

Описание

Delphi ведет список пакетов, из которых состоит приложение. Когда Delphi выгружает пакет, вызывается серия процедур выгрузки, каждой из которых передается дескриптор экземпляра библиотеки DLL. Программист может добавить собственную процедуру в начало этого списка, передав ее адрес процедуре `AddModuleUnloadProc`. При завершении приложения Delphi, в том числе, вызовет и пользовательскую процедуру выгрузки исполняемого модуля.

`AddModuleUnloadProc` – это реальная процедура.

Пример

```
// Графический сервер управляет графическими ресурсами.
// Когда приложение загружает графический ресурс, сервер
// проверяет глубину цвета для ресурса и, если она больше текущей
// глубины цвета дисплея, делает копию графического объекта
// с глубиной цвета дисплея и возвращает ссылку на новый объект.
// Использование высококачественного алгоритма формирования
// изображения дает лучшие результаты, чем встроенный
// алгоритм подбора цветов Windows.
//
// При выгрузке модуля, все его ресурсы освобождаются.

type
  PResource = ^TResource;
  TResource = record
    Module: THandle;
    Resource: TGraphicsObject;
  case Boolean of
    True:  (Name: PChar;);
    False: (ID: LongInt;);
  end;
var
  List: TList;

procedure ByeBye(HInstance: THandle);
var
  I: Integer;
  Resource: PResource;
begin
  for I := List.Count-1 downto 0 do
  begin
    Resource := List[I];
    if Resource.Module = HInstance then
    begin
      List.Delete(I);
      Resource.Resource.Free;
      Dispose(Resource);
    end;
  end;
```

```
end;

initialization
  List := TList.Create;
  AddModuleUnloadProc(ByeBye);
finalization
  RemoveModuleUnloadProc(ByeBye);
  FreeAndNil(List);
end.
```

Смотрите также

Переменная ModuleUnloadList, типы PModuleUnloadRec, TModuleUnloadRec, процедуры RemoveModuleUnloadProc, UnregisterModule.

Addr, функция

Синтаксис

```
function Addr(var X): Pointer;

Addr(Переменная)
Addr(Подпрограмма)
```

Описание

Функция Addr возвращает адрес переменной или подпрограммы. Возвращается значение типа Pointer — нетипизированного указателя. Даже если используется директива $T или $TypedAddress, эта функция все равно возвращает нетипизированный указатель. Оператор @ аналогичен функции Addr, однако @ при использовании директивы $T или $TypedAddress возвращает типизированный указатель.

Функция Addr встроена в компилятор.

Смотрите также

Тип Pointer, директивы компилятора $T, $TypedAddress.

AllocMemCount, переменная

Синтаксис

```
var AllocMemCount: Integer;
```

Описание

В переменной AllocMemCount хранится количество блоков, выделенных менеджером памяти Delphi.

Советы и приемы

- Delphi не использует переменную AllocMemCount — она предназначена только для получения информации. Изменение ее значения, хотя и не имеет смысла, не принесет вреда.

- Если вы пишете собственный менеджер памяти, проверьте значение AllocMemCount до вызова SetMemoryManager. Переменная AllocMemCount должна быть равна нулю. Если это не так, значит, встроенный менеджер памяти уже выделил, как минимум, один блок. Проблема в том, что Delphi, возможно, попытается освободить этот блок с помощью вашего менеджера памяти. Если он не может справиться с этой ситуацией, лучше прервать выполнение программы.
- Разрабатывая собственный менеджер памяти, можно установить значение AllocMemCount, чтобы отразить количество выделенных им блоков памяти.
- Для библиотек динамической компоновки переменная AllocMemCount может не отражать количество блоков, выделенных другими модулями. Если требуется вычислить количество блоков памяти, выделенных всеми модулями, которые используют модуль ShareMem, вызовите функцию GetAllocMemCount.

Смотрите также

Переменная AllocMemSize, процедуры Dispose, FreeMem, GetHeapStatus, GetMem, GetMemoryManager, New, ReallocMem, SetMemoryManager.

AllocMemSize, переменная

Синтаксис

```
var AllocMemSize: Integer;
```

Описание

В переменной AllocMemSize содержится общее количество байт во всех блоках памяти, выделенных менеджером памяти Delphi. Таким образом, в ней представлен объем динамической памяти, отведенной приложению.

Советы и приемы

- Delphi не использует переменную AllocMemSize для внутренних целей. Изменение ее значения, хотя и не имеет смысла, не принесет вреда.
- Разрабатывая собственный менеджер памяти, можно установить значение AllocMemSize для отражения размера памяти, выделенной приложению.
- Для библиотек динамической компоновки значение AllocMemSize может не отражать блоки, выделенные в других модулях. Если требуется определить размер блоков, выделенных всем модулям, которые используют ShareMem, вызовите функцию GetAllocMemSize.

Смотрите также

Переменная AllocMemCount, процедуры Dispose, FreeMem, GetHeapStatus, GetMem, GetMemoryManager, New, ReallocMem, SetMemoryManager.

And, ключевое слово

Синтаксис

Логическое выражение and *Логическое выражение*
Целое выражение and *Целое выражение*

Описание

Оператор and выполняет логическую операцию «и», если операнды относятся к типу Boolean, и битовое «и», если операнды целые. Целые операнды могут быть любого целого типа, в том числе Int64. Логическое «и» равно False, если хотя бы один операнд ложный, и True, если оба операнда истинны.

Советы и приемы

- В отличие от стандартного Паскаля, если левый операнд ложный, Delphi не вычисляет правый операнд, т. к. результат в любом случае – ложь. Можно отключить этот сокращенный алгоритм и вернуться к стандарту с помощью директивы компилятора $BoolEval или $B.
- Целое «и» обрабатывает каждый бит своего операнда, устанавливая бит результата в 0, если хотя бы в одном операнде имеется нулевой бит, и 1, если в одном из двух операндов этот бит равен 1. Если один из операндов короче другого, Delphi дополняет меньший операнд нулями в старших битах. Результат имеет размер большего операнда.

Примеры

```
var
  I, J: Integer;
  S: string;
begin
  I := $F0;
  J := $8F;
  WriteLn(I and J); // Выводит 128 (= $80)

  // Укороченная схема вычисления AND не допускает обращения к
  // несуществующему элементу строки S[1] в следующем примере.
  S := '';
  if (Length(S) > 0) and (S[1] = 'X') then
    Delete(S, 1, 1);
```

Смотрите также

Типы Boolean, ByteBool, LongBool, WordBool, ключевые слова Not, Or, Shl, Shr, Xor, директивы компилятора $B, $BoolEval.

AnsiChar, тип

Синтаксис

```
type AnsiChar = #0..#255;
```

Описание

Тип AnsiChar представляет собой 8-битовый расширенный символ ANSI. В текущей версии Delphi тип Char аналогичен AnsiChar, но в будущих версиях он может быть переопределен, а тип AnsiChar всегда будет 8-битовым, несмотря на переопределение типа Char.

Смотрите также

Типы AnsiString, Char, WideChar

AnsiString, тип

Синтаксис

```
type AnsiString;
```

Описание

Тип AnsiString – это длинная строка со счетчиком ссылок, состоящая из символов типа AnsiChar. По умолчанию в Delphi тип string является синонимом AnsiString. Однако если включена директива компилятора $H- или $LongString, string может стать синонимом ShortString.

Delphi хранит AnsiString в виде указателя на запись, но вместо ссылки на начало записи указатель AnsiString ссылается на начало поля Data. Поля Length и RefCount предшествуют содержимому строки.

```
type
  // Ниже приведена логическая структура AnsiString, но это
  // лишь описательное определение, и оно не предназначено
  // для компиляции.
  TAnsiString = record
    RefCount: LongWord;
    Length: LongWord;
    Data: array[1..Length+1] of AnsiChar;
  end;
```

Советы и приемы

- Delphi управляет жизненным циклом строк AnsiString с помощью счетчика ссылок. Можно, по необходимости, манипулировать счетчиком ссылок с помощью процедур Initialize и Finalize.

- Присвоение строки переменной типа AnsiString копирует указатель на строку и увеличивает счетчик ссылок. Новую переменную можно рассматривать как имеющую собственную копию строки, т. к. Delphi использует семантику «копирование при записи». Если изменить содержимое строки, счетчик ссылок которой больше единицы, Delphi автоматически создает уникальную копию строки и модифицирует копию.
- В каждой строке, кроме того, содержится ее длина в виде целого значения. Значение длины строки можно установить с помощью процедуры SetLength. Delphi автоматически записывает символ #0 в конец строки (но не включает этот символ в общую длину строки), после чего строку легко преобразовать к типу PChar, который требуется для Windows API и других функций в стиле языка C.

Смотрите также

Типы AnsiChar, PChar, ShortString, WideString, процедуры Finalize, Initialize, SetLength, SetString, функция Length, ключевое слово String, директивы компилятора $H, $LongStrings.

Append, процедура

Синтаксис

```
procedure Append(var F: TextFile);
```

Описание

Процедура Append открывает существующий текстовый файл для записи. Начальная позиция записи устанавливается в конец файла и новые данные дописываются в эту точку. Процедура Append встроена в компилятор. Пример смотрите в описании процедуры AssignFile.

Ошибки

- Необходимо вызвать AssignFile до процедуры Append, иначе Delphi выдаст ошибку ввода/вывода 102.
- Если файл по какой-либо причине не может быть открыт, вызов процедуры Append генерирует ошибку Windows.

Советы и приемы

Заметьте, что типизированный или нетипизированный двоичный файл нельзя открыть для дозаписи – только текстовые файлы. Чтобы дописать в двоичный файл, вызовите Reset и перейдите в конец файла.

Смотрите также

Процедуры AssignFile, CloseFile, Reset, Rewrite, функции Eof, IOResult, тип TextFile, директивы компилятора $I, $IOChecks.

ArcTan, функция

Синтаксис

```
function ArcTan(Number: Вещественный тип): Extended;
```

Описание

Функция ArcTan возвращает арктангенс в радианах от Number. Функция ArcTan – встроенная.

Советы и приемы

- Delphi автоматически преобразует аргументы типа Integer и Variant в вещественные числа. Для преобразования Int64 в вещественное число прибавьте к нему 0.0.
- Если Number равно плюс бесконечности, результат равен $\pi/2$ (или точнее, наилучшее приближение $\pi/2$ в Delphi); если Number – минус бесконечность, то приближенному значению $-\pi/2$.
- Если Number «спокойное» NaN, результат равен Number.
- Если Number «сигнализирующее» NaN, ArcTan вызывает ошибку времени выполнения 6 (EInvalidOp).

Смотрите также

Функции Cos, Sin.

Array, ключевое слово

Синтаксис

```
type Имя = array[Тип индекса] of Базовый тип;      // статический массив
type Имя = array[Тип индекса, ...] of Базовый тип;// статический массив
type Имя = array of Базовый тип;                  // динамический массив
Имя: array of Базовый тип      // открытый массив в качестве параметра
                                // подпрограммы
Имя: array of const        // открытый вариантный массив в качестве
                           // параметра подпрограммы
```

Описание

В Delphi имеется несколько различных типов массивов: статические, динамические и открытые.

- Статический массив – это традиционный массив языка Паскаль. В качестве индекса может использоваться любой порядковый тип, кроме того массив может иметь несколько индексов. Размер статического массива не меняется во время выполнения программы.
- Динамический массив – массив, тип индекса которого Integer, и размер которого может меняться в процессе работы программы. Нижняя граница индекса всегда равна нулю, верхняя граница

устанавливается процедурой SetLength. Чтобы скопировать динамический массив, вызовите процедуру Copy. Присваивание динамического массива означает присваивание ссылки на массив без переноса его содержимого. Для управления жизненным циклом динамического массива предназначены счетчики ссылок. В отличие от строк, для динамических массивов Delphi не использует принцип «копирования при записи».

- Параметр подпрограммы может быть открытым массивом. В подпрограмму можно передать любой статический или динамический массив. Delphi передает дополнительный, скрытый параметр, содержащий верхнюю границу массива. Подпрограмма не может изменить размер динамического массива, переданного как открытый массив. Независимо от типа индекса фактического массива параметр, переданный как открытый массив, использует целый индекс с нижней границей, равной нулю.
- Специальный вид открытого массива – это вариантный открытый массив, который объявляется как array of const. Каждый элемент массива преобразуется в запись TVarRec. Самая распространенная сфера применения вариантного открытого массива – это написание подпрограммы, которая может принимать переменное количество аргументов (такой, как функция Format из модуля SysUtils).
- Массив AnsiChar, Char или WideChar с индексом, являющимся целым диапазоном, начинающимся с нуля, – это особый случай. Delphi рассматривает такой массив как строку или «широкую» строку (если только вы не выключите директиву компилятора $ExtendedSyntax или $X), с той оговоркой, что нельзя передать символьный массив в подпрограмму, в которой объявлен var параметр строкового типа. Также можно передать ссылку на массив в качестве аргумента подпрограммы, которой требуется параметр типа PChar или PWideChar. Delphi автоматически передает в этом случае адрес первого элемента массива.
- Массив хранится в порядке развертывания столбцов, т. е. самый правый индекс массива меняется быстрее всего.

Примеры

```
// Добавления сообщения в файл протокола.
// Смотрите пример для процедуры AssignFile. Там приведена
// другая версия процедуры Log.
procedure Log(const Fmt: string; const Args: array of const); overload;
begin
  Log(Format(Fmt, Args));
end;

// Добавляет случайное число в целый динамический массив.
// Поскольку динамические и открытые массивы имеют одинаковый
// синтаксис, для параметра типа «динамический массив» приходится
```

```
// использовать именованный тип.
type
  TIntArray = array of integer;

procedure AppendRandomInt(var Ints: TIntArray);
begin
  SetLength(Ints, Length(Ints) + 1);
  Ints[High(Ints)] := Random(MaxInt);
end;

var
  Counter: Integer;
  TestInfo: string;
  Ints: TIntArray;
  I: Integer;
begin
  ...
  Log('Это тест #%d: %s', [Counter, TestInfo]);
  for I := 1 to 10 do
    AppendRandomInt(Ints);
end.
```

Смотрите также

Процедуры Copy, SetLength, функции High, Length, Low, Slice, типы PAnsiChar, PChar, PWideChar, TVarRec, ключевое слово Type, директивы компилятора $ExtendedSyntax, $X Compiler Directive.

As, ключевое слово

Синтаксис

Ссылка на объект as *тип класса*
Ссылка на объект или *интерфейс* as *тип интерфейса*
Описание

Оператор as преобразует ссылку на объект к другому типу класса или ссылку на интерфейс к другому типу интерфейса. Тип результирующего выражения соответствует типу класса или интерфейса правой части оператора as.

Советы и приемы

- Если ссылка на объект или интерфейс равна nil, результат также равен nil.
- Объявленный класс объекта должен быть потомком или предком класса, указанного в правой части. Если тип ссылки на класс не совместим с этим классом, компилятор выдает ошибку. Если объявленный тип совместим, но истинный тип объекта во время выполнения не равен указанному типу или не является его потомком, Delphi выдает ошибку времени выполнения (EInvalidCast).

- Для преобразования типа объекта следует применять оператор as, а не стандартный синтаксис преобразования типов. Единственное исключение из этого правила имеет место, если вы точно знаете тип объекта, выполнив ранее оператор is.
- Если в правой части указан интерфейсный тип, Delphi вызывает метод QueryInterface, передав в качестве первого аргумента GUID интерфейсного типа. Если объект не реализует этот интерфейс, Delphi выдает ошибку времени выполнения 23 (EIntFCastError).

Пример

```
// Если checkbox включен, разблокировать кнопку OK.
// Этот обработчик можно использовать для нескольких checkbox.
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if (Sender as TCheckBox).Checked then
    OkButton.Enabled := True;
end;
```

Смотрите также

Ключевые слова Interface, Is, тип TObject

Asm, ключевое слово

Синтаксис

```
asm
 инструкции ассемблера
end;
```

Описание

Ключевое asm обозначает начало блока ассемблерных инструкций.

Советы и приемы

- Блок asm – это предложение языка Паскаль, и его можно использовать везде, где требуется предложение или блок, например, в качестве тела подпрограммы.
- В ассемблерном блоке можно ссылаться на имена переменных и переходить на метки, объявленные в любом месте процедуры. Не выполняйте переходов внутрь цикла, если не уверены в том что делаете. Метка, начинающаяся с символа @, является локальной в пределах подпрограммы и не должна объявляться заново.
- Встроенный ассемблер Delphi обычно отстает от современных технологий, и потому не стоит надеяться, что в нем имеется последний и самый полный набор инструкций. Тем не менее, допускается применение директив DB, DW или DD для явной компиляции машинных кодов.

- **В блоке** `asm` **можно изменять значения регистров** `EAX`, `ECX` **и** `EDX`, **но необходимо сохранять значения** `EBX`, `ESI`, `EDI`, `EBP` **и** `ESP`. **Как правило, нельзя быть уверенным, что в каком-либо регистре содержится определенное значение, но, соблюдая осторожность, можно обращаться к параметрам подпрограммы через соответствующие регистры. Передача параметров в процедуры рассматривается в описании директив соглашений о вызовах (**`cdecl`, `pascal`, `register`, `safecall` **и** `stdcall`**).**
- **Ручное написание ассемблерного кода редко помогает улучшить производительность. Самая частая причина включения блока** `asm` **– необходимость выполнения инструкций, отсутствующих в Delphi. Например, инструкции** `CPUID`, **используемой в следующем примере.**

Примеры

```
unit cpuid;

// Определение типа CPU.
// В этом модуле приведена функция GetCpuID, которая использует
// инструкцию CPUID для определения типа процессора. GetCpuID
// возвращает истину, если инструкция CPUID поддерживается и ложь,
// если не поддерживается. Старые 486-е и более ранние процессоры
// не поддерживают CPUID.

interface

const
  VendorIntel = 'GenuineIntel';
  VendorAMD   = 'AuthenticAMD';
  VendorCyrix = 'CyrixInstead';

type
  TCpuType = (cpuOriginalOEM, cpuOverdrive, cpuDual, cpuReserved);
  TCpuFeature = (cfFPU, cfVME, cfDE, cfPDE, cfTSC, cfMSR, cfMCE, cfCX8,
                 cfAPIC, cfReserved10, cfReserved11, cfMTRR, cfPGE, cfMCA,
                 cfCMOV, cfPAT, cfReserved17, cfReserved18, cfReserved19,
                 cfReserved20, cfReserved21, cfReserved22, cfReserved23,
                 cfMMX, cfFastFPU, cfReserved26, cfReserved27,
                 cfReserved28, cfReserved29, cfReserved30, cfReserved31
                );
  TCpuFeatureSet = set of TCpuFeature;
  UInt4 = 0..15;
  TCpuId = packed record
    CpuType: TCpuType;
    Family: UInt4;
    Model: UInt4;
    Stepping: UInt4;
    Features: TCpuFeatureSet;
    Vendor: string[12];
  end;
```

```
// Получение информации о CPU и запись в Cpuid.
function GetCpuid(var Cpuid: TCpuid): Boolean; register;

implementation

function GetCpuid(var Cpuid: TCpuid): Boolean;
asm
  // GetCpuid использует передачу параметров через регистры,
  // и потому параметр Cpuid записывается в EAX. Так как это
  // VAR-параметр, в EAX содержится указатель на запись.
  // Проверяет, поддерживает ли процессор инструкцию CPUID.
  // При проверке изменяются значения ECX и EDX.
  pushfd
  pop ecx             // Извлечение EFLAGS в ECX.
  mov edx, ecx        // Сохранение копии EFLAGS в EDX.
  xor ecx, $200000    // Переключение флага ID.
  push ecx            // Попытка установки EFLAGS.
  popfd
  pushfd              // Теперь проверяем, произошли ли изменения.
  pop ecx             // Извлечение нового значения EFLAGS в ECX.
  xor ecx, edx        // Сравнение с EDX.
  je @NoCpuId         // Если биты равны, процессор не поддерживает
                      // инструкции CPUID.

  // Теперь можно воспользоваться CPUID. Восстанавливаем исходное
  // значение EFLAGS.
  push edx
  popfd

  // Инструкция CPUID изменит значение EAX, поэтому записываем аргумент
  // Cpuid в ESI. Delphi требует восстановления ESI после выхода из
  // блока ASM, поэтому сохраняем его предыдущее значение. Также сохраняем
  // EBX, оно будет затерто инструкцией CPUID, и может нам потребоваться.
  push esi
  push ebx
  mov esi, eax

  // Извлекаем наименование производителя, которое является сцеплением
  // содержимого регистров EBX, EDX и EAX, рассматриваемых как три
  // четырехбайтовых символьных массива.
  xor eax, eax                      // EAX = 0 значит «извлечение
                                    // наименования производителя»
  dw $a20f                          // инструкция CPUID
  mov BYTE(TCpuid(esi).Vendor), 12         // длина строки
  mov DWORD(TCpuid(esi).Vendor+1), ebx     // содержимое строки
  mov [OFFSET(TCpuid(esi).Vendor)+5], edx
  mov [OFFSET(TCpuid(esi).Vendor)+9], ecx

  // Получение информации о процессоре.
  // Теперь EAX не равен 0, поэтому извлекается информация о
```

```
    // процессоре.
    dw $a20f               // инструкция CPUID
    mov TCpuid(esi).Features, edx

    // Сигнатура извлекается по частям, большая часть которых имеет
    // длину 4 бита. Delphi не поддерживает битовые поля, поэтому
    // в записи TCpuid для хранения этих полей используются байты.
    // Это требует перепаковки полубайтов в байты.
    mov edx, eax
    and al, $F
    mov TCpuid(esi).Stepping, al

    shr edx, 4
    mov eax, edx
    and al, $F
    mov TCpuid(esi).Model, al

    shr edx, 4
    mov eax, edx
    and al, $F
    mov TCpuid(esi).Family, al

    shr edx, 4
    mov eax, edx
    and al, $3
    mov TCpuid(esi).CpuType, al

    pop ebx             // Восстановление содержимого регистров EBX и ESI.
    pop esi
    mov al, 1           // Возвращаем инстину в случае успеха.
    ret

@NoCpuId:
    xor eax, eax        // Возвращаем ложь, если CPUID не поддерживается.
end;

end.
```

Смотрите также

Директивы `CDecl`**,** `Pascal`**,** `Register`**,** `SafeCall`**,** `StdCall`

Assembler, директива

Синтаксис

Заголовок подпрограммы; `assembler;`

Описание

Директива `assembler` **смысла не имеет. Существует для обратной совместимости с Delphi 1.**

Смотрите также

Ключевое слово `Asm`.

Assert, процедура

Синтаксис

```
procedure Assert(Test: Boolean);
procedure Assert(Test: Boolean; const Message: string);
```

Описание

Процедура `Assert` предназначена для документирования и проверки истинности допущений, сделанных при написании кода. `Assert` фактически не является процедурой. Компилятор специальным образом обрабатывает вызовы `Assert` и генерирует имя файла и номер строки, где была нарушена истинность утверждения для облегчения поиска приведшей к этому ошибки.

Если условие `Test` равно `False`, Delphi вызывает процедуру по указателю, определяемому переменной `AssertErrorProc`. Модуль `SysUtils` устанавливает значение этой переменной равным процедуре, вызывающей исключительную ситуацию `EAssertionFailed`. Если значение `AssertErrorProc` равно `nil`, Delphi вызывает ошибку 21 (`EAssertError`).

Можно указать необязательное сообщение, которое Delphi передаст процедуре `AssertErrorProc`. Если сообщение не указано, Delphi использует строку по умолчанию, «Assertion failed».

Советы и приемы

- Правильный способ применения процедуры `Assert` предполагает проверку условий, которые должны выполняться, чтобы код работал корректно. Все программисты делают допущения – о внутреннем состоянии объекта, значении и корректности аргументов подпрограммы или значении, возвращаемом функцией. Лучше всего относиться к допущениям как к проверкам на наличие ошибок программиста, а не пользователя.
- Можно отключить допущения с помощью директивы компилятора `$Assertions` или `$C`, но причины для этого появляются редко. Ошибка «assertion failed» может сбить пользователя с толку, но в значительно большей степени его расстроит потеря данных.

Пример

В этой главе содержится несколько примеров использования процедуры `Assert`: смотрите описание процедуры `Move`, функций `TypeInfo`, `VarArrayLock` и `VarIsArray`.

Смотрите также

Переменная `AssertErrorProc`, директивы компилятора `$Assertions`, `$C`.

AssertErrorProc, переменная

Синтаксис

```
var AssertErrorProc: Pointer;


procedure ErrorProc(const Message, FileName: string;
    LineNumber: Integer; ErrorAddress: Pointer);
AssertErrorProc := @ErrorProc
```

Описание

В переменной AssertErrorProc хранится адрес процедуры, вызываемой Delphi, если допущение нарушается. Компилятор передает процедуре сообщение и точку, где была вызвана процедура Assert.

Советы и приемы

- С помощью этой переменной можно реализовать процедуру обработки ошибок, выполняющую какие-либо действия, например, протоколирование сбоя, отправку сообщения по электронной почте персоналу и т. п. Причем, в отличие от других процедур обработки ошибок, она может вернуть управление, в этом случае программа будет продолжена с того оператора, где была вызвана процедура Assert.
- Если переменная AssertErrorProc равна nil, Delphi вызывает ошибку времени выполнения 21 (EAssertError).
- Модуль SysUtils устанавливает эту переменную на процедуру, вызывающую исключительную ситуацию EAssertError.

Смотрите также

Переменная AbstractErrorProc, ErrorProc, ExceptProc, процедура Assert.

Assign, процедура

Синтаксис

```
procedure Assign(var F: File; const FileName: string);
procedure Assign(var F: TextFile; const FileName: string);
```

Описание

Процедура Assign делает то же самое, что и AssignFile, но при разработке программ следует применять AssignFile. Assign – это имя метода, часто используемое в Delphi, и два одинаковых наименования могут привести к путанице. Assign не является настоящей процедурой.

Смотрите также

Процедура AssignFile

Assigned, функция

Синтаксис

```
function Assigned(P: Pointer): Boolean;
function Assigned(Obj: TObject): Boolean;
function Assigned(Method: TMethod): Boolean;
```

Описание

Функция Assigned возвращает истину, если аргумент не равен nil, и ложь, если аргумент равен nil. Assigned не является настоящей функцией.

Советы и приемы

- Аргумент может быть указателем, ссылкой на объект или методом.
- Вызов Assigned вместо сравнения указателя с nil не ухудшает производительность.
- Если аргумент является указателем на функцию, использование Assigned проясняет отсутствие у вас намерения вызвать саму функцию и сравнить ее результат с nil. Поэтому Assigned часто применяется для проверки указателей на функцию и метод.
- Указатель на метод состоит из двух частей: указателя на код и указателя на данные. Assigned проверяет только самое старшее слово в ссылке: если старшее слово равно нулю, ссылка на указатель нулевая. Assigned игнорирует ссылку на данные.

Смотрите также

Ключевое слово Nil.

AssignFile, процедура

Синтаксис

```
procedure AssignFile(var F: File; const FileName: string);
procedure AssignFile(var F: TextFile; const FileName: string);
```

Описание

Процедура AssignFile вызывается для присваивания имени типизированному, нетипизированному или текстовому файлу перед его открытием. AssignFile не является настоящей процедурой.

Советы и приемы

- Последующий вызов Append, Reset или Rewrite открывает файл. Если предварительно не была вызвана процедура AssignFile, вызов Append, Reset или Rewrite приведет ошибке ввода/вывода 102.
- Delphi интерпретирует пустую строку как ссылку на консольное устройство. В консольном приложении файлы Input и Output авто-

матически назначаются на консоль. Попытка использования консольного файла в графическом приложении приведет к ошибке ввода/вывода 105.

Пример

```
var
  LogFile: string = 'c:\log.txt';

// Добавление сообщения в файл протокола. В описании ключевого слова
// Array приведена другая версия процедуры Log.
procedure Log(const Msg: string); overload;
var
  F: TextFile;
begin
  AssignFile(F, LogFile);
  // Попытка дописывания данных к файлу. Может увенчаться успехом только
  // если файл существует.
{$IoChecks Off}
  Append(F);
{$IoChecks On}
  if IOResult <> 0 then
    // Файл не существует, создаем его.
    Rewrite(F);
  WriteLn(F, Msg);
  CloseFile(F);
end;
```

Смотрите также

Процедуры Append, CloseFile, Reset, Rewrite, функции Eof, IOResult, типы File, TextFile, директивы компилятора $I, $IOChecks.

At, директива

Синтаксис

raise *Exception* at *Address*;

Описание

Директива at предназначена для вызова исключительной ситуации с указанием конкретного адреса возникновения ошибки. Address может быть любым целым выражением.

Советы и приемы

Директива at в большинстве приложений не применяется, однако она может быть полезна при написании определенных библиотек и собственных пакетов обработки ошибок.

Пример

В следующем примере приведена подпрограмма, вызывающая исключительную ситуацию с указанием вызывающего адреса. Эту подпрограмму можно вызвать в любой точке приложения, при этом отладчик остановит приложение не в том месте, где фактически была вызвана исключительная ситуация, а там, где была вызвана процедура RaiseExceptionInCaller, что значительно полезнее и информативнее.

```
procedure RaiseExceptionInCaller;
  // Извлекает адрес возврата из стека и сохраняет его в EAX,
  // в котором обычно хранится результат выполнения функции.
  // Значение ESP - это адрес RaiseExceptionInCaller; возвращаемся
  // еще на один фрейм в [ESP+4] для получения адреса вызова
  // RaiseExceptionInCaller. Если вам потребуются локальные
  // переменные для RaiseExceptionInCaller, вы, возможно захотите
  // воспользоваться смещением от значения EBP.
  function CallerAddress: Pointer;
  asm
    mov eax, [esp+4]
  end;
begin
  raise Exception.Create('Пример') at CallerAddress;
end;

procedure Demo;
begin
  RaiseExceptionInCaller; // Здесь отладчик покажет исключительную
                          // ситуацию.
end;
```

Смотрите также

Ключевое слово Raise.

Automated, директива

Синтаксис

```
type Описание класса
automated
Описания методов и свойств
end;
```

Описание

Директива automated обозначает секцию объявления класса, в которой все последующие методы и свойства используются для сервера COM-автоматизации. Описания методов automated аналогичны открытым (public) методам за исключением того, что для них создается дополнительная информация RTTI, а именно – тип каждого аргумента и тип возвращаемого значения, если метод является функцией. В главе 3

«Информация о типе времени выполнения» подробно описан формат таблиц RTTI, в которых хранятся сигнатуры методов.

Описания `automated` – это устаревшая конструкция. Вместо них следует использовать библиотеки типов и интерфейсы, которые обеспечивают большую гибкость и мощность.

Смотрите также

Ключевые слова Class, DispInterface, Interface, директива Public

Begin, ключевое слово

Синтаксис

```
begin
  Предложения...
end
```

Описание

Ключевое слово `begin` обозначает начало блока. Блок – это основное тело программы, библиотеки, процедуры, функции или модуля. Блок может включать любое количество предложений и может применяться везде, где требуется одиночное предложение, например, в условии или цикле.

Ключевое слово `begin` в Delphi работает аналогично стандарту языка Паскаль. Также можно написать блок на языке ассемблера, использовав вместо `begin` ключевое слово `asm`.

Смотрите также

Ключевые слова `Asm`, `End`, `Initialization`, `Library`, `Program`, `Unit`.

BeginThread, функция

Синтаксис

```
function BeginThread(SecurityAttributes: Pointer; StackSize: LongWord;
  ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlags: LongWord;
  var ThreadId: LongWord): Integer;
```

Описание

Функция `BeginThread` вызывается для запуска потока в многопоточном приложении и обращается к функции Windows API `CreateThread`, которая начинает новый поток и, в свою очередь, вызывает функцию потока (`ThreadFunc`) в контексте нового потока. Когда функция потока заканчивает работу, поток завершается. За дополнительной информацией об атрибутах безопасности и флагах обратитесь к документации Windows API по функции `CreateThread`.

`BeginThread` возвращает дескриптор нового потока или ноль, если Windows не создаст поток. `BeginThread` – это настоящая функция.

Советы и приемы

- Следует использовать функцию `BeginThread` вместо функции `CreateThread` Windows API, т. к. `BeginThread` выставляет глобальную переменную `IsMultiThread` в истину. `BeginThread` также определяет параметры `ThreadFunc` и `ThreadID` в стиле языка Паскаль.
- Функция потока должна перехватывать и обрабатывать все исключительные ситуации. Если функция потока вызывает исключительную ситуацию, которую не может обработать, ее перехватывает `BeginThread` и прерывает программу.
- Более детальную информацию о потоках можно найти в главе 4 «Создание многопоточных приложений».
- Как и для любого другого ресурса, следует вызвать `CloseHandle` после окончания работы потока, чтобы Windows освободил все связанные с ним ресурсы. Если вы создадите поток в приостановленном состоянии и затем закроете его, ни разу не запустив, в Delphi произойдет небольшая утечка памяти. Чтобы предотвратить утечку, всегда запускайте поток до того, как его закроете.

Пример

В следующем примере показан фоновый поток, который вычисляет множество Мандельброта и рисует его изображение в растре. Фоновый поток уведомляет основной поток программы о том, что формирование изображения закончено, и оно может быть выведено на экран. В фоновом потоке используется свойство `Scanline`, т. к. это обеспечивает быстрый и удобный доступ к данным растра без привлечения Windows API. Если потоку требуются функции Windows GDI, следует использовать метод `TThread.Synchronize` из модуля `Classes`.

```
const
  // Фоновый поток посылает это сообщение основному.
  Wm_Finished = Wm_User;

type
  TThreadInfo = class;
  TWmFinished = packed record
    Msg: Cardinal;
    Aborted: Boolean;
    Bitmap: TBitmap;
    Result: LongInt;
  end;
  // Каждому потоку передается объект ThreadInfo. Объект создает
  // растровое изображение, в котором создается множество Мандельброта
  // и флаг, который поток периодически проверяет, чтобы определить
  // необходимость заранее прервать свою работу.
  TThreadInfo = class
  private
    fBitmap: TBitmap;
    fAborted: Boolean;
```

```
  public
    constructor Create(Width, Height: Integer);
    destructor Destroy; override;
    procedure Abort;

    property Bitmap: TBitmap read fBitmap;
    property Aborted: Boolean read fAborted;
  end;

// Число итераций устанавливаем равным 360, чтобы было легче
// преобразовать количество итераций в значение Hue (цветовое
// смещение) в цветовой схеме HSV.
const
  MaxIterations = 360;

// Смотрите пример для процедуры Exit, чтобы найти функцию
// ComputeIterations.

// Эти начальные точки смотрятся неплохо. Вы можете их изменить,
// если хотите увидеть что-нибудь другое.
const
  XOffset = -0.03;
  YOffset = 0.78;
  Zoom = 450000.0;
  Background = clBlack;

// Растр использует 24-разрядный формат пикселов, так что
// каждый пиксел занимает три байта. Массив TRgb облегчает доступ
// к красной, зеленой и синей компонентам цвета строки изображения.
type
  TRgb = array[0..2] of Byte;
  PRgb = ^TRgb;

function MandelbrotThread(Param: Pointer): Integer;
var
  Info: TThreadInfo;
  R, C: Integer;                    // Положение в растре.
  Color: TColor;                    // Цвет пикселя.
  Count: Integer;                   // Количество итераций.
  X, Y: Double;                     // Положение на воображаемой
                                    // плоскости.
  XIncrement, YIncrement: Double;   // Шаг изменения координат X и Y.
  Scanline: PRgb;                   // Доступ к изображению выполняется
                                    // построчно.
begin
  Result := 0;
  Info := TThreadInfo(Param);

  XIncrement := Info.Bitmap.Width / Zoom;
  YIncrement := Info.Bitmap.Height / Zoom;
```

```
  Y := YOffset;
  for R := 0 to Info.Bitmap.Height-1 do
  begin
    X := XOffset;
    Scanline := Info.Bitmap.ScanLine[R];
    for C := 0 to Info.Bitmap.Width-1 do
    begin
      Count := ComputeIterations(X, Y); // Смотрите процедуру Exit.
      X := X + XIncrement;
      // Преобразуем значение максимального числа итераций в цвет
      // фона,а другие значения количества итераций - в различные
      // цвета путем использования count в качестве значения компо-
      // ненты цветового смещения в цветовой схеме.
      if Count = MaxIterations then
        Color := Background
      else
        Color := HSV(Count, 255, 255); // Смотрите функцию HSV в
      Scanline[0] := GetBValue(Color); // описании ключевого слова
      Scanline[1] := GetGValue(Color); // Case.
      Scanline[2] := GetRValue(Color);
      Inc(Scanline);
    end;
    Y := Y + YIncrement;
    if Info.Aborted then
    begin
      PostMessage(Form1.Handle, Wm_Finished, 1, LParam(Param));
      Exit;
    end;
 end;
  // Сообщаем основному потоку, что фоновый поток завершил свою
  // работу. Передаем объект ThreadInfo в качестве параметра
  // сообщения.
  PostMessage(Form1.Handle, Wm_Finished, 0, LParam(Param));
end;

// Когда фоновый поток завершается, рисуем изображение и освобождаем
// объект ThreadInfo.
procedure TForm1.WmFinished(var Msg: TWmFinished);
begin
  if not Msg.Info.Aborted then
    Image1.Picture.Bitmap := Msg.Info.Bitmap;
  FreeAndNil(Msg.Info);
  CloseHandle(Thread);
  Thread := 0;
end;

// Создаем новый поток для вычисления и формирования множества
// Мандельброта. Info - это запись типа TThreadInfo, которая
// является закрытым полем класса TForm1. Thread - это THandle,
// тоже закрытое поле.
procedure TForm1.StartThread;
```

```
var
  Id: Cardinal;
begin
  Info := TThreadInfo.Create(Image1.Width, Image1.Height);
  Thread := BeginThread(nil, 0, @MandelbrotThread, Info, 0, Id);
end;

// Когда форма закрывается, она прерывает поток. Возможно, что сообщение
// Wm_Finished придет после разрушения формы, но это не является
// большой проблемой, т. к. Windows уничтожит поток при завершении
// программы.
procedure TForm1.FormClosed(Sender: TObject);
begin
  if Info <> nil then
    Info.Abort;
end;
```

Смотрите также

Переменная IsMultiThread, ключевое слово ThreadVar, тип TThreadFunc

BlockRead, процедура

Синтаксис

```
procedure BlockRead(var F: File; var Buffer; Count: Integer);
procedure BlockRead(var F: File; var Buffer; Count: Integer;
  var RecordCount: Integer);
```

Описание

Процедуру `BlockRead` вызывают для того, чтобы прочитать в `Buffer` `Count` записей из двоичного файла. Если `F` – нетипизированный файл, `BlockRead` использует размер записи, указанный при открытии файла процедурой `Reset`. Если вы указываете переменную `RecordCount`, `BlockRead` записывает в нее количество фактически прочитанных записей. В случае ошибки или конца файла, `RecordCount` может быть меньше, чем `Count`. `BlockRead` не является настоящей процедурой.

Ошибки

- Если аргумент `RecordCount` не указан, и функция `BlockRead` столкнулась с ошибкой или концом файла, возникает ошибка ввода/вывода 100.
- Если файл не открыт, `BlockRead` выдает ошибку ввода/вывода 103.

Советы и приемы

- Аргумент `Buffer` – это не указатель, а нетипизированный `var`-параметр. Передавайте саму переменную, а не ее адрес. Если вы передаете указатель на динамически выделенный буфер, разыменуйте ссылку при вызове `BlockRead`.

- Чаще всего BlockRead применяется для чтения сразу нескольких записей или сложных структур данных, которые не удается аккуратно записать в простой типизированный файл. Предположим, что первые четыре байта файла содержат длину строки, а непосредственно за ними следует сама строка, и вы хотите прочитать эти данные в длинную строку. В таком случае придется читать длину строки и ее содержимое по отдельности, как показано в примере.

Пример

```
// Чтение строки из двоичного файла.
// Строка начинается с четырехбайтовой длины.
function ReadString(var F: File): string;
var
  Len: LongInt;
begin
  BlockRead(F, Len, SizeOf(Len));
  SetLength(Result, Len);
  if Len > 0 then
    BlockRead(F, Result[1], Len);
end;
```

Смотрите также

Процедуры Assign, BlockWrite, CloseFile, Reset, Rewrite, функция IOResult, директивы компилятора $I, $IOCheck.

BlockWrite, процедура

Синтаксис

```
procedure BlockWrite(var F: File; const Buffer; Count: Integer);
procedure BlockWrite(var F: File; const Buffer; Count: Integer;
    var RecordCount: Integer);
```

Описание

Процедура BlockWrite записывает Count записей из Buffer в двоичный файл. Если переменная RecordCount указана в аргументах процедуры, то BlockWrite сохраняет в ней количество фактически добавленных записей. Если диск переполнен или возникла другая ошибка, RecordCount может быть меньше Count. BlockWrite не является настоящей процедурой.

Ошибки

- Если указан аргумент RecordCount и в BlockWrite возникает ошибка, выдается ошибка ввода/вывода 101 или код ошибки Windows.
- Если файл не был открыт, BlockWrite выдает ошибку ввода/вывода 103.

Советы и приемы

- Аргумент Buffer – это не указатель, а нетипизированный var-параметр. Передавайте саму переменную, а не ее адрес. Если в программе есть указатель на динамически выделенный буфер, разыменуйте ссылку при вызове BlockWrite.
- Чаще всего BlockWrite используется для записи нескольких записей одновременно и для сохранения сложных структур данных, которые не удается аккуратно записать в простой типизированный файл. Чтобы записать, например, длинную строку, сначала записывается длина строки в виде четырехбайтового двоичного значения, а затем содержимое строки.

Пример

```
// Запись строки в двоичный файл. Перед строкой записывается
// ее длина в виде четырехбайтового целого.
procedure WriteString(var F: File; const Str: string);
var
  Len: LongInt;
begin
  Len := Length(Str);
  BlockWrite(F, Len, SizeOf(Len));
  if Len > 0 then
    BlockWrite(F, Str[1], Len);
end;
```

Смотрите также

Процедуры Append, AssignFile, BlockRead, CloseFile, Reset, Rewrite, функция IOResult, директивы компилятора $I, $IOChecks.

Boolean, тип

Синтаксис

```
type Boolean = (False, True);
```

Описание

Тип Boolean – это перечисляемый тип, принимающий два значения. Все операции сравнения и логические операции выдают результат типа Boolean. Условия в операторах if, while и repeat-until также должны иметь тип Boolean или любой другой логический тип (ByteBool, WordBool или LongBool).

Советы и приемы

Не преобразуйте целое значение в Boolean. Для того чтобы добиться совместимости с C или C++, где любое ненулевое значение рассматривается как истина, используйте вместо Boolean типы ByteBool, WordBool или LongBool, которые имеют одну и ту же семантику, но разный

размер. В отличие от Boolean, эти три типа интерпретируют любое ненулевое порядковое значение как истину.

Смотрите также

Ключевые слова And, Not, Or, Xor, тип ByteBool, LongBool, WordBool.

Break, процедура

Синтаксис

```
Break;
```

Описание

Break выполняет выход из цикла, аналогично следующему предложению goto:

```
label BreakOut;
begin
  while Condition do
  begin
    DoSomething;
    if AnotherCondition then
      goto BreakOut;  // аналогично Break;
  end;
 BreakOut:
  // Передача управления предложению, следующему после цикла
```

Если Break вызывается из вложенных циклов, выполняется выход только из самого внутреннего цикла. Чтобы выйти сразу из нескольких циклов, используйте предложение goto.

Смотрите также

Процедуры Continue, Exit, ключевые слова For, Repeat, While, Byte.

Byte, тип

Синтаксис

```
type Byte = 0..255;
```

Описание

Тип Byte – это поддиапазон целого. В массиве или упакованной записи значения типа Byte занимают один байт (8 бит). Информация о других целых типах содержится в описании типа Integer.

Смотрите также

Типы Integer, ShortInt.

ByteBool, тип

Синтаксис

```
type ByteBool;
```

Описание

Тип ByteBool – это логический тип, размер которого совпадает с размером типа Byte. Значение типа ByteBool ложно, если порядковое значение равно нулю, и истинно, если оно не равно нулю. Порядковым значением для константы True в ByteBool (ByteBool(True)) является –1.

Советы и приемы

- Значение типа ByteBool применяется там же, где и Boolean. Этот тип полезнее всего для интерфейса с C и C++, где любое ненулевое значение рассматривается как истина.
- Типы WordBool и LongBool аналогичны ByteBool, но имеют другой размер.

Смотрите также

Ключевые слова And, Not, Or, Xor, типы Boolean, LongBool, WordBool.

Cardinal, тип

Синтаксис

```
type Cardinal = 0..4294967295;
```

Описание

Тип Cardinal – это диапазон целых чисел без знака, размер которого соответствует естественному размеру целого. В Delphi 5 этот размер равен 32 разрядам, но в будущих версиях может увеличиться. Для целого без знака, которое должно состоять из 32 битов, независимо от естественного размера целого используйте LongWord. Смотрите описание типа Integer по информации о других целых типах.

Советы и приемы

- Чаще всего Cardinal применяется для вызова Windows API или других внешних функций, которым требуется параметр типа DWORD (unsigned long в C и C++).
- Если требуется целый тип для натуральных или целых чисел, следует определить собственные поддиапазоны, как показано ниже:

```
// Для расчетных задач данный способ предпочительнее, чем применение Cardinal
type
  Whole = 1..MaxInt;
  Natural = 0..MaxInt;
```

- **Использование** `Cardinal` **в качестве обычного целого типа может часто дать результат, отличный от ожидаемого, т. к. может получиться любое значение типа** `Integer` **или** `Cardinal`**. Диапазон обоих типов – 32 бита, однако их комбинации требуют 33 бита. Таким образом, любая арифметическая операция, манипулирующая значениями** `Integer` **и** `Cardinal`**, заставляет компилятор расширить операнды до минимум 33 битов, поэтому Delphi преобразует их к типу** `Int64`**:**

```
type
  I: Integer;
  C: Cardinal;
begin
  ReadLn(I, C);
  // Результат I+C может быть от Low(Integer)..High(Cardinal),
  // который требует 33 бита, поэтому Delphi должен использовать
  // для результата Int64.
  WriteLn(I + C);

  // Сравнение Integer и Cardinal требует преобразования I и C к
  // Int64 для корректного сравнения чисел. Подумайте, что, например,
  // получится, если I=Low(Integer) и C=High(Cardinal), и вы
  // попытаетесь сравнить эти значения как 32-битовые целые.
  if I < C then
    WriteLn('I < C');
end;
```

Смотрите также

Типы `Int64`**,** `Integer`**,** `LongWord`**.**

Case, ключевое слово

Синтаксис

```
case Порядковое выражение of
  Порядковый диапазон: Предложение;
  Порядковый диапазон, Порядковый диапазон, ...: Предложение
  else Предложения...;
end;
```

```
type Имя = record
  Описания...
case Ordinal type of
  Порядковый диапазон: (Объявления...);
  Порядковый диапазон, Порядковый диапазон, ...: (Declarations...);
end;
```

```
type Имя = record
  Описания...
case ИмяПоля: Ordinal type of
```

```
    Порядковый диапазон: (Объявления...);
    Порядковый диапазон, Порядковый диапазон, ...: (Объявления...);
  end;
```

Описания

Оператор case выбирает одну из многих возможных ветвей в зависимости от значения порядкового типа. По сравнению со стандартным Паскалем, в Delphi он расширен включением необязательного условия else. Если другие элементы case не соответствуют выражению, выполняются инструкции, следующие за ключевым словом else, аналогичным otherwise в других расширенных версиях Паскаля.

Ключевое слово case также используется для описания записи с вариантами. Подробности приведены в описании ключевого слова record.

Выражение, следующее за оператором case, должно относиться к порядковому типу, т. е. быть целым, символом или перечислением. Каждый из селекторов case должен быть константой соответствующего типа либо константным диапазоном. Селекторы могут идти в любом порядке, но конструкция else должна быть последней. Селекторы case должны быть уникальными. В каждом из элементов case должна быть только одна инструкция, кроме else, где их может быть несколько.

Советы и приемы

Если выражение case относится к перечисляемому типу и покрывает все возможные значения этого типа, все равно имеет смысл добавить конструкцию else для вызова исключительной ситуации. Перехват такой ситуации сразу уведомит вас о том, что в программе возникли проблемы, и защитит в том случае, если в будущей версии программы вы дополните перечисляемый тип, не дополнив предложение case.

Пример

Следующая функция преобразует цвета, выраженные в виде цветового смещения, насыщенности и значения (модель HSV), в цвета модели RGB (красный-зеленый-синий). Цветовое смещение (Hue) можно представить в виде позиции на цветовой окружности и потому находится в диапазоне 0—359. Значение цветового смещения, деленное на 60, всегда будет находиться в диапазоне от 0 до 5. Тем не менее, предложение case содержит конструкцию else, которая вызывает исключительную ситуацию в случае ошибки. Такое защищенное программирование не требует дополнительных затрат во время выполнения, но поможет идентифицировать ошибки. (Если вы не предвидите возникновение ошибки в этой функции, подумайте, что может быть, если значение цветового смещения будет меньше 0. В этом случае результат выражения Hue div 60 будет находиться в диапазоне −5—0, и функция HSV вызовет исключительную ситуацию.)

```
// Преобразование HSV-цвета в TColor, т.е. RGB-цвет.
```

```
function HSV(Hue: Integer; Saturation, Value: Byte): TColor;
var
  P, Q, R, S: Byte;
begin
  if Saturation = 0 then
    Result := RGB(Value, Value, Value)
  else
  begin
    Hue := Hue mod 360;
    S := Round(Saturation * Frac(Hue / 60));
    P := MulDiv(Value, 255 - Saturation, 255);
    Q := MulDiv(Value, 255 - S, 255);
    R := MulDiv(Value, 255 - (Saturation-S), 255);
    case Hue div 60 of
    0: Result := RGB(Value, R, P);
    1: Result := RGB(Q, Value, P);
    2: Result := RGB(P, Value, R);
    3: Result := RGB(P, Q, Value);
    4: Result := RGB(R, P, Value);
    5: Result := RGB(Value, P, Q);
    else
      raise Exception.CreateFmt('Невозможно: неверное значение Hue = %d',
                                [Hue]);
    end;
  end;
end;
```

Смотрите также

Ключевые слова `If`, `Record`.

CDecl, директива

Синтаксис

Объявление подпрограммы; cdecl;

Описание

Директива `cdecl` сообщает компилятору о необходимости использовать для функции или процедуры соглашение о вызовах в формате языка C. При вызове подпрограммы аргументы помещаются в стек начиная с самого правого аргумента. После завершения подпрограммы аргументы извлекаются из стека.

Функции помещают порядковые значения, указатели, небольшие записи или множества в регистр EAX, вещественные значения – в стек FPU. Строки, динамические массивы, вариантные записи, большие записи и множества передаются как скрытый `var`-параметр. Этот скрытый параметр оказывается первым параметром и помещается в стек последним. Если подпрограмма является методом, переменная

Self помещается еще раньше результата функции (если он передается в виде var-параметра).

Смотрите также

Функция Function, директивы Pascal, Register, ключевое слово Procedure.

ChangeAnyProc, процедура

Синтаксис

```
Syntax
var ChangeAnyProc: Pointer;

procedure ChangeAny(var V: Variant);
ChangeAnyProc := @ChangeAny;
```

Описание

Процедура ChangeAnyProc преобразует вариантное значение varAny к типу Variant, используемому в Delphi. Значение varAny представляет собой скрытый тип, который в Delphi может применяться только для присваивания и передачи в качестве аргумента подпрограммы.

Значение ChangeAnyProc по умолчанию – это процедура, которая вызывает ошибку времени выполнения 15 (EVariantError).

Советы и приемы

Модуль CorbaObj устанавливает значение переменной на процедуру, поддерживающую тип Any из CORBA. Если вы не работаете с CORBA, то можете использовать значение varAny для собственных целей.

Пример

Предположим, требуется использовать в приложении вариантные переменные и хранить в них значения типа Int64. Тип Variant в Delphi не поддерживает тип Int64, но для хранения значений этого типа подходит varAny. Когда Delphi требуется конкретное значение, процедура ChangeAnyProc преобразует Int64 в строку, с которой Delphi может работать. Поле VAny – это указатель, и процедура SetVerInt64 выделяет динамическую память для записи значения типа Int64 и сохраняет этот указатель в вариантной переменной. Процедура ClearAnyProc освобождает эту память, когда Delphi закончит использование вариантного значения.

```
// Изменяет значение Int64, записанное как varAny, к допустимому для
// Variant значению, строке.
procedure ChangeVarInt64(var V: Variant);
var
  Value: Int64;
begin
```

```
    if TVarData(V).VType = varAny then
    begin
      Value := PInt64(TVarData(V).VAny)^;
      V := IntToStr(Value);
    end;
  end;
  ...
  ChangeAnyProc := @ChangeVarInt64;
```

Смотрите также

Переменные `ClearAnyProc`, `RefAnyProc`, типы `TVarData`, `Variant`.

Char, тип

Описание

Тип `Char` представляет собой одиночный символ, как и в стандарте языка Паскаль. Любой целый тип можно преобразовать к символьному типу, указав тип `Char` в конструкции преобразования типов. В отличие от всех других вариантов преобразования типов, размер целого не обязательно должен соответствовать размеру типа `Char`.

Вы можете записать символьную константу несколькими разными способами:

- В виде строки длиной 1 байт, например `'A'` (наилучший вариант для печатных символов).
- В виде управляющего символа со знаком вставки, например `^A` (подходит для управляющих символов ANSI, но имеет смысл дать символу осмысленное имя, как показано ниже):

```
TAB = ^I;
CRLF = ^M^J;
```

- С помощью символа `#` и целой константы, например, `#65` или `#$41` (подходит для управляющих символов, например, #0 или #255)
- С помощью функции `Chr`, например `Chr(65)` (предназначена для преобразования целого в символ, в данном случае «A»)
- Преобразованием целого числа, например `Char(65)` (аналогично `Chr`).

Советы и приемы

В текущей версии Delphi тип `Char` аналогичен `AnsiChar`, но в дальнейшем его смысл может измениться. Например, этот тип может стать синонимом `WideChar`. Тип `Char` годится для большинства применений, однако это не означает, что `Char` и, например, `Byte` всегда будут занимать одинаковый объем памяти.

Смотрите также

Типы AnsiChar, PChar, WideChar, функция Chr, ключевое слово String.

ChDir, процедура

Синтаксис

```
procedure ChDir(const Directory: string);
```

Описание

Процедура ChDir изменяет текущий рабочий каталог и диск на путь, указанный в аргументе Directory. Если ChDir не может по какой-либо причине изменить текущий рабочий каталог, выдается ошибка ввода/вывода с кодом, полученным от Windows. ChDir не является настоящей процедурой.

Смотрите также

Функции GetDir, IOResult, процедуры MkDir, RmDir, директивы компилятора $I, $IOChecks.

Chr, функция

Синтаксис

```
function Chr(IntValue: Integer): AnsiChar;
function Chr(IntValue: Integer): WideChar;
```

Описание

Функция Chr преобразует целое к эквивалентному символу AnsiChar либо WideChar, в зависимости от контекста вызова. Chr не является настоящей функцией, и ее вызов не компилируется в какой-либо код. Delphi автоматически преобразует IntValue в диапазон, необходимый для символьного типа (#0..#255 для AnsiChar или #0..#65535 для WideChar).

Советы и приемы

- Компилятор одинаково интерпретирует символьные выражения Chr(27), #27 и ^[, и конкретный выбор является делом вкуса. Для управляющих символов предпочтительным является вариант с использованием символа вставки, например, ^M для возврата каретки. Для специальных символов лучше подходят символьные константы, например, #0 для отметки конца строки PChar. Применяйте Chr для преобразования переменных.
- Вызов Chr аналогичен преобразованию типов, однако функция Chr более понятна для опытных программистов языка Паскаль.

Смотрите также

Типы AnsiChar, Char, Ord, WideChar.

Class, ключевое слово

Синтаксис

```
type Имя = class
  Описания...
  class function ...;
  class procedure...;
end;

type Имя = class(БазовыйКласс)
  ...
end;

type Имя = class(БазовыйКласс);

type Объявление прототипа = class;

type Имя = class(Базовый Класс, Имя интерфейса...)
   Объявления...
end;

type Имя = packed class...

type МетаКласс = class of ТипКласса;
```

Описание

Ключевое слово `class` обозначает объявление класса, а также описание метода класса. Если сразу за ключевым словом `class` следует точка с запятой, это — *объявление прототипа*: оно уведомляет компилятор о том, что данный тип является классом, не предоставляя об этом классе никакой информации. Следует полностью описать класс ниже в том же блоке объявлений типов. В последнем примере приведено объявление метакласса. Переменная типа «метакласс» может содержать ссылку на класс. Эта переменная может входить в любое выражение, где требуется ссылка на класс, например, при вызове конструктора или метода класса, а также выступать как правая часть оператора `is`. (С другой стороны, для оператора `as` требуется статическое имя класса, а не переменная, содержащая ссылку на класс, т. к. оператор `as` выполняет преобразование типа, и компилятор должен знать целевой тип.)

В объявлении класса указывается один базовый класс и любое количество интерфейсов. Базовый класс можно опустить, и в этом случае Delphi использует `TObject`.

Объявление класса содержит одну или более секций, каждая из которых имеет определенный уровень доступа. По умолчанию принят уровень доступа `public`, если класс не имеет RTTI (из-за указания ди-

ректива компилятора $M или $TypeInfo или из-за того, что является потомком класса, имеющего RTTI), иначе – published.

Каждая из секций начинается с нуля и более описаний полей. После всех полей следуют описания методов и свойств. Количество секций и их порядок могут быть любыми, а секции с одинаковыми уровнями доступа могут повторяться.

Если указывается базовый класс, и точка с запятой стоит сразу же за закрывающей скобкой, ключевое слово end можно опустить.

По умолчанию все поля выровнены по словам, однако если указать директиву packed, то поля будут начинаться с границ байтов. Подробности рассмотрены в описании ключевого слова packed. Для того чтобы все классы и записи были упакованными, используйте директиву компилятора $A или $Align.

Методы класса

Методы классов аналогичны обычным методам, за исключением следующих отличий:

- Объявление методов класса начинается с ключевого слова class. Необходимо указать ключевое слово class как в описании класса, так и в реализации метода.
- Обычный метод вызывается с помощью ссылки на объект. Метод класса вызывается либо с помощью ссылки на объект, либо ссылки на класс.
- Внутри метода класса переменная Self относится к классу, а не к объекту. Это означает, что метод класса не может обращаться к полям, т. к. отсутствует объект для доступа к этим полям.

Советы и приемы

Класс может реализовывать любое количество интерфейсов. Delphi устанавливает соответствие между именами методов интерфейса и методов класса. Можно перенаправить методы интерфейса на другие методы в описании класса с помощью конструкции следующего вида:

```
procedure Interface.InterfaceMethod = MethodName;
function Interface.InterfaceMethod = MethodName;
```

MethodName – это имя метода в объявлении класса. Перенаправление особенно полезно, когда класс реализует несколько интерфейсов, в которых имеются методы с одинаковыми именами и аргументами.

Примеры

```
type
  TSimpleStream = class;
  TSimpleClass = class       // Неявное наследование от TObject.
  private                    // Секции могут располагаться в любом порядке.
    fStream: TSimpleStream;  // Принято перечислять секции в порядке
```

```
public                      // увеличения уровня доступа.
  constructor Create(const FileName: string = '');
  destructor Destroy; override;
  property Stream: TSimpleStream read fStream;
end;

TSimpleStream = class(TPersistent, ISequentialStream)
private
  fHandle: THandle;
  fReadOnly: Boolean;
protected
  // метод для ISequentialStream
  function Read(Data: Pointer; Count: LongInt; BytesRead: PLongInt):
    HResult; stdcall;
  function Write(Data: Pointer; Count: LongInt;
    BytesWritten: PLongint): HResult; stdcall;
public
  constructor Create(const FileName: string); overload;
  constructor Create(Handle: THandle); overload;
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); override;
published
  property ReadOnly: Boolean read fReadOnly write fReadOnly;
end;

// В Delphi Open Tools API интерфейсы IOTAWizard и IOTAFileSystem
// оба содержат метод с именем GetIDString. Класс TVfsWizard
// реализует оба интерфейса и использует перенаправление методов для
// выбора методов, реализующих методы интерфейса.
TVfsWizard = class(TInterfacedObject, IOTAWizard, IOTAFileSystem)
private
// методы для IOTAFileSystem
  function DeleteFile(const FileName: string): Boolean;
  function FileAge(const FileName: string): LongInt;
  function FileExists(const FileName: string): Boolean;
  function GetBackupFileName(const FileName: string): string;
  function GetFileStream(const FileName: string; Mode: Integer):
      IStream;
  function GetFileSystemIDString: string;
  function IOTAFileSystem.GetIDString = GetFileSystemIDString;
  function GetTempFileName(const FileName: string): string;
  function IsFileBased: Boolean;
  function IsReadonly(const FileName: string): Boolean;
  function RenameFile(const OldName, NewName: string): Boolean;

// методы для IOTAWizard
  function GetWizardIDString: string;
  function IOTAWizard.GetIDString = GetWizardIDString;
  function GetName: string;
  function GetState: TWizardState;
  procedure Execute;
```

```
  // Следующая функция предназначена для удобства получения ссылки на
  // глобальный интерфейс IOTAServices.
    class function Services: IOTAServices;
  end;
```

Смотрите также

Директива Automated, ключевые слова Constructor, Destructor, Function, Interface, Is, Packed, Private, Procedure, Property, Protected, Public, Published, Type, переменная Self, директивы компилятора $A, $Align, $M, $TypeInfo.

ClearAnyProc, переменная

Синтаксис

```
var ClearAnyProc: Pointer;

procedure ClearAny(var V: Variant);
ClearAnyProc := @ClearAny;
```

Описание

Когда Delphi завершает использование значения varAny, вызывается ClearAnyProc для освобождения всей памяти, связанной со скрытым значением varAny. По умолчанию выполняется процедура, которая вызывает ошибку времени выполнения 16 (EVariantError).

Советы и приемы

Модуль CorbaObj устанавливает значение этой переменной на процедуру, которая поддерживает тип Any, принятый в CORBA. Если вы не работаете с CORBA, то можете использовать значения типа varAny для собственных целей.

Пример

Смотрите описание этого примера в разделе о переменной ChangeAnyProc.

```
// Очистка вариантного значения типа varAny, в котором содержится
// указатель на значение типа Int64.
procedure ClearVarInt64(var V: Variant);
var
  Ptr: Pointer;
begin
  if TVarData(V).VType = varAny then
  begin
    Ptr := TVarData(V).VAny;
    TVarData(V).VType := varEmpty;
    FreeMem(Ptr);
  end;
end;
```

```
...
ClearAnyProc := @ClearVarInt64;
```

Смотрите также

Переменные ChangeAnyProc, RefAnyProc, типы TVarData, Variant.

Close, процедура

Синтаксис

```
procedure Close(var F: File);
procedure Close(var F: TextFile);
```

Описание

Процедура Close оставлена для обратной совместимости с Turbo Pascal и стандартным Паскалем. В новых Delphi-программах следует использовать CloseFile, чтобы избежать конфликтов с методами Close. Close не является настоящей процедурой.

Смотрите также

Процедура CloseFile.

CloseFile, процедура

Синтаксис

```
procedure CloseFile(var F: File);
procedure CloseFile(var F: TextFile);
```

Описание

Процедура CloseFile вызывается для того, чтобы закрыть файл, открытый с помощью Append, Reset или Rewrite. CloseFile не является настоящей процедурой.

Пример смотрите в описании процедуры AssignFile.

Ошибки

- Если процедура CloseFile вызывается для файла, который уже закрыт, Delphi выдает ошибку ввода/вывода 103.
- Если попытаться закрыть файл, для которого еще не была вызвана процедура AssignFile, результат будет непредсказуемым.

Смотрите также

Процедуры Append, AssignFile, Reset, Rewrite, функция IOResult.

CmdLine, переменная

Синтаксис

```
var CmdLine: PChar;
```

Описание

Переменная CmdLine содержит командную строку, которая использовалась при запуске программы. Для библиотек – это пустая строка. Функции ParamCount и ParamStr преобразуют командную строку в отдельные аргументы; обычно удобнее вызвать одну из этих функций, чем самостоятельно обрабатывать строку. Значения переключателей в командной строке можно проверить с помощью функции FindCmdLineSwitch из модуля SysUtils.

Смотрите также

Переменная CmdShow, функции ParamCount, ParamStr.

CmdShow, переменная

Синтаксис

```
var CmdShow: Integer;
```

Описание

Переменная CmdShow определяет вид основного окна программы. Подробности изложены в документации Windows API по функции ShowWindow. Если в приложении имеется основная форма, Delphi автоматически использует переменную CmdShow для запуска приложения в нормальном, минимизированном или развернутом состоянии.

Смотрите также

Переменная CmdLine.

Comp, тип

Синтаксис

```
type Comp;
```

Описание

Тип Comp – это 64-битовое знаковое целое, для которого требуется математический сопроцессор. Тип Comp существует только для обратной совместимости. Новые программы должны вместо него использовать Int64 – настоящий целочисленный тип с полным диапазоном 64-битового типа, поддерживающий битовые операции и команды сдвига и не зависящий от управляющего состояния FPU.

Советы и приемы

- Для использования полного диапазона точности типа Comp управляющее состояние FPU должно быть установлено в режим расширенной точности. Некоторые библиотеки Microsoft и другие DLL устанавливают управляющее состояние в режим одинарной или

двойной точности, таким образом сокращая точность типа Comp до 53 или меньшего количества бит.

- Подпрограммы ввода/вывода и форматирования из модуля SysUtils не поддерживают полный диапазон типа Comp. Значения вблизи пределов типа Comp печатаются с меньшей точностью, чем фактически хранится в переменной. Большинство приложений это ограничение не затронет.

Смотрите также

Функции CompToCurrency, CompToDouble, процедуры CurrencyToComp, DoubleToComp, типы Extended, Int64.

CompToCurrency, функция

Синтаксис

```
function CompToCurrency(Value: Comp): Currency; cdecl;
```

Описание

Функция CompToCurrency преобразует значение типа Comp в Currency. Это настоящая функция.

Советы и приемы

- Функцию CompToCurrency можно вызвать в программе, написанной на C++ Builder. Delphi автоматически преобразует Comp в Currency, и потому вызывать эту функцию в программе на Delphi не потребуется.
- Comp может хранить значения вне диапазона типа Currency, и потому преобразование Comp в Currency может привести к ошибке времени выполнения (EInvalidOp).

Смотрите также

Типы Comp, Currency, функция CompToDouble, процедура CurrencyToComp.

CompToDouble, функция

Синтаксис

```
function CompToDouble(Value: Comp): Double; cdecl;
```

Описание

CompToDouble преобразует значение Comp в Double. Это настоящая функция.

Советы и приемы

- Можно вызвать функцию CompToDouble в программе, разработанной с помощью C++ Builder. Delphi автоматически преобразует Comp в

Double, и потому вызывать эту функцию в программе на Delphi не потребуется.

- Comp может хранить значения вне диапазона типа Double, и поэтому преобразование Comp в Double может привести к ошибке времени выполнения (EInvalidOp).

Смотрите также

Типы Comp, Double, функция CompToCurrency, процедура DoubleToComp.

Concat, функция

Синтаксис

```
function Concat(const S1, S2, ....: string): string;
```

Описание

Функция Concat соединяет все строки, перечисленные в качестве аргументов, в одну строку. Она выполняет те же действия, что и оператор '+': S1 + S2 + ... Функция Concat встроена в компилятор и не является настоящей функцией. Между использованием оператора + и вызовом функции Concat нет разницы в производительности.

Смотрите также

Функции Copy, Delete, процедуры Insert, SetLength, SetString.

Const, ключевое слово

Синтаксис

```
const
  Имя = Выражение;
  Имя: Тип = Выражение;


Заголовок подпрограммы(...; const Имя: Тип; const Имя: array of const);
```

Описание

В Delphi расширен синтаксис ключевого слова const стандартного Паскаля, допуская использование в качестве значения константы любого выражения, составленного из констант, и указание конкретного типа константы.

Если указан тип, создается типизированная константа, которая фактически является не константой, а проинициализированной переменной. Жизненный цикл типизированной константы соответствует жизненному циклу программы или библиотеки, даже если типизированная константа объявлена локально в подпрограмме.

Можно также объявить параметр подпрограммы как константу или открытый массив констант. Подпрограмма не может изменить значе-

ние константного параметра, но его использование дает несколько преимуществ:

- Значение параметра изменяется прозрачно для читателя подпрограммы, что улучшает читаемость и ясность кода.
- Компилятор проверяет выполнение ограничения. Если программист случайно попытается присвоить значение const параметру, компилятор выдаст сообщение об ошибке. Заметьте, что константная ссылка на объект означает, что нельзя изменить ссылку. Сам объект не является константой.
- Передача строк, динамических массивов и интерфейсов в качестве параметров типа `const` несколько эффективнее, т. к. Delphi не приходится увеличивать счетчик ссылок в том случае, когда подпрограмма не предполагает изменение значения параметра.

Советы и приемы

- Чаще всего так называемые типизированные константы применяются для объявления переменных, лексический диапазон которых ограничен подпрограммой, но при этом значение сохраняется при всех ее вызовах.
- По умолчанию типизированные константы можно изменять. Некоторые применения типизированных констант соответствуют их прямому назначению, и можно указать директиву компилятора `$J` или `$WriteableConst`, чтобы обеспечить использование типизированных констант только в качестве постоянных.

Пример

```
// Простой пример, в котором функция NextFruitName
// возвращает имя следующего фрукта при каждом ее вызове.
// В перечисляемом типе TFruit приведен список имеющихся
// фруктов. В примере показано несколько способов использования
// констант. Первый вариант – обычная константа. Второй –
// типизированная константа, значение которой постоянно.
// Третий вариант – типизированная константа, значение
// которой изменяется, но сохраняется при последовательных
// вызовах функции NextFruitName.
type
  TFruit = (fKumquat, fMango, fStar, fStrawberry, fMarionberry);
function NextFruitName: string;
const
  InitialValue = fStar;
{$WriteableConst Off}
  FruitNames: array[TFruit] of string =
    ('Kumquat', 'Mango', 'Star Fruit', 'Strawberry', 'Marionberry');
{$WriteableConst On}
  Fruit: TFruit = InitialValue;
begin
  if Fruit = High(Fruit) then
```

```
    Fruit := Low(Fruit)
  else
    Inc(Fruit);
  Result := FruitNames[Fruit];
end;
```

Смотрите также

Тип AnsiString, ключевые слова Array, Interface, String, Var, директива Out, директивы компилятора $J, $WriteableConst.

Constructor, ключевое слово

Синтаксис

```
type Описание класса
...
  constructor Имя;
  constructor Имя(Аргументы...);
end;
```

Описание

Конструктор – это специальный тип метода. Если конструктор вызывается с указанием ссылки на класс, Delphi создает новый экземпляр этого класса, инициализирует экземпляр и затем соответствующим образом вызывает конструктор. Если при вызове указывается ссылка на объект, Delphi вызывает конструктор как обычный метод.

Естественным следствием принятых в Delphi правил является то, что конструктор может вызывать другой конструктор того же или базового класса. При этом вызове используется ссылка на объект (а именно переменная Self), так что конструктор вызывается как обычный метод.

При вызове с указанием ссылки на класс Delphi вызывает метод NewInstance для создания экземпляра класса. TObject.NewInstance выделяет память для нового объекта и заполняет эту область памяти нулями, но программист может переопределить метод NewInstance для изменения алгоритма создания объекта.

Советы и приемы

- Обычной ошибкой новичков в Delphi является вызов конструктора с помощью переменной объектного типа. Вместо создания объекта такая конструкция неизбежно приведет к ошибке нарушения доступа, т. к. наверняка ссылка на объект является неверной. Компилятор может предупреждать о таких ошибках, только в том случае, если включены предупреждения в настройках проекта. Например:

```
var
  ObjRef: TSomething;
```

```
begin
  ObjRef.Create;                  // неверно
  ObjRef := TSomething.Create;    // верно
```

- Если вы создаете абстрактный базовый класс и объявляете виртуальный конструктор, чтобы можно было создать «фабрику классов», не делайте конструктор абстрактным. Производные классы всегда вызывают унаследованный конструктор, но если он является абстрактным, это становится невозможным. Если абстрактный базовый класс ничего не делает в своем конструкторе, создайте его так, чтобы там не было ничего, кроме вызова базового конструктора. Например:

```
constructor TAbstractBaseClass.Create;
begin
  inherited Create;
end;
```

- Всегда вызывайте унаследованный конструктор. Даже если точно известно, что он ничего не делает (например, `TObject.Create`), все равно обязательно вызовите его. Накладные расходы вызова унаследованного конструктора небольшие, зато потенциальный выигрыш огромен. Представьте, например, что в будущей версии базовый класс вашего класса изменится. Конструктор нового базового класса может выполнять какие-то важные действия. Даже если базовый класс остался прежним, его конструктор может измениться и тоже выполнять какие-то действия. Изменение базового класса при последующей поддержке кода должно быть легким и не должно требовать трудоемкого просмотра всех конструкторов производных классов на предмет наличия вызова унаследованного конструктора.

Смотрите также

Ключевые слова `Class`, `Destructor`, `Inherited`, тип `TObject`.

Contains, директива

Синтаксис

```
package Имя;
contains Модуль, Модуль in Имя файла;
...
```

Описание

Директива `contains` обозначает начало списка имен модулей, из которых состоит пакет. Имена модулей разделяются запятыми, и каждый из них представлен либо только именем, либо именем модуля и именем файла в виде строки.

Советы и приемы

- Все модули, которые используются модулями пакета, должны либо находиться в самом пакете, либо в одном из пакетов, перечисленных в директиве `requires`. При компиляции пакета необходимо обеспечить Delphi возможность найти каждый из указанных модулей.
- Нельзя указать какой-либо модуль более одного раза, и ни один из модулей, перечисленных в директиве `contains`, не должен присутствовать в пакетах, перечисленных в директиве `requires`. Это правило позволяет не допустить включение модуля в проект более одного раза.
- Обычно добавление и удаление модулей из пакета выполняется с помощью редактора пакетов Delphi. Редактировать исходный текст пакета можно и вручную, но при этом окно редактора пакетов не обновляется. Закройте и откройте проект или группу проектов, чтобы среда разработки Delphi заново прочитала новый исходный текст пакета.
- IDE не всегда рассматривает пакет как проект. Если возникли проблемы с IDE, создайте новую группу проектов и добавьте в нее пакет.

Смотрите также

Ключевые слова `Package`, `Unit`, `Uses`, директива `Requires`.

Continue, процедура

Синтаксис

```
Continue;
```

Описание

Процедура `Continue` позволяет выйти из тела цикла, аналогично следующему оператору `goto`.

```
label Continue;
begin
  while Condition do
  begin
    DoSomething;
    if AnotherCondition then
      goto Continue;
    DoSomeMore;
 Continue:
  end;
```

Если процедура `Continue` вызывается из вложенного цикла, управление передается только в пределах самого внутреннего цикла. Чтобы выйти сразу из нескольких циклов, используйте оператор `goto`.

Процедура Continue встроена в компилятор и не является настоящей процедурой. При попытке вызвать Continue вне цикла компилятор выдаст ошибку.

Смотрите также

Процедура Break, ключевые слова For, Repeat, While.

Copy, функция

Синтаксис

```
function Copy(Source: string; StartingIndex, Count: Integer): string;
function Copy(Source: array; StartingIndex, Count: Integer): array;
```

Описание

Функция Copy создает копию части строки или динамического массива. Результатом является новая строка или динамический массив. Первый элемент новой строки или массива имеет номер StartingIndex в Source. Новая строка или массив содержат не более Count элементов. Если Count превышает количество элементов, оставшихся в исходной строке или массиве, копируются элементы от StartindexIndex до конца строки или массива.

Copy не является настоящей функцией.

Советы и приемы

- Если StartingIndex меньше нуля или выходит за границы Source, результатом будет пустая строка или массив.
- Длина новой строки или массива равна Count, если только в Source не остается меньше чем Count элементов для копирования. В последнем случае копируются все элементы до конца строки или массива.
- Первый элемент строки имеет номер 1. Первый элемент динамического массива имеет номер 0.
- Удобный способ скопировать все элементы Source, начиная с определенного начального элемента, – это указать в качестве Count константу MaxInt.

Смотрите также

Функции Delete, High, Length, Low, Slice, процедуры Insert, SetLength, SetString.

Cos, функция

Синтаксис

```
function Cos(Number: Вещественный тип): Extended;
```

Описание

Функция Cos вычисляет и выдает в качестве результата косинус Number, являющийся величиной угла в радианах.

Советы и приемы

- Delphi автоматически преобразует аргументы типа Integer и Variant в вещественные числа. Чтобы преобразовать аргумент типа Int64 в вещественное, прибавьте к нему 0.0.
- Если Number равно «сигнализирующему» NaN, плюс бесконечности или минус бесконечности, возникает ошибка времени выполнения 6 (EInvalidOp).
- Если Number равно «спокойному» NaN, результат равен Number.

Смотрите также

Функции ArcTan, Sin.

Currency, тип

Синтаксис

```
type Currency;
```

Описание

Тип Currency – это 64-битовый тип с фиксированной точкой, с четырьмя знаками после десятичной точки. Тип Currency использует FPU, рассматривающий значение Currency как целое. Ввод и вывод значений типа Currency выполняется в формате с четырьмя цифрами после запятой, вычислемые значения Currency умножаются на 10000 для учета цифр после запятой.

Советы и приемы

- Для реализации полной 64-битной точности типа Currency управляющее состояние FPU должно быть установлено в режим расширенной точности. Некоторые библиотеки Microsoft и другие DLL устанавливают управляющее состояние в режим одинарной или двойной точности, таким образом сокращая точность типа Currency до 53 или меньшего количества бит.
- Подпрограммы ввода/вывода и форматирования из модуля SysUtils не поддерживают полный диапазон типа Currency. Значения вблизи границ типа Currency печатаются с меньшей точностью, чем фактически хранится в переменной. Большинство приложений это ограничение не затрагивает.

Смотрите также

Типы Extended, Int64, процедура Set8087CW.

CurrencyToComp, процедура

Синтаксис

```
procedure CurrencyToComp(Value: Currency; var result: Comp); cdecl;
```

Описание

Функция CurrencyToComp преобразует значение типа Currency в Comp. Это настоящая функция.

Советы и приемы

- Функция CurrencyToComp может вызываться в программе на C++ Builder. Delphi при необходимости автоматически преобразует Currency в Comp, так что вызывать эту функцию в программе на Delphi не потребуется.
- Преобразование Currency в Comp может привести к потере точности.

Смотрите также

Типы Comp, Currency, функция CompToCurrency.

DataMark, переменная

Синтаксис

```
unit SysInit;
var DataMark: Integer;
```

Описание

DataMark – это первая переменная сегмента данных модуля. Ее адрес – это виртуальный адрес начала доступных для записи данных.

DebugHook, переменная

Синтаксис

```
var DebugHook: Byte;
```

Описание

С помощью переменной DebugHook отладчик Delphi определяет, что делать с исключительными ситуациями. Когда приложение работает нормальным образом, DebugHook равна 0. Если приложение отлаживается в IDE Delphi, IDE устанавливает DebugHook в 1. Когда IDE перехватывает исключительную ситуацию, или когда программа выполняется в отладчике в пошаговом режиме, IDE устанавливает DebugHook в 2.

Смотрите также

Переменная JITEnable.

Dec, процедура

Синтаксис

```
procedure Dec(var Variable);
procedure Dec(var Variable; Count: Integer);
```

Описание

Процедура Dec уменьшает значение переменной любого порядкового типа или указателя, но ее нельзя использовать для уменьшения значения вещественной переменной.

Уменьшение значения указателя изменяет его значение на величину, равную размеру базового типа. Например, значение указателя на AnsiChar уменьшается на 1, а указателя на WideChar – на 2.

По умолчанию переменная уменьшается на одну единицу, однако можно указать целый параметр для изменения на другую величину. При уменьшении указателя значение переменной Count умножается на размер базового типа.

Процедура Dec – встроенная и не является настоящей процедурой.

Советы и приемы

- Значение переменной Count может быть отрицательным, в этом случае оно увеличивается.
- Между Dec, Pred и операцией вычитания разница производительности незначительна. То есть следующие конструкции компилируются в похожий объектный код:

```
Dec(X);
X := Pred(X);
X := X - 1;     // Если X – целое или PChar
```

- Вместо переменной нельзя использовать свойство, т. к. значение свойства не может выступать в качестве параметра var. Следует предпочесть простое присваивание.

Смотрите также

Функции High, Low, Pred, Succ, процедура Inc.

Default, директива

Синтаксис

```
property Описание; default;
property Описание default Ordinal constant;
```

Описание

Директива default имеет два применения при описании свойства:

- Объявление свойства-массива свойством по умолчанию.
- Указание значения по умолчанию для свойства порядкового типа.

В классе одно из свойств-массивов может быть объявлено как свойство по умолчанию. Свойство по умолчанию позволяет обращаться к ссылке на объект как к массиву без указания имени свойства. Например, в классе `TList` свойство `Items` объявлено как свойство по умолчанию для данного класса. Если `List` – это переменная типа `TList`, то можно ссылаться на элемент списка как `List[Index]`, что является сокращением записи `List.Items[Index]`. В классе может быть только одно свойство по умолчанию. Производные классы могут определить свойство по умолчанию отличное от родительского класса.

Свойства порядкового типа (целые, перечисления, множества или символы) могут иметь значение по умолчанию. Значение по умолчанию не влияет на исходное значение свойства. Оно хранится в RTTI класса, и среда разработки и исполнения Delphi использует значение по умолчанию для сокращения размера *dfm*-файла или ресурса. Для непорядковых свойств (например, вещественных, строк, классов) значением по умолчанию всегда является ноль (пустая строка, `nil` и т. п.).

Советы и приемы

- Если директива `default` отсутствует, Delphi подразумевает директиву `nodefault` и в качестве значения по умолчанию для данного свойства устанавливает наименьшее целое (–2,147,483,648). Побочным эффектом этого правила является то, что нельзя использовать это значение в качестве значения по умолчанию. Иначе Delphi будет интерпретировать его как `nodefault`.
- Поскольку директива `default` отражается только на RTTI класса, конструктор должен сам устанавливать значение по умолчанию для каждого свойства. Можно прочитать значения по умолчанию из RTTI и установить соответствующим образом значения всех свойств, как показано ниже:

```
// Установка значения по умолчанию свойств объекта по значениям
// из RTTI. Вы можете вызвать эту процедуру из конструктора
// любого класса, использующего директиву default.
//
procedure SetDefaultValues(Obj: TObject);
const
  tkOrdinal = [tkEnumeration, tkSet, tkInteger, tkChar, tkWChar];
  NoDefault = Low(Integer);
var
  Count, I: Integer;
  PropList: PPropList;
begin
  // Убедитесь в том, что класс имеет публикуемые свойства и
  // RTTI.
  if (Obj <> nil) and (Obj.ClassInfo <> nil) then
```

```
  begin
    // Определяем количество свойств порядкового типа
    Count := GetPropList(Obj.ClassInfo, tkOrdinal, nil);
    GetMem(PropList, Count * SizeOf(PPropInfo));
    try
      // Извлекаем информацию о свойствах порядковых типов.
      GetPropList(Obj.ClassInfo, tkOrdinal, PropList);
      for I := 0 to Count-1 do
        // Инициализируем все свойства, которые имеют значения по умолчанию,
        // этими значениями.
        if PropList[I].Default <> NoDefault then
          SetOrdProp(Obj, PropList[I], PropList[I].Default);
    finally
      FreeMem(PropList);
    end;
  end;
end;
```

Пример

Смотрите примеры использования директивы `default` в описании ключевого слова `property`.

Смотрите также

Ключевые слова `Class`, `Dispinterface`, `Interface`, `Property`, директивы `Nodefault`,`Read`, `Stored`, `Write`.

Default8087CW, переменная

Синтаксис

```
var Default8087CW: Word;
```

Описание

Delphi устанавливает управляющее состояние FPU равным значению `Default8087CW`. Управляющее состояние можно установить с помощью процедуры `Set8087CW`, и Delphi автоматически сохранит новое значение управляющего состояния в `Default8087CW`.

Заметьте, что некоторые DLL изменяют управляющее состояние FPU. Вы можете вызвать процедуру `SysUtils.SafeLoadLibrary`, которая гарантирует, что управляющее состояние FPU будет восстановлено после загрузки библиотеки и останется равным `Default8087CW`.

Обратитесь к описанию процедуры `Set8087CW` за информацией об управляющем состоянии FPU.

Смотрите также

Процедура `Set8087CW`.

Delete, процедура

Синтаксис

```
procedure Delete(var Str: string; StartingIndex, Count: Integer);
```

Описание

Процедура Delete удаляет Count символов из строки, начиная с номера StartingIndex. Если Count превышает количество символов, оставшихся в строке, Delete удаляет весь остаток строки, начиная с номера StartingIndex. Delete не является настоящей процедурой.

Советы и приемы

- Первый символ в строке имеет номер 1.
- Если StartingIndex не положителен или больше длины строки, Delete не изменяет содержимое строки.
- Если Count больше, чем количество символов после StartingIndex, удаляется весь остаток строки.
- Для удаления символов в конце строки можно воспользоваться процедурой SetLength, которая работает быстрее, чем более универсальная процедура Delete.

Пример

```
// Удаление буквы устройства из начала строки пути.
procedure RemoveDriveLetter(var Path: string);
begin
  // Сначала убедимся, что буква устройства присутствует
  // в начале строки
  if (Length(Path) >= 2) and (Path[2] = ':') then
    // Удаляем два первых символа.
    Delete(Path, 1, 2);
end;
```

Смотрите также

Функция Copy, процедуры Insert, SetLength, SetString.

Destructor, ключевое слово

Синтаксис

```
type Объявление класса
  destructor Destroy; override;
end;
```

Описание

Деструктор уничтожает объект и освобождает память, занятую им. Деструктор имеет скрытый параметр, равный 1, если деструктор вызывается из обычного метода, что сообщает деструктору о необхо-

димости вызова процедуры BeforeDestruction, затем собственно деструктора и, наконец, процедуры FreeInstance для освобождения памяти объекта. Когда деструктор вызывает унаследованный деструктор, скрытый параметр равен нулю.

Советы и приемы

- Для уничтожения объекта вызывайте функцию Free. Не пользуйтесь Destroy, т. к. Free проверяет, не равна ли ссылка на объект нулю и вызывает Destroy только для ненулевой ссылки.
- Хотя и допускается объявление деструктора класса с любым именем и аргументами, следует объявить один деструктор с именем Destroy. Поскольку Destroy объявлен как виртуальный метод класса TObject, при его объявлении следует указать директиву override в описании класса.
- Причина объявления метода Destroy с директивой override в том, что деструктор часто вызывается полиморфным образом – когда объявленный тип ссылки на объект может отличаться от фактического класса объекта.
- Delphi автоматически вызывает Destroy, если в конструкторе возникла исключительная ситуация. Поэтому следует применять защищенный стиль программирования деструктора. Не все поля могут быть проинициализированы в момент вызова деструктора, и потому всегда проверяйте их на ноль или значение nil. Заметьте, что процедуры Free, FreeMem и Dispose автоматически проверяют значение ссылки на nil перед освобождением объекта или участка памяти.
- Если освобождается объект, ссылка на который хранится в глобальной переменной или в поле другого объекта, вызывайте вместо метода Free процедуру SysUtils.FreeAndNil.

Смотрите также

Ключевые слова Class, Constructor, тип TObject.

DispCallByIDProc, переменная

Синтаксис

```
var DispCallByIDProc: Pointer;

type
  PDispRec = ^TDispRec;
  TCallDesc = packed record
    CallType: Byte;
    ArgCount: Byte;
    NamedArgCount: Byte;
    Args: array[0.255] of Byte;
  end;
```

```
  TDispDesc = packed record
    Dispid: Integer;
    ResultType: Byte;
    CallDesc: TCallDesc;
  end;
procedure CallProc(Result: Pointer; const Dispatch: IDispatch;
  DispDesc: PDispDesc; Params: Pointer); cdecl;

DispCallByIDProc := @CallProc;
```

Описание

Delphi вызывает процедуру DispCallByIDProc для вызова метода-диспетчера, идентифицируемого по номеру в описании диспетчерского интерфейса, и заполняет запись DispDesc необходимой информацией о номере диспетчера и формальных параметрах метода. Params является указателем на фактические параметры, указанные при вызове метода. Процедура должна записать результат в буфер, на который указывает Result. Тип буфера Result определяется дескриптором диспетчера. По умолчанию DispCallByIDProc – это процедура, вызывающая ошибку времени выполнения 17 (EVariantError).

Советы и приемы

- Модуль ComObj устанавливает DispCallByIDProc на процедуру, которая реализует код, необходимый для вызова IDispatch.Invoke. Кроме того, в ComObj объявлены типы TCallDesc и TDispDesc.
- Написание собственной процедуры DispCallByIDProc – серьезное мероприятие. Чтобы узнать, как это делать, изучите исходный текст модуля *ComObj.pas*.

Смотрите также

Директивы Cdecl, Dispid, ключевое слово Dispinterface, интерфейс IDispatch, переменная VarDispProc.

Dispid, директива

Синтаксис

```
Объявление Dispinterface
  Описание метода; dispid Целая константа;
  Описание свойства dispid Целая константа;
  ...
end;
```

Описание

По умолчанию Delphi самостоятельно присваивает номера диспетчеров методам и свойствам в описании диспетчерского интерфейса, но их можно указать явно с помощью директивы dispid. Не допускается

указание в директиве dispid индекса, который уже был использован в данном диспетчерском интерфейсе.

Пример

Пример смотрите в описании ключевого слова Dispinterface.

Смотрите также

Ключевое слово Dispinterface, интерфейс IDispatch.

Dispinterface, ключевое слово

Синтаксис

```
type Имя = dispinterface
  ['{Guid...}']
  Описания методов и свойств...
end;
```

Описание

Ключевое слово dispinterface аналогично ключевому слову interface, предназначенному для объявления интерфейсного типа, но объявляет диспетчерский интерфейс. В Delphi диспетчерский интерфейс служит для доступа к серверу автоматизации COM через метод Invoke интерфейса IDispatch. Класс не может реализовывать диспетчерский интерфейс, а диспетчерский интерфейс не может быть потомком другого интерфейса (ни обычного, ни диспетчерского).

В описании интерфейса могут быть указаны идентификаторы диспетчеров с помощью директивы dispid. Все идентификаторы одного интерфейса должны быть уникальными. Никакие другие директивы методов не разрешаются.

Объявления свойств не могут содержать спецификаторы read и write. По умолчанию свойства разрешают доступ на чтение и на запись, но допускается объявление свойства с директивой readonly или writeonly. Свойство-массив может содержать директиву default. Никакие другие директивы для свойств не разрешаются.

Советы и приемы

- Объявление dispinterface предназначено для обеспечения доступа к диспетчерскому интерфейсу на этапе компиляции, чтобы неверное написание имени метода не было выявлено во время выполнения приложения. Автор COM-сервера должен предоставить объявление диспетчерского интерфейса для своего COM-объекта. При наличии библиотеки типов можно воспользоваться редактором библиотек типов Delphi для генерирования исходного кода и объявления dispinterface.
- COM-сервер может реализовывать обычный интерфейс, диспетчерский интерфейс или оба. Обратитесь к библиотеке типов, чтобы

узнать, какой тип интерфейса поддерживается сервером. Обычный интерфейс быстрее, чем диспетчерский, но если вы вынуждены использовать последний, это все равно удобнее, чем применять для доступа к COM-объекту переменную типа Variant.

- COM-сервер создает объект, реализующий диспетчерский интерфейс, с помощью «фабрики классов» или чего-либо аналогичного, например, процедуры CreateComObject (из модуля ComObj). Точное описание процесса создания COM-объекта приведено в его документации.

Пример

```
// Ниже приведен диспетчерский интерфейс IStrings из stdvcl.pas.
// Когда вы создаете управляющий элемент ActiveX
// из визуального компонента Delphi
// ActiveX IStrings используется как COM-интерфейс к свойствам
// типа TStrings, таких как TMemo.Lines или TListView.Items.

type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0;
        default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

Смотрите также

Директивы Default, Dispid, Readonly, Writeonly, ключевое слово Interface.

Dispose, процедура

Синтаксис

```
procedure Dispose(var P: Тип указателя);
procedure Dispose(var P: ^object; Деструктор);
```

Описание

Память, выделенная с помощью процедуры New, должна быть освобождена процедурой Dispose, которая игнорирует попытку освободить нулевой указатель. Если вы попытаетесь освободить указатель, который уже был освобожден или не был выделен с помощью процедуры New, результат непредсказуем. Dispose не является настоящей процедурой.

При освобождении экземпляра «старого» объекта также можно указать в качестве второго аргумента деструктор. Используйте имя деструктора и любое количество необходимых аргументов.

Советы и приемы

- До освобождения памяти Delphi вызывает процедуру `Finalize`.
- После окончания работы `Dispose` указатель `P` содержит недопустимую величину. Если указатель не является локальной переменной, обязательно обнулите его.
- Освобождайте память, выделенную процедурой `GetMem`, с помощью процедуры `FreeMem`. Используйте процедуру `Dispose` только для памяти, выделенной процедурой `New`.

Пример

```
type
  PLink = ^TLink;
  TLink = record
    Info: string;
    Next: PLink;
    Previous: PLink;
  end;
const
  FreePattern = Pointer($BADOOBAD);

// Уничтожение связанного элемента двусвязного списка.
procedure FreeLink(var Link: PLink);
var
  Tmp: PLink;
begin
  if Link.Previous <> nil then
    Link.Previous.Next := Link.Next;
  if Link.Next <> nil then
    Link.Next.Previous := Link.Previous;

  // Ссылка на Link или Link.Next и т. п. после уничтожения Link
  // будет ошибкой. Облегчим поиск подобных ошибок, записав
  // в Link определенное фиктивное значение. Если программа вызывает
  // исключительную ситуацию, и ошибочный указатель равен этому
  // значению, проблема наверняка вызвана «висящей» ссылкой на
  // Link.


  Link.Next := FreePattern;
  Link.Previous := FreePattern;
  Tmp := Link;
  Link := FreePattern;
  Dispose(Tmp);
end;
```

Смотрите также

Процедуры Finalize, FreeMem, New, переменная IsMultThread, ключевое слово Object.

Div, ключевое слово

Синтаксис

```
Делимое div Делитель
```

Описание

Оператор div выполняет целочисленное деление, при котором дробная часть отбрасывается без округления. Если делитель равен нулю, Delphi выдает ошибку времени выполнения 3 (EDivByZero).

Смотрите также

Ключевое слово mod, оператор /.

DllProc, переменная

Синтаксис

```
unit SysInit;
var DllProc: Pointer;

procedure DllHandler(Reason: Integer);
begin ... end;
DllProc := @DllHandler;
```

Описание

Загружая или выгружая DLL, запуская или останавливая поток, Windows запускает процедуру DllProc из DLL, передавая причину вызова в качестве аргумента Reason. Windows обычно вызывает процедуру DLL, когда библиотека впервые загружается процессом, но Delphi при обработке этого события инициализирует все модули библиотеки и исполняет блок из файла проекта библиотеки. Таким образом, вызов DllProc выполняется только для оставшихся трех случаев, т. е. Reason может быть равен одной из следующих констант (объявленных в модуле Windows):

Dll_Thread_Attach

: Процесс, который загрузил библиотеку, создал новый поток. Если к библиотеке подключено более одного потока, не забудьте установить переменную IsMultiThread в True.

Dll_Thread_Detach

: Поток прерван.

`Dll_Process_Detach`

Процесс остановлен или DLL выгружается.

Советы и приемы

- Windows вызывает процедуру `DllProc` в контексте нового потока, поэтому с помощью переменных `threadvar` можно присоединять DLL к новому потоку. Однако не используйте эти переменные при отсоединении. Delphi очищает все локальные переменные потока до вызова `DllProc`.
- Событие `Dll_Process_Detach` выгоднее всего применять, если процедура `DllProc` находится в файле проекта библиотеки. Для модуля можно использовать раздел завершения.

Пример

```
// Отслеживает потоки приложения. Сохраняются идентификаторы
// всех потоков, однако вы можете легко модифицировать этот
// пример для сохранения другой информации.
library ThreadMonitor;

uses Windows, SysUtils, Classes;

var
  Threads: TThreadList;

procedure DllHandler(Reason: Integer);
begin
  case Reason of
  Dll_Thread_Attach:
    with Threads.LockList do
      try
        IsMultiThread := Count >= 1;
        Add(Pointer(GetCurrentThreadID));
      finally
        Threads.UnlockList;
      end;
  Dll_Thread_Detach:
    with Threads.LockList do
      try
        Remove(Pointer(GetCurrentThreadID));
        IsMultiThread := Count <= 1;
      finally
        Threads.UnlockList;
       end;
  Dll_Process_Detach:
    FreeAndNil(Threads);

  // Либо в новой версии Windows придумали новые события. Игнорируем их.
  end;
end;
```

```
begin
  Threads := TThreadList.Create;
  DllProc := @DllHandler;
end.
```

Смотрите также

Ключевые слова Finalization, Initialization, Library, Threadvar, переменная IsMultiThread.

Do, ключевое слово

Синтаксис

```
for Переменная := Выражение to Выражение do Предложение

while Выражение do Предложение

try Предложение... except on Класс исключительной ситуации
  do Предложение end;

with Выражение do Предложение
```

Описание

Ключевое слово do является частью конструкций for и while, обработчика исключительной ситуации try-except и предложения with языка Delphi. Подробности смотрите в описаниях этих конструкций.

Смотрите также

Ключевые слова For, Try, While, With.

Double, тип

Синтаксис

```
type Double;
```

Описание

Тип Double – тип с плавающей точкой, соответствующий стандарту IEEE, который занимает 8 байт: знаковый бит, 11 бит экспоненты и 52 бита мантиссы. Мантисса обычно нормализована, т. е. имеет незначащий бит со значением 1 перед старшим битом. Если, однако, экспонента равна нулю, мантисса денормализуется и сохраняется без неявного бита. Таким образом, значение +0,0 представлено нулевыми битами. Экспонента со всеми битами, равными 1, представляет бесконечность (мантисса равна нулю) или нечисловое значение (NaN, мантисса не равна нулю).

Границы значений типа Double – это примерно от $2{,}23 \times 10^{-308}$ до $1{,}79 \times 10^{308}$ с точностью около 15 десятичных знаков. В таблице 5-1 приведен формат конечных и специальных значений Double.

Табл. 5-1. Формат вещественных чисел Double

Класс значений	Знак	Биты экспоненты	Биты мантиссы
Положительные значения			
Нормализованные	0	От 0…1 до 1…10	От 0…0 до 1…1
Денормализованные	0	0…0	От 0…1 до 1…1
Ноль	0	0…0	0…0
Бесконечность	0	1…1	0…0
«Сигнализирующий» NaN	0	1…1	От 0…1 до 01…1
«Спокойный» NaN	0	1…1	От 1…0 до 1…1
Отрицательные значения			
Нормализованные	1	От 0…1 до 1…10	От 0…0 до 1…1
Денормализованные	1	0…0	От 0…1 до 1…1
Ноль	1	0…0	0…0
Бесконечность	1	1…1	0…0
«Сигнализирующий» NaN	1	1…1	От 0…1 до 01…1
«Спокойный» NaN	1	1…1	От 1…0 до 1…1

Советы и приемы

- Double – это популярный тип, обеспечивающий хороший баланс между производительностью и точностью.
- Тип Double соответствует типу double в Java, C и C++.
- В документации по архитектуре Intel (например, «Pentium Developer's Manual», том 3, «Architecture and Programming Manual») или в описании стандарта IEEE 754 можно найти более подробную информацию о бесконечности и NaN (нечисловое значение). В Delphi использование NaN вызывает ошибку времени выполнения 6 (EInvalidOp).

Пример

```
type
  TDouble = packed record
    case Integer of
    0: (Float: Double;);
    1: (Bytes: array[0..7] of Byte;);
    2: (Words: array[0..3] of Word;);
```

```
    3: (LongWords: array[0..1] of LongWord;);
    4: (Int64s: array[0..0] of Int64;);
  end;
  TFloatClass = (fcPosNorm, fcNegNorm, fcPosDenorm, fcNegDenorm,
            fcPosZero, fcNegZero, fcPosInf, fcNegInf, fcQNaN, fcSNan);

// Возвращает класс вещественного числа: конечное, бесконечное,
// нечисловое; положительное или отрицательное, нормализованное или
// денормализованное. Класс определяется путем проверок
// экспоненты, знакового бита и мантиссы.
function fp_class(X: Double): TFloatClass; overload;
var
  XParts: TDouble absolute X;
  Negative: Boolean;
  Exponent: Word;
  Mantissa: Int64;
begin
  Negative := (XParts.LongWords[1] and $80000000) <> 0;
  Exponent := (XParts.LongWords[1] and $7FF00000) shr 20;
  Mantissa :=  XParts.Int64s[0] and $000FFFFFFFFFFFFF;

  // Три первых варианта могут быть положительными или
  // отрицательными. Предположим, что значение положительно, и
  // проверим знаковый бит позднее.
  if (Mantissa = 0) and (Exponent = 0) then
    // Мантисса и экспонента равны нулю, следовательно, число равно
    // нулю.
    Result := fcPosZero
  else if Exponent = 0 then
    // Если экспонента равна нулю, а мантисса не равна нулю,
    //значит, число конечное, но денормализированное.
    Result := fcPosDenorm
  else if Exponent <> $7FF then
    // Иначе, если экспонента не целиком состоит из единиц, число
    // нормализовано.
    Result := fcPosNorm
  else if Mantissa = 0 then
    // Экспонента состоит из 1, мантисса из 0, следовательно,
    // бесконечность.
    Result := fcPosInf

  else
  begin
    // Экспонента состоит из 1, мантисса – ненулевая, следовательно,
    // значение нечисловое. Проверяем, «спокойный» или
    // «сигнализирующий» NaN.
    if (Mantissa and $8000000000000) <> 0 then
      Result := fcQNaN
    else
      Result := fcSNaN;
    Exit; // Не выделяем отрицательный NaN.
```

```
    end;

    if Negative then
      Inc(Result);
  end;
```

Смотрите также

Функция CompToDouble, процедура DoubleToComp, типы Extended, Real, Single.

DoubleToComp, процедура

Синтаксис

```
procedure DoubleToComp(Value: Double; var result: Comp); cdecl;
```

Описание

DoubleToComp преобразует значение типа Double в Comp. Это настоящая процедура.

Советы и приемы

- Процедуру DoubleToComp можно вызвать из программы, созданной на C++ Builder. Delphi при необходимости автоматически преобразует Double в Comp, и вызывать эту функцию в программе на Delphi не потребуется.
- Double поддерживает более широкий диапазон значений, чем Comp. Преобразование Double в Comp может вызвать ошибку времени выполнения 6 (EInvalidOp).

Смотрите также

Типы Comp, Double функция CompToDouble.

Downto, ключевое слово

Синтаксис

```
for Variable := Expression downto Expression do Statement
```

Описание

Downto используется в цикле for для реализации обратного отсчета. Подробности смотрите в описании цикла for.

Смотрите также

Ключевые слова For, To.

Dynamic, директива

Синтаксис

```
Описание метода; dynamic;
```

Описание

Можно объявить виртуальный метод в базовом классе с помощью директивы `virtual` или `dynamic`. Семантика обеих директив одинакова, различие состоит только в реализации метода и способе его вызова. Подробная информация приведена в главе 3.

Так же как и для виртуального метода, производные классы должны использовать директиву `override` для переопределения метода.

Советы и приемы

- Вместо `dynamic` почти всегда следует использовать директиву `virtual`. В большинстве случаев виртуальные методы работают быстрее и занимают меньше памяти, чем динамические.
- Директива `dynamic` должна следовать за директивами `reintroduce` и `overload` и предшествовать директивам соглашений о вызовах и директиве `abstract` (если эти директивы присутствуют в объявлении метода).

Смотрите также

Директивы `Abstract`, `Overload`, `Override`, `Reintroduce`, `Virtual`, тип `Class`.

Else, ключевое слово

Синтаксис

```
if Условие then Предложение else Предложение

try Предложение... except Исключительные ситуации...
else Предложения... end;

case Выражение of Селекторы case... else Предложения... end;
```

Описание

Ключевое слово `else` обозначает начало универсальной части нескольких языковых конструкций. Подробности рассмотрены в описаниях соответствующих предложений.

Заметьте, что в `else` части конструкции `if` может быть только одно предложение, а в `else` части `try-except` и `case` – несколько.

Смотрите также

Ключевые слова `Case`, `If`, `Try`.

EmptyParam, переменная

Синтаксис

```
var EmptyParam: OleVariant;
```

Описание

Некоторые COM-серверы имеют дополнительные параметры, которые можно опустить при вызове метода. В Delphi, однако, объявления интерфейса и диспетчерского интерфейса не поддерживают дополнительные параметры. Можно вызвать метод с дополнительными параметрами с помощью значения типа Variant или передав в качестве параметра EmptyParam. Следует предпочесть интерфейс или диспетчерский интерфейс (если он есть) типу Variant, т. к. при этом можно воспользоваться преимуществами проверок на этапе компиляции.

Значение EmptyParam по умолчанию равно varError с кодом ошибки, равным Disp_E_ParamNotFound. Значение можно изменить, но на это должна быть веская причина.

Пример

COM-объект ShellWindows имеет метод Item, которому требуется дополнительный параметр Index. Delphi реализует дополнительный параметр объявлением двух перегруженных методов. Реализация одного из методов передает в качестве значения Index EmptyParam:

```
// Ниже приведен фрагмент shdocvw.pas
function  TShellWindows.Item: IDispatch;
begin
  Result := DefaultInterface.Item(EmptyParam);
end;

function  TShellWindows.Item(index: OleVariant): IDispatch;
begin
  Result := DefaultInterface.Item(index);
end;
```

Смотрите также

Ключевые слова Dispinterface, Interface, тип OleVariant.

End, ключевое слово

Синтаксис

```
asm Предложения... end;
begin Предложения... end;
case Выражение of Селекторы Case... else Предложения... end;
library Имя; ... end.
package Имя; requires Пакеты... contains Модули... end.
program Имя; ... end.
```

```
try Предложения... except Исключительные ситуации...
else Предложения... end;
try Предложения... finally Предложения... end;
type Имя = class Описания... end;
type Имя = object Описания... end;
type Имя = record Описания... end;
type Имя = interface Описания... end;
type Имя = dispinterface Описания... end;
unit Имя; interface ... implementation ... end.
```

Описание

Ключевое слово end завершает практически все, что состоит из нескольких частей: блоки, описания классов, конструкции case и т. п. Подробности можно найти в описаниях каждого из этих ключевых слов.

Смотрите также

Ключевые слова Asm, Begin, Case, Class, Dispinterface, Interface, Library, Object, Program, Record, Try, Unit, директива Package.

EnumModules, процедура

Синтаксис

```
procedure EnumModules(Func: TEnumModuleFuncLW; Data: Pointer);
```

Описание

Процедура EnumModules вызывает пользовательскую функцию обратного вызова (callback function) для каждого исполняемого модуля приложения, т. е. для исполняемого модуля, для каждого пакета и DLL, которые загружаются приложением.

Пользовательская функция получает два параметра: дескриптор экземпляра каждого модуля и Data. Функция возвращает истину для продолжения перечисления модулей и ложь, чтобы остановить работу процедуры EnumModules и вызвать ее немедленное завершение.

EnumModules – настоящая процедура.

Советы и приемы

Для обратной совместимости EnumModules перегружена и может использовать функцию обратного вызова, которая объявлена с параметром HInstance типа Integer. Однако в новых программах следует предпочесть тип LongWord или, если речь идет о модуле Windows, THandle.

Пример

```
// Загружаем список модулей в строковый список. Data должна
// быть ссылкой на TStrings.
function GetModules(Instance: THandle; Data: Pointer): Boolean;
var
```

```
  FileName: array[0..Max_Path] of Char;
begin
  if GetModuleFileName(Instance, FileName, SizeOf(FileName)) = 0 then
    TStrings(Data).Add(FileName)
  else
    RaiseLastWin32Error;
  Result := True;
end;
...
EnumModules(GetModules, ListBox1.Items);
```

Смотрите также

Процедуры EnumResourceModules, RegisterModule, UnregisterModule, функции FindClassHInstance, FindHInstance, переменная LibModuleList, типы PLibModule, TEnumModuleFuncLW, TLibModule.

EnumResourceModules, процедура

Синтаксис

```
procedure EnumResourceModules(Func: TEnumModuleFuncLW; Data: Pointer);
```

Описание

Процедура EnumResourceModules вызывает пользовательскую функцию обратного вызова для каждого модуля ресурсов приложения, т. е. динамических библиотек ресурсов для исполняемого модуля, всех пакетов и DLL, загружаемых приложением.

Пользовательская функция получает два аргумента: дескриптор экземпляра каждого из модулей ресурсов и параметр Data. Функция возвращает истину для продолжения перечисления модулей и ложь, чтобы остановить работу процедуры EnumResourceModules и вызвать ее немедленное завершение.

EnumResourceModules – настоящая процедура.

Советы и приемы

- Для обратной совместимости EnumResourceModules перегружена и может использовать функцию обратного вызова, которая объявлена с параметром HInstance типа Integer. Однако в новых программах следует предпочесть тип LongWord или, если речь идет о модуле Windows, THandle.
- Если модуль не имеет отдельного файла ресурсов, EnumResourceModules использует дескриптор экземпляра исполняемого модуля.

Смотрите также

Процедуры EnumModules, RegisterModule, UnregisterModule, функции FindClassHInstance, FindHInstance, FindResourceHInstance, LoadResourceModule, тип TEnumModuleFuncLW.

Eof, функция

Синтаксис

```
function Eof(var F: File): Boolean;
function Eof(var F: TextFile): Boolean;
```

Описание

Функция Eof возвращает истину, если позиция файла F находится в конце файла. Eof не является настоящей функцией.

Ошибки

- Если файл F не был открыт, Eof выдает ошибку ввода/вывода 103.
- Файл должен быть открыт процедурой Reset. Если файл был открыт с помощью Rewrite или Append, Eof выдает ошибку ввода/вывода 104.

Смотрите также

Функции Eoln, IOResult, ключевое слово File, процедуры Reset, SeekEof, Truncate, тип TextFile.

Eoln, функция

Синтаксис

```
function Eoln(var F: TextFile): Boolean;
```

Описание

Функция Eoln возвращает истину, если позиция текстового файла F находится в конце строки или в конце файла. Интерпретирует в качестве конца страницы «возврат каретки» (#13). Eoln не является настоящей функцией.

Ошибки

- Если файл F не был открыт, Eoln выдает ошибку ввода/вывода 103.
- Файл должен быть открыт процедурой Reset. Если файл был открыт с помощью Rewrite или Append, функция Eoln выдает ошибку ввода/вывода 104.

Смотрите также

Функции Eof, IOResult, процедуры Reset, SeekEoln, тип TextFile.

Erase, процедура

Синтаксис

```
procedure Erase(var F: File);
procedure Erase(var F: TextFile);
```

Описание

Процедура Erase удаляет файл, назначенный переменной F. Файл должен быть закрыт, но ему должно быть присвоено имя процедурой AssignFile или Assign. Не является настоящей процедурой.

Ошибки

- Если AssignFile не была вызвана, Erase генерирует ошибку ввода/вывода 102.
- Если файл не может быть удален, выполнение процедуры Erase вызовет ошибку ввода/вывода с кодом ошибки Windows. В частности, если файл уже открыт, код ошибки Windows будет равен 32 (файл используется), если файл доступен только для чтения, код ошибки Windows равен 5 (доступ запрещен).

Смотрите также

Процедуры AssignFile, CloseFile, Rename, ключевое слово File, функция IOResult, тип TextFile.

ErrorAddr, переменная

Синтаксис

```
var ErrorAddr: Pointer;
```

Описание

Когда Delphi генерирует ошибку времени выполнения, в ErrorAddr записывается указатель на точку возникновения ошибки в исполняемом коде. Этот адрес используется в сообщениях об ошибках как стартовая точку для отслеживания стека вызовов и для других целей.

Процедура RunError устанавливает ErrorAddr и затем вызывает процедуру Halt, которая и генерирует сообщение об ошибке, если ErrorAddr не равен nil.

Смотрите также

Переменные ErrorProc, NoErrMsg, процедуры Halt, RunError.

ErrorProc, переменная

Синтаксис

```
var ErrorProc: Pointer;

procedure MyErrorProc(ErrorCode: Integer; ErrorAddr: Pointer);
begin
  ...
end;
ErrorProc := @MyErrorProc;
```

Описание

Если возникает ошибка времени выполнения, и переменная ErrorProc не равна nil, Delphi вызывает процедуру, на которую указывает эта переменная. Процедуре требуются два аргумента: номер ошибки и ее адрес.

Если ErrorProc равна nil, Delphi вызывает RunError для выдачи сообщения об ошибке и остановки программы. В графическом приложении Delphi отображает сообщение об ошибке в диалоговом окне. В консольном приложении Delphi выводит сообщение на консольное устройство. Коды ошибок ввода/вывода хранятся в Delphi отдельно. Если ErrorCode равен нулю, скорее всего возникла ошибка ввода/вывода, для получения кода которой вызывается функция IOResult.

Пример

```
// Заносит ошибку в протокол событий Windows NT.
procedure LogError(ErrorCode: Integer; ErrorAddr: Pointer);
var
  ApplicationName: string;
  Handle: THandle;
  Strings: array[0..0] of PChar;
begin
  // Меняет путь приложения на его имя, например, 'App'.
  // Приложение должно было создать соответствующий ключ в реестре.
  ApplicationName := ChangeFileExt(ExtractFileName(ParamStr(0)), '');
  Handle := RegisterEventSource(nil, PChar(ApplicationName));

  if ErrorCode = 0 then
    ErrorCode := IOResult;

  // Формирование сообщения об ошибке.
  Strings[0] := PChar('Ошибка времени выполнения №' +
                      IntToStr(ErrorCode));

  // Категория ошибки определяется в другом месте.
  ReportEvent(Handle, EventLog_Error_Type, Category, ErrorCode,
    nil, 0, 1, @Strings, nil);

  DeregisterEventSource(Handle);
end;
...
ErrorProc := @LogError;
```

Смотрите также

Переменные ErrorAddr, ExceptProc, процедуры Halt, RunError, функция IOResult.

Except, ключевое слово

Синтаксис

```
try Предложения... except Предложения... end;
try Предложения... except Обработчики исключительных ситуаций... end;
```

Описание

Ключевое слово except является частью блока try-except. В секции try перечислены исполняемые Delphi предложения, в секции except – обработчики исключительных ситуаций, которые Delphi исполняет, только если предложения в секции try вызвали исключительную ситуацию.

Конструкция try-except существует в двух вариантах:

- В первом варианте в секции except перечислены предложения. Если возникает исключительная ситуация, Delphi исполняет предложения секции except. Если эти предложения не генерируют другое исключение, программа продолжает выполнение предложений до конца конструкции try-except.
- Во втором варианте в секции except перечислены обработчики исключительных ситуаций. Каждый обработчик начинается с директивы on и указания класса исключительной ситуации. Секция except перехватывает только те исключения, которые перечислены в указанных обработчиках. Если исключительная ситуация не соответствует ни одному из указанных обработчиков и в секции except отсутствует ключевое слово else, Delphi заново генерирует исключительную ситуацию (как если бы в качестве последнего обработчика было бы указано else raise).

Примеры и дополнительная информация об обработке исключительных ситуаций приведены в описании ключевого слова try.

Смотрите также

Ключевые слова Finally, Raise, Try, директива On.

ExceptClsProc, переменная

Синтаксис

```
var ExceptClsProc: Pointer;

function ExceptClassProc(var Rec: TExceptionRecord): TClass;
begin
  ...
end;
ExceptClsProc := @ExceptClassProc;
```

Описание

Delphi вызывает функцию, на которую указывает ExceptClsProc, когда внутри блока try-except, содержащего обработчики, возникает исключительная ситуация Windows (т. е. когда Delphi должна проверить, какому обработчику соответствует возникшая исключительная ситуация). Единственный аргумент вызываемой функции – это указатель на запись об исключительной ситуации.

Класс, возвращаемый функцией, на которую указывает ExceptClsProc, должен соответствовать классу объекта, возвращаемого функцией ExceptObjProc. При помощи класса из ExceptClsProc Delphi определяет, какой из обработчиков использовать перед тем, как вызвать функцию, указанную в ExceptObjProc.

Советы и приемы

- В модуле Windows определяется тип TExceptionRecord.
- В модуле SysUtils переменная ExceptClsProc устанавливается на функцию, которая преобразует все исключительные ситуации Windows в соответствующие исключительные ситуации Delphi, например: Status_Integer_Overflow преобразуется в EIntOverflow.

Пример

```
// Простой пример преобразования исключительных ситуаций Windows в
// собственные исключительные ситуации
class CustomException = class(Exception);
class CustomMathException = class(CustomException);
class CustomRangeException = class(CustomException);
function CustomExceptClsFunc(var Ex: TExceptionRecord): TClass;
begin
  case Ex.ExceptionCode of
    STATUS_ARRAY_BOUNDS_EXCEEDED,
    STATUS_INTEGER_OVERFLOW:
      Result := CustomRangeException;
    STATUS_INTEGER_DIVIDE_BY_ZERO,
    STATUS_FLOAT_INEXACT_RESULT,
    STATUS_FLOAT_INVALID_OPERATION,
    STATUS_FLOAT_STACK_CHECK,
    STATUS_FLOAT_DIVIDE_BY_ZERO,
    STATUS_FLOAT_OVERFLOW,
    STATUS_FLOAT_UNDERFLOW,
    STATUS_FLOAT_DENORMAL_OPERAND:
      Result := CustomMathException;
    else
      Result := CustomException;
  end;
end;

ExceptClsProc := @CustomExceptClsFunc;
```

Смотрите также

Переменные ErrorProc, ExceptObjProc, ExceptProc, ключевые слова Except, Try, директива On.

ExceptionClass, переменная

Синтаксис

```
var ExceptionClass: TClass;
```

Описание

IDE Delphi использует переменную ExceptionClass чтобы определить, какие исключительные ситуации обрабатывает отладчик. Интегрированный отладчик IDE прерывается только на исключительных ситуациях, которые унаследованы от класса, указанного в ExceptionClass. Модуль SysUtils устанавливает значение переменной в Exception.

Советы и приемы

IDE Delphi позволяет определить, будет ли отладчик получать управление в случае возникновения в программе исключительной ситуации. Кроме того, можно управлять этим из программы, устанавливая значение переменной ExceptionClass. В частности, вы можете отменить вызов отладчика, установив значение переменной в nil.

Смотрите также

Ключевые слова Raise, Try.

ExceptObjProc, переменная

Синтаксис

```
var ExceptObjProc: Pointer;

function ExceptProc(var Rec: TExceptionRecord): TObject;
begin ... end;
...
ExceptObjProc := @ExceptProc;
```

Описание

При возникновении исключительной ситуации Delphi вызывает функцию, на которую указывает значение переменной ExceptObjProc (если оно не равно nil). Вызываемая функция *ExceptProc* возвращает новый объект Exception, соответствующий возникшей исключительной ситуации Windows. Если значение переменной ExceptObjProc равно nil или функция, на которую она указывает, возвращает nil, исключительная ситуация Windows прерывает программу.

Единственный аргумент функции *ExceptProc* – указатель на запись об исключительной ситуации.

Советы и приемы

- В модуле Windows определяется тип TExceptionRecord.
- В модуле SysUtils переменная ExceptObjProc устанавливается на функцию, которая преобразует все исключительные ситуации Windows в соответствующие исключительные ситуации Delphi, например: Status_Integer_Overflow преобразуется в EIntOverflow.

Пример

```
// Вызывает функцию CustomExceptObjFunc для определения класса
// исключительной ситуации, создает и возвращает объект Exception.
function CustomExceptObjFunc(var Rec: TExceptionRecord): TObject;
begin
  Result := CustomExceptClsFunc(Rec).Create;
end;

ExceptObjProc := @CustomExceptObjFunc;
```

Смотрите также

Переменные ExceptClsProc, ExceptProc, тип TExceptionRecord.

ExceptProc, переменная

Синтаксис

```
var ExceptProc: Pointer;

procedure MyProc(ExceptObject: TObject; ExceptAddr: Pointer);
begin ... end;
...
ExceptProc := @MyProc;
```

Описание

Если исключительная ситуация не обрабатывается блоком try-except и переменная ExceptProc не равна nil, Delphi вызывает процедуру, указанную в ExceptProc, передав в качестве аргумента ссылку на объект Exception и адрес, где возникла исключительная ситуация. Если ExceptProc равна nil, Delphi вызывает ошибку времени выполнения 217.

Пример

```
// Записывает исключительные ситуации в протокол событий
// Windows NT.
procedure LogException(ExceptObject: TObject; ExceptAddr: Pointer);
var
  ApplicationName: string;
  Handle: THandle;
  Msg: string;
  Strings: array[0..0] of PChar;
begin
```

```
    ApplicationName := ChangeFileExt(ExtractFileName(ParamStr(0)), '');
    // Меняет путь приложения на его имя, например, 'App'.

  // Считаем что, приложение уже создало подходящий ключ в реестре.
    Handle := RegisterEventSource(nil, PChar(ApplicationName));

    // Get the exception message.
    if ExceptObject is Exception then
      Msg := Exception(ExceptObject).Message
    else
      Msg := ExceptObject.ClassName;
    Strings[0] := PChar(Msg);

    // Category и Exception_ID определяются в другом месте.
    ReportEvent(Handle, EventLog_Error_Type, Category, Exception_ID,
      nil, 0, 1, @Strings, nil);

    DeregisterEventSource(Handle);
  end;
  ...
  ExceptProc := @LogException;
```

Смотрите также

Переменная `ErrorProc`, ключевое слово `Raise`.

Exclude, процедура

Синтаксис

```
procedure Exclude(var ASet: Множество; Value: Порядковый тип);
```

Описание

Процедура `Exclude` удаляет элемент `Value` из множества, хранящегося в переменной `ASet`. Тип `Value` должен соответствовать базовому типу `ASet`. `Exclude` не является настоящей процедурой.

Советы и приемы

Заметьте, что нельзя использовать свойство для ссылки на множество, т. к. значение свойства не может выступать в качестве параметра `var`. Элемент можно удалить из множества с помощью оператора «-», например,

```
Font.Style := Font.Style - [fsBold];
```

Пример

```
// Редактор TFontStyleProperty переключает «жирность» шрифта
// при выборе свойства TFont.Style в инспекторе
// объектов.
type
  TFontStyleProperty = class(TSetProperty)
```

```
  public
    procedure Edit; override;
  end;
...
procedure TFontStyleProperty.Edit;
var
  Style: TFontStyles;
begin
  Style := TFontStyles(Byte(GetOrdValue));// Получение стиля текущего
                                          // шрифта.
  if fsBold in Style then
    Exclude(Style, fsBold)
  else
    Include(Style, fsBold);
  SetOrdValue(Byte(Style));           // Установка стиля шрифта
                                      // компонента.
end;
```

Смотрите также

Ключевое слово `In`, `Set`, процедура `Include`.

Exit, процедура

Синтаксис

```
Exit;
```

Описание

Вызов процедуры `Exit` приводит к моментальному возврату из функции или процедуры. Если `Exit` вызывается из блока `try-finally`, перед выходом из подпрограммы исполняется секция `finally`.

Процедура `Exit` встроена в компилятор и не является настоящей процедурой. Если она вызывается вне подпрограммы или метода, Delphi завершает приложение.

Пример

```
// Вычисляет количество итераций в точке (X, Y).
// Останавливается на MaxIterations, что является грубым
// приближением бесконечности. Эта функция используется
// в примере, приведенном в описании BeginThread.
function ComputeIterations(X, Y: Double): Integer;
const
  Threshold = 4.0;
var
  XNew, YNew: Double;
  XC, YC: Double;
begin
  XC := X;
  YC := Y;
  for Result := 0 to MaxIterations-1 do
```

```
    begin
      XNew := X * X - Y * Y + XC;
      YNew := 2 * X * Y + YC;
      if (XNew * XNew + YNew * YNew) > Threshold then
        Exit;
      X := XNew;
      Y := YNew;
    end;
    Result := MaxIterations;
  end;
```

Смотрите также

Процедура `Break`, ключевые слова `Goto`, `Try`.

ExitCode, переменная

Синтаксис

```
var ExitCode: Integer;
```

Описание

В переменную `ExitCode` записывается код возврата программы. Процедура выхода или раздел инициализации могут использовать `ExitCode` для выполнения определенных действий. Код возврата передается как параметр в процедуру `Halt`.

Советы и приемы

Командный файл может использовать код возврата программы в предложении `if`. Подробности смотрите в файле помощи Windows в разделе «пакетные команды».

Смотрите также

Ключевое слово `Finalization`, процедуры `Halt`, `RunError`.

ExitProc, переменная

Синтаксис

```
var ExitProc: Pointer;
```

Описание

Когда приложение завершает свою работу, вызывается процедура, на которую указывает `ExitProc`. Процедуре не требуются аргументы. `ExitProc` существует только для обратной совместимости со старыми программами на Delphi. Новые программы должны вместо этого содержать раздел завершения. Если вам необходимо использовать `ExitProc`, обратитесь к процедуре `AddExitProc` из модуля `SysUtils` как более правильному способу установки процедуры выхода.

Смотрите также

Ключевое слово Finalization.

Exp, функция

Синтаксис

```
function Exp(X: Вещественный тип): Extended;
```

Описание

Функция Exp вычисляет e^x. Функция Exp – встроенная.

Советы и приемы

- Delphi автоматически преобразует аргументы типа Integer и Variant в вещественные числа. Чтобы преобразовать аргумент типа Int64 в вещественное число, прибавьте к нему 0,0.
- Если X равно «сигнализирующему» NaN, плюс бесконечности или минус бесконечности, возникает ошибка времени выполнения 6 (EInvalidOp).
- Если X равно «спокойному» NaN, результат равен X.
- Если результат слишком велик, Delphi выдает ошибку времени выполнения 8 (EOverflow).

Смотрите также

Функция Ln.

Export, директива

Синтаксис

```
Объявление подпрограммы; export;
```

Описание

Delphi игнорирует директиву export. Эта директива оставлена только для обратной совместимости с Delphi 1.

Смотрите также

Ключевое слово Exports.

Exports, ключевое слово

Синтаксис

```
exports
  Подпрограмма,
  Подпрограмма name Идентификатор,
  Подпрограмма index Константа,
  Подпрограмма index Константа name Идентификатор,
```

```
...;
```

Описание

В объявлении `exports` перечислены имена или сигнатуры подпрограмм, которые должны быть экспортированы из DLL. Подпрограммы для экспорта могут быть объявлены в любом модуле или в файле проекта библиотеки.

Если подпрограмма перегружена, программист может указать, какую именно подпрограмму следует экспортировать, включив в объявление `exports` аргументы подпрограммы. Допускается экспорт нескольких перегруженных подпрограмм, но в этом случае необходимо обеспечить вызывающему приложению возможность идентифицировать конкретную подпрограмму присвоением каждой из них уникального идентификатора.

По умолчанию подпрограмма экспортируется под собственным именем, но можно указать другое имя или индекс. Если индекс не указан, Delphi присваивает его автоматически.

Советы и приемы

- Допускается одновременное использование директив `index` и `name` для одной экспортируемой подпрограммы (в указанном порядке).
- Delphi не проверяет индексы на уникальность, поэтому будьте осторожны.
- Delphi 5 не разрешает применение директивы `index` для перегружаемых процедур.

Пример

```
unit Debug;
interface

// Простые отладочные процедуры. Отладочные сообщения записываются
// в файл протокола. Вы можете присоединить этот модуль к
// приложению или вызывать из DLL. В описании директивы External
// смотрите пример, как приложение может использовать эти процедуры
// из DLL.
procedure Log(const Msg: string); overload;
procedure Log(const Fmt: string; const Args: array of const); overload;
procedure SetDebugLog(const FileName: string);
function GetDebugLog: string;

implementation

uses SysUtils;

var
  DebugLog: string = 'c:\debug.log';
```

```
procedure SetDebugLog(const FileName: string);
begin
  DebugLog := FileName;
end;

function GetDebugLog: string;
begin
 Result := DebugLog;
end;

procedure Log(const Msg: string); overload;
var
  F: Text;
begin
  AssignFile(F, DebugLog);
  if FileExists(DebugLog) then
    Append(F)
  else
    Rewrite(F);
  try
    WriteLn(F, '[', DateTimeToStr(Now), '] ', Msg);
  finally
    CloseFile(F);
  end;
end;

procedure Log(const Fmt: string; Args: array of const); overload;
begin
  WriteLn(Format(Fmt, Args));
end;

exports
  GetDebugLog,
  SetDebugLog,
  Log(const Msg: string) name 'Log',
  Log(const Fmt: string; const Args: array of const) name 'LogFmt';
end.
```

Смотрите также

Директивы `External`, `Index`, `Name`, ключевое слово `Library`.

Extended, тип

Синтаксис

```
type Extended;
```

Описание

Тип `Extended` – тип с плавающей точкой, соответствующий стандарту IEEE, который занимает 10 байт: знаковый бит, 15 бит экспоненты и

64 бита мантиссы. Тип Extended соответствует минимальным требованиям стандарта IEEE 754 для типа с расширенной двойной точностью.

Пределы значений типа Extended – это примерно от $3{,}37\times10^{-4932}$ до $1{,}18\times10^{4932}$ с точностью около 19 десятичных знаков.

В отличие от Single и Double, все биты Extended – значимые. Нормализованные значения, бесконечность и нечисловое значение содержат явный старший бит со значением 1. В табл. 5-2 подробно рассмотрен формат конечных и специальных значений Extended. Не все битовые шаблоны являются допустимыми значениями Extended. Если попытаться использовать неверный битовый шаблон для вещественного числа, Delphi генерирует ошибку времени выполнения 6 (EInvalidOp).

Таблица 5-2. Формат вещественных чисел Extended

Класс значений	Знак	Биты экспоненты	Биты мантиссы
Положительные числа			
Нормализованные	0	От 0…1 до 1…0	От 10…0 до 11…1
Денормализованные	0	0…0	От 0…1 до 01…1
Ноль	0	0…0	0…0
Бесконечность	0	1…1	10…0
«Спокойный» NaN	0	1…1	От 110…1 до 11…1
«Сигнализирующий» NaN	0	1…1	От 100…0 до 101…1
Отрицательные числа			
Нормализованные	1	От 0…1 до 1…10	От 10…0 до 11…1
Денормализованные	1	0…0	От 0…1 до 01…1
Отрицательные числа			
Ноль	1	0…0	0…0
Бесконечность	1	1…1	10…0
«Спокойный» NaN	1	1…1	От 110…1 до 11…1
«Сигнализирующий» NaN	1	1…1	От 100…0 до 101…1

Советы и приемы

- Применяйте тип Extended для того, чтобы сохранить максимальную точность или диапазон экспоненты, но имейте в виду, что за это придется платить падением производительности из-за неудобного размера значений Extended.

- Delphi устанавливает управляющее состояние FPU в режим расширенной точности, и потому все промежуточные вычисления производятся с полной точностью, соответствующей типу `Extended`. Если требуется сохранить вещественное значение в переменной типа `Single` или `Double`, Delphi отбрасывает лишние биты точности.
- В документации по архитектуре Intel (например, «Pentium Developer's Manual», том 3, «Architecture and Programming Manual») или в стандарте IEEE 754 можно найти более подробную информацию о бесконечности и NaN (нечисловое значение). В Delphi использование NaN вызывает ошибку времени выполнения 6 (`EInvalidOp`).

Пример

```
type
  TExtended = packed record
    case Integer of
    0: (Float: Extended;);
    1: (Bytes: array[0..9] of Byte;);
    2: (Words: array[0..4] of Word;);
    3: (LongWords: array[0..1] of LongWord; LWExtra: Word;);
    4: (Int64s: array[0..0] of Int64; Exponent: Word;);
  end;
  TFloatClass = (fcPosNorm, fcNegNorm, fcPosDenorm, fcNegDenorm,
            fcPosZero, fcNegZero, fcPosInf, fcNegInf, fcQNaN, fcSNan);

// Возвращает класс вещественного числа: конечное, бесконечное,
// нечисловое; положительное или отрицательное, нормализованное или
// денормализованное. Класс определяется путем проверок
// экспоненты, знакового бита и мантиссы.
function fp_class(X: Extended): TFloatClass; overload;
var
  XParts: TExtended absolute X;
  Negative: Boolean;
  Exponent: LongWord;
  Mantissa: Int64;
begin
  Negative := (XParts.Exponent and $8000) <> 0;
  Exponent :=  XParts.Exponent and $7FFF;
  Mantissa :=  XParts.Int64s[0];

  // Три первых варианта могут быть положительными или
  // отрицательными. Предположим, значение положительно и
  // проверим знаковый бит позднее.
  if (Exponent = 0) and (Mantissa = 0) then
    // Мантисса и экспонента - нулевые, следовательно, число равно
    // нулю.
    Result := fcPosZero
  else if (Exponent = 0) and (Mantissa < 0) then
    // Если экспонента равна нулю, а в мантиссе старший бит
    // равен 0, число денормализовано. Обратите внимание на
    // наличие в Extended старшего бита, равного 1.
```

```
      Result := fcPosDenorm
    else if Exponent <> $7FFF then
      // Иначе, если экспонента не целиком состоит из единиц, число
      // нормализовано.
      Result := fcPosNorm
    else if Mantissa = $8000000000000000 then
      // Все биты экспоненты равны 1, в мантиссе старший бит равен 1,
      // следовательно - бесконечность.
      Result := fcPosInf

    else
    begin
      // Экспонента состоит из 1, мантисса - ненулевая, следовательно,
      // значение нечисловое. Проверяем, «спокойный» или
      // «сигнализирующий» NaN. Старший бит всегда 1. Следующий бит
      // равен 1 для «спокойного» NaN и 0 для сигнализирующего.
      if (Mantissa and $4000000000000000) <> 0 then
        Result := fcQNaN
      else
        Result := fcSNaN;
      Exit; // Не различаем отрицательных NaN.
    end;

    if Negative then
      Inc(Result);
  end;
```

Смотрите также

Типы `Currency`**,** `Double`**,** `Real`**,** `Real48`**,** `Single`**, процедура** `Set8087CW`**.**

External, директива

Синтаксис

```
Описание подпрограммы; external;
Описание подпрограммы; external Имя DLL;
Описание подпрограммы; external Имя DLL name Строка;
Описание подпрограммы; external Имя DLL index Константа;
```

Описание

Все подпрограммы, объявленные в разделе интерфейса модуля, должны быть реализованы в разделе реализации модуля. Подпрограммы могут быть реализованы с директивой `external`, тогда фактическая реализация находится в отдельном объектном файле или DLL. Если `Имя DLL` не указано, внешняя реализация должна находиться в совместимом объектном файле, присоединенном к приложению с помощью директивы компилятора `$L` или `$Link`. `Имя DLL` должно быть строковой константой. Для максимальной переносимости обязательно укажите расширение *dll* в имени файла библиотеки. (Windows NT, например, требует расширение файла.) По умолчанию Delphi ищет в DLL имя

указанной процедуры, однако можно указать другое имя для поиска или числовой индекс.

Пример

```
// Импорт отладочных процедур, которые экспортируются
// библиотекой Debug. (Смотрите описание директивы Exports.)
procedure Log(const Msg: string); overload; external 'Debug.dll';
procedure Log(const Fmt: string; const Args: array of const); overload;
    external 'Debug.dll' name 'LogFmt';
```

Смотрите также

Директивы Exports, Index, Name, директивы компилятора $L, $Link.

Far, директива

Синтаксис

```
Описание подпрограммы; far;
```

Описание

Delphi игнорирует директиву far, которая существует только для обратной совместимости с Delphi 1.

Смотрите также

Директива Near.

File, ключевое слово

Синтаксис

```
var
  BinaryFile: file;
  TypedFile: file of Какой-либо тип;
```

Описание

File – это двоичная последовательность данных некоторого типа. Содержимое файла обычно хранится на некотором постоянном носителе, например, на диске. Если тип файла не указан, нетипизированный файл рассматривается как последовательность записей фиксированного размера. Этот размер указывается при открытии файла процедурой Reset или Rewrite.

Для внутреннего представления двоичного файла в Delphi используется запись TFileRec, для удобства объявленную в модуле SysUtils, но программист может сделать это и самостоятельно:

```
type
  TFileRec = packed record
    Handle: Integer;                  // Дескриптор файла Windows
    Mode: Integer;               // fmInput, fmOutput, fmInOut, fmClosed
```

```
    RecSize: Cardinal;                // Размер записи
    Private: array[1..28] of Byte;    // зарезеривировано Delphi
    UserData: array[1..32] of Byte;   // можно использовать для
                                      // своих целей
    Name: array[0..259] of Char;      // путь к файлу
  end;
```

Советы и приемы

- Тип `file` предназначен для двоичных файлов. Текстовые файлы рассматриваются в описании типа `TextFile`. Двоичные и текстовые файлы в Delphi часто используются совершенно по-разному. Заметьте, кроме того, что тип `TextFile` не аналогичен `file of Char`.
- Базовый тип файла может быть скаляром, массивом или записью. Он не может быть указателем, динамическим массивом или длинной строкой. Сохранение указателя в файле бесполезно, т. к. значения указателей все время меняются. Длинные и «широкие» строки являются указателями, поэтому, если требуется сохранять в файле строки, объявляйте их как тип `short string` или вызывайте процедуры `BlockRead` и `BlockWrite`.
- Для открытия существующего файла используется процедура `Reset`, для создания нового файла – процедура `Rewrite`.
- Delphi не буферизует операции файлового ввода/вывода, поэтому чтение и запись отдельных записей может быть медленным. Для работы с несколькими записями за одну операцию с целью повышения производительности применяйте процедуры `BlockRead` и `BlockWrite`.
- Избегайте типов `Integer`, `Cardinal` и любых других, размер которых может меняться от версии к версии. Используйте данные определенного размера, такие как `Word` и `LongInt`. В этом случае файлы, записанные в одной версии, можно будет читать в другой.
- Выравнивание полей внутри записи может различаться в разных версиях Delphi. Для уменьшения вероятности расхождения форматов записей в разных версиях, применяйте упакованные (`packed`) записи.
- Delphi не поддерживает стандартный подход языка Паскаль к файловому вводу/выводу: вызовы `Get` и `Put` и ссылки на файловую переменную как на указатель. Вместо этого используйте процедуры `Read`, `Write`, `BlockRead` и `BlockWrite`.

Пример

```
type
  TStudent = packed record
    ID: string[9];
    Name: string[40];
    GPA: Single;
  end;
```

```
  TStudentFile = file of TStudent;
var
  F: TStudentFile;
  S: TStudent;
begin
  AssignFile(F, 'students.dat');
  Reset(F);
  try
    while not Eof(F) do
    begin
      Read(F, S);
      ProcessStudent(S);
    end;
  finally
    CloseFile(F);
  end;
end;
```

Смотрите также

Процедуры AssignFile, BlockRead, BlockWrite, Read, Seek, Write, функции Eof, FilePos, FileSize, IOResult, ключевые слова Packed, Record, тип TextFile.

FileMode, переменная

Синтаксис

```
var FileMode: Integer;
```

Описание

При открытии двоичного файла процедурой Reset Delphi устанавливает режим, указанный в переменной FileMode. Возможны следующие значения FileMode:

0 Только для чтения

1 Только для записи

2 Запись и чтение

Значение по умолчанию равно 2.

Советы и приемы

- Чтобы открыть существующий файл с доступом только на чтение, установите FileMode в ноль до вызова процедуры Reset. Если этого не сделать, процедура Reset не сможет открыть файл, доступный только для чтения, например, находящийся на CD-ROM.
- Чтобы дописать двоичный файл, установите FileMode равной 1 или 2, откройте файл с помощью процедуры Reset, перейдите в конец файла, а затем приступайте к записи.

- FileMode не используется при открытии двоичных файлов процедурой Rewrite, а также при открытии текстовых файлов.

Смотрите также

Процедура Reset.

FilePos, функция

Синтаксис

```
function FilePos(var F: File): LongInt;
function FilePos(var F: TextFile): LongInt;
```

Описание

Функция FilePos возвращает текущую позицию (в виде номера записи) в файле F. Начало файла – нулевая позиция. Если F – текстовый файл, размер записи может быть произвольно задан размером буфера, который по умолчанию равен 128 байтам. Если значение, возвращаемое функцией Eof(F), равно True, FilePos возвращает количество записей в файле. FilePos не является настоящей функцией.

Ошибки

- Если файл F не был открыт, FilePos выдает ошибку ввода/вывода 103.
- Хотя определить позицию в текстовом файле можно, но нельзя использовать это значение для позиционирования. Процедура Seek работает только с двоичными файлами. Чтобы определить позицию в текстовом файле, используйте Windows API:

```
// Возвращает позицию в текстовом файле в байтах, если его буфер
// пуст.
function TextFilePos(var F: TextFile): LongInt;
begin
  Result := SetFilePointer(TTextRec(F).Handle, 0, nil, File_Current);
end;
```

- FilePos не поддерживает файлы свыше 2 Гбайт. Обратитесь к описанию функции FileSeek из модуля SysUtils или используйте Windows API для работы с большими файлами.

Смотрите также

Функции Eof, FileSize, IOResult, ключевое слово File, процедура Seek, Truncate, тип TextFile.

FileSize, функция

Синтаксис

```
function FileSize(var F: File): LongInt;
```

```
function FileSize(var F: TextFile): LongInt;
```

Описание

Функция FileSize возвращает размер файла F в записях. Если F – текстовый файл, размер записи может быть произвольно задан размером буфера, который по умолчанию равен 128 байтам. Если F – нетипизированный двоичный файл, размер записи задается при открытии файла. FileSize не является настоящей функцией.

Ошибки

- Если файл F не был открыт, FileSize выдает ошибку ввода/вывода 103.
- Настоящие текстовые файлы не имеют записей фиксированного размера, и потому функция FileSize бесполезна для текстовых файлов. Используйте вместо FileSize потоки (streams) или функцию Windows API GetFileSize.

```
// Возвращает размер текстового файла в байтах или -1
// в случае ошибки.
function TextFileSize(var F: TextFile): LongInt;
begin
  case TTextRec(F).Mode of
    fmInput, fmOutput: Result := GetFileSize(TTextRec(F).Handle, nil);
    else               Result := -1;
  end;
end;
```

- FileSize не поддерживает файлы размером свыше 2 Гбайт. Обратитесь к описанию функции FileSeek из модуля SysUtils или используйте Windows API для работы с большими файлами.

Смотрите также

Функции Eof, FilePos, IOResult, ключевое слово File, процедуры Reset, Rewrite, Seek, Truncate, тип TextFile.

FillChar, процедура

Синтаксис

```
procedure FillChar(var Buffer; Count: Integer; const Fill);
```

Описание

Процедура FillChar заполняет переменную Count байтами, копируя аргумент Fill необходимое число раз. Fill может быть константой порядкового типа размером в Byte. FillChar не является настоящей процедурой.

Советы и приемы

- Заметьте, что Buffer не является указателем. Передавайте не адрес переменной, а саму переменную. Если выделена динамическая память, не забудьте разыменовать указатель при вызове FillChar.
- Если порядковое значение константы Fill выходит из диапазона типа Byte, Delphi использует в качестве байта заполнения ее младший байт. Delphi не генерирует ошибку даже в случае включенной проверки на переполнение.
- Чаще всего процедура FillChar применяется для заполнения Buffer нулями. Также можно использовать функцию SysUtils.AllocMem, которая вызывает GetMem и затем FillChar для заполнения всей выделенной памяти нулями.
- При создании новой записи или массива, содержащего длинные строки, динамические массивы, интерфейсы или значения типа Variant, необходимо проинициализировать эти элементы. Процедура Initialize обычно наилучшее средство для этого, однако она не инициализирует другие элементы массива или записи. Вместо нее можно использовать процедуру FillChar для заполнения всей памяти нулями, что является корректным начальным значением для строк, динамических массивов, интерфейсов и значений типа Variant. При освобождении записи не забудьте вызывать процедуру Finalize для освобождения памяти, связанной со строками, динамическими массивами, интерфейсами и Variant.
- Если Count меньше нуля, FillChar ничего не делает.

Пример

```
// Создание динамического массива из Count целых и инициализация
// их нулями.
type
  TIntArray = array of Tnteger;
function MakeZeroArray(Count: Integer): TIntArray;
begin
  SetLength(Result, Count);
  if Count > 0 then
    FillChar(Result[0], Count*SizeOf(Integer), 0);
end;
```

Смотрите также

Процедуры GetMem, Initialize, New, функция StringOfChar.

Finalization, ключевое слово

Синтаксис

```
unit Имя;
interface Объявления...
implementation Объявления...
```

```
initialization Инструкции...
finalization Инструкции...
end.
```

Описание

Раздел завершения (finalization) модуля содержит инструкции, которые запускаются при выходе из программы или выгрузке библиотеки. Разделы завершения запускаются в порядке, обратном инициализации.

Заметьте, что модуль может содержать раздел finalization, только если в нем создан раздел initialization. Раздел initialization может быть пустым.

Советы и приемы

- Раздел finalization используется для очистки глобально выделенной памяти и других глобальных настроек, например, изменений глобальных переменных.
- Всегда освобождайте всю память и другие ресурсы, используемые модулем. Модуль может быть загружен в пакет, который может загружаться и выгружаться одним приложением несколько раз. Небольшие утечки памяти или других ресурсов могут быстро перерасти в серьезные проблемы в случае вашей неаккуратности.

Пример

```
procedure ErrorHandler(ErrorCode: Integer; ErrorAddr: Pointer);
begin
  ... // Обработка ошибок времени выполнения.
end;

// Установка нового обработчика ошибок с сохранением старого.
// Восстанавливаем предыдущий обработчик по завершении работы
// модуля (это может произойти при выгрузке пакета, содержащего
// модуль, и тогда приложение продолжит использование старого
// обработчика ошибок).
var
  OldErrorProc: Pointer;
initialization
  OldErrorProc := ErrorProc;
  ErrorProc := @ErrorHandler;
finalization
  ErrorProc := OldErrorProc;
end.
```

Смотрите также

Ключевые слова Initialization, Unit.

Finalize, процедура

Синтаксис

```
procedure Finalize(var Buffer);
procedure Finalize(var Buffer; Count: Integer);
```

Описание

Процедура Finalize очищает строки, динамические массивы, интерфейсы, значения типа Variant, а также записи или массивы, содержащие элементы этих типов. Delphi автоматически очищает переменные строкового, интерфейсного, вариантного типа и динамических массивов, но если память для этих значений была выделена динамически, то до освобождения памяти необходимо выполнить процедуру Finalize.

Для того чтобы очистить более одного элемента массива, передайте количество элементов массива в параметре Count. Count – это количество элементов массива, а не количество освобождаемых байт памяти.

Finalize не является настоящей процедурой.

Советы и приемы

- Dispose вызывает Finalize автоматически, поэтому процедуру Finalize необходимо вызывать только при использовании FreeMem.
- Первый аргумент Finalize – это не указатель, а переменная или разыменованный указатель.
- Процедура Finalize должна знать тип буфера, чтобы определить, каким образом очистить массив или поля записи. В случае преобразования типа нетипизированного указателя используйте для него правильный тип. Преобразование типов указателей – обычный источник труднообнаруживаемых ошибок.

Пример

```
type
  TSample = record
    Str: string;
    List: array of Integer;
    Intf: IUnknown;
    V: Variant;
  end;
  TSampleArray = array[0..MaxInt div SizeOf(TSample)-1] of TSample;
  PSampleArray = ^TSampleArray;

// Используйте процедуру Initialize для выделения памяти под массив
// TSample.

procedure FreeSamples(Samples: PSampleArray; Count: Integer);
begin
```

```
    Finalize(Samples^, Count);
    FreeMem(Samples);
  end;
```

Смотрите также

Процедуры FreeMem, Initialize.

Finally, ключевое слово

Синтаксис

```
try Инструкции... finally Инструкции... end;
```

Описание

Ключевое слово finally обозначает финальную секцию блока try-finally. Операторы секции finally запускаются всегда, независимо от того, каким образом завершилась секция try: исключительной ситуацией, процедурами Exit или Break.

Дополнительная информация и пример приведены в описании ключевого слова try.

Смотрите также

Ключевые слова Except, Raise, Try.

FindClassHInstance, функция

Синтаксис

```
function FindClassHInstance(ClassRef: TClass): LongInt;
```

Описание

Функция FindClassHInstance возвращает дескриптор экземпляра модуля, содержащего данный класс. Модуль может быть либо приложением, либо пакетом для выполнения, загруженным приложением. Если передается неверная ссылка на класс, FindClassHInstance возвращает ноль.

Ссылка на класс указывает на VMT класса, располагающейся в кодовом сегменте модуля, поэтому FindClassHInstance вызывает функцию FindHInstance. Программист может вызвать по своему желанию либо FindHInstance, либо FindClassHInstance. FindClassHInstance – настоящая функция.

Смотрите также

Функция FindHInstance, переменные HInstance, MainInstance.

FindHinstance, функция

Синтаксис

```
function FindHInstance(Ref: Pointer): LongInt;
```

Описание

Функция FindHInstance возвращает дескриптор экземпляра модуля, содержащего данный указатель на код. Модуль может быть приложением, DLL или пакетом для выполнения, загруженным приложением или DLL. Если указатель не является корректным указателем на код, FindHInstance возвращает ноль.

Вызывать можно как FindHInstance, так и FindClassHInstance. FindHInstance – настоящая функция.

Советы и приемы

Ранние версии Delphi требовали вызова функции FindHInstance для определения дескриптора экземпляра пакета. Текущая версия позволяет делать это при помощи переменной HInstance, поэтому большинству приложений вызов функции FindHInstance не требуется.

Смотрите также

Функции FindClassHInstance, FindResouceHInstance, переменные HInstance, MainInstance.

FindResouceHInstance, функция

Синтаксис

```
function FindResourceHInstance(Instance: LongWord): LongInt;
```

Описание

Если какие-то ресурсы требуется перевести на другой язык, можно воспользоваться средствами локализации Delphi для построения соответствующих DLL. Функция FindResourceHInstance позволяет получить дескриптор экземпляра DLL с этими ресурсами. В качестве параметра передается дескриптор исполняемого модуля, который содержит необходимый ресурс (обычно HInstance). Если модуль не содержит ресурсов, FindResourceHInstance возвращает дескриптор экземпляра основного модуля. В любом случае, для загрузки языковых ресурсов можно использовать дескриптор экземпляра, возвращаемый FindResourceHInstance.

Delphi автоматически загружает локализованные ресурсы из языковой DLL, так что обычно не требуется вызывать функцию FindResourceHInstance. Однако если языковая DLL была изменена и в нее были добавлены другие ресурсы, эту функцию можно вызвать для загрузки дополнительных ресурсов.

FindResouceHInstance — настоящая функция.

Смотрите также

Функции FindClassHInstance, FindHInstance, LoadResouceModule, переменная HInstance.

Flush, процедура

Синтаксис

```
procedure Flush(var F: TextFile);
```

Описание

Процедура Flush заставляет вывести буферизованный текст в текстовый файл, открытый для записи (процедурами Append или Rewrite). По умолчанию текстовый файл накапливает строки текста в буфере до записи в файл. Flush — это не настоящая процедура.

Советы и приемы

Flush сбрасывает буфер Delphi, но не буферы операционной системы. Чтобы быть уверенным в записи всех данных, закройте файл.

Ошибки

- Если файл не был открыт, Flush генерирует ошибку ввода/вывода 103.
- Если процедура Flush не может записать файл, например, из-за того что диск полон, в качестве ошибки ввода/вывода выдается код ошибки Windows.
- Допускается вызов функции Flush и для входного файла (открытого процедурой Reset), но в этом случае процедура не выполняет никаких действий.

Смотрите также

Функция IOResult, процедуры Rewrite, Write, WriteLn, тип TextFile.

For, ключевое слово

Синтаксис

```
for Переменная := Выражение1 to Выражение2 do Оператор
for Переменная := Выражение1 downto Выражение2 do Оператор
```

Описание

Ключевое слово for обозначает начало цикла for. Цикл for вычисляет выражения, определяющие пределы цикла, а затем многократно исполняет тело цикла, присваивая новое значение управляющей переменной цикла перед каждой итерацией. Управляющая переменная цикла должна быть локальной переменной.

Типы *Переменной*, *Выражение1* и *Выражение2* должны соответствовать друг другу и должны иметь порядковый тип: целый, символьный или перечисляемый. Целые значения вне диапазона типа `Integer` не поддерживаются. В этом случае Delphi преобразует *Выражение1* и *Выражение2* в `Integer`, таким образом, полученные результаты могут отличаться от ожидаемых.

Советы и приемы

- Delphi оптимизирует цикл, вычисляя количество итераций до начала выполнения цикла. Изменение пределов цикла (*Выражения1* и *Выражения2*) не повлияет на количество прохождений тела цикла.
- Цикл `for` работает только со значениями типа `Integer`, поэтому при необходимости итераций по значениям типа `Cardinal` или `Int64` следует предпочесть цикл `while`.
- После обычного завершения цикла значение управляющей переменной не определено. Если цикл завершается оператором `Break` или `goto`, переменная сохраняет свое последнее значение.

Смотрите также

Ключевые слова `Do`, `Repeat`, `While`, тип `Integer`.

Forward, директива

Синтаксис

```
Заголовок подпрограммы; forward;
```

Описание

В разделе реализации модуля или тела программы или библиотеки можно объявить функцию или процедуру до ее вызова, указав после заголовка (имени, параметров и типа возвращаемого значения) директиву `forward`.

При компиляции Delphi читает файл от начала до конца. При обнаружении вызова процедуры или функции должно быть известно количество и тип аргументов подпрограммы или метода и тип возвращаемого функцией значения (сигнатура подпрограммы). Директива `forward` – единственный способ объявить подпрограмму заранее и привести полное ее определение позднее.

Советы и приемы

- Чаще всего директива `forward` применяется для взаимно рекурсивных процедур.
- Все объявления в разделе интерфейса модуля уже являются объявлениями прототипов (forward declarations), и потому Delphi игнорирует директиву `forward` в разделе интерфейса.

- В объявлении класса приведены сигнатуры всех методов класса, и потому для методов не требуется директива `forward`.
- Допускается объявление прототипа класса, однако в этом случае директива `forward` не требуется. Подробности смотрите в описании ключевого слова `class`.

Пример

```
// Процедура WalkDirectory рекурсивно обрабатывает все файлы
// указанного каталога и его подкаталогов. Все файлы добавляются
// в управляющий элемент TTreeView, который показывает иерархию
// каталогов и файлов.

procedure WalkDirectory(const Dir: string; Node: TTreeNode); forward;

// Добавляем файл в управляющий элемент. Если файл является каталогом,
// рекурсивно обрабатывает этот каталог.
procedure WalkFile(const DirName, FileName: string;
    Attr: Integer; Parent: TTreeNode);
var
  Node: TTreeNode;
begin
  Node := Parent.Owner.AddChild(Parent, FileName);
  if (faDirectory and Attr) = faDirectory then
    WalkDirectory(DirName + FileName, Node);
end;

procedure WalkDirectory(const Dir: string; Node: TTreeNode);
var
  Rec: TSearchRec;
  Path: string;
begin
  Path := IncludeTrailingBackslash(Dir);
  if FindFirst(Path + '*.*', faAnyFile, Rec) = 0 then
    try
      repeat
        // Пропускаем текущий и родительский каталоги.
        if (Rec.Name = '.') or (Rec.Name = '..') then
          Continue;
        WalkFile(Path, Rec.Name, Rec.Attr, Node)
      until FindNext(Rec) <> 0;
    finally
      FindClose(Rec);
    end;
end;
```

Смотрите также

Ключевые слова `Class`, `Function`, `Procedure`.

Frac, функция

Синтаксис

```
function Frac(X: Вещественный тип): Extended;
```

Описание

Функция Frac возвращает дробную часть вещественного значения. Это не настоящая функция.

Советы и приемы

- Если X – бесконечность или «сигнализирующий» NaN, Frac генерирует ошибку времени выполнения 6 (EInvalidOp).
- Если X – «спокойный» NaN, результат равен X.

Смотрите также

Функции Int, Round, Trunc.

FreeMem, процедура

Синтаксис

```
procedure FreeMem(P: Pointer);
```

Описание

Процедура FreeMem освобождает память, на которую указывает P, с помощью встроенного менеджера памяти. FreeMem не является настоящей процедурой.

Советы и приемы

- P может быть равен nil, в этом случае FreeMem возвращает управление сразу.
- При освобождении памяти никогда не следует освобождать указатель более одного раза и обращаться к памяти после ее освобождения.
- Чтобы избежать ошибок, особенно в многопоточной программе, следует обнулить глобальные переменные и поля объектов до освобождения памяти. Необходимо удостовериться в отсутствии участков программы, в которых переменная или поле содержит неверный указатель. Например:

```
procedure FreeMemAndNil(var P);
var
  Tmp: Pointer;
begin
  Tmp := Pointer(P);
  Pointer(P) := nil;
```

```
    FreeMem(Tmp);
  end;
```

- В случае применения встроенного менеджера памяти FreeMem поддерживает многопоточность, т.е. может быть вызывана одновременно из нескольких потоков (в том случае, если IsMultiThread равна True).
- Используйте процедуру FreeMem для освобождения памяти, выделенной GetMem, и Dispose – для освобождения памяти, выделенной процедурой New.

Смотрите также

Процедуры Dispose, GetMem, ReallocMem, SetMemoryManager, переменная IsMultiThread, функция SysFreeMem.

FreeMemory, функция

Синтаксис

```
function FreeMemory(P: Pointer): Integer; cdecl;
```

Описание

FreeMemory вызывает процедуру SysFreeMem для освобождения указателя P с помощью встроенного менеджера памяти Delphi. В случае успеха возвращается ноль, и ненулевое значение – в случае неудачи. Если P равно nil, FreeMemory возвращает ноль.

FreeMemory предназначена для использования в C++ Builder. При написании менеджера памяти в Delphi, следует применять SysFreeMem.

FreeMemory – настоящая функция.

Смотрите также

Процедуры Dispose, FreeMem, функции GetMemory, ReallocMemory, SysFreeMem, переменная IsMultiThread.

Function, ключевое слово

Синтаксис

```
function Имя: Возвращаемое значение; Директивы...
function Имя (Параметры...): Возвращаемое значение; Директивы...

type Имя = function(Параметры...): Возвращаемое значение;
type Имя = function(Параметры...): Возвращаемое значение of object;
```

Описание

Функция – это подпрограмма, возвращающая значение. В интерфейсном разделе модуля может присутствовать только заголовок функции (ее имя, параметры, возвращаемое значение и директивы, без кода

функции). Может быть также определен функциональный тип. В разделе реализации необходимо предоставить полное определение всех функций, объявленных в разделе интерфейса. (Смотрите описание директивы external о том, как определить функцию без указания ее кода.) В разделе реализации можно опустить параметры функции в том случае, если функция не перегружена. Если указываются параметры, они должны быть заданы точно так же, как и в разделе интерфейса (в том числе и имена параметров).

В реализации функции не следует повторять директивы, указанные при первом описании ее заголовка.

Советы и приемы

- Указание копии заголовка функции в разделе реализации может показаться излишним, однако это большой плюс для того, кто сопровождает код – не очень удобно постоянно обращаться к заголовку функции, только чтобы узнать о ее параметрах.
- С помощью директивы overload можно объявить несколько функций с одним именем, но разными параметрами.
- Функция может быть объявлена в описании класса, в этом случае она называется методом. Виртуальный метод можно создать, объявляя его с директивами dynamic или virtual. Если виртуальный метод объявлен с директивой abstract, то реализация функции не требуется.
- По умолчанию используется передача параметров через регистры (register). Выбрать другое соглашение о вызове можно с помощью директив cdecl, pascal, safecall или stdcall.
- Все директивы являются необязательными, но если они указываются, то должны быть перечислены для методов в следующем порядке:
  - reintroduce
  - overload
  - virtual, dynamic, override
  - cdecl, pascal, register, safecall, stdcall
  - abstract (только если ранее указаны virtual, dynamic или override)

Примеры

```
type
  TRandFunc = function(Min, Max: Integer): Integer;
  TRandMethod = function(Max: Integer): Integer of object;
  TRandClass = class
  private
    fMin: Integer;
  public
    constructor Create(Min: Integer);
```

```
    function IntFunc(Max: Integer): Integer;
    property Min: Integer read fMin;
  end;

function TestFunc(Min, Max: Integer): Integer;
begin
  Result := Random(Max - Min + 1) + Min;
end;

function TRandClass.IntFunc(Max: Integer): Integer;
begin
  Result := Random(Max - Min + 1) + Min;
end;

var
  F: TRandFunc;
  M: TRandMethod;
  O: TRandClass;
begin
  O := TRandClass.Create(10);
  try
    F := TestFunc;
    WriteLn(F(1, 6));
    M := O.IntFunc;
    WriteLn(M(50));
  finally
    O.Free;
  end;
end.
```

Смотрите также

Директивы `Abstract`, `CDecl`, `External`, `Overload`, `Pascal`, `SafeCall`, `StdCall`, `Virtual`, ключевые слова `Class`, `Procedure`, `Type`, объект `Object`, переменная `Result`.

GetDir, функция

Синтаксис

```
procedure GetDir(Drive: Byte; var Directory: string);
```

Описание

Процедура `GetDir` записывает в `Directory` каталог по умолчанию, находящийся на диске, определяемом параметром `Drive`. Если `Drive` имеет значение от 1 (диск A:) до 26 (диск Z:), `GetDir` извлекает текущий каталог указанного диска. Если `Drive` имеет другое значение или указанный диск недоступен, `GetDir` определяет текущий диск и каталог. Значение, сохраняемое в `Directory`, всегда начинается с буквы диска, например, `X:\Directory`.

`GetDir` не является настоящей процедурой.

Смотрите также

Процедуры ChDir, MkDir, RmDir.

GetHeapStatus, функция

Синтаксис

```
function GetHeapStatus: THeapStatus;
```

Описание

Функция GetHeapStatus возвращает запись, содержащую информацию о менеджере памяти Delphi. Если установлен новый менеджер памяти, не использующий встроенный менеджер памяти Delphi, все поля THeapStatus будут равны нулю. GetHeapStatus – это настоящая функция.

Советы и приемы

Для того чтобы определить, установлен ли специальный менеджер памяти, вызовите функцию IsMemoryManagerSet. Если она возвращает True, данные, полученные с помощью GetHeapStatus, могут быть некорректными.

Пример

```
procedure TForm1.Button1Click(Sender: TObject);
  procedure AddFmt(const Fmt: string; Args: array of const);
  begin
    Memo1.Lines.Add(Format(Fmt, Args));
  end;
var
  Status: THeapStatus;
begin
  Status := GetHeapStatus;
  AddFmt('TotalAddrSpace = %d', [Status.TotalAddrSpace]);
  AddFmt('TotalUncommitted = %d', [Status.TotalUncommitted]);
  AddFmt('TotalCommitted = %d', [Status.TotalCommitted]);
  AddFmt('TotalAllocated = %d', [Status.TotalAllocated]);
  AddFmt('TotalFree = %d', [Status.TotalFree]);
  AddFmt('FreeSmall = %d', [Status.FreeSmall]);
  AddFmt('FreeBig = %d', [Status.FreeBig]);
  AddFmt('Unused = %d', [Status.Unused]);
  AddFmt('Overhead = %d', [Status.Overhead]);
  AddFmt('HeapErrorCode = %d', [Status.HeapErrorCode]);
end;
```

Смотрите также

Переменные AllocMemCount, AllocMemFree, функция IsMemoryManagerSet, процедура SetMemoryManager, тип THeapStatus.

GetMem, процедура

Синтаксис

```
procedure GetMem(var P: Pointer; Size: LongInt);
```

Описание

Процедура GetMem выделяет Size байт динамической памяти и сохраняет указатель на выделенную область памяти в P. Она не инициализирует выделенную память. GetMem не является настоящей процедурой.

Советы и приемы

- Память, выделенная GetMem, не инициализируется. При создании строки, динамического массива или Variant, следует вызвать Initialize или обнулить область памяти процедурой FillChar.
- При использовании встроенного менеджера памяти GetMem поддерживает многопоточность, т. е. может быть вызвана из нескольких потоков одновременно (только в том случае, если IsMultiThread равна True).
- Создавать запись или массив фиксированного размера следует с помощью New, а не GetMem.
- Процедура GetMem обычно применяется для создания массива, размер которого неизвестен на этапе компиляции. И хотя динамические массивы в значительной степени исключили необходимость в использовании этой процедуры, она все еще встречается в существующем коде Delphi.

Пример

```
// Создает серую палитру и возвращает дескриптор палитры. Хотя
// можно создать палитру до 255 оттенков серого, многие видеоадаптеры
// могут показывать только 15 или 16 битов на пиксел, т. е. 5 бит
// для оттенка, или 32 различных серых оттенка.
function CreateGrayScalePalette(NumShades: Bytes);
var
  LogPalette: PLogPalette;
  I: Integer;
begin
  // TLogPalette уже содержит место для одного элемента палитры,
  // поэтому выделяем память для NumShades-1 дополнительного
  // элемента.
  GetMem(LogPalette,
         SizeOf(TLogPalette) + (NumShades-1)*SizeOf(TPaletteEntry));
  try
    LogPalette.palVersion := $300;
    LogPalette.palNumEntries := NumShades;
    // TLogPalette определяет массив palPalEntry с границами 0..0,
    // поэтому выключаем проверку на границы диапазона для установки
    // других границ массива.
```

```
{$R-}
    for I := 0 to NumShades-1 do
    begin
      LogPalette.palPalEntry[I].peRed   := I * 256 div NumShades;
      LogPalette.palPalEntry[I].peGreen := I * 256 div NumShades;
      LogPalette.palPalEntry[I].peBlue  := I * 256 div NumShades;
      LogPalette.palPalEntry[I].peFlags := 0;
    end;
{$R+}
    Result := CreatePalette(LogPalette);
  finally
    FreeMem(LogPalette)
  end;
end;
```

Смотрите также

Процедуры FillChar, FreeMem, Initialize, ReallocMem, функция GetMemory, SysGetMem, переменная IsMultiThread.

GetMemory, функция

Синтаксис

```
function GetMemory(Size: LongInt): Pointer; cdecl;
```

Описание

Функция GetMemory вызывает встроенный менеджер памяти Delphi для выделения Size байт. Она возвращает указатель на выделенную память или nil, если запрос не был выполнен.

Функция GetMemory используется C++ Builder. Написать собственный менеджер памяти в Delphi можно с помощью SysGetMem.

GetMemory – настоящая функция.

Смотрите также

Функции FreeMemory, ReallocMemory, SysGetMem, процедуры GetMem, New, переменная IsMultiThread.

GetMemoryManager, процедура

Синтаксис

```
procedure GetMemoryManager(var Mgr: TMemoryManager);
```

Описание

Процедура GetMemoryManager извлекает указатели на функции, которые Delphi использует для выделения и освобождения памяти, и сохраняет эти указатели в записи Mgr. GetMemoryManager – настоящая процедура.

Советы и приемы

При выполнении глобальных изменений хорошим правилом является сохранение старых настроек, чтобы их можно было восстановить в разделе завершения модуля. Установка нового менеджера памяти является исключением из этого правила. Объяснение смотрите в описании процедуры `SetMemoryManager`.

Смотрите также

Функция `IsMemoryManagerSet`, процедура `SetMemoryManager`, тип `TMemoryManager`.

GetPackageInfoTable, тип

Синтаксис

```
type GetPackageInfoTable = function: PackageInfo;
```

Описание

Каждый пакет экспортирует функцию `@GetPackageInfoTable`. Тип этой функции – `GetPackageInfoTable`. Функция возвращает указатель на таблицу, содержащую указатели на разделы инициализации и завершения всех модулей пакета. Когда Delphi загружает пакет, вызывается функция `@GetPackageInfoTable` для инициализации всех модулей пакета.

Советы и приемы

Углубляться на такой уровень сложности большинству программистов никогда не придется. Delphi делает все это автоматически.

Смотрите также

Ключевые слова `Finalization`, `Initialization`, директива `Package`, типы `PackageInfo`, `PackageInfoTable`, `PUnitEntryTable`, `UnitEntryTable`.

Goto, ключевое слово

Синтаксис

```
goto Метка
```

Описание

Оператор `goto` передает управление на указанную метку. Метка может быть любым идентификатором или строкой символов, содержащей до четырех цифр. Нельзя использовать для локальной переменной идентификатор, применяемый в том же блоке в качестве метки.

Советы и приемы

- Беспорядочное применение `goto` приводит к классической проблеме программ-«спагетти» – исходный текст оказывается спутанным, как «блюдо-спагетти» (bowl of spaghetti).
- Чаще всего оператор `goto` применяется для выхода из глубоко вложенных циклов. В этом случае его можно рассматривать как «супер-`Break`».
- Весь код VCL содержит только два оператора `goto`, и оба они используются для выхода из вложенного цикла. Следует считать это нормой употребления `goto`.
- Переход извне внутрь блока (цикла, условия или вложенной подпрограммы) может привести к непредсказуемым результатам. Delphi не позволяет войти внутрь или выйти из любой части конструкций `try-except` и `try-finally`.
- В отличие от стандартного Паскаля, в Delphi не разрешается применение `goto` для выхода из вложенной подпрограммы во внешнюю. Для преждевременного выхода из функции или процедуры используйте `Exit`.

Пример

```
// Ищет первую непустую клетку сетки и возвращает ее содержимое,
// преобразованное в прописные буквы. Если вся сетка пуста,
// возвращает строку по умолчанию. Пропускаем фиксированные
// строки и столбцы, т. к. они являются лишь идентифицирующими
// заголовками.
procedure UpcaseCell(Grid: TStringGrid; const Default: string): string;
var
  R, C: Integer;
label
  NestedBreak;
begin
  Result := Default;
  for R := Grid.FixedRows to Grid.RowCount-1 do
    for C := Grid.FixedCols to Grid.ColCount-1 do
       if Grid.Cells[C, R] <> '' then
       begin
         Result := Grid.Cells[C, R];
         goto NestedBreak;
       end;
NestedBreak:
  Result := UpperCase(Result);
end;
```

Смотрите также

Процедуры `Break`, `Continue`, `Exit`, ключевое слово `Label`.

Halt, процедура

Синтаксис

```
procedure Halt(ExitCode: Integer);
```

Описание

Процедура `Halt` сразу прерывает работу приложения, не давая ему возможности «убрать за собой». Windows NT автоматически освобождает все ресурсы, которые использовались приложением, но Windows 95 и Windows 98 не столь снисходительны. Применяйте эту процедуру `Halt` только в особых ситуациях, например, для обработки критических ошибок.

`Halt` не является настоящей процедурой.

Советы и приемы

- `Halt` сохраняет свой аргумент в глобальной переменной `ExitCode`.
- Если `ErrorAddr` не равна `nil`, `Halt` выдает сообщение об ошибке перед завершением программы.
- Перед завершением программы исполняются процедура `ExitProc` и разделы завершения модулей.
- Поцедура `Halt`, закрывая программу, не освобождает все объекты и формы. Для завершения графического приложение следует закрыть главную форму, а не вызывать процедуру `Halt`.

Пример

```
// Сообщает об ошибке времени выполнения по указанному адресу.
procedure ReportError(ErrorCode: Integer; Addr: Pointer);
begin
  ErrorAddr := Addr;
  if ErrorProc <> nil then
    ErrorProc(ErrorCode, Addr);
  Halt(ErrorCode);
end;
```

Смотрите также

Переменные `ErrorProc`, `ExceptProc`, `ExitCode`, `ExitProc`, `NoErrMsg`, ключевое слово `Finalization`, процедура `RunError`.

HeapAllocFlags, переменная

Синтаксис

```
var HeapAllocFlags: Word;
```

Описание

Переменная HeapAllocFlags оставлена для обратной совместимости с Delphi 1, где она использовалась при вызове процедуры GlobalAlloc, теперь не существующей.

Hi, функция

Синтаксис

```
function Hi(Value: Integer): Byte;
```

Описание

Функция Hi возвращает самый старший значащий бит 16-битного слова, игнорируя биты старше 16-го. Функция Hi не является настоящей функцией, а преобразуется компилятором в inline-код.

Советы и приемы

Функция Hi эквивалентна следующей:

```
function Hi(Value: Integer): Byte;
begin
  Result := (Value shr 8) and $FF;
end;
```

Смотрите также

Функция Lo.

High, функция

Синтаксис

```
function High(Тип или переменная): Порядковый тип;
```

Описание

Функция High возвращает самое большое значение перечисляемого типа, верхнюю границу индекса массива, либо аналогичную информацию о порядковой переменной или о массиве.

Функция High встроена в компилятор и не является настоящей функцией.

Советы и приемы

- В цикле for, объявлении поддиапазона или в любой другой ситуации, где используются границы порядкового типа или массива, всегда вызывайте функцию High вместо явного указания старшего порядкового значения. В будущей версии программы вы можете изменить тип и вряд ли захотите пересматривать весь код в поисках явных ссылок на перечисляемое значение, которое следует заменить вызовом функции High.

- При вызове функции High для значения типа ShortString возвращается максимальный индекс элемента строки. Для AnsiString или WideString функцию High применять нельзя.
- Значение функции High для открытого массива всегда равно длине массива минус единица независимо от типа или диапазона фактического аргумента.

Пример

```
// Смотрите функцию GetCpuId в примере к ключевому слову asm.

// Идентифицируем CPU и выводим результат в Output.
procedure WriteCpuId;
const
  CpuTypes: array[TCpuType] of string =
    ('Original', 'Overdrive', 'Dual', '?');
var
  ID: TCpuId;
  F:  TCpuFeature;
begin
  if not GetCpuId(ID) then
    WriteLn('Инструкция CPUID не поддерживается')
  else
  begin
    WriteLn(ID.Vendor, ' ', CpuTypes[ID.CpuType]);
    WriteLn('Family: ', ID.Family);
    WriteLn('Model: ', ID.Model);
    WriteLn('Stepping: ', ID.Stepping);
    Write('Features:');
    for F := Low(TCpuFeature) to High(TCpuFeature) do
       if F in ID.Features then
         Write(' ', GetEnumName(TypeInfo(TCpuFeature), Ord(F)));
    WriteLn;
  end;
end;
```

Смотрите также

Процедуры Dec, Inc, функции Length, Low, Pred, Succ.

HInstance, переменная

Синтаксис

```
unit SysInit;
var HInstance: LongWord;
```

Описание

Переменная HInstance содержит дескриптор экземпляра исполняемого модуля (приложения, библиотеки или пакета), в котором находится ссылка на HInstance.

Советы и приемы

- Чаще всего дескриптор экземпляра модуля применяется для загрузки из него ресурсов. Модуль может использовать HInstance для загрузки ресурсов, присоединенных к этому модулю (с помощью директивы компилятора $R). Для загрузки ресурсов из другого модуля (возможно, находящегося в другом пакете) вызывайте функцию FindHInstance.
- Для загрузки локализуемых ресурсов (например, форм или строк) используйте FindResourceHInstance.
- Дескриптор экземпляра основного исполняемого модуля хранится в MainInstance.
- Более ранние версии Delphi сохраняли в HInstance дескриптор экземпляра основного исполняемого модуля и требовали вызова FindHInstance для определения дескриптора экземпляра другого модуля. Текущая версия записывает в HInstance дескриптор экземпляра текущего модуля.

Пример

```
// Создайте в Image Editor RES-файл, содержащий эмблему
// компании в ресурсе по имени LOGO.
// Процедура GetLogo устанавливает значение аргумента
// BitMap на битовый образ эмблемы.

{$R 'Logo.res'}

procedure GetLogo(Bitmap: TBitmap);
begin
  Bitmap.LoadFromResourceName(HInstance, 'LOGO');
end;
```

Смотрите также

Функции FindClassHInstance, FindHInstance, переменная MainInstance.

HPrevInst, переменная

Синтаксис

```
var HPrevInst: Integer;
```

Описание

HPrevInst всегда равна нулю. Это устаревшая особенность Delphi 1, и она больше не служит никаким полезным целям.

Советы и приемы

Большинство программ, написанных на Delphi 1, используют HPrevInst для определения того, что приложение уже было запущено. В Win32 есть много способов решения этой задачи без применения HPre-

vInst, но определение того, что приложение уже запущено, – это только самая простая часть проблемы. Сложнее понять, как вести себя в этой ситуации: должен ли новый экземпляр передать существующему приложению свою командную строку? должно ли приложение вести себя по-другому, будучи запущенным не в первый раз? Ни одна книга не ответит за вас на этот вопрос, однако в главе 4 рассматривается обработка подобной ситуации.

HResult, тип

Синтаксис

```
type HResult = LongWord;
```

Описание

HResult – это тип, возвращаемый большинством методов COM. Старший бит результата равен нулю в случае успеха или 1 в случае ошибки. Дополнительную информацию о типе HResult и стандартных кодах ошибок можно найти в документации по Windows API.

В модуле SysUtils имеется несколько методов для облегчения работы со значениями HResult, например, Succeeded и Failed.

Смотрите также

Директива SafeCall.

IDispatch, интерфейс

Синтаксис

```
IDispatch = interface(IUnknown)
['{00020400-0000-0000-C000-000000000046}']
  function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
  function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
    HResult; stdcall;
  function GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DispIDs: Pointer):
    HResult; stdcall;
  function Invoke(DispID: Integer; const IID: TGUID;
    LocaleID: Integer; Flags: Word; var Params;
    VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
end;
```

Описание

IDispatch – это стандартный интерфейс ActiveX, который позволяет вызывать COM-методы, не зная имени метода, аргументов и типа возвращаемого значения на этапе компиляции.

Советы и приемы

- В случае применения COM-объекта, реализующего интерфейс IDispatch, Delphi автоматически делает все для осуществления динамического вызова метода. Ссылку на объект можно сохранить в переменной типа Variant или использовать диспетчерский интерфейс.
- Для создания COM-сервера, реализующего интерфейс IDispatch, просто унаследуйте свой класс от TAutoObject (из модуля ComObj) или одного из его потомков. Если применяется редактор библиотек типов Delphi, установите флаг Dual, и Delphi сделает за вас все остальное.
- Дополнительная информация об интерфейсе IDispatch приведена в документации на Windows Platform SDK.

Смотрите также

Ключевые слова Dispinterface, Interface, интерфейс IUnknown, переменная VarDispProc, тип Variant.

If, ключевое слово

Синтаксис

```
if Условие then Оператор(ы)
if Условие then Оператор(ы) else Оператор(ы)
```

Описание

Оператор if в Delphi аналогичен if в стандартном Паскале. Условие должно быть логическим выражением. Раздел else не обязателен. Если используются вложенные операторы if, раздел else привязывается к ближайшему if. Например:

```
if X > Y then
  if A > B then
    WriteLn('A > B')
  else
    WriteLn('X > Y and A <= B');
```

Советы и приемы

- Программисты C привыкли использовать точку с запятой для завершения операторов, но в Паскале точка с запятой служит разделителем. В частности, после первого предложения перед ключевым словом else точка с запятой не разрешается. Delphi ослабляет некоторые ограничения Паскаля на применение точек с запятой, поэтому программисты C, C++ и Java могут ошибочно решить, что точки с запятой предназначены в Delphi для завершения предложений.
- По умолчанию логические операторы в Delphi реализуют сокращенную логику вычисления (как их собратья в C, C++ и Java). Подробности рассмотрены в описании операторов and и or.

Смотрите также

Ключевые слова And, Begin, Else, Or, тип Boolean.

Implementation, ключевое слово

Синтаксис

```
unit Имя;
interface Объявления...
implementation Объявления...
end.
```

Описание

Раздел implementation в модуле является обязательным, хотя может быть пустым. Раздел реализации содержит определения всех функций, процедур и методов, объявленных в разделе interface модуля. Кроме того, программист может объявить дополнительные типы, подпрограммы и переменные в разделе реализации, которые будут являться внутренними для модуля.

Изменения в разделе реализации модуля не вызывают перекомпиляции зависимых модулей.

Советы и приемы

Все используемые модули, не требуемые для раздела interface, должны быть перечислены в предложении uses раздела implementation. Это уменьшает необходимость в перекомпиляции при изменении одного из этих модулей. Изменения в разделе интерфейса – в том числе изменения в модулях из uses – вызывают перекомпиляцию всех зависимых модулей, тогда как изменения в разделе реализации не распространяются на зависимые модули.

Пример

Пример рассмотрен в описании ключевого слова unit.

Смотрите также

Ключевые слова Interface, Unit, Uses.

Implements, директива

Синтаксис

```
Объявление свойства read Getter implements Интерфейсы...;
```

Описание

Класс может делегировать реализацию одного или нескольких интерфейсов свойству, обозначенному директивой implements. Аргумент *Getter* должен быть полем или простым методом (массивы и индексированные свойства не разрешаются). Метод может быть виртуальным, но

не динамическим и не обработчиком сообщений. Поле или метод *Getter* должен принадлежать к объектному или интерфейсному типу. Если свойство реализует более одного интерфейса, идентификаторы интерфейсов разделяются запятыми.

Перечисленные интерфейсы должны присутствовать в объявлении класса.

Советы и приемы

- Директива `implements` часто применяется для агрегирования в стиле COM. В модуле `ComObj` объявлен класс `TAggregatedObject` в качестве базового класса для внутреннего объекта.
- Если *Getter* имеет объектный тип, следует быть осторожным в отношении жизненного цикла этого объекта. Когда Delphi преобразует ссылку на объект в интерфейс, автоматическая система счетчиков ссылок Delphi управляет жизненным циклом объекта. Как только последний из интерфейсов выходит из своей области действия, объект автоматически уничтожается. Простейший способ справиться с этой ситуацией – это создавать новый метод в каждом вызове *Getter*.
- Если *Getter* имеет объектный тип, то при помощи конструкции для перенаправления метода можно перенаправить один из методов интерфейса на класс-контейнер.
- Если *Getter* имеет интерфейсный тип, возвращаемый интерфейс должен реализовывать все перечисленные интерфейсы. Для изменения интерфейсов нельзя применять перенаправление методов.

Пример

```
interface
// В интерфейсном разделе модуля объявляем класс TWizardComponent.
// Пользователь помещает класс на форму для создания мастера, т. е.
// расширения IDE Delphi с помощью Open Tools API. Мастер должен
// реализовывать интерфейс IOTAWizard, но детали реализации могут
// быть скрыты в разделе implementation. Делегирование реализации
// свойству позволяет решить эту задачу.
type
  TWizardComponent = class(TComponent, IOTAWizard)
  private
    fWizard: IOTAWizard;
    fOnExecute: TNotifyEvent;
    function GetIDString: string;
    procedure SetIDString(const Value: string);
    function GetWizardName: string;
    procedure SetWizardName(const Value: string);
    property Wizard: IOTAWizard read fWizard implements IOTAWizard;
  protected
    procedure Execute; virtual;
  public
```

```
    constructor Create(Owner: TComponent); override;
  published
    property IDString: string read GetIDString write SetIDString;
    property WizardName: string read GetWizardName write SetWizardName;
    property OnExecute: TNotifyEvent read fOnExecute write fOnExecute;
  end;

implementation

// Конструктор TWizardComponent создает объект типа TWizard и
// сохраняет его в поле fWizard. Фактическая реализация
// интерфейса IOTAWizard выполняется объектом TWizard.
type
  TWizard = class(TInterfacedObject, IOTAWIzard)
  private
    fIndex: Integer;
    fOwner: TWizardComponent;
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
  public
    constructor Create(Owner: TWizardComponent);
    destructor Destroy; override;
  end;
```

Смотрите также

Ключевые слова `Class`**,** `Interface`**,** `Property`**.**

In, ключевое слово

Синтаксис

Выражение порядкового типа `in` *Выражение множественного типа*

Описание

Оператор `in` **проверяет принадлежность множеству. Возвращает** `True` **в том случае, если** *Выражение порядкового типа* **принадлежит множеству, иначе возвращает** `False`**.**

Советы и приемы

Базовый тип множества ограничен максимум 256 членами. Оператор `in` **проверяет только последний значимый байт** *Выражения порядкового типа*. **Поэтому в следующем примере процедура** `ShowMessage` **вызывается всегда:**

```
var
  X: set of 0..7;
begin
  X := [0];
```

```
if 512 in X then
  ShowMessage('Оператор in проверяет только младший байт 512.');
```

Пример

Пример рассмотрен в описании ключевого слова `set`.

Смотрите также

Процедуры `Exclude`, `Include`, функции `High`, `Low`, ключевое слово `Set`.

Inc, процедура

Синтаксис

```
procedure Inc(var Переменная);
procedure Inc(var Переменная; Count: Integer);
```

Описание

Процедура `Inc` увеличивает значение любой переменной порядкового типа или указателя. Для увеличения значения переменной вещественного типа процедуру `Inc` использовать нельзя.

Увеличение значения указателя изменяет его в соответствии с размером базового типа. Например, значение указателя на `AnsiChar` увеличивается на 1, указателя на `WideChar` – на 2. По умолчанию переменная увеличивается на один элемент, однако можно указать целый параметр для увеличения значения переменной на другую величину. При увеличении указателя `Count` умножается на размер базового типа.

Процедура `Inc` встроенная и не является настоящей процедурой.

Советы и приемы

- `Count` может быть отрицательным, в этом случае значение переменной уменьшается.
- Между `Inc`, `Succ` и операцией сложения разница в производительности минимальна. То есть, следующие конструкции компилируются в похожий объектный код:

```
Inc(X);
X := Succ(X);
X := X - 1;    // Если X – целое или PChar
```

- Вместо переменной нельзя использовать свойство, т. к. значение свойства не может быть параметром `var`. В этом случае следует предпочесть простое присваивание.

Смотрите также

Процедура `Dec`, функции `High`, `Low`, `Pred`, `Succ`.

Include, процедура

Синтаксис

```
procedure Include(var ASet: Множество; Value: Порядковый тип);
```

Описание

Процедура Include добавляет элемент Value в множество, хранящееся в переменной ASet. Тип Value должен соответствовать базовому типу ASet. Include не является настоящей процедурой.

Советы и приемы

Заметьте, что не допускается использование свойства для ссылки на множество, т. к. свойство не может выступать в качестве var параметра. Для добавления элемента в множество следует применять оператор «+», например,

```
Font.Style := Font.Style + [fsBold];
```

Пример

Пример рассмотрен в описании процедуры Exclude.

Смотрите также

Процедура Exclude, ключевые слова In, Set.

Index, директива

Синтаксис

```
function Getter(Index: Integer): Тип свойства;
procedure Setter(Index: Integer; Value: Тип свойства);
property Name: Тип свойства index Константа read Getter write Setter;

function ArrayGet(Index: Integer; ArrayIndex: Тип индекса):
                                                    Тип свойства;
procedure ArraySet(Index: Integer; ArrayIndex: Тип индекса;
  Value: Тип свойства);
property Name[ArrayIndex: Тип индекса]: Тип свойства index Константа
  read Getter write Setter;

exports
  Подпрограмма index Константа;

Заголовок подпрограммы; external Имя DLL index Константа;
```

Описание

Директива index применяется в трех различных случаях: объявление индексированного свойства, экспорт подпрограмм и импорт подпрограмм.

- Несколько свойств могут использовать один метод *Getter* или *Setter* с помощью директивы index и уникального значения индекса. Значение индекса передается первым аргументом в методы *Getter* и *Setter*. Дополнительную информацию можно найти в описании ключевого слова property.
- Модуль или библиотека могут экспортировать подпрограммы с помощью директивы exports. Каждая экспортируемая подпрограмма имеет уникальный индекс. Если директива index не указана, Delphi присваивает уникальный индекс автоматически. Дополнительная информация приведена в описании ключевого слова exports.
- В объявлении подпрограммы может быть указана директива external для импорта подпрограммы из DLL. По умолчанию Delphi ищет подпрограмму по ее имени. Если директива index указана, Delphi ищет подпрограмму по номеру индекса, что несколько быстрее (но номер индекса может измениться в новой версии DLL). Дополнительная информация приведена в описании директивы external.

Советы и приемы

- Для свойства, объявленного с директивой index, в директивах read и write не может быть указано имя поля.
- Свойство-массив может использовать директиву index. Номер индекса – первый параметр, индексы массива – второй и все последующие параметры методов *Getter* и *Setter*.
- Для одной и той же экспортируемой подпрограммы можно указать и директиву index, и директиву name (именно в таком порядке).

Смотрите также

Ключевые слова Exports, Property, директивы External, Name.

Inherited, ключевое слово

Синтаксис

```
Заголовок подпрограммы;
begin
  ...
  inherited;
  inherited Имя(Аргументы...);
```

Описание

Метод производного класса может вызвать метод своего базового класса, указав перед именем метода ключевое слово inherited вместо ссылки на объект.

Для вызова унаследованной процедуры с тем же именем и аргументами достаточно указать только ключевое слово (inherited;). Чтобы вы-

звать унаследованную функцию с другими аргументами или вызвать другой метод объекта, нужно указать имя метода и аргументы так же, как и при обычном вызове метода.

Советы и приемы

- Большинство конструкторов вызывают унаследованный конструктор в начале метода. Большинство деструкторов вызывают унаследованный деструктор в конце.
- Учитывайте потребности производных классов. Одни методы расширяют функциональность базового класса, другие – заменяют его. В приведенном ниже примере показаны методы обоих типов.
- Указание одного ключевого слова `inherited` в обработчике сообщения вызывает обработчик родительского класа для сообщения с тем же номером. Если ни один из родительских классов не имеет обработчика с тем же номером сообщения, ключевое слово `inherited` вызывает метод `DefaultHandler`. Заметьте, что родительский обработчик сообщений может иметь другое имя и даже быть закрытым (`private`).
- Если указано только ключевое слово `inherited`, и ни один из родительских классов не содержит метода с тем же именем, Delphi игнорирует предложение `inherited`.

Пример

```
// Объявление класса приведено в описании директивы override.

// Создание растрового изображения при создании управляющего
// элемента.
constructor TTile.Create(Owner: TComponent);
begin
  inherited;
  fBitmap := TBitmap.Create;
  fBitmap.OnChange := BitmapChanged;
end;

// Рисуем управляющий элемент, формируя мозаики. Если растр
// отсутствует, ничего не рисуем. Не вызываем inherited,
// т. к. этот метод Paint заменяет родительский.
procedure TTile.Paint;
var
  X, Y: Integer;
begin
  if (Bitmap.Width = 0) or (Bitmap.Height = 0) then
    Exit;

  Y := 0;
  while Y < ClientHeight do
  begin
    X := 0;
```

```
    while X < ClientWidth do
    begin
      Canvas.Draw(X, Y, Bitmap);
      Inc(X, Bitmap.Width);
    end;
    Inc(Y, Bitmap.Height);
  end;
end;
```

Смотрите также

Ключевое слово `Class`, директивы `Dynamic`, `Message`, `Override`, `Virtual`, переменная `Self`.

Initialization, ключевое слово

Синтаксис

```
unit Имя;
interface Обьявления...
implementation Обьявления
initialization Инструкции...
end.
```

Описание

Инструкции в разделе `initialization` выполняются, когда Windows загружает исполняемый файл (приложение, DLL или пакет), содержащий этот модуль. Сначала Delphi запускает разделы инициализации всех модулей, которые использует модуль *Имя*, в порядке их указания, в первую очередь — модули из раздела `interface`, затем — из раздела реализации. Раздел инициализации запускается после того, как Delphi исполнит разделы инициализации всех используемых модулей.

Delphi отслеживает, какие модули были проинициализированы, так что ни один из разделов инициализации не исполняется более одного раза.

Советы и приемы

- Delphi инициализирует модули начиная с самого глубокого уровня вложенности. Для того чтобы некоторый модуль был проинициализирован раньше, укажите его первым в *dpr*-файле. Хорошим примером модуля, который должен быть инициализирован в первую очередь, является новый менеджер памяти.
- Порядок инициализации модулей определяется таким образом, что программист никогда не знает наверняка, когда будет инициализироваться его модуль. Но в любом случае все модули, используемые в разделе интерфейса данного модуля, будут проинициализированы раньше.
- Модули завершаются в порядке, обратном инициализации.

- Даже если в модуле не объявлен раздел `initialization`, Delphi создает его автоматически. В большом проекте запуск приложения или загрузка библиотеки может привести к большому количеству ошибок page fault, т. к. Windows не будет успевать загрузить код инициализации всех модулей.
- Можно использовать вместо `initialization` ключевое слово `begin`, однако в этом случае модуль не должен иметь раздела `finalization`.

Пример

Пример приведен в описании ключевого слова `unit`.

Смотрите также

Ключевые слова `Begin`, `Finalization`, `Implementation`, `Unit`, процедура `SetMemoryManager`.

Initialize, процедура

Синтаксис

```
procedure Initialize(var Value);
procedure Initialize(var Value; Count: Integer);
```

Описание

Процедура `Initialize` очищает строки, динамические массивы, интерфейсы и значения типа `Variant`. Аргумент `Value` может быть одиночной переменной, а также записью или массивом, содержащим строки, динамические массивы, интерфейсы или `Variant`.

Если требуется проинициализировать более одного элемента массива, передайте количество элементов массива в параметре `Count`. `Count` – это количество элементов массива, а не количество освобождаемых байтов памяти.

`Initialize` не является настоящей процедурой.

Советы и приемы

- Процедура `New` вызывает `Initialize` автоматически, поэтому вызывать последнюю явным образом необходимо только при использовании `GetMem`.
- Первый аргумент `Initialize` – это не указатель, а переменная или разыменованный указатель.
- Процедура `Initialize` должна "знать" тип буфера, а следовательно – размер массива и структуру записи. В случае преобразования типа нетипизированного указателя используйте корректный тип. Преобразование типов указателей – обычный источник труднообнаруживаемых ошибок.

Пример

```
type
  TSample = record
    Str: string;
    List: array of Integer;
    Intf: IUnknown;
    V: Variant;
  end;
  // Тип TSampleArray объявляется с максимально возможным размером.
  // На этапе выполнения программа выделяет память для необходимого
  // количества элементов. Это обычные действия для программ, которым
  // требуется обращаться к функциям Windows API или другим программам
  // C и C++, «не знающим» о динамических массивах Delphi.
  TSampleArray = array[0..MaxInt div SizeOf(TSample)-1] of TSample;
  PSampleArray = ^TSampleArray;


// Уничтожение массива TSample
// рассмотрено в описании процедуры Finalize.

function AllocateSamples(Count: Integer): PSampleArray;
begin
  GetMem(Result, Count * SizeOf(TSample));
  Initialize(Result^, Count);
end;
```

Смотрите также

Процедуры `Finalize`, `GetMem`.

InitProc, переменная

Синтаксис

```
var InitProc: Pointer;

procedure MyInitProc;
begin ... end;
InitProc := @MyInitProc;
```

Описание

Указатель `InitProc` ссылается на процедуру без параметров, которая выполняет вторичную инициализацию. Например, метод `TApplication.Initialize` вызывает процедуру `InitProc`.

Вторичную процедуру инициализации можно определить, например, если модулю требуется дополнительная инициализация, но после инициализации других модулей.

Советы и приемы

- Сохраните предыдущее значение InitProc и вызовите сохраненную процедуру после завершения своей.
- Delphi не гарантирует, что процедура, на которую указывает InitProc, будет вызвана, – это обеспечивается наличием раздела инициализации. Можно только рассчитывать, что вторичная инициализация будет выполнена в одном из других участков кода. Библиотека VCL «ведет себя правильно», и все графические и серверные приложения вызывают процедуру, указанную в InitProc. Создавая собственную структуру приложения, не забудьте вызвать процедуру вторичной инициализации, т. к. некоторые модули VCL этого требуют.
- Везде, где это возможно, используйте раздел инициализации. Прибегайте к InitProc только в том случае, если у вас нет другого способа выполнить инициализацию.
- Позаботьтесь о завершении работы модуля в разделе finalization.

Пример

```
...
procedure LocalInitProc;
begin
  SetUpUnit;                 // Инициализируем модуль.
  TProcedure(SaveInitProc); // Вызываем следующую процедуру из
                             // InitProc в цепочке.
end;

var
  SaveInitProc: Pointer;
initialization
  SaveInitProc := InitProc;
  InitProc := @LocalInitProc;
finalization
  CleanUpUnit;
end.
```

Смотрите также

Переменная ExitProc, ключевые слова Finalization, Initialization.

Inline, ключевое слово

Описание

Ключевое слово inline зарезервировано. В Delphi 1 ключевое слово inline использовалось, но в Delphi 2 и последующих версиях оно не применяется. И хотя оно может поддерживаться в будущих версиях Delphi, основной задачей резервирования этого слова является поиск ошибок при переносе кода из Delphi 1 в более поздние версии.

Input, переменная

Синтаксис

```
var Input: TextFile;
```

Описание

Переменная Input – это текстовый файл, который автоматически открывается для чтения, но только в консольных приложениях. Файл Input – это обычно консоль, однако можно перенаправить ввод из командной строки.

Это обычный текстовый файл, который можно закрыть или открыть другой. Открытие файла без имени аналогично открытию консольного файла. Например, при запуске консольного приложения Delphi автоматически выполняет следующее:

```
AssignFile(Input, '');
Reset(Input);
```

Вызов Read или ReadLn без указания файла в первом аргументе аналогичен вызову Read или ReadLn с первым аргументом Input.

Смотрите также

Процедуры AssignFile, Read, ReadLn, Reset, переменная Output, тип TextFile.

Insert, процедура

Синтаксис

```
procedure Insert(const Ins: string; var Str: string; Index: Integer);
```

Описание

Процедура Insert вставляет строку Ins в строку Str в позиции Index. Если Index ≤ 1, Ins вставляется в начало строки. Если Index находится за концом строки, Ins дописывается в конец Str.

Insert не является настоящей процедурой.

Советы и приемы

Индекс первого символа строки равен 1.

Пример

```
// Вставляем букву диска в начало строки пути.
procedure InsertDriveLetter(var Path: string; const Drive: Char);
begin
  // Сначала убеждаемся, что путь не содержит буквы диска в начале.
  if (Length(Path) < 2) or (Path[2] <> ':') then
    // Insert the drive at the start of the path.
    Insert(Drive + ':', Path, 1);
end;
```

Смотрите также

Функция Copy, процедуры Delete, SetLength, SetString.

Int, функция

Синтаксис

```
function Int(X: Вещественный тип): Extended
```

Описание

Функция Int урезает вещественное число, отбрасывая его дробную часть (округляя в сторону нуля). Заметьте, что Int возвращает вещественное число. Это не настоящая функция.

Советы и приемы

- Если X равен «сигнализирующему» NaN, Delphi выдает ошибку времени выполнения 6 (EInvalidOp).
- Если X равен «спокойному» NaN, плюс бесконечности или минус бесконечности, результат равен X.
- Для отбрасывания дробной части и получения целочисленного результата используйте функцию Trunc.

Смотрите также

Функции Frac, Round, Trunc.

Int64, тип

Синтаксис

```
type Int64 = -9223372036854775808..9223372036854775807;
```

Описание

Int64 — это 64-битовый целый тип со знаком.

Советы и приемы

- Используйте Int64 вместо типа Comp. Int64 не зависит от модуля операций с плавающей точкой и от режима точности, установленного в слове состояния FPU. Кроме того, Int64 поддерживает целочисленные операторы, такие как shl и shr.
- Целая константа, значение которой слишком велико для типа Integer, автоматически преобразуется к типу Int64.
- Все обычные целые функции (например, IntToStr) имеют перегруженные версии, поддерживающие аргументы Integer и Int64.
- В собственных подпрограммах, работающих с типом Integer, рассмотрите возможность создания перегруженных версий, поддерживающих Int64. Если, например, вы пишете функцию IntToBi-

nary для преобразования целого в строку символов '0' и '1', создайте две перегруженные функции – одну для аргумента типа Integer и другую – для Int64.

- 64-битового целого типа без знака не существует.

Смотрите также

Тип Integer.

Integer, тип

Синтаксис

```
type Integer = -2147483648..2147483647;
```

Описание

Тип Integer – это базовый целый тип. Размер Integer может различаться в разных версиях Delphi. В текущей версии Integer – это 32-битовый тип, но в будущих версиях его размер может стать другим.

В Delphi определено несколько целых типов с различными диапазонами значений. Предопределенные целые типы приведены ниже.

Тип	Минимум	Максимум
Byte	0	255
ShortInt	-128	127
Word	0	65,535
SmallInt	-32,768	32,767
LongWord	0	4,294,967,295
LongInt	-2,147,483,648	2,147,483,647
Integer	Естественный (определяется операционной системой)	Естественный
Cardinal	0	Естественный
Int64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Советы и приемы

- Числовые константы имеют тип Integer, если попадают в диапазон типа Integer; иначе они относятся к типу Int64. Результат арифметических выражений, в которых присутствует хотя бы один аргумент типа Int64, имеет тип Int64. Если все аргументы имеют тип Integer, то и результат также принадлежит к этому типу.
- Размеры типов Cardinal и Integer могут меняться от одной версии Delphi к другой. В Delphi 5 оба являются 32-битовыми, но в будущих версиях, без сомнения, размер изменится на 64 бита.

- Везде, где это возможно, определяйте свой собственный поддиапазон. Например, если известно, что значение всегда положительно, объявите переменную соответствующим образом. Дополнительные ограничения типов помогают в поиске ошибок (случайное присваивание переменной неположительного значения), кроме того, код программы будет документирован, что даст читателю больше информации о переменной.
- `Cardinal` является поддиапазоном `Int64`, поэтому смешивание в выражении значений `Cardinal` и `Integer` приводит к преобразованию всех аргументов к типу `Int64`. Если нужно выполнить арифметические действия с неотрицательными целыми, используйте поддиапазон `Integer`.

Пример

```
// Count никогда не может быть отрицательным, поэтому не
// используйте Integer. Cardinal также не лучший выбор, поскольку
// при выполнении арифметических операций с Count и другим
// значением типа Integer будет выполняться преобразование в Int64.
// Поэтому объявляем поддиапазон внутри диапазона Integer,
// при этом ограничиваем тип значений, доступных для счетчика.
var Count: 0..MaxInt;

// Если определяется много типов, выбирайте полезные имена.
type
  WholeNumber = 1..MaxInt;
  Natural = 0..MaxInt;
```

Смотрите также

Типы `Byte`, `Cardinal`, `Int64`, `LongInt`, `LongWord`, `ShortInt`, `SmallInt`, `Word`.

Interface, ключевое слово

Синтаксис

```
unit Имя;
interface
  Объявления...
implementation
  Объявления...
end.


type Имя = interface
  ...
end;

type Имя = interface(Базовый Интерфейс, ...)
  ...
end;
```

Описание

Ключевое слово `interface` служит для выполнения двух абсолютно несвязанных функций.

Чаще всего ключевое слово `interface` обозначает начало интерфейсного раздела модуля. Каждый модуль должен иметь раздел интерфейса, возможно, пустой.

Объявления в разделе интерфейса экспортируются и могут использоваться любым другим модулем, включившим данный модуль в список директивы `uses`.

Второе применение ключевого слова `interface` – объявление интерфейсного типа. Интерфейс определяет абстрактный протокол, реализуемый классом. Он может содержать описания методов и свойств, но не может иметь полей. Все объявления в интерфейсе являются открытыми (public).

Интерфейсный раздел

- Если присутствует объявление `uses`, оно должно быть первым в разделе интерфейса.
- Все функции, процедуры и неабстрактные методы, объявленные в разделе интерфейса, должны быть определены в разделе реализации. Имена и типы параметров в определении должны соответствовать объявленным в разделе интерфейса.

Интерфейсные типы

- Хотя интерфейсы выглядят и работают как COM-интерфейсы, программист не ограничен только использованием COM. Интерфейсы – мощная (и недооцененная) технология создания объектно-ориентированного кода. Интерфейсы обеспечивают автоматическое управление памятью и улучшенный полиморфизм. В главе 2 «Объектная модель Delphi» эта тема обсуждается подробно.
- При необходимости Delphi автоматически преобразует ссылку на интерфейс в GUID. Поэтому можно использовать имя интерфейса для вызова метода `QueryInterface`, например:

```
var
  Stream: IStream;
begin
  if Something.QueryInterface(IStream, Stream) = S_OK then
    Stream.Revert;
```

Пример

Пример интерфейсного раздела смотрите в описании ключевого слова `unit`. Примеры интерфейсных типов смотрите в описаниях интерфейсов `IDispatch` и `IUnknown`.

Смотрите также

Ключевые слова Class, Dispinterface, Implementation, Type, Unit, Uses, директива External, интерфейсы IDispatch, IUnknown.

IOResult, функция

Синтаксис

```
function IOResult: Integer;
```

Описание

В Delphi имеется два способа уведомления об ошибке ввода/ввода: ошибки времени выполнения и функция IOResult. По умолчанию Delphi сообщает об ошибках ввода/вывода как о других ошибках времени выполнения, и модуль SysUtils преобразует эти ошибки в исключительные ситуации. Если для отключения ошибок ввода/вывода применяется директива $I- или $IOChecks Off, Delphi возвращает статус операций ввода и вывода через функцию IOResult. Программист сам отвечает за своевременный вызов IOResult для проверки успеха или неуспеха вызова каждой процедуры или функции ввода/вывода.

IOResult возвращает ноль в случае успеха или код ошибки в случае неудачи. Код ошибки может быть либо кодом ошибки Windows, либо одним из следующих кодов ошибок Delphi.

Код ошибки	Описание
100	Чтение за границами файла
101	Диск переполнен
102	Не была вызвана процедура AssignFile
103	Файл закрыт
104	Файл не открыт для ввода
105	Файл не открыт для вывода
106	Неверный формат данных для процедуры Read

После вызова IOResult код ошибки ввода/вывода сбрасывается в ноль. IOResult – настоящая функция.

Советы и приемы

- В каждом потоке имеется собственный код результата операций ввода/вывода, поэтому вызывайте IOResult в контексте нужного вам потока.
- Программист сам отвечает за вызов IOResult после обращения к каждой подпрограмме ввода/вывода. Если не проверить значение

IOResult, последующий вызов подпрограммы ввода/вывода перепишет старый код ошибки новым значением.

- Модуль SysUtils преобразует ошибки ввода/вывода в исключительную ситуацию EInOutError. Этот класс содержит текстовые сообщения лишь для небольшого числа кодов ошибок ввода/вывода. В реальном приложении следует перехватывать EInOutError и анализировать код ошибки самостоятельно. Например, функции обработчика OnException в приложении может выполнять следующий код:

```
procedure TForm1.AppEventsException(Sender: TObject; E: Exception);
resourcestring
  sEOF         = 'Попытка чтения за границами файла';
  sNotAssigned = 'Файл не назначен;
  sNotOpen     = 'Файл не был открыт';
  sNotRead     = 'Файл не был открыт для ввода';
  sNotWrite    = 'Файл не был открыт для вывода';
  sBadRead     = 'Ошибка входного формата';
begin
  if E is EInOutError then
    case EInOutError(E).ErrorCode of
    100: E.Message := sEOF;
    101: E.Message := SysErrorMessage(Error_Disk_Full);
    102: E.Message := sNotAssigned;
    103: E.Message := sNotOpen;
    104: E.Message := sNotRead;
    105: E.Message := sNotWrite;
    106: E.Message := SBadRead;
    else E.Message := SysErrorMessage(EInOutError(E).ErrorCode);
    end;
  Application.ShowException(E);
end;
```

Пример

```
// Создание каталога. Если каталог существует, ничего делать не нужно.
procedure CreateDir(const Dir: string);
type
  TErrorProc = procedure(Error: integer; Addr: Pointer);
var
  Error: Integer;
begin
  {$IOChecks Off}
  MkDir(Dir);
  {$IOChecks On}
  Error := IOResult;
  if (Error <> 0) and (Error <> Error_Already_Exists) then
    // Some error other than that the directory already exists.
    TErrorProc(ErrorProc)(Error, @CreateDir);
end;
```

Смотрите также

Процедуры Append, AssignFile, BlockRead, BlockWrite, ChDir, CloseFile, Erase, Flush, MkDir, Read, ReadLn, Rename, Reset, Rewrite, RmDir, Seek, SetTextBuf, Truncate, Write, WriteLn, функции Eof, Eoln, FilePos, FileSize, SeekEof, SeekEoln, директивы компилятора $I, $IOChecks.

Is, ключевое слово

Синтаксис

Ссылка на объект is *Ссылка на класс*

Описание

Оператор is проверяет, является ли тип ссылки на объект таким же, что и для данной ссылки на класс или для потомка этого класса. True возвращается в том случае, если объект имеет тот же или производный тип от указанного класса; иначе возвращается False.

Советы и приемы

- Для проверки типа интерфейса используйте метод QueryInterface или функцию Supports из модуля SysUtils.
- Оператор is вызывает метод объекта InheritsFrom.
- При проверке типа объекта с помощью is применение оператора as не требуется. Большей производительности можно достичь с помощью простого преобразования типов.
- Оператор is требует в левой части наличие ссылки на объект. Для проверки ссылки на класс служит метод InheritsFrom для TObject.

Пример

```
// Двойной щелчок очищает любое поле ввода. Установите обработчик
// события OnDblClick для всех полей ввода на эту процедуру.
procedure TForm1.EditDblClick(Sender: TObject);
begin
  if Sender is TCustomEdit then
    TCustomEdit(Sender).Clear;
end;
```

Смотрите также

Ключевое слово As, типы Class, TObject.

IsConsole, переменная

Синтаксис

```
var IsConsole: Boolean;
```

Описание

Delphi устанавливает переменную IsConsole в True для консольного приложения или в False – для графического. Можно проверить IsConsole перед попыткой чтения из Input или записи в Output.

Пример

```
// Выдача сообщения об ошибке.
procedure ErrorHandler(Code: Integer; Addr: Pointer);
var
  ErrorMessage: string;
begin
  ErrorMessage := Format('Ошибка %d в $%p', [Code, Addr]);
  if IsConsole then
    WriteLn(ErrorMessage)
  else
    ShowMessage(ErrorMessage);
  Halt(Code);
end;
```

Смотрите также

Переменные Input, Output, директива компилятора $AppType.

IsLibrary, переменная

Синтаксис

```
var IsLibrary: Boolean;
```

Описание

Delphi устанавливает следующие значения переменной IsLibrary: True – в проекте библиотеки (DLL), False – в программном проекте (EXE).

Советы и приемы

- В большинстве случаев следует употреблять переменную ModuleIsLib вместо IsLibrary, чтобы избежать сложностей, связанных с пакетами.
- Если библиотека работает с пакетом *vcl50* и загружается в приложение, также использующее пакет *vcl50*, то библиотека и программа будут использовать общий экземпляр пакета *vcl50*. Приложение устанавливает IsLibrary в ложь и библиотека видит это значение IsLibrary из пакета *vcl50*. Любой написанный программистом модуль также может быть одновременно загружен приложением и библиотекой в одном процессе, и потому возможность применения функции IsLibrary ограничена. Переменной ModuleIsLib эта проблема не свойственна.

Смотрите также

Ключевое слово Library, переменные ModuleIsLib, ModuleIsPackage.

IsMemoryManagerSet, функция

Синтаксис

```
function IsMemoryManagerSet: Boolean;
```

Описание

Функция IsMemoryManagerSet возвращает True, если текущий менеджер памяти отличен от встроенного менеджера памяти Delphi. Другой менеджер памяти устанавливается процедурой SetMemoryManager.

IsMemoryManager — настоящая функция.

Советы и приемы

Программист, реализующий новый менеджер памяти, наверняка знает, что его программа вызывает функцию SetMemoryManager. Функция IsMemoryManagerSet позволяет определить, имеет ли смысл применять такие функции, как GetHeapStatus. Если IsMemoryManagerSet возвращает True, следует предполагать, что функция GetHeapStatus не выдаст какой-либо полезной информации. Изучите документацию по новому менеджеру памяти, чтобы узнать, какая функция (если она существует) заменяет GetHeapStatus.

Смотрите также

Функция GetHeapStatus, процедуры GetMemoryManager, SetMemoryManager.

IsMultiThread, переменная

Синтаксис

```
var IsMultiThread: Boolean;
```

Описание

Delphi автоматически устанавливает значение переменной IsMultiThread в True, когда создается поток с помощью BeginThread или класса TThread из модуля Classes. Для многопоточной защиты можно проверить переменную IsMultiThread и, если ее значение равно False, исключить исполнение кода.

Советы и приемы

- Менеджер памяти Delphi проверяет IsMultiThread и с помощью критических секций защищает целостность своих внутренних структур данных. Однако, если поток создается с помощью функции CreateThread из Windows API, необходимо явно установить значение IsMultiThread в True.

- Delphi также устанавливает IsMultiThread в True для веб-серверов и COM-серверов, кроме тех, которые используют однопоточную модель (tmSingle).
- При написании своей DLL, которая может быть загружена многопоточным приложением, установите IsMultiThread в True. Применяя DllProc для получения информации о присоединении к DLL дополнительного потока, следует понимать, что приложение уже могло запустить несколько потоков до загрузки DLL.

Смотрите также

Функция BeginThread, переменная DllProc, ключевое слово ThreadVar.

IUnknown, интерфейс

Синтаксис

```
type IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

Описание

Интерфейс IUnknown является базовым для всех интерфейсов в Delphi. Каждый класс, реализующий какой-либо интерфейс, должен также реализовывать IUnknown. Для удобства программиста в Delphi определен класс TInterfacedObject, реализующий IUnknown.

В случае присваивании интерфейса переменной или передачи его в качестве параметра подпрограммы Delphi вызывает метод _AddRef. При выходе переменной интерфейсного типа из своей области действия, автоматически вызывается метод _Release.

Советы и приемы

- IUnknown является базовым интерфейсом COM, следовательно, любой класс Delphi может реализовывать любой COM-интерфейс. Delphi-программа может использовать COM-объекты так же, как и любые другие интерфейсы. Однако применение COM совершенно не обязательно. Многие варианты употребления интерфейсов никак не связаны с COM.
- Хотя принято соглашение, что методы _AddRef и _Release управляют счетчиком ссылок, программист может реализовать их по своему усмотрению. Можно воспользоваться преимуществами интерфейсов и без счетчика ссылок, реализовав эти методы как заглушки. Возвращайте ненулевое значение, т. к. нулевой результат этих методов означает, что счетчик ссылок объекта достиг нуля.

- Delphi реализует оператор as для интерфейсов с помощью вызова метода QueryInterface. Логичнее всего реализовать QueryInterface вызовом метода GetInterface, но можно и любым другим способом. Обязательно возвращайте S_OK (ноль) в случае успеха; любой другой результат, возвращаемый QueryInterface, сообщает оператору as о необходимости генерирования ошибки времени выполнения (EIntFCastError).
- Если нужно лишь проверить, поддерживается ли данный интерфейс классом, и код для вызова исключительной ситуации не требуется, вызывайте метод QueryInterface напрямую вместо оператора as, либо воспользуйтесь функцией Supports модуля SysUtils.

Смотрите также

Ключевые слова As, Interface.

JITEnable, переменная

Синтаксис

```
var JITEnable: Byte;
```

Описание

Переменная JITEnable определяет, какие исключительные ситуации вызывают запуск оперативного (just-in-time) отладчика.

Советы и приемы

- По умолчанию значение JITEnable равно 0, и только необработанные исключительные ситуации вызывают оперативный отладчик.
- Значение 1 означает, что оперативный отладчик вызывается только при возникновении исключений, не обрабатываемых Delphi.
- Значение 2 и более означает, что оперативный отладчик запускается при возникновении любой исключительной ситуации.

Смотрите также

Переменные DebugHook, ErrorProc, ExceptionClass, ExceptProc, ключевое слово Raise.

Label, ключевое слово

Синтаксис

```
label Цифры, Идентификатор, ...;
```

Описание

Ключевое слово label объявляет одну или несколько меток. Метка может быть строкой до четырех цифр (как в стандартном Паскале) или идентификатором (расширение стандартного Паскаля). Метка может

быть использована далее в этом блоке для идентификации точки, в которую передает управление оператор goto. Можно также применять метку для инструкции JMP в ассемблерном блоке. Дополнительную информацию смотрите в описании ключевого слова goto.

Советы и приемы

В ассемблерном блоке можно объявить метки, обозначенные символом @. Такие метки являются локальными для подпрограммы.

Смотрите также

Ключевые слова Asm, Goto.

Length, функция

Синтаксис

```
function Length(const S: String): Integer;
function Length(const A: array): Integer;
```

Описание

Функция Length возвращает количество элементов в строке или массиве. Функция Length не является настоящей функцией.

Советы и приемы

- Обычно Length служит для определения размера динамического массива, однако можно применять эту функцию для нахождения длины открытого и статического массива.
- Для массива Length(A) всегда возвращает Ord(High(A)) - Ord(Low(A)) + 1. Однако для ShortString функция Length возвращает значение, хранящееся в байте длины, которое может быть меньше размера строковой переменной.

Пример

```
// Переключаем регистр символов строки.
procedure ToggleCase(var S: string);
var
  I: Integer;
begin
  for I := 1 to Length(S) do
    if S[I] in ['a'..'z'] then
      S[I] := UpCase(S[I])
    else if S[I] in ['A'..'Z'] then
      S[I] := DownCase(S[I]);
end;
```

Смотрите также

Функции High, Low, SizeOf, процедура SetLength.

LibModuleList, переменная

Синтаксис

```
var LibModuleList: PLibModule;
```

Описание

Переменная LibModuleList указывает на головной элемент односвязного списка записей TLibModule. Каждый узел списка содержит информацию об исполняемом модуле, т. е. о приложении, библиотеке, пакете и других DLL, загруженных приложением.

Советы и приемы

- Не связывайтесь с переменной LibModuleList понапрасну. Delphi предоставляет несколько функций, которые могут сами работать со списком исполняемых модулей: EnumModules, EnumResourceModules, RegisterModule и UnregisterModule.
- Delphi автоматически вставляет узел в список при загрузке пакета или DLL. Чтобы просмотреть содержимое списка, используйте EnumModules или, для просмотра соответствующих модулей ресурсов, EnumResouceModules.

Смотрите также

Процедуры EnumModules, EnumResourceModules, RegisterModule, UnregisterModule, директива Package, типы PLibModule, TLibModule.

Library, ключевое слово

Синтаксис

```
library Имя;
  Описания...
Блок.
```

Описание

Библиотечный проект, обозначаемый ключевым словом library, служит для создания динамически присоединяемой библиотеки (DLL). Когда Windows загружает библиотеку, Delphi сначала запускает разделы инициализации всех модулей. Затем исполняется основной блок библиотеки.

Советы и приемы

- Если библиотека передает строки, динамические массивы или значения типа Variant между исполняемыми модулями (приложением и другими библиотеками), необходимо использовать в качестве первого модуля ShareMem. Подробности приведены в главе 2.

- Если библиотека выделяет память (например, объекты), освобождающуюся приложением или другой библиотекой, или освобождает объекты, созданные другим исполняемым модулем, следует применять модуль ShareMem.
- Передавая в качестве аргументов или возвращаемых значений ссылки на объекты, имейте в виду, что операторы is и as могут работать некорректно. Каждый исполняемый модуль (программа или библиотека) содержит собственную копию RTTI каждого класса. Операторы is и as используют для проверки классов RTTI, и потому разные RTTI обозначают разные классы. Если этот момент важен, применяйте вместо библиотек пакеты. Подробное описание RTTI можно найти в главе 3.

Пример

```
library FontNames;

// Delphi позволяет расширять возможности IDE, загружая
// пакеты или DLL, использующие Open Tools API. Это довольно
// тривиальное расширение IDE должно компилироваться с пакетом
// времени выполнения VCL50. Чтобы установить расширение, добавьте
// в реестр запись по ключу
// HKEY_CURRENT_USER\Software\Borland\Delphi\5.0\Experts. Имя
// должно быть уникальной строкой, значение – указывать путь к
// этой DLL.
//
// После компиляции этой DLL создайте запись в реестре,
// перезапустите Delphi, откройте инспектор объектов и выберите
// свойство Font.Name. Раскройте список имен шрифтов и
// понаблюдайте. Если у вас установлено много шрифтов, это
// займет много времени. Удалите запись из реестра и перезапустите
// Delphi для удаления расширения IDE.

uses ShareMem, DsgnIntf, ToolsAPI;

function Init(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean; stdcall;
begin
  FontNamePropertyDisplayFontNames := True;
  Result := True;
end;

exports Init name WizardEntryPoint;

end.
```

Смотрите также

Ключевые слова Asm, Begin, Exports, Program, **переменная** DllProc, **директивы** External, Package.

Ln, функция

Синтаксис

```
function Ln(X: Вещественный тип): Extended;
```

Описание

Функция Ln возвращает значение натурального логарифма X. Функция Ln – встроенная.

Советы и приемы

- Delphi автоматически преобразует аргументы Integer и Variant в вещественное число. Для преобразования аргумента Int64 к вещественному типу прибавьте к нему 0,0.
- Если X равен «сигнализирующему» NaN или отрицательному значению (в том числе – минус бесконечности), Delphi генерирует ошибку времени выполнения 6 (EInvalidOp).
- Если X равен нулю, Delphi выдает ошибку времени выполнения 7 (EZeroDivide).
- Если X равен плюс бесконечности, результат равен плюс бесконечности.
- Если X равен «спокойному» NaN, результат равен X.

Смотрите также

Функция Exp.

Lo, функция

Синтаксис

```
function Lo(Value: Integer): Byte;
```

Описание

Функция Lo возвращает самый младший значащий байт целого. Функция Lo не является настоящей функцией, а компилируется в inline код.

Советы и приемы

Функция Lo эквивалентна следующей функции:

```
function Lo(Value: Integer): Byte;
begin
  Result := Value and $FF;
end;
```

Смотрите также

Функция Hi.

LoadResourceModule, функция

Синтаксис

```
function LoadResourceModule(ModuleName: PChar): LongWord;
```

Описание

Delphi автоматически вызывает LoadResourceModule для загрузки языковых DLL, необходимых приложению. Если такая DLL не найдена, функция возвращает ноль; в противном случае DLL загружается как файл данных и возвращается дескриптор ее экземпляра.

Когда Delphi загружает исполняемый модуль (приложение, DLL или пакет), автоматически вызывается функция LoadResourceModule для поиска DLL, имя которой соответствует имени модуля и с расширением, соответствующим региональным установкам. Функция LoadResouceModule ищет языковое расширение в трех местах. На каждом этапе функция пытается загрузить языковую DLL, и если это не удается, переходит к следующему шагу в последовательности:

1. Во-первых, проверяется реестр. Происходит поиск полного пути приложения в следующих ключах реестра:

   – HKEY_CURRENT_USER\Software\Borland\Delphi\Locales

   – HKEY_CURRENT_USER\Software\Borland\Locales

   Если приложение находит запись в одном из ключей, значение этой записи воспринимается как языковое расширение для DLL. Если приложение не находит запись, но в ключе имеется значение по умолчанию, оно используется как языковое расширение.

2. Затем происходит поиск DLL, расширение которой соответствует местному коду языка и страны.

3. И наконец, выполняется поиск DLL, расширение которой равно коду языка. Если LoadResourceModule находит DLL с языковым расширением, функция загружает DLL и возвращает ее дескриптор экземпляра. Кроме того, дескриптор экземпляра сохраняется, чтобы функция FindResourceHInstance искала ресурсы в локализованной DLL. LoadResourceModule – настоящая функция.

Смотрите также

Функции FindResourceHInstance, LoadResString.

LoadResString, функция

Синтаксис

```
function LoadResString(Rec: PResStringRec): string;
```

Описание

Функция LoadResString загружает строковый ресурс. Когда объявляется строка с помощью ключевого слова resourcestring, Delphi автоматически создает ресурс со строковой таблицей и присваивает строке уникальный идентификатор. Для поиска строки компилятор генерирует код с вызовом LoadResString и указанием идентификатора ресурса.

LoadResString – настоящая функция.

Советы и приемы

Delphi работает со строковыми ресурсами автоматически, и потому нет нужды самостоятельно вызывать эту функцию. Если такая необходимость возникает, можно передать в качестве аргумента адрес идентификатора resourcestring.

Смотрите также

Функция LoadResouceModule, тип PResStringRec, TResStringRec, ключевое слово ResouceString.

LongBool, тип

Синтаксис

```
type LongBool;
```

Описание

Тип LongBool – это логический тип, размер которого равен размеру LongWord. Значение LongBool ложно, если его порядковое значение равно нулю, и истинно, если порядковое значение не равно нулю. Значению «истина» для типа LongBool – (LongBool(True)) LongBool соответствует –1.

Советы и приемы

- Значения типа LongBool могут быть использованы в тех же случаях, что и Boolean. Больше всего они подходят для интерфейса с C и C++, где любое ненулевое целое рассматривается как истина.
- LongBool особенно полезен как тип возвращаемого значения многих функций Windows API, возвращающих, согласно документации, ноль в случае неудачи и любое ненулевое значение в случае успеха.
- ByteBool и WordBool аналогичны LongBool, но имеют другой размер.

Смотрите также

Ключевые слова And, Not, Or, Xor, типы Boolean, ByteBool, WordBool.

LongInt, тип

Синтаксис

```
type LongInt = -2147483648..2147483647;
```

Описание

LongInt – 32-битовое целое со знаком. Даже если размер типа Integer изменится в будущей версии Delphi, LongInt останется прежним. Обратитесь к описанию типа Integer за дополнительной информацией об этом и других целых типах.

Смотрите также

Типы Integer, LongWord, константа MaxLongInt.

LongWord, тип

Синтаксис

```
type LongWord = 0..4294967295;
```

Описание

LongWord – это 32-битовое целое без знака. Даже если размер Integer и Cardinal изменится в будущей версии Delphi, LongWord останется прежним.

О применении этого типа читайте в описании типа Cardinal. Дополнительную информацию об этом и других целых типах можно найти в описании типа Integer.

Смотрите также

Типы Cardinal, Integer.

Low, функция

Синтаксис

```
function Low(Тип или переменная): Порядковый тип;
```

Описание

Функция Low возвращает наименьшее значение перечисляемого типа, нижнюю границу индекса массива или подробную информацию о переменной порядкового типа или типа «массив».

Функция Low встроена в компилятор и не является настоящей функцией.

Советы и приемы

- В цикле for, объявлении поддиапазона или в любой другой ситуации, где присутствуют границы порядкового типа или типа массива, всегда применяйте функцию Low вместо явного указания нижнего порядкового значения. В последующей версии своего кода программист может изменить тип, и вряд захочет просматривать весь код в поисках явных ссылок на перечисляемое значение, вместо которого должен был быть вызов функции Low.

- Вызов Low для ShortString возвращает нулевое значение, которое является индексом байта длины строки. Не допускается применение Low для AnsiString или WideString, поэтому используйте единицу, т. к. первый элемент длинной строки всегда имеет индекс 1.
- Значение функции Low для открытого массива всегда равно нулю, независимо от типа или диапазона фактического параметра-массива.

Пример

Пример рассмотрен в описании функции High.

Смотрите также

Процедуры Dec, Inc, Length, функции High, Pred, Succ.

MainInstance, переменная

Синтаксис

```
var MainInstance: LongWord;
```

Описание

В переменной MainInstance хранится дескриптор экземпляра основного исполняемого модуля программы (*exe*-файла).

Советы и приемы

- Любой модуль в пакете может обратиться к MainInstance для получения дескриптора основной программы. Даже пакеты, загруженные отдельными DLL, могут использовать MainInstance.
- Если библиотека не работает с пакетом *vcl50*, то значение переменной MainInstance, равное в этом случае нулю, для нее не имеет смысла.

Пример

```
// Ресурс MAINICON содержится в файле ресурсов проекта, который
// находится внутри основного исполняемого модуля. Для загрузки
// ресурса требуется дескриптор экземпляра основного модуля. Эта
// функция загружает ресурс MAINICON и возвращает его дескриптор.
function LoadMainIcon: HICON;
begin
  Result := LoadIcon(MainInstance, 'MAINICON');
end;
```

Смотрите также

Функции FindClassHInstance, FindHInstance, переменные HInstance, ModuleIsLib, ModuleIsPackage.

MainThreadID, переменная

Синтаксис

```
var MainThreadID: LongWord;
```

Описание

Переменная MainThreadID содержит значение Windows-идентификатора основного потока приложения. С помощью MainThreadID можно, например, отправить сообщение основному потоку.

Смотрите также

Функция BeginThread, переменная IsMultiThread.

MaxInt, константа

Синтаксис

```
const MaxInt = High(Integer);
```

Описание

Константа MaxInt – это максимально возможное значение типа Integer. Она является удобным сокращением для High(Integer). Точное значение MaxInt может меняться от одной версии Delphi к другой. В текущей версии эта константа равна 2,147,483,647.

Советы и приемы

Для кода, который взаимодействует с библиотеками, написанными на C или C++ (в том числе Windows API), принято общее правило, в соответствии с которым массив объявляется с максимальными размерами. Один из параметров функции или поля записи содержит истинный размер массива. Максимальный размер массива зависит от размера базового типа, поэтому можно объявить тип массива следующим образом:

```
type CArray = array[0..MaxInt div SizeOf(BaseType) - 1] of BaseType;
```

Смотрите также

Тип Integer, константа MaxLongInt.

MaxLongInt, константа

Синтаксис

```
const MaxLongInt = High(LongInt);
```

Описание

Константа MaxLongInt – удобное сокращения для High(LongInt), которое имеет значение 2,147,483,647.

Смотрите также

Типы Integer, LongInt, константа MaxInt.

Message, директива

Синтаксис

```
procedure Имя(var Msg); message Константа;
```

Описание

Если указана директива message, метод является обработчиком события. Обработчик события должен иметь один параметр var; обычно этот аргумент имеет тип TMessage или один из аналогичных типов из модуля Messages. Единственное требование, накладываемое Delphi – аргумент Msg должен иметь размер не меньше, чем Word. Чтобы определить, какой обработчик сообщения необходимо вызвать, Delphi проверяет первое слово параметра Msg. В определении обработчика сообщения должна быть указана константа – номер сообщения. Этот номер сохраняется в RTTI класса.

Советы и приемы

- Обработчики сообщений и динамические методы используют общую таблицу RTTI. Delphi присваивает динамическим методам в качестве индексов небольшие отрицательные значения. Чтобы избежать конфликтов с динамическими методами, не употребляйте для обработчиков событий номера с установленным старшим битом, т. е. используйте номера меньше $8000 (32768).
- В любом классе – не только в управляющих элементах и компонентах – могут быть определены обработчики сообщений. Визуальные элементы преобразуют сообщения Windows в сообщения Delphi, и метод Dispatch класса TObject вызывает соответствующий обработчик сообщения. Раньше сообщения были необходимы для реализации межтипового полиморфизма, когда объекты могли реагировать на сообщение независимо от их базового класса. На смену этому механизму пришли интерфейсы, и потому единственным применением обработчиков сообщений в новом коде остается обработка сообщений Windows.
- В модуле Control определяется много сообщений Delphi. Каждый, кто создает новые управляющие элементы или компоненты, должен ознакомиться с этими сообщениями и решить, какие сообщения новый элемент должен обрабатывать и какие – отправлять.

Пример

```
// Управляющий элемент, изображающий время дня на циферблате.
// Если Windows изменяет системное время, всем окнам верхнего
// уровня рассылается сообщение Wm_TimeChange. Форма, в свою
// очередь, рассылает Cm_TimeChange всем своим элементам.
```

```
// Этот управляющий элемент содержит обработчик сообщения
// для перевывода времени.
type
  TClock = class(TCustomControl)
  private
    fTime: TDateTime;
    procedure CmTimeChange(var Msg: TWmTimeChange);
        message Cm_TimeChange;
  ...
  end;
```

Смотрите также

Ключевые слова `Class`, `Procedure`, директива `Dynamic`, тип `TObject`.

MkDir, процедура

Синтаксис

```
procedure MkDir(const Directory: string)
```

Описание

Процедура `MkDir` служит для создания каталога. Если каталог не может быть создан, Delphi генерирует ошибку ввода/вывода с кодом ошибки Windows, например, `Error_Already_Exists` – если каталог уже существует.

Включать в имя каталога обратную косую черту не обязательно.

`MkDir` не является настоящей процедурой.

Пример

```
// Создаем каталог, но только в том случае, если он
// еще не существует.
procedure MakeDirectory(const Directory: string);
var
  Search: TSearchRec;
begin
  if FindFirst(Directory, faDirectory, Search) = 0 then
    FindClose(Search)
  else
    MkDir(Directory);
end;
```

Смотрите также

Процедуры `ChDir`, `GetDir`, функцию `IOResult`, процедуру `RmDir`.

Mod, ключевое слово

Синтаксис

Целое выражение `mod` *Целое выражение*

Описание

Оператор `mod` вычисляет модуль или остаток от деления. Результат операции `A mod B` равен `A - (A div B) * B`.

Советы и приемы

- Если `A` и `B` – положительны, модуль и остаток равны.
- `A mod B` всегда имеет одинаковое абсолютное значение, независимо от знака `A` и `B`. Знак `A mod B` совпадает со знаком `A`.
- Если `B` = 0, Delphi выдает ошибку 3 (`EDivByZero`).

Смотрите также

Ключевое слово `Div`.

ModuleIsCpp, переменная

Синтаксис

```
unit SysInit;
var ModuleIsCpp: Boolean;
```

Описание

Переменная `ModuleIsCpp` равна `True`, если исполняемый модуль компилировался и собирался в C++ Builder и `False`, если исполняемый модуль компилировался и собирался в Delphi.

Советы и приемы

В отличие от большинства других системных переменных, `ModuleIsCpp` не разделяется между всеми пакетами приложения.

ModuleIsLib, переменная

Синтаксис

```
unit SysInit;
var ModuleIsLib: Boolean;
```

Описание

Значение переменной `ModuleIsLib` равно `True` для модуля, являющегося частью DLL или пакета, и `False`, если модуль статически присоединен к приложению.

Советы и приемы

В отличие от большинства других системных переменных, `ModuleIsLib` не разделяется между всеми пакетами приложения. Это делает ее более полезной, чем переменная `IsLibrary`.

Смотрите также

Переменные HInstance, IsLibrary, ModuleIsPackage, ключевое слово Library, директива Package.

ModuleIsPackage, переменная

Синтаксис

```
unit SysInit;
var ModuleIsPackage: Boolean;
```

Описание

ModuleIsPackage равна True для модуля, находящегося в пакете, и False для модуля, статически присоединенного к библиотеке или приложению.

Советы и приемы

В отличие от большинства других системных переменных, ModuleIsPackage не разделяется между всеми пакетами приложения.

Смотрите также

Переменные HInstance, ModuleIsLib, директива Package.

ModuleUnloadList, переменная

Синтаксис

```
var ModuleUnloadList: PModuleUnloadRec;
```

Описание

Переменная ModuleUnloadList указывает на начало односвязного списка записей TModuleUnloadRec. Каждая из записей указывает на процедуру, вызываемую при выгрузке исполняемого модуля (приложения, библиотеки или пакета).

Советы и приемы

- Новые процедуры добавляются в начало списка с помощью процедуры AddModuleUnloadProc; удаление из списка происходит с помощью процедуры RemoveModuleUnloadProc. Delphi автоматически обходит список и вызывает все процедуры при выгрузке пакета, библиотеки и завершении приложения.
- Большинству программ никогда потребуется значение переменной ModuleUnloadList.

Смотрите также

Процедуры AddModuleUnloadProc, RemoveModuleUnloadProc, UnregisterModule, типы PModuleUnloadRec, TModuleUnloadRec.

Move, процедура

Синтаксис

```
procedure Move(const Source; var Dest; Count: Integer);
```

Описание

Процедура Move копирует Count байт из Source в Dest. Move – настоящая процедура.

Советы и приемы

- Source и Dest не являются указателями, поэтому передавайте сами переменные или разыменованные указатели на динамически выделенную памяти.
- Move может корректно работать с перекрывающимися участками памяти.

Пример

```
// Вставляем элемент в середину массива. Игнорируем текущее
// значение A[High(A)].
procedure ArrayInsert(var A: array of Integer; Value, Index: Integer);
begin
  Assert((Low(A) <= Index) and (Index <= High(A));
  // Сначала освобождаем место для нового значения.
  Move(A[Index], A[Index+1], (High(A)-Index) * SizeOf(Integer));
  // Записываем в массив значение.
  A[Index] := Value;
end;
```

Смотрите также

Процедура FillChar.

Name, директива

Синтаксис

```
exports Сигнатура подпрограммы name String, ... ;
Объявление подпрограммы; external Имя DLL name Строка;
```

Описание

Модуль или библиотека могут экспортировать любую подпрограмму и указать для нее другое имя, которое будет присутствовать в DLL. Без указания директивы name Delphi использует имя функции или процедуры. Директива name позволяет включать в имена небуквенные и нецифровые символы, или просто указать другое имя.

Советы и приемы

- При экспорте перегруженных подпрограмм приходится применять директиву name для того, чтобы эти подпрограммы имели уникальные имена.
- Допускается указание директивы index и name для одной экспортируемой подпрограммы (в указанном порядке).
- Дополнительную информацию и пример смотрите в описании ключевого слова exports.
- Чтобы экспортировать подпрограмму только с индексом, укажите для имени пустую строку.

Смотрите также

Ключевое слово Exports, директивы External, Index, Name.

Near, директива

Синтаксис

```
Заголовок подпрограммы; near;
```

Описание

Директива near значения не имеет. Она оставлена только для обратной совместимости с Delphi 1.

Смотрите также

Директива Far.

New, процедура

Синтаксис

```
procedure New(var P: Тип указателя);
procedure New(var P: ^object; Constructor);
```

Описание

Процедура New выделяет память для новой переменной типа «указатель» или для «старого» объекта. Чаще всего эта процедура служит для установки в переменной-указателе ссылки на динамически выделенную память. New вызывает процедуру GetMem для выделения памяти и затем процедуру Initialize для инициализации строк, динамических массивов, интерфейсов или значений типа Variant, сформированных в этой памяти. Заметьте, что поля других типов, например, скаляры, статические массивы и короткие строки, не инициализируются.

Кроме того, с помощью процедуры New можно создать «старый» объект. Первый параметр – это переменная-указатель, необязательный второй параметр – имя конструктора и его необязательные

параметры. Delphi вызывает конструктор, выделяет память для объекта, инициализирует новый участок памяти нулями и устанавливает указатель на выделенную память. New не является настоящей процедурой.

Советы и приемы

- В отличие от стандартного Паскаля, Delphi Pascal создает записи с вариантами с помощью обычного вызова процедуры New. Значения вариантных меток указывать не нужно. New всегда выделяет максимальный объем памяти, необходимый для всех комбинаций вариантных меток.
- Чтобы выделить память для массива переменного размера, применяйте GetMem, а для создания одиночной записи или массива фиксированного размера — New.

Пример

```
type
  PEmployee = ^TEmployee;
  TEmployee = record
    Name: string;
    TIN: string[9];
    Salary: Currency;
  end;
var
  E: PEmployee;
begin
  New(E);
  try
    E.Salary := 0; // До установки фактического значения
                   // инициализируем нулем.
    AbuseByPointyHairBoss(E);
  finally
    Dispose(E);
  end;
end;
```

Смотрите также

Процедуры Dispose, GetMem, Initialize, переменная IsMultiThread, ключевое слово Object.

Nil, ключевое слово

Синтаксис

```
const nil = Pointer(0);
```

Описание

Ключевое слово nil — это специальное значение указателя, которое гарантированно отличается от значения любого реального указателя.

Числовое значение nil равно нулю. При создании нового объекта и последующей инициализации всех полей объекта нулями можно быть уверенным, что поля-указатели, поля-объекты, поля-интерфейсы и поля-методы инициализируются значением nil.

Советы и приемы

- Можно присваивать значения nil переменным и полям типа «метод», при этом Delphi сохраняет значение nil в качестве указателей на код и на данные. При сравнении метода с nil Delphi проверяет только указатель на код. Другими словами, следующие два предложения эквивалентны:

```
if Notify = nil then ...
if TMethod(Notify).Code = nil then ...
```

- Чаще всего nil применяется, чтобы пометить переменные типа «указатель» или «метод» значением-«неуказателем». Например, событийное свойство, которому не присвоен обработчик события, имеет значение nil.
- Присваивание nil переменной интерфейсного типа приводит к вызову метода _Release для предыдущего значения переменной (если до этого переменная не имела значения nil).
- В Delphi пустой динамический массив, длинная строка или «широкая» строка представляются как nil.
- Многие программисты предпочитают функцию Assigned сравнению с nil. В основном это вопрос стиля (смотрите дополнительную информацию в описании функции Assigned).

Смотрите также

Функция Assigned, тип Pointer.

Nodefault, директива

Синтаксис

Объявление свойства nodefault;

Описание

Директива nodefault сообщает Delphi, что свойство не имеет значения по умолчанию, и в *dfm*-файле всегда содержится значение этого свойства.

Советы и приемы

- Значения по умолчанию могут быть только у свойств порядковых типов, поэтому директива nodefault имеет смысл только для таких свойств. Строковые, вещественные, объектные, Variant-, Int64- и

интерфейсные свойства имеют фиксированное значение по умолчанию – «ноль» (пустая строка, nil, Unassigned и т.д.)

- В качестве признака nodefault в Delphi служит наименьшее целое число, т. е. nodefault практически эквивалентно default -2147483648. («Практически», потому что свойства перечисляемого, символьного и множественного типов не могут иметь в качестве значения по умолчанию целую константу, что является одной из причин, почему в Delphi используется в качестве признака nodefault наименьшее отрицательное целое.)
- Директива nodefault является необязательной. Если она опущена, Delphi все равно предполагает nodefault. Nodefault в объявлении свойств следует употреблять как напоминание тому, кто будет читать и сопровождать ваш код, что это свойство не имеет значения по умолчанию.
- Delphi всегда создает новые объекты с нулями во всех полях. Чтобы использовать ноль как значение по умолчанию для свойства, необходимо явно указать ноль в директиве default при описании свойства. В конструкторе в этом случае никаких специальных действий предпринимать не нужно.

Пример

```
property Color nodefault;
property Area: Integer read fArea write SetArea nodefault;
```

Смотрите также

Директивы Default, Stored, ключевое слово Property.

NoErrMsg, переменная

Синтаксис

```
var NoErrMsg: Boolean;
```

Описание

Когда графическое приложение завершается с ошибкой, оно выдает диалоговое окно с сообщением об ошибке и адресом. Если флаг NoErrMsg установлен в True, диалоговое окно не отображается. По умолчанию переменная имеет значение False. В консольном приложении сообщение об ошибке печатается всегда, независимо от значения NoErrMsg.

Смотрите также

Переменные ErrorAddr, ErrorProc, процедуры Halt, RunError.

Not, ключевое слово

Синтаксис

```
not Логическое выражение
not Целое выражение
```

Описание

Оператор `not` выполняет операцию отрицания. Если операнд логического типа, выполняется логическое отрицание (`not False = True` и `not True = False`). Если операнд – целое, оператор `not` выполняет побитовое отрицание каждого бита целого значения. Другими словами, выполняется операция дополнения.

Советы и приемы

Типы `ByteBool`, `LongBool` и `WordBool` интерпретируют любое ненулевое порядковое значение как `True`. Когда операция `not` возвращает `True`, порядковое значение результата всегда равно –1. Другими словами, порядковое значение `not not X` не всегда равно порядковому значению `X`. Если для вас важно порядковое значение логической переменной, следует использовать целый, а не логический тип.

Смотрите также

Ключевые слова `And`, `Or`, `Xor`, типы `Boolean`, `Integer`, константы `False`, `True`.

Null, переменная

Синтаксис

```
var Null: Variant;
```

Описание

Переменная `Null` – это нулевое значение типа `Variant`. Значение этой переменной изменять не следует.

Советы и приемы

- Переменные типа `Variant` инициализируются значением `Unassigned`. Если нужно присвоить значение вариантной переменной, но нет возможности назначить конкретную, известную величину, используйте `Null` как неизвестное или отсутствующее значение. В частности, компоненты, производные от `TField`, используют `Null` для представления значений NULL из SQL (если требуется, чтобы значение поля имело тип `Variant`).
- Вариантные выражения, в которых в качестве операнда присутствует `Null`, производят результат, равный `Null`.

- Попытка преобразовать значение `Null` в число приводит к ошибке времени выполнения 15 (`EVariantError`).
- В качестве эквивалента `NULL` из C и C++ используйте `nil`.

Пример

```
// Вычисляем среднее всех не-Null значений в вариантном
// массиве Data.
function ComputeAverage(Data: Variant): Variant;
var
  Sum: Double;
  Count: Integer;
  I: Integer;
begin
  Sum := 0.0;
  Count := 0;
  for I := VarArrayLowBound(Data) to VarArrayHighBound(Data) do
    if not VarIsNull(Data[I]) then
    begin
      Sum := Sum + Data[I];
      Inc(Count);
    end;
 if Count = 0 then
    Result := Null
  else
    Result := Sum / Count;
end;
```

Смотрите также

Переменные `EmptyParam`, `Unassigned`, тип `Variant`, функция `VarIsNull`.

Object, ключевое слово

Синтаксис

```
type Имя = Заголовок подпрограммы of object;

type Имя = object
  Объявления...
end;

type Имя = object(Базовый класс)
  Объявления...
end;
```

Описание

Ключевое слово `object` употребляется в двух различных, несвязанных между собой случаях: для объявления типа метода и объявления «старых» классов.

- Delphi Pascal позволяет объявить процедурный или функциональный тип как обычную подпрограмму или как метод. Процедурный тип является обычным указателем. Ему можно присваивать указатели на процедуры, сигнатуры которых соответствуют этому типу, и значением указателя является обычный указатель на код.

  В случае применения ключевых слов `of object` процедурный тип становится типом метода. Такой указатель состоит из двух частей: указатель на код и указатель на данные. Результатом извлечения адреса из переменной типа «метод», является указатель на код метода (аналогичный обычному процедурному типу) и указатель на данные (являющийся ссылкой на объект или ссылкой на класс для метода класса).

- Объявления `object` являются устаревшей формой и должны быть заменены на объявления `class`. «Старые» объекты существуют в основном для обратной совместимости с Turbo Pascal. В новых программах на Delphi вместо них должны быть классы.

Советы и приемы

- Свойства типа «метод» называются событиями.
- Тип `TMethod` – это запись, содержащая обобщенный указатель на метод. Для того чтобы вызвать метод, можно преобразовать `TMethod` к определенному виду, например, `TNotifyEvent`.

Примеры

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TSimpleComponent = class(TComponent)
  private
    fOnChange: TNotifyEvent;       // Здесь сохраняется ссылка
                                   // на метод.
  protected
    procedure Changed; virtual;
  published
    property OnChange: TNotifyEvent read fOnChange write fOnChange;
  end;
...
procedure TSimpleComponent.Changed;
begin
  if Assigned(fOnChange) then      // Проверка указателя на равенство nil.
    fOnChange(Self);               // Таким образом вызывается метод.
end;
```

Смотрите также

Ключевые слова `Class`, `Function`, `Packed`, `Procedure`, `Record`, `Type`, процедуры `Dispose`, `New`, тип `TMethod`, директивы компилятора `$A`, `$Align`.

Odd, функция

Синтаксис

```
function Odd(Value: Integer): Boolean;
function Odd(Value: Int64): Boolean;
```

Описание

Odd – стандартная функция языка Паскаль, возвращает True, если Value – нечетное число. Если Value – четное, возвращается False. Odd не является настоящей функцией.

Смотрите также

Типы Boolean, Integer.

Of, ключевое слово

Синтаксис

```
type Имя = array[Индексный тип] of Базовый тип;

type Имя = Заголовок подпрограммы of object;

case Выражение of
  Селектор: ...
end;
```

Описание

Ключевое слово of выполняет в Delphi много функций, однако оно всегда является вспомогательным для других ключевых слов. Подробности рассмотрены в описаниях ключевых слов array, object и case.

Смотрите также

Ключевые слова Array, Case, Function, Object, Procedure, Type.

OleStrToString, функция

Синтаксис

```
function OleStrToString(OleStr: PWideChar): string;
```

Описание

Функция OleStrToString преобразует строку в формате Unicode в многобайтовую строку. OleStrToString – настоящая функция.

Советы и приемы

В описании типа WideString обсуждается формат Unicode и многобайтовые наборы символов.

Смотрите также

Процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String, функции StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString.

OleStrToStrVar, процедура

Синтаксис

```
procedure OleStrToString(OleStr: PWideChar; var Dest: string);
```

Описание

Процедура OleStrToStrVar преобразует строку Unicode в многобайтовую строку. OleStrToStrVar – настоящая процедура.

Советы и приемы

Смотрите описание типа WideString, где обсуждается формат Unicode и многобайтовые наборы символов.

Смотрите также

Функции OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString, типы PWideChar, WideChar, WideString, ключевое слово String, процедуры WideCharLenToStrVar, WideCharToStrVar.

OleVariant, тип

Синтаксис

```
type OleVariant;
```

Описание

Тип OleVariant аналогичен типу Variant, за исключением того, что в OleVariant могут храниться только значения типов, совместимых с OLE. В частности, в OleVariant нельзя хранить строки Delphi, а допускаются только «широкие» строки.

Советы и приемы

Присваивая значение переменной типа OleVariant, Delphi автоматически преобразует его в OLE-совместимый формат, т. е. преобразует AnsiString (varString) в «широкую» строку (varOleStr).

Смотрите также

Типы AnsiString, PWideChar, Variant, WideString, ключевое слово String, процедуры VarCast, VarCopy.

On, директива

Синтаксис

```
try
  Операторы...
except
  on Переменная: Имя класса do Предложение;
  on Имя класса do Предложение;
  else Предложения...
end;
```

Описание

Директива on обозначает начало блока обработки исключительной ситуации в конструкции try-except. Обработчиков может быть любое количество, каждый из них обозначается классом исключительной ситуации, возможно, с указанием имени переменной.

Delphi проверяет объект исключительной ситуации на соответствие каждому из классов в порядке их перечисления. Поиск останавливается на первом обработчике, в котором класс объекта исключительной ситуации равен классу обработчика или является производным от класса обработчика (оператор is возвращает True). Затем Delphi вызывает связанное с этим обработчиком предложение (или блок), и если это предложение не генерирует новой исключительной ситуации, выполнение программы продолжается с предложения, следующего за ключевым словом end предложения try-except.

Если в обработчике исключительной ситуации указано имя переменной, Delphi присваивает объект исключительной ситуации этой переменной. Лексический диапазон этой переменной совпадает с конкретным обработчиком, поэтому на нее нельзя ссылаться из других обработчиков или извне конструкции try-except.

Если объект исключительной ситуации не соответствует ни одному из классов, и присутствует ключевое слово else, Delphi выполняет предложения, следующие за else. Если ни один из классов не подходит, и в конструкции try-except отсутствует ключевое слово else, снова вызывается та же исключительная ситуация, и другое предложение try-except получает возможность ее обработать.

По завершении обработчика исключительной ситуации Delphi освобождает объект исключительной ситуации, если только внутри обработчика с помощью оператора raise не была вызвана та же исключительная ситуация.

Советы и приемы

- Обработчик исключительной ситуации может вызвать другую исключительную ситуацию. В этом случае выполнение конструк-

ции try-except сразу прекращается, и Delphi ищет следующий обработчик событий выше по стеку вызовов.

- Поскольку Delphi ищет обработчик исключительной ситуации в соответствии с порядком их перечисления, следует всегда указывать более специфические классы исключений первыми.
- Любой объект может быть объектом исключительной ситуации. По соглашению Delphi использует классы, производные от SysUtils.Exception. Большинство классов исключительных ситуаций объявлены в модуле SysUtils. В приложении В «Модуль SysUtils» перечислены стандартные классы исключительных ситуаций.

Пример

Пример смотрите в описании ключевого слова except.

Смотрите также

Ключевые слова Else, Except, Raise, Try, переменные ExceptClsProc, ExceptObjProc.

OpenString, тип

Синтаксис

```
Объявление подпрограммы(...; var Param: OpenString; ...);
```

Описание

Объявление OpenString позволяет передать аргумент ShortString подпрограмме, которая не знает точного размера строки. Обычно тип строкового параметра var должен точно соответствовать типу фактического параметра. Для коротких строк максимальный размер – это часть определения типа, и он должен быть известен, чтобы подпрограмма не пыталась записать в строку больше символов, чем она может вместить.

Параметр OpenString ослабляет это ограничение и позволяет передать в подпрограмму строку любого размера. Delphi передает дополнительный, скрытый параметр, содержащий максимальную длину строки. Поэтому ссылки на Param в подпрограмме становятся корректными по отношению к типу фактического параметра.

Советы и приемы

- Директива компилятора $OpenString заставляет параметры var типа ShortString вести себя как параметры OpenString. Эта директива по умолчанию включена. Разработчику придется объявлять параметры как OpenString только в том случае, если директива $OpenStrings выключена.
- Если директива компилятора $LongStrings выключена, тип string становится короткой строкой, а строковые параметры var – Open-

String (если только не отключена директива компилятора $OpenStrings).

- Если используются длинные строки, необходимость в параметрах OpenString отпадает.

Смотрите также

Типы AnsiString, ShortString, ключевые слова String, Var, директивы компилятора $H, $LongStrings, $OpenStrings, $P.

Or, ключевое слово

Синтаксис

Логическое выражение or *Логическое выражение*
Целое выражение or *Целое выражение*

Описание

Оператор or выполняет логическое «или», если операнды относятся к логическому типу, и битовое «или», если операнды целые. Целые операнды могут относиться к любому целому типу, в том числе Int64. Логическое «или» ложно, если оба операнда ложны, и истинно, если хотя бы один операнд истинен.

Советы и приемы

- В отличие от стандартного Паскаля, если левый операнд равен True, Delphi не вычисляет правый операнд, т. к. результат в любом случае – True. Отключить эту сокращенную схему и вернуться к стандарту Паскаля можно, установив директиву компилятора $BoolEval или $B.
- Целое «или» оперирует с каждым битом операнда, устанавливая бит результата в 1, если хотя бы один из операндов содержит бит 1, и в 0, если оба операнда содержат нулевые биты. Если одни из операндов меньше другого, Delphi дополняет меньший операнд слева нулевыми битами. Результат имеет размер наибольшего операнда.

Примеры

```
var
  I, J: Integer;
  S: string;
begin
  I := $25;
  J := $11;
  WriteLn(I or J); // Печатает 53 (что равно $35)
...
  // Сокращенная схема вычисления OR в следующем примере предотвращает
  // обращение к несуществующему элементу в конце строки,
  // если строка является пустой.
  if (Length(S) = 0) or (S[Length(S)] <> '\') then
    S := S + '\';
```

Смотрите также

Ключевые слова And, Not, Shl, Shr, Xor, типы Boolean, ByteBool, LongBool, WordBool, директивы компилятора $B, $BoolEval.

Ord, функция

Синтаксис

```
function Ord(A: AnsiChar): Integer;
function Ord(C: Char): Integer;
function Ord(W: WideChar): Integer;
function Ord(E: Enumerated type): Integer;
function Ord(I: Integer): Integer;
function Ord(I: Int64): Int64;
```

Описание

Функция Ord возвращает порядковое значение символа или элемента перечисления в виде неотрицательного целого. Вызов Ord с целым аргументом – холостая операция, возвращающая этот аргумент. Ord не является настоящей функцией.

Советы и приемы

Значение порядковой переменной можно получить преобразованием ее к целому типу. Вызов функции Ord предпочтительнее, т. к. ясно говорит, что делает код.

Пример

```
// В модуле TypInfo присутствует функция GetEnumName, которая
// возвращает имя перечисляемого значения по указателю TypeInfo
// и порядковому значению. С помощью GetEnumName можно писать
// собственные функции, такие как эта, преобразующая
// значение типа Boolean в строку.
function BoolToStr(B: Boolean): string;
begin
  Result := GetEnumName(TypeInfo(Boolean), Ord(B));
end;
```

Смотрите также

Типы AnsiChar, Char, WideChar, функция Chr, ключевое слово Type.

Out, директива

Синтаксис

```
Заголовок подпрограммы(Параметры...; out Имя: Тип; ...);
```

Описание

Директива out применяется для объявления параметров подпрограммы. Она аналогична ключевому слову var при объявлении параметра,

значение которого может быть изменено внутри подпрограммы. Разница между var и out заключается в том, что параметр out не передает в подпрограмму какое-либо осмысленное значение – он служит только как выходной параметр при возврате из подпрограммы.

Советы и приемы

- Out-параметры часто используются в COM-интерфейсах.
- Out-параметры полезны для структур данных, поддерживающих счетчики ссылок, например, строк, динамических массивов и интерфейсов. Out-параметры дают некоторое повышение производительности по сравнению с параметрами var, т. к. компилятору не требуется увеличивать счетчик ссылок при передаче параметра подпрограмме. Подпрограмма должна изменить значение переменной, и в этот момент счетчик ссылок, как обычно, увеличится.
- Даже если параметр не поддерживает счетчик ссылок, можно с помощью out-параметра дать понять читателю кода, что подпрограмма не использует входное значение данного параметра.

Пример

```
type
  ICollection = interface
    ...
    // Любая коллекция может иметь произвольное количество
    // методов-перечислителей, которые обходят все ее элементы
    // коллекции. Функция Enum создает новый перечислитель
    // и сохраняет его в параметре Enumerator.
    function Enum(out Enumerator: IEnumerator): HResult;
    ...
```

Смотрите также

Ключевые слова Const, Interface, Var.

Output, переменная

Синтаксис

```
var Output: TextFile;
```

Описание

Переменная Output – это текстовый файл, который Delphi автоматически открывает для записи, но только в консольном приложении. Файл output – это обычно консоль, однако пользователь может перенаправить его из командной строки.

Output – это обычный текстовый файл, и можно закрыть его и открыть другой. Открытие файла без имени аналогично открытию консольного файла. Например, при запуске консольного приложения Delphi всегда выполняет следующие действия:

```
AssignFile(Output, '');
Rewrite(Output);
```

Вызов процедур Write и WriteLn без указания файла в качестве первого аргумента аналогичен вызову Write или WriteLn с первым аргументом output.

Смотрите также

Процедуры AssignFile, Write, Writeln, переменная Input, тип TextFile.

Overload, директива

Синтаксис

Объявление подпрограммы; overload; *другие директивы*...

Описание

Директива overload сообщает Delphi, что объявляется другая подпрограмма с тем же именем, но с другими параметрами.

Советы и приемы

- Если в объявлении метода присутствует директива overload, она должна быть указана до директив virtual, dynamic или abstract.
- Альтернативой перегруженным методам являются параметры по умолчанию. Для простых типов применение параметров по умолчанию обычно приводит к уменьшению исходного кода. Для параметров составных типов, например, объектов, создание перегруженных подпрограмм, вызывающих друг друга, может оказаться проще.
- Для того чтобы определить, какую из перегруженных подпрограмм вызвать, используются количество и типы фактических параметров. Из возможных целых типов выбирается наиболее узкий тип. Если компилятор не может принять решение о выборе конкретной подпрограммы, он выдает ошибку.

Пример

```
// Если параметр Time равен нулю, сохраняем текущую дату и время
// в виде временной метки.
// Записываем сообщение в протокол.
procedure Log(const Message: string; Time: TDateTime = 0); overload;
// Записываем в протокол форматированное сообщение.
procedure Log(const Fmt: string; const Args: array of const;
    Time: TDateTime = 0); overload;

// Компилятор выбирает конкретный вариант функции Min на основе
// аргументов.
function Min(I, J: Integer): Integer; overload;
function Min(I, J: Int64): Int64; overload;
function Min(I, J: Extended): Extended; overload;
```

```
X := Min(10, 100);          // Вызов Min(I, J: Integer)
Y := Min(10, 10000000000); // Вызов Min(I, J: Int64);
Z := Min(10, 100.0);        // Вызов Min(I, J: Extended);
```

Смотрите также

Ключевые слова `Function`, `Procedure`.

Override, директива

Синтаксис

Объявление подпрограммы; override;

Описание

Директива `override` предназначена для объявления метода, который переопределяет виртуальный или динамический метод, объявленный в базовом классе. Виртуальные методы и полиморфизм обсуждаются в главе 2.

Пример

```
type
  // Формирование мозаики.
  TTile = class(TGraphicControl)
  private
    fBitmap: TBitmap;
    procedure SetBitmap(NewBitmap: TBitmap);
    procedure BitmapChanged(Sender: TObject);
  protected
    procedure Paint; override;
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
  published
    property Align;
    property Bitmap: TBitmap read fBitmap write SetBitmap;
    property OnClick;
    // Есть и другие полезные свойства, но для краткости мы их опускаем.
    // Полный список приведен в определении типа Tcontrol.
  end;
```

Смотрите также

Ключевые слова `Class`, `Function`, `Inherited`, `Procedure`, **директивы** `Dynamic`, `Reintroduce`, `Virtual`.

Package, директива

Синтаксис

package *Имя*;

```
requires
  Имена...;
contains
  Имена...;
end.
```

Описание

Директива package обозначает начало исходного файла пакета. Пакет – это DLL специального вида, содержащая модули, которые могут быть динамически связаны с другим проектом.

Советы и приемы

- Исходный файл проекта имеет расширение *dpk* (сокращение от Delphi package). В одном файле может содержаться только одна директива package.
- Можно редактировать исходный текст пакета вручную, но обычно для этого применяется менеджер пакетов из Delphi IDE.
- Если исходный файл пакета редактируется вручную, убедитесь в правильности синтаксиса файла. Компилятор Delphi не всегда корректно реагирует на ошибки синтаксиса в исходном файле пакета.

Смотрите также

Директивы Contains, Requires.

PackageInfo, тип

Синтаксис

```
type PackageInfo = ^PackageInfoTable;
```

Описание

Значения типа PackageInfo являются указателями на запись PackageInfoTable.

Смотрите также

Типы GetPackageInfoTable, PackageInfoTable, PUnitEntryTable, UnitEntryTable.

PackageInfoTable, тип

Синтаксис

```
type
  PackageInfoTable = packed record
    UnitCount: Integer;
    UnitInfo : PUnitEntryTable;
  end;
```

Описание

Каждый пакет содержит запись PackageInfoTable, ссылающуюся на таблицу модулей (UnitEntryTable), каждая из которых содержит указатель на разделы инициализации и завершения модуля. Каждый модуль, содержащийся с пакете, имеет запись в таблице модулей. Когда Delphi загружает или выгружает пакет, таблица PackageInfoTable используется для инициализации и завершения модулей пакета.

Delphi работает с пакетами автоматически. Программистам редко приходится напрямую обращаться к внутренним данным пакета.

Смотрите также

Ключевые слова Finalization, Initialization, Unit, типы GetPackageInfoTable, PackageInfoTable, PUnitEntryTable, UnitEntryTable, директива Package.

PackageUnitEntry, тип

Синтаксис

```
type
  PackageUnitEntry = packed record
    Init, FInit: procedure;
  end;
```

Описание

Записи PackageUnitEntry содержат указатель на код разделов инициализации и завершения модуля. Delphi автоматически создает такую запись для каждого модуля, и все приложения, библиотеки и пакеты содержат таблицу из записей обо всех модулях.

Delphi работает с модулями автоматически. Прямые обращения к внутренним данным о модулях требуются редко.

Смотрите также

Ключевые слова Finalization, Initialization, Unit, типы GetPackageInfoTable, PackageInfoTable, PUnitEntryTable, UnitEntryTable.

Packed, ключевое слово

Синтаксис

```
type Имя = packed record ... end;
type Имя = packed class ... end;
type Имя = packed object ... end;
type Имя = packed array[...] of ...;
```

Описание

Delphi выравнивает поля записей и объектов по естественным границам для улучшения производительности. Если указывается директи-

ва packed, Delphi не вставляет выравнивающие байты после полей объекта или записи.

После выравнивания запись также дополняется байтами до размера, кратного 4 байтам. Упакованная запись не имеет дополнительного выравнивания, но переменные и динамическая память выравниваются по четырехбайтовым границам, так что экономия памяти на упаковке записи имеет смысл, только если создается массив записей или записи используются в качестве полей других структурированных типов.

Выравнивание неупакованных полей зависит от размера поля:

- Поля размером в байт и множества любого размера выравниваются по границе байта.
- Поля размером в слово (Word) выравниваются по границам слов (2 байта).
- Поля размером LongWord (в том числе Single) выравниваются по границам длинных слов (4 байта).
- Другие типы (Comp, Currency, Double, Extended, Int64, Real48, Variant) выравниваются по восьмибайтовым границам.
- Массивы выравниваются в соответствии с выравниванием базового типа массива.
- Записи выравниваются с соответствии с максимальным выравниванием, используемым для полей записи.

Советы и приемы

- Доступ к упакованному полю осуществляется медленнее, чем к выровненному. Используйте ключевое слово packed только в том случае, если необходимость в экономии памяти для вас важнее падения производительности.
- Delphi не упаковывает поля типа Boolean в одиночные биты, как делают некоторые компиляторы Паскаля.
- Массивы всегда упакованы. Можно указать ключевое слово packed в объявлении массива для совместимости со стандартом Паскаля, однако это не дает никакого эффекта.

Пример

```
type
  Big = record    // SizeOf(Big) = 16
    B1: Byte;
    W: Word;      // Выравниваем W по границе слова вставкой
                  // заполняющего байта
    B2: Byte;
    L: LongWord;  // Выравниваем L по 4-байтовой границе вставкой
                  // 3 байтов.
    B3: Byte;     // Размер записи выровнен по границе длинного слова.
  end;
```

```
Small = packed record // SizeOf(Small) = 9
  B1: Byte;
  W: Word;      // Выравниваем W по границам байтов: дополнительные
                // байты не требуются.
  B2: Byte;
  L: LongWord;
  B3: Byte;     // Запись в целом не дополняется.
end;
```

Смотрите также

Ключевые слова Array, Class, Object, Record.

PAnsiChar, тип

Синтаксис

```
type PAnsiChar = ^AnsiChar;
```

Описание

Тип PAnsiChar – это указатель на AnsiChar, и в расширенном синтаксисе Delphi он может также рассматриваться как AnsiString и как указатель на array of AnsiChar.

Советы и приемы

- PAnsiChar чаще всего используется как тип параметра для DLL, написанных на C или C++, например для Windows API.
- Указатель PAnsiChar можно рассматривать как указатель на массив AnsiChar. Индекс массива – поддиапазон целого, начинающийся с нуля. Delphi не выполняет проверок на границы массива. Используется соглашение, по которому конец строки обозначается символом #0.
- Арифметика указателей для типа PAnsiChar при добавлении и вычитании целых значений работает аналогично процедурам Inc и Dec.
- Расширенный синтаксис Delphi по умолчанию включен. Эта возможность выключается с помощью директивы компилятора $X и $ExtendedSyntax. В этом случае поведение компилятора становится более похожим на стандартный Паскаль.

Смотрите также

Типы AnsiChar, AnsiString, Char, PChar, PWideChar, ключевые слова Array, String, директивы компилятора $ExtendedSyntax, $X.

PAnsiString, тип

Синтаксис

```
type PAnsiString = ^AnsiString;
```

Описание

Тип PAnsiString определен для удобства работы с указателями на AnsiString. Заметьте, что AnsiString сама является указателем, поэтому PAnsiString используется редко.

Смотрите также

Типы AnsiString, String, PString, PWideString.

ParamCount, функция

Синтаксис

```
function ParamCount: Integer;
```

Описание

Функция ParamCount возвращает количество параметров командной строки, полученной приложением. Для библиотеки значение функции равно нулю. ParamCount – настоящая функция.

Пример

Пример рассмотрен в описании функции ParamStr.

Смотрите также

Переменная CmdLine, функция ParamStr.

ParamStr, функция

Синтаксис

```
function ParamStr(Number: Integer): string;
```

Описание

Функция ParamStr возвращает параметр командной строки под номером Number. ParamStr – настоящая функция.

Советы и приемы

- Параметр номер ноль – это путь к приложению.
- Параметры нумеруются от 1 до ParamCount. Если в Number передается неверное значение, ParamStr возвращает пустую строку.
- При разбиении командной строки на параметры Delphi использует в качестве разделителей пробелы. Заключайте текст, содержащий пробелы, в двойные кавычки, чтобы формировались параметры с пробелами (например, длинные имена файлов).
- Для просмотра переключателей командной строки применяется функция FindCmdLineSwitch из модуля SysUtils.

Пример

```
program Echo;
// Печатает аргументы командной строки, разделенные пробелами.
{$AppType Console}
var
  I: Integer;
begin
  if ParamCount > 0 then
    Write(ParamStr(1));
  for I := 2 to ParamCount do
    Write(' ', ParamStr(I));
  WriteLn;
end.
```

Смотрите также

Процедура `CmdLine`, функция `ParamCount`.

Pascal, директива

Синтаксис

Объявление подпрограммы; pascal;

Описание

Директива `pascal` сообщает компилятору о необходимости использовать передачу параметров функции или процедуре в стиле языка Паскаль. При вызове подпрограммы аргументы помещаются в стек начиная с крайнего левого аргумента. При возврате из подпрограммы аргументы выталкиваются из стека.

Функции, возвращающие порядковые значения, указатели, небольшие записи и множества, передают их через регистр `EAX`, вещественные – через стек FPU. Строки, динамические массивы, значения типа `Variant`, большие записи и множества передаются как скрытые параметры `var`. Если подпрограмма является методом, переменная `Self` передается в качестве первого аргумента, скрытый `var`-параметр – вторым, и эти параметры помещаются в стек в первую очередь.

Советы и приемы

Пусть имя этой директивы не вводит вас в заблуждение. По умолчанию в Delphi параметры передаются через регистры, а не в стиле Паскаль. Соглашение `pascal` оставлено для обратной совместимости, и его применение имеет смысл только для обеспечения интерфейса с архаичными DLL.

Смотрите также

Директивы `CDecl`, `Register`, `SafeCall`, `StdCall`, ключевые слова `Function`, `Procedure`.

PChar, тип

Синтаксис

```
type PChar = ^Char;
```

Описание

Тип PChar – это указатель на Char и в расширенном синтаксисе Delphi он также рассматривается как строка или указатель на array of Char.

Советы и приемы

- PChar чаще всего используется как тип параметра для DLL, написанных на C или C++, например для Windows API.
- Можно рассматривать указатель PAnsiChar как указатель на массив AnsiChar. Индекс массива – поддиапазон целого, начинающийся с нуля. Delphi не выполняет проверку на границы массива. Используется соглашение, по которому конец строки обозначается символом #0.
- Арифметика указателей для типа PChar при добавлении и вычитании целых значений работает аналогично процедурам Inc и Dec.
- Расширенный синтаксис Delphi по умолчанию включен. Его можно выключить с помощью директивы компилятора $X и $ExtendedSyntax. В этом случае поведение компилятора становится более похожим на стандартный Паскаль.

Смотрите также

Типы AnsiChar, AnsiString, Char, PAnsiChar, PWideChar, ключевые слова Array, String, директивы компилятора $ExtendedSyntax, $X.

PCurrency, тип

Синтаксис

```
type PCurrency = ^Currency;
```

Описание

Тип PCurrency определен для удобства работы с указателями на тип Currency.

Смотрите также

Тип Currency.

PDateTime, тип

Синтаксис

```
type PDateTime = ^TDateTime;
```

Описание

Тип PDateTime определен для удобства работы с указателями на тип TDateTime.

Смотрите также

Тип DateTime

PExtended, тип

Синтаксис

```
type PExtended = ^Extended;
```

Описание

Тип PExtended определен для удобства применения указателей на тип Extended.

Смотрите также

Тип Extended.

PGUID, тип

Синтаксис

```
type PGUID = ^TGUID;
```

Описание

Тип PGUID определен для удобства работы с указателями на тип TGUID.

Смотрите также

Тип TGUID.

Pi, функция

Синтаксис

```
function Pi: Extended
```

Описание

Функция Pi возвращает приближенное значение числа π. Компилятор преобразует вызов этой функции в inline-код, поэтому можно рассматривать функцию Pi как константу и формировать с ее помощью константные выражения. Компилятор распознает контекст вызова функции и выбирает подходящий тип и точность вычисления (т. е. Single, Double или Extended). В результате вы получаете лучшее приближение к π, чем при указании десятичной константы.

Смотрите также

Типы Double, Extended, Single.

PInt64, тип

Синтаксис

```
type PInt64 = ^Int64;
```

Описание

Тип PInt64 определен для удобства работы с указателями на тип Int64.

Смотрите также

Тип Int64.

PInterfaceEntry, тип

Синтаксис

```
type PInterfaceEntry = ^TInterfaceEntry;
```

Описание

Тип PInterfaceEntry определен для удобства работы с указателями на тип TInterfaceEntry.

Смотрите также

Ключевые слова Class, Interface, типы PInterfaceTable, TInterfaceEntry, TInterfaceTable, TObject, функция TypeInfo.

PInterfaceTable, тип

Синтаксис

```
type PInterfaceTable = ^TInterfaceTable;
```

Описание

Тип PInterfaceTable определен для удобства работы с указателями на тип TInterfaceTable.

Смотрите также

Ключевые слова Class, Interface, типы TInterfaceTable, TInterfaceEntry, TObject, функция TypeInfo.

PLibModule, тип

Синтаксис

```
type PLibModule = ^TLibModule;
```

Описание

Тип PLibModule определен для удобства работы с указателями на тип TLibModule.

Смотрите также

Процедуры EnumModules, EnumResourceModules, RegisterModule, UnregisterModule, переменная LibModuleList, тип TLibModule.

PMemoryManager, тип

Синтаксис

```
type PMemoryManager = ^TMemoryManager;
```

Описание

Тип PMemoryManager определен для удобства работы с указателями на тип TMemoryManager.

Смотрите также

Процедуры GetMemoryManager, SetMemoryManager, функция IsMemoryManagerSet, тип TMemoryManager.

PModuleUnloadRec, тип

Синтаксис

```
type PModuleUnloadRec = ^TModuleUnloadRec;
```

Описание

Тип PModuleUnloadRec определен для удобства работы с указателями на тип TModuleUnloadRec.

Смотрите также

Процедуры AddModuleUnloadProc, RemoveModuleUnloadProc, переменная ModuleUnloadList, тип TModuleUnloadProcLW.

Pointer, тип

Синтаксис

```
type Pointer;
```

Описание

Тип Pointer — это обобщенный тип указателя. Не прибегая к преобразованию типов, можно присвоить любое выражение-указатель переменной типа Pointer, а также присвоить выражение типа Pointer переменной конкретного типа указателя.

Советы и приемы

- Поскольку ссылки на объекты фактически являются указателями, можно сколько угодно раз свободно преобразовывать их в Pointer и обратно.

- Применение типизированных указателей везде, где это возможно, является хорошим стилем программирования. Типы помогают избежать ошибок и документируют код, что облегчает чтение и сопровождение программы. Неверное преобразование указателя – обычный источник ошибок.

Пример

```
// Тип TList содержит список указателей. Delphi автоматически
// преобразует типизированные указатели и ссылки на объекты к
// типу Pointer. Извлечение элементов списка также выполняется
// просто.
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TList;
  I: Integer;
  P: ^Integer;
  F: TForm;
begin
  List := TList.Create;
  List.Add(@I);     // Получает аргумент типа Pointer
  List.Add(Self);

  P := List[0];     // List[0] возвращает Pointer
  P^ := 10;
  F := List[1];
  F.Show;

  List.Free;
end;
```

Смотрите также

Функции `Addr`, `Ptr`, ключевое слово `Nil`, директивы компилятора `$T`, `$TypedAddress`, оператор `@`.

POleVariant, тип

Синтаксис

```
type POleVariant = ^TOleVariant;
```

Описание

Тип `POleVariant` определен для удобства работы с указателями на тип `TOleVariant`.

Смотрите также

Типы `OleVariant`, `PVariant`, `Variant`.

Pos, функция

Синтаксис

```
function Pos(const SubStr, Str: string): Integer;
```

Описание

Функция Pos возвращает индекс первого вхождения фрагмента SubStr в строке Str или ноль, если SubStr нигде в Str не встречается. Pos не является настоящей функцией.

Советы и приемы

- Индексы символов в строке начинаются с 1.
- Поиск выполняется с учетом регистра.
- Pos не обрабатывает многобайтовые символы. Вместо Pos используйте AnsiPos из модуля SysUtils.

Пример

```
// В переменных окружения, например PATH, содержатся списки
// каталогов, разделенных точками с запятой. Функция SplitPath
// принимает такую строку в качестве аргумента и разделяет
// строку на отдельные имена файлов. Имена файлов записываются
// в аргумент FileList.
// Заметьте, что имена файлов могут содержать многобайтовые
// символы, поэтому SplitPath должна вызывать функцию AnsiPos
// вместо Pos. Тем не менее, она является примером использования
// обеих функций.
procedure SplitPath(const Path: string; FileList: TStrings);
var
  Semicolon: Integer;
  FileName: string;     // Первое имя файла в оставшейся части строки
  Remaining: string;    // Остаток строки после извлечения первого
                        // имени файла
begin
  Remaining := Path;
  FileList.BeginUpdate;
  try
    FileList.Clear;
    while Remaining <> '' do
    begin
      Semicolon := Pos(';', Remaining);
      if Semicolon = 0 then
        Semicolon := MaxInt;
      FileName := Copy(Remaining, 1, Semicolon-1);
      Delete(Remaining, 1, Semicolon);
      FileList.Add(FileName);
    end;
  finally
    FileList.EndUpdate;
```

```
    end;
  end;
```

Смотрите также

Типы AnsiString, ShortString, String, функция Copy, процедуры Delete, Insert.

Pred, функция

Синтаксис

```
function Pred(const Value): Порядковый тип;
```

Описание

Функция Pred возвращает предыдущее порядковое значение, обычно для перечисляемой величины. То есть возвращается перечисляемое значение, порядковая величина которого на единицу меньше Value. Pred не является настоящей функцией.

Советы и приемы

- Процедура Dec, функция Pred и операция вычитания имеют почти одинаковую производительность, поэтому выбирайте наиболее понятный и ясный для себя вариант.
- Вызов Pred(Low(SomeType)) генерирует ошибку времени выполнения 4 (ERangeError). Однако без проверки на переполнение Pred возвращает значение с соответствующей порядковой величиной, хотя оно и не является корректным для данного типа.

Пример

```
type TDay =
 (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
var
  Day: TDay;
begin
  ...
  Day := Pred(Day);
  ...
```

Смотрите также

Процедуры Dec, Inc, функции High, Low, Succ, директивы компилятора $OverflowChecks, $Q.

PResStringRec, тип

Синтаксис

```
type PResStringRec = ^ResStringRec;
```

Описание

Тип PResStringRec определен для удобства работы с указателями на тип TResStringRec. Адрес идентификатора resourcestring является указателем PResStringRec.

Смотрите также

Функция LoadResString, ключевые слова ResourceString, String, тип TResStringRec.

Private, директива

Синтаксис

```
type Объявление класса
private
  Описания полей...
  Описания свойств и методов ...
end;
```

Описание

Директива private обозначает начало private-секции в объявлении класса. Закрытые (private) поля, методы и свойства недоступны другим классам и подпрограммам, кроме описанных в том же модуле.

Советы и приемы

- Объявляйте все поля как закрытые (private), а свойства для доступа к полям – с более высоким уровнем доступа: защищенные (protected), открытые (public) или публикуемые (published). Это обеспечивает гибкость в изменении метода доступа к полю, позволяя, например, заменить прямую ссылку на поле вызовом метода. Такие изменения будут отражаться только на реализации класса, а не на его применении.
- Директива private предотвращает прямой вызов метода, но класс может иметь и другие способы доступа к нему. Закрытый (private) метод может быть обработчиком сообщения, поддерживать открытое (public) свойство или реализовывать метод интерфейса.

Пример

Пример рассмотрен в описании ключевого слова class.

Смотрите также

Ключевое слово Class, директивы Protected, Public, Published.

Procedure, ключевое слово

Синтаксис

```
procedure Имя;
```

```
procedure Имя (Параметры...);

type Имя = procedure(Параметры...);
type Имя = procedure(Параметры...) of object;
```

Описание

Ключевое слово `procedure` служит для объявления подпрограммы, которая не возвращает значение. Кроме того, программист может объявить процедурный тип.

В интерфейсном разделе модуля может присутствовать только заголовок процедуры (ее имя, параметры и директивы, но не код процедуры).

В разделе реализации необходимо предоставить полное определение всех процедур, объявленных в разделе интерфейса. (Способ определения процедуры без указания ее кода рассмотрен в описании директивы `external`). В разделе реализации можно опустить параметры процедуры в том случае, если процедура не перегружена. Параметры следует указывать точно так же (в том числе и имена параметров), как и в разделе интерфейса.

В реализации процедуры не должны присутствовать директивы, указанные при первом описании ее заголовка.

Советы и приемы

- Хотя указание копии заголовка процедуры в разделе реализации может показаться излишним, это дает большое преимущество тому, кто сопровождает код. Неудобно постоянно обращаться к заголовку процедуры только для того, чтобы узнать о ее параметрах.
- Можно объявить несколько процедур с одним именем, но с разными параметрами с помощью директивы `overload`.
- Процедура может быть объявлена в описании класса, в этом случае она называется методом. Метод может быть объявлен с директивами `dynamic` или `virtual` для создания виртуального метода. Виртуальный метод может быть объявлен с директивой `abstract`, и в этом случае предоставлять реализацию процедуры не требуется.
- По умолчанию используется передача параметров через регистры (`register`). Программист может выбрать другое соглашение о вызове с помощью директив `cdecl`, `pascal`, `safecall` или `stdcall`.
- Все директивы являются необязательными, но если они указываются, то должны быть перечислены для методов в следующем порядке:
  - `reintroduce`
  - `overload` (не может быть в сочетании с `message`)
  - `virtual`, `dynamic`, `override`, `message`

— cdecl, pascal, register, safecall, stdcall

— abstract (только если ранее указаны директивы virtual, dynamic или override)

Пример

```
type
  TRandProc = procedure(var Rand: Integer; Max: Integer);
  TRandMethod = procedure(var Rand: Integer) of object;
  TRandClass = class
  private
    fMin, fMax: Integer;
  public
    constructor Create(Min, Max: Integer);
    procedure GetRand(var Rand: Integer);
    property Min: Integer read fMin;
    property Max: Integer read fMax;
  end;

procedure TestRand(var Rand: Integer; Max: Integer);
begin
  Rand := Random(Max);
end;

procedure TRandClass.GetRand(var Rand: Integer);
begin
  Rand := Random(Max - Min + 1) + Min;
end;

var
  P: TRandProc;
  M: TRandMethod;
  O: TRandClass;
  I: Integer;
begin
  O := TRandClass.Create(1, 6);
  try
    P := RandProc;
    P(I, 100);
    WriteLn(I);
    M := O.GetRand;
    M(I);
    WriteLn(I);
  finally
    O.Free;
  end;
end.
```

Смотрите также

Директивы Abstract, CDecl, Dynamic, External, Message, Overload, Pascal, Register, Reintroduce, SafeCall, StdCall, Virtual, ключевые слова Class, Function, Object, Type.

Program, ключевое слово

Синтаксис

```
program Имя;
Объявления...
Блок.
```

Описание

Ключевое слово program обозначает начало приложения. Для исходного файла программы используется расширение *.dpr*.

Советы и приемы

- Delphi допускает наличие параметров программы для совместимости со стандартным Паскалем, но игнорирует их.
- При запуске программы сначала запускаются разделы инициализации всех модулей, затем – блок программного файла. После завершения блока вызывается процедура ExitProc и затем разделы завершения всех модулей.
- Разделы инициализации запускаются в порядке перечисления модулей в предложении uses программы, но с учетом объявлений uses каждого модуля. Разделы завершения выполняются в порядке, обратном порядку запуска разделов инициализации. Другими словами, первый модуль в предложении uses программы инициализируется в первую очередь. Затем инициализируются перечисленные в нем модули, при этом сначала инициализируются модули из их предложений uses, а затем запускаются разделы инициализации.
- *Блок* может быть просто предложением end, если в программе не запускается какой-либо код. В этом случае сразу после запуска всех разделов инициализации программа завершается, вызвав ExitProc и разделы завершения.

Пример

```
Program Echo;
// Печатает аргументы командной строки, разделенные пробелами.
{$AppType Console}
var
  I: Integer;
begin
  if ParamCount > 0 then
    Write(ParamStr(1));
  for I := 2 to ParamCount do
```

```
    Write(' ', ParamStr(I));
  WriteLn;
end.
```

Смотрите также

Ключевые слова `Asm`, `Begin`, `Initialization`, `Library`, `Uses`, переменные `ExitProc`, `InitProc`, директива компилятора `$AppType`.

Property, ключевое слово

Синтаксис

```
property Имя: Тип read Getter write Setter;
property Имя: Тип read Getter write Setter
   index Константа default Константа stored Stored;
property Имя [Index: Тип индекса]: Базовый тип read Getter write Setter;
    default;
property Имя;

property Имя: Тип;              // только в объявлениях dispinterface
property Имя: Тип readonly;
property Имя: Тип writeonly;

property Имя: Тип read Getter implements Интерфейсы...;
```

Описание

Ключевое слово `property` обозначает начало описания свойства класса, интерфейса или диспетчерского интерфейса. Свойство вызывается со ссылкой на объект или интерфейс, аналогично обращению к полю, но для реализации доступа к свойству можно использовать метод. Это позволяет без потери производительности сочетать лучшие черты обоих средств языка: удобный синтаксис и гибкую семантику. В объявлении диспетчерского интерфейса свойство может быть объявлено как `readonly` (только для чтения) или как `writeonly` (только для записи).

Свойство также может предоставлять реализацию одного или нескольких интерфейсов. Дополнительную информации о данном применении свойств можно найти в описании ключевого слова `implements`.

Советы и приемы

- Delphi позволяет использовать директивы `default` и `stored` для любого свойства, однако они имеют смысл только для публикуемых свойств. Компилятор генерирует для публикуемых свойств RTTI, которая включает значения директив `default` и `stored`. Эти директивы больше никак не влияют на работу компилятора, но IDE читает RTTI для того чтобы определить, какие свойства следует сохранять в *dfm*-файле.
- Заметьте, что директива `default` применяется в двух случаях: указание значения по умолчанию для скалярного свойства и обозначе-

ние свойства-массива по умолчанию. Конкретная интерпретация директивы `default` зависит от ее расположения (до или после точки с запятой).

- Все директивы не обязательны, но если они используются, то должны указываться в следующем порядке: `index`, `read`, `write`, `default` или `nodefault`, `stored`, `implements`.
- Параметры *`Getter`* и *`Setter`* – не обязательны, однако хотя бы один из них должен присутствовать. Для публикуемого свойства должны быть указаны оба. *`Getter`*, *`Setter`* и *`Stored`* могут быть ссылками на поля или именами методов. Метод может быть виртуальным, но не может быть динамическим или обработчиком сообщений. Значение *`Stored`* также может быть константой или выражением типа `Boolean`. Дополнительная информация приведена в описании директив `read`, `stored` и `write`.
- Разработчик может заново определить свойство, унаследованное от базового класса, указав только ключевое слово `property` и имя свойства. Свойство сохраняет все директивы, унаследованные от базового класса. Можно переопределить свойство с более высоким уровнем доступа, чем тот, который был установлен в базовом классе. Этот прием часто используется в компонентах, где настраиваемый базовый класс объявляет свойства как защищенные, а конкретные производные классы объявляют их заново как открытые или публикуемые.

Пример

```
{
  Красно-черные деревья:
  Каждый узел имеет красный или черный цвет. Дерево сбалансировано
  по определенным правилам в отношении красных и черных узлов.
}

  type
  TRbColor = (rbBlack, rbRed);
  TRbNode = class
  private
    fLeft, fRight: TRbNode;
    fItem: ICollectible;
    fParent: Integer;
    function GetParent: TRbNode;
    procedure SetParent(Parent: TRbNode);
  protected
    constructor Create(Item: ICollectible; Leaf, Parent: TRbNode;
                       Color: TRbColor);
    function IsRed: Boolean;
    function IsBlack: Boolean;
    function GetColor: TRbColor; virtual;
    procedure SetColor(Color: TRbColor); virtual;
```

```
    property Left: TRbNode read fLeft write fLeft;
    property Right: TRbNode read fRight write fRight;
    property Item: ICollectible read fItem write fItem;

    Property Color: TRbColor read GetColor write SetColor;
    Property Parent: TRbNode read GetParent write SetParent;

  end;
```

Смотрите также

Ключевые слова Class, Dispinterface, Interface, директивы Default, Implements, Index, NoDefault, Read, Readonly, Stored, Write, Writeonly.

Protected, директива

Синтаксис

```
type Объявление класса
protected
   Описания полей...
   Описания методов и свойств...
end;
```

Описание

Директива protected обозначает начало защищенной секции в объявлении класса. Protected (защищенные) поля, методы и свойства доступны только для методов самого класса и его потомков. Классам и подпрограммам, объявленным в том же модуле, также доступны объявления в секции protected.

Советы и приемы

- Большинство методов и свойств, не являющихся частью общедоступного интерфейса класса, обычно объявляются как защищенные. При объявлении закрытого (private) метода или отсутствии защищенного свойства для доступа к закрытому полю программист ограничивает возможности повторного применения своего класса. Нельзя предвидеть способы использования класса в других проектах, и потому сокрытие важных описаний в секции private лишь сокращает количество его возможных пользователей.
- С другой стороны, подробности реализации класса иногда имеет смысл помещать в секцию private, чтобы иметь свободу изменений реализации в будущих версиях класса. Обязательно определяйте соответствующие защищенные методы, которые позволят повторно использовать ваш класс, даже если они лишь вызывают закрытые (private) методы.
- Многие защищенные (protected) методы следует объявлять виртуальными. Поскольку трудно предвидеть, каким образом производный класс захочет расширить или изменить поведение роди-

тельского класса, лучше ошибиться в сторону большей совместимости. Объявляйте защищенные методы статическими только в том случае, если для этого есть причины.

Пример

Пример рассмотрен в описании ключевого слова `class`.

Смотрите также

Ключевое слово `Class`, директивы `Private`, `Public`, `Published`.

PShortString, тип

Синтаксис

```
type PShortString = ^ShortString;
```

Описание

Тип `PShortString` объявлен для удобства работы с указателями на `ShortString`.

Смотрите также

Типы `PAnsiString`, `PString`, `ShortString`.

PString, тип

Синтаксис

```
type PString = ^String;
```

Описание

Тип `PString` объявлен для удобства работы с указателями на `String`. Заметьте, что строковый тип является указателем, поэтому `PString` применяется редко.

Смотрите также

Ключевое слово `PWideString`, типы `PAnsiString`, `PShortString`, `String`.

Ptr, функция

Синтаксис

```
function Ptr(Value: Integer): Pointer;
```

Описание

Функция `Ptr` преобразует `Value` в нетипизированный указатель. Функция `Ptr` не является настоящей функцией.

Советы и приемы

Вызов Ptr лишь выполняет преобразование значения в указатель. По возможности преобразуйте значение к конкретному типу указателя, используйте, например, PInt64 вместо нетипизированного Pointer. Типизированные указатели обеспечивают лучшую защиту.

Смотрите также

Функции Addr, Assigned, тип Pointer.

Public, директива

Синтаксис

```
type Объявление класса
public
   Описания полей...
   Описания методов и свойств...
end;
```

Описание

Директива public обозначает начало открытой секции в объявлении класса. Открытые (public) поля, методы и свойства доступны для всех подпрограмм и методов того же модуля или любого модуля, использующего данный (если объявление класса находится в разделе интерфейса модуля).

Советы и приемы

- Поля никогда не следует объявлять в секции public. Объявляйте их в секции private, а в секции public – свойства для доступа к этим полям. Свойства обеспечивают большую гибкость, чем поля, и дают возможность вносить в класс изменения, которые не отражаются на применении класса.
- Открытые методы и свойства определяют внешнее поведение класса и его объектов. После того как класс начал использоваться, избегайте вносить изменения в открытые методы и свойства, насколько это возможно. Вместо изменения метода объявите новый, работающий по-другому. Изменения в секции public позволяют как усовершенствовать, так и нарушить работу других частей программы.
- Объявление класса может начинаться с первой, неименованной секции. По умолчанию для неименованной секции подразумевается директива public, если только класс не компилировался с директивами компилятора $M или $TypeInfo, включающими поддержку RTTI, и если класс не является потомком класса со включенным RTTI. Класс TPersistent из модуля Classes использует RTTI, и потому все перманентные классы (включая все компоненты, управляющие элементы и формы) имеют RTTI. Начальная, неименованная секция для классов с RTTI является публикуемой (published).

Пример

Пример рассмотрен в описании ключевого слова `class`.

Смотрите также

Ключевое слово `Class`, директивы `Private`, `Protected`, `Published`, директивы компилятора `$M`, `$TypeInfo`.

Published, директива

Синтаксис

```
type Объявление класса
published
   Описания полей...
   Описания методов и свойств...
end;
```

Описание

Директива `published` обозначает начало публикуемой секции в объявлении класса. Как и открытые, публикуемые поля, методы и свойства доступны для всех подпрограмм и методов того же модуля или любого модуля, использующего данный (если объявление класса находится в разделе интерфейса модуля).

Советы и приемы

- Разница между секциями `public` и `published` состоит в том, что для полей, методов и свойств, объявленных в секции `published`, компилятором формируется информация времени выполнения о типах (RTTI).
- Компилятор рассматривает секцию `published` так же, как `public`, если только директивой компилятора `$M` и `$TypeInfo` не включена поддержка RTTI и класс не является наследником класса с RTTI. Класс `TPersistent` из модуля `Classes` использует RTTI, и потому все перманентные классы (включая все компоненты, управляющие элементы и формы) имеют RTTI. Начальная, неименованная секция для классов с RTTI является публикуемой (`published`).
- Delphi IDE использует для хранения полей и методов формы начальную, неименованную секцию объявления класса формы. Хотя компилятор и позволяет переместить эти объявления в любую другую публикуемую (`published`) секцию класса, IDE требует, чтобы они находились в неименованной секции. Поскольку формы и модули данных являются потомками класса `TPersistent`, начальная секция их описаний является публикуемой (`published`).
- Публикуемые поля могут быть объектного типа, и этот класс также должен поддерживать RTTI.

- Публикуемые свойства не могут иметь тип «массив». Тип публикуемого свойства может быть порядковым, множеством в пределах Integer, объектом (class), поддерживающим RTTI, Variant, вещественным типом, кроме Real48, любым строковым типом, но не может быть массивом, указателем и «старым» объектом (object).

Пример

Пример рассмотрен в описании ключевого слова class.

Смотрите также

Ключевое слово Class, директивы Private, Protected, Public, директивы компилятора $M, $TypeInfo.

PUniEntryTable, тип

Синтаксис

```
type PUnitEntryTable = ^UnitEntryTable;
```

Описание

Тип PUnitEntryTable объявлен для удобства работы с указателями на UnitEntryTable.

Смотрите также

Ключевые слова Finalization, Initialization, Unit, типы PackageInfo, PackageInfoTable, UnitEntryTable.

PVarArray, тип

Синтаксис

```
type PVarArray = ^TVarArray;
```

Описание

Тип PVarArray объявлен для удобства работы с указателями на TVarArray. Запись TVarData хранит массив Variant в виде указателя PVarArray. Дополнительную информацию можно найти в описании типа TVarData.

Смотрите также

Типы TVarArray, TVarArrayBound, TVarData, Variant.

PVarData, тип

Синтаксис

```
type PVarData = ^TVarData;
```

Описание

Тип PVarData объявлен для удобства работы с указателями на TVarData.

Смотрите также

Типы TVarData, Variant

PVariant, тип

Синтаксис

```
type PVariant = ^Variant;
```

Описание

Тип PVariant объявлен для удобства работы с указателями на Variant.

Смотрите также

Тип Variant.

PVarRec, тип

Синтаксис

```
type PVarRec = ^TVarRec;
```

Описание

Тип PVarRec объявлен для удобства работы с указателями на TVarRec. Тип TVarRec применяется для интерпретации вариантного открытого массива (array of const) внутри подпрограммы.

Смотрите также

Ключевое слово Array, тип Variant.

PWideChar, тип

Синтаксис

```
type PWideChar = ^WideChar;
```

Описание

Тип PWideChar – это указатель на WideChar, и в расширенном синтаксисе Delphi он может также рассматриваться как WideString и как указатель на array of WideChar.

Советы и приемы

- PWideChar чаще всего используется как тип параметра для DLL, написанных на C или C++, например для Windows API.
- Указатель PWideChar может рассматриваться как указатель на массив WideChar. Индекс массива – поддиапазон целого, начинающийся с нуля. Delphi не выполняет проверку на границы массива. Используется соглашение, по которому конец строки обозначается символом #0.

- Арифметика указателей для типа PWideChar при добавлении и вычитании целых значений работает аналогично процедурам Inc и Dec.
- Расширенный синтаксис Delphi по умолчанию включен. Он может быть выключен с помощью директивы компилятора $X и $ExtendedSyntax. В этом случае поведение компилятора становится более похожим на стандартный Паскаль.

Смотрите также

Типы AnsiChar, AnsiString, Char, PAnsiChar, PChar, WideString, ключевые слова Array, String, директивы компилятора $ExtendedSyntax, $X.

PWideString, тип

Синтаксис

```
type PWideString = ^WideString;
```

Описание

Тип PWideString определен для удобства работы с указателями на WideString. Заметьте, что WideString сама является указателем, поэтому PWideString используется редко.

Смотрите также

Типы PAnsiString, PShortString, PString, WideString, ключевое слово String.

Raise, ключевое слово

Синтаксис

```
raise Ссылка на объект
raise Ссылка на объект at Pointer
raise
```

Описание

Предложение raise генерирует исключительную ситуацию. Объект исключительной ситуации передается обработчику, по завершении которого Delphi автоматически освобождает объект. Обычно при указании ссылки на объект создается новый экземпляр объекта исключительной ситуации, класс которого является производным от класса Exception из модуля SysUtils, но для этого может использоваться любая ссылка на объект.

По умолчанию в качестве адреса исключительной ситуации Delphi сохраняет адрес оператора raise. С помощью директивы at можно указать другой адрес. Внутри обработчика для повторного вызова текущей исключительной ситуации можно использовать простую форму оператора raise. В этом случае Delphi не освобождает объект Exception

до тех пор, пока следующий обработчик из стека вызовов не получит возможность обработать эту исключительную ситуацию.

Советы и приемы

- Низкоуровневый код и утилиты должны сообщать о чрезвычайных условиях, генерируя исключительные ситуации. Высокоуровневый и пользовательский код должен перехватывать и обрабатывать исключительные ситуации с помощью предложения `try-except` или обработчика `TApplication.OnException` удобным для пользователя образом.
- Генерирование и обработка исключительной ситуации занимает время и ресурсы процессора. Делайте это только при возникновении чрезвычайных условий. Не используйте исключительные ситуации как обычный способ выдачи информации. Например, в классе коллекции может быть реализован метод `Contains`, который возвращает `True`, если элемент присутствует в коллекции, и `False`, если он отсутствует. Нормальное возвращаемое значение – `True` или `False`, т. е. метод не вызывает исключительную ситуацию. С другой стороны, метод `Delete` удаляет элемент коллекции. Если элемент отсутствует, это чрезвычайное условие, и метод `Delete` должен генерировать исключительную ситуацию.
- При необходимости создавайте новые классы исключительных ситуаций. Класс передает информацию обработчику, и обработчик использует этот класс для выполнения действий, специфических для определенного типа исключительных ситуаций.
- Применяйте для сообщения об исключительной ситуации `resourcestring`, чтобы облегчить локализацию приложения.

Пример

```
// Генерируем обобщенную исключительную ситуацию по вызывающему
// адресу.
procedure Error(const Fmt: string; const Args: array of const);
  function Caller: Pointer;
  asm
    // Извлекаем вызывающий адрес из стека. Реализация может
    // отличаться в зависимости от версии и настроек компилятора.
    MOV EAX, [ESP+4]
  end;

begin
  raise Exception.CreateFmt(Fmt, Args) at Caller;
end;

// Удаляем файл, генерируем исключительную ситуацию
// в случае любой неудачи, в том числе отсутствия файла.
procedure RemoveFile(const FileName: string);
begin
```

```
  if not DeleteFile(FileName) then
    raise Exception.CreateFmt('Невозможно удалить %s: %s',
      [FileName, SysErrorMessage(GetLastError)]);
end;
```

Смотрите также

Директива `At`, ключевые слова `Except`, `Finally`, `Try`, функция `IOResult`.

RaiseList, функция

Синтаксис

```
function RaiseList: Pointer;
```

Описание

Функция `RaiseList` возвращает указатель на текущий фрейм исключительной ситуации, который может быть первым узлом списка. В модуле `System` не объявлен тип для фрейма исключительной ситуации; структура записи приведена в примере.

Первый фрейм в списке – это текущая исключительная ситуация. Если ошибка возникла в обработчике исключительной ситуации, в списке может быть несколько фреймов.

`RaiseList` – настоящая функция.

Советы и приемы

- Обычно выполнение действий, необходимых для обработки исключительных ситуаций, обеспечивает конструкция `try-except`. Не вызывайте функцию `RaiseList`, если только не хотите обрабатывать исключительные ситуации нестандартным образом.
- Каждый поток поддерживает собственный список исключительных ситуаций.

Пример

```
type
  PRaiseFrame = ^TRaiseFrame;
  TRaiseFrame = packed record
    NextRaise: PRaiseFrame;
    ExceptAddr: Pointer;
    ExceptObject: TObject;
    ExceptionRecord: PExceptionRecord; // Тип, определенный в модуле
                                       // Windows.
  end;
...
try
  ...
except
  Code := PRaiseFrame(RaiseList).ExceptionRecord.ExceptionCode;
  if Code = Exception_BreakPoint then
```

```
    ...
end;
```

Смотрите также

Переменные ErrorProc, ExceptProc, ключевые слова Except, Raise, Try, процедура SetRaiseList.

Random, функция

Синтаксис

```
function Random: Extended;
function Random(Limit: Integer): Integer;
```

Описание

Функция Random возвращает псевдослучайное число. При вызове без аргументов функция возвращает вещественное значение в диапазоне $0 \leq$ Результат < 1. Вторая форма функции принимает целый аргумент и возвращает целое значение в диапазоне $0 \leq$ Результат $<$ Limit.

Не является настоящей функцией.

Советы и приемы

- Delphi использует генератор псевдослучайных чисел с циклом, равным 2^{32}. Он удовлетворяет потребностям простого моделирования, однако не подходит для шифрования и других применений, где требуется более качественный генератор.
- Для того чтобы последовательность псевдослучайных чисел начиналась с разных значений при каждом запуске программы, используйте процедуру Randomize.

Пример

```
type
  TDieRoll = 1..6;
  TDiceRoll = 2..12;

// Имитация броска одиночной игральной кости. Заметьте, что
// Random возвращает значение в диапазоне [0, 5], поэтому
// добавляем 1, чтобы получить [1, 6].
function Die: TDieRoll;
begin
  Result := Random(6) + 1;
end;

// Имитация броска двух костей.
function Dice: TDiceRoll;
begin
  Result := Die + Die;
end;
```

Смотрите также

Процедура Randomize, переменная RandSeed.

Randomize, процедура

Синтаксис

```
procedure Randomize;
```

Описание

Вызовите процедуру Randomize один раз в начале программы для того, чтобы при каждом запуске последовательность псевдослучайных чисел начиналась с различных значений. Randomize устанавливает значение переменной RandSeed, используя текущее время дня. Randomize — настоящая процедура.

Советы и приемы

- При тестировании программы не вызывайте процедуру Randomize. В этом случае будет получена одна и та же последовательность псевдослучайных чисел при каждом запуске программы, и нельзя будет сравнить результаты запусков. После тестирования добавьте вызов Randomize, чтобы при каждом запуске программы последовательность чисел была различной.
- Не вызывайте Randomize более одного раза в программе. Это разрушает случайность последовательности псевдослучайных чисел.
- Randomize использует для инициализации переменной RandSeed текущее время дня. Если программа каждый день запускается в одно и то же время, выберите другой способ инициализации RandSeed.

Пример

```
unit RandomDice;
interface
...
function Die: TDieRoll;
function Dice: TDiceRoll;
implementation
...
initialization
  Randomize;
end.
```

Смотрите также

Функция Random, переменная RandSeed.

RandSeed, переменная

Синтаксис

```
var RandSeed: LongInt;
```

Описание

В RandSeed содержится начальное значение для генератора псевдослучайных чисел. Если не была вызвана процедура Randomize, начальное значение RandSeed равно 0.

Пример

```
// Инициализируйте RandSeed наименее предсказуемым начальным
// значением. В этой версии используется время дня, идентификатор
// процесса и время, прошедшее с момента загрузки. Маловероятно,
// что все три величины будут совпадать при различных запусках
// программы.
procedure MyRandomize;
begin
  Randomize; // Начинаем со времени дня.
  RandSeed := ((RandSeed shl 8) or GetCurrentProcessID) xor
     GetTickCount;
end;
```

Смотрите также

Функция Random, процедура Randomize.

Read, директива

Синтаксис

```
property Имя: Тип... read Getter ...;
```

Описание

Директива read сообщает Delphi, каким образом получить значение свойства. *Getter* может быть именем поля, методом данного класса или методом одного из родительских классов.

Если *Getter* – поле, его тип должен совпадать с типом свойства. Уровни доступа учитываются как обычно, поэтому поле не может быть закрытым полем родительского класса, если только родительский класс не определен в том же модуле. Обычно поле определяется в секции private того же класса, в котором определено свойство. Поле может быть составным (записью или массивом), и тогда в *Getter* должно быть указано поле записи или элемент массива (с константным индексом) соответствующего типа. Можно использовать вложенные записи и массивы.

Если *Getter* – метод, он должен быть функцией, тип возвращаемого значения которой совпадает с типом свойства. Метод может быть статическим или виртуальным, но не может быть динамическим.

Если свойство является индексированным или массивом, *Getter* должен быть методом. Первый параметр – это целое значение индекса метода. Последующие аргументы – это индексы массива. Тип каждого из аргументов метода *Getter* должен соответствовать типу индекса массива.

Когда пользователь запрашивает значение свойства, Delphi получает это значение от поля *Getter* или путем вызова метода *Getter*.

Советы и приемы

- Если поле *Getter* указано, Delphi компилирует все ссылки на свойство в прямые обращения к полю, поэтому использование свойства не приводит к падению производительности.
- В «хорошем» стиле программирования принято объявлять все поля закрытыми, и создавать защищенные, открытые и публикуемые свойства для доступа к этим полям. Если позднее потребуется изменить класс, можно будет заменить поле на вызов метода, не затрагивая код, который зависит от класса, и его свойства.

Пример

Примеры рассмотрены в описании ключевого слова `property`.

Смотрите также

Ключевые слова `Class`, `Property`, директивы `Index`, `Write`.

Read, процедура

Синтаксис

```
procedure Read(var F: File; var Переменная; ...);
procedure Read(var F: TextFile; var Переменная; ...);
procedure Read(var Переменная; ...);
```

Описание

Процедура `Read` читает данные из двоичного или текстового файла. Она не является настоящей процедурой.

Для чтения типизированного двоичного файла *Переменная* должна быть того же типа, что и базовый тип файла. Delphi читает одну запись из файла в *Переменную* и перемещает позицию файла, готовясь к чтению следующей записи. Если файл нетипизированный, Delphi читает столько байт, сколько определено размером записи при открытии файла процедурой `Reset`. Вы можете перечислить более одной переменной в аргументах `Read`, и в этом случае `Read` читает несколько записей и присваивает значение каждой из них отдельной переменной.

При чтении текстового файла Read выполняет форматированное чтение. Delphi читает символы из текстового файла и интерпретирует их в соответствии с типом каждой переменной. При чтении числовых значений Read пропускает пустые символы (пробелы, табуляции и переводы строк), читает число и останавливает чтение при получении следующего пустого символа.

При чтении строк и символов Read не пропускает пустых символов. Если *Переменная* является длинной строкой, Read читает в переменную целую строку, но без символов конца строки. Если *Переменная* – короткая строка, Read останавливается в конце строки или в соответствии с размером строки, смотря что произойдет раньше.

Ошибки

- Если файлу не назначено имя процедурой AssignFile или Assign, Read генерирует ошибку ввода/вывода 102.
- Если файл не открыт для чтения, Read выдает ошибку ввода/вывода 103.
- Если входные данные сформатированы неверно (например, попытка прочитать 3.14 в целую переменную), Read выдает ошибку ввода/вывода 106.
- Если в процедуре Read произошел сбой по другой причине (например, ошибка сети), Read выдает в качестве ошибки ввода/вывода код ошибки Windows.
- Если входное значение находится вне диапазона типа (предположим, 257 при чтении Byte), Read преобразует значение к корректному типу, не вызывая исключительной ситуации, или выдает ошибку времени выполнения.

Советы и приемы

- Delphi не буферизует ввод из двоичного файла; для чтения нескольких записей за один раз можно использовать процедуру BlockRead.
- Чтение за концом двоичного файла вызывает ошибку ввода/вывода 100. В текстовом файле Read всегда успешно выполняет чтение, извлекая символ #26. При чтении числа или строки в этом случае получается ноль или пустая строка.
- Если первый аргумент – файл – не указан, Read читает данные из текстового файла Input.

Пример

```
var
  I: Integer;
  D1, D2: Double;
  S1, S2: string;
begin
```

```
Read(S1); // Читает строку целиком, но без символа конца строки.
Read(S2); // Всегда читает символ конца строки.
Read(I);  // Пропускает конец строки и пустые символы, чтобы
          // прочитать число.
Read(D1, D2);  // Пробелы также служат для ограничения и разделения
               // числовых значений.
```

Смотрите также

Процедура BlockRead, ReadLn, Write, ключевое слово File, переменная Input, функция IOResult, тип TextFile.

ReadLn, процедура

Синтаксис

```
procedure ReadLn(var F: TextFile; var Переменная; ...);
procedure ReadLn(var Переменная; ...);
procedure ReadLn(var F: TextFile);
procedure ReadLn;
```

Описание

Процедура ReadLn аналогична Read, но по завершении чтения данных в *Переменные* ReadLn пропускает остаток строки и готовится к чтению следующей. Без указания аргументов *Переменная* ReadLn только пропускает остаток строки.

ReadLn не может использоваться для двоичного файла и не является настоящей процедурой.

Советы и приемы

- Для ReadLn строка заканчивается переводом каретки (#13) или концом файла. За переводом каретки может следовать либо перевод строки (#10), либо конец файла.
- Для чтения строк предпочтительнее вызывать ReadLn вместо Read. Процедура Read читает все символы длинной строки, за исключением символа конца строки. При повторном вызове Read строка снова будет прочитана до конца, но в результате будет получена пустая строка. Процедура ReadLn читает строку, пропускает конец строки, помещая следующую строку текста в текстовую переменную.
- Если первый аргумент – текстовый файл – не указан, ReadLn читает данные из Input.

Пример

```
program bmindex;
// Вычисляет индекс массы тела по росту человека в сантиметрах и весу
// в килограммах. Если индекс больше 26, человек имеет лишний вес.
var
  Height, Weight: Single;
  BMI: Integer;
```

```
begin
  WriteLn('Введите рост в сантиметрах: ');
  ReadLn(Height);
  WriteLn('Введите вес в килограммах: ');
  ReadLn(Weight);
  BMI := Round((Weight * 10000) / (Height * Height));
  Write('Индекс веса тела', BMI, '.');
  if BMI > 26 then
    WriteLn(' Советую вам сбросить вес.')
  else
    WriteLn;

  WriteLn('Нажмите клавишу ENTER для выхода из программы.');
  ReadLn;
end.
```

Смотрите также

Процедуры BlockRead, Read, Write, WriteLn, переменная Input, функция IOResult, тип TextFile.

Readonly, директива

Синтаксис

```
property Имя: Тип readonly;
```

Описание

Директива readonly применяется только для свойств, объявляемых в диспетчерском интерфейсе. Подробности приведены в описании ключевого слова dispinterface.

Смотрите также

Ключевые слова Dispinterface, Property, директива Writeonly.

Real, тип

Синтаксис

```
type Real = Double;
```

Описание

Тип Real — это альтернативное имя для Double; он оставлен для совместимости с программами, написанными на стандартном Паскале и Turbo Pascal. Для чтения файла, хранящего значения Real, принятые в Turbo Pascal, используйте тип Real48. Директива компилятора $RealCompatibility позволяет рассматривать ссылки на тип Real как Real48.

Смотрите также

Типы Double, Real48, директива компилятора $RealCompatibility.

Real48, тип

Синтаксис

```
type Real48;
```

Описание

Тип Real48 оставлен для обратной совместимости с Turbo Pascal. Он определяет шестибайтовый вещественный тип. Значения Real48 должны быть преобразованы в Extended перед использованием в расчетах, поэтому данный тип будет работать медленнее, чем Double или Extended.

Тип Real48 состоит из 8-битовой экспоненты и 39-битовой нормализованной мантиссы. Он не поддерживает денормализованные значения, бесконечность и «нечисловые» значения. Если экспонента равна нулю, число также равно нулю.

Смотрите также

Типы Extended, Real, директива компилятора $RealCompatibility.

ReallocMem, процедура

Синтаксис

```
procedure ReallocMem(var P: Pointer; NewSize: Integer);
```

Описание

Процедура ReallocMem изменяет размер динамически выделенного блока памяти, максимально сохраняя его предыдущее содержимое.

Если P равно nil, ReallocMem работает аналогично GetMem. Если NewSize равно нулю – аналогично FreeMem, т. е. значение P не изменяется и после завершения процедуры содержит некорректное значение.

По мере возможности ReallocMem пытается изменить размер текущего блока памяти, но если это невозможно, выделяется новый блок требуемого размера и старые данные копируются в новый блок памяти. В любом случае, если NewSize больше текущего размера, дополнительная память не инициализируется.

ReallocMem не является настоящей процедурой.

Советы и приемы

- Необходимо прогнозировать рост блока, увеличивающегося постепенно. Выгоднее увеличивать размер блока реже, но большими порциями. Старайтесь вызывать эту процедуру как можно реже, т. к. ей часто не удается заново использовать текущий блок. Кроме того, в процессе увеличения размера блока потери производительности при копировании старых данных в новый блок увеличиваются.

- Процедура ReallocMem во встроенном менеджере памяти Delphi поддерживает многопоточность, т. е. можно вызывать ReallocMem одновременно из нескольких потоков, но только если переменная IsMultiThread равна True.
- Встроенный менеджер памяти Delphi предполагает, что программа часто выделяет и освобождает блоки разного размера. Большинство программ соответствуют этому допущению, но не все. Если программа часто вызывает ReallocMem для постепенного увеличения размера большого блока, могут возникнуть проблемы с производительностью и чрезмерным расходом памяти. Решение состоит в том, чтобы либо реже вызывать ReallocMem, либо, если ее частые вызовы необходимы, установить собственный менеджер памяти, который сможет увеличивать размер блока без выделения новых блоков большего размера.

Пример

```
// Тривиальный список, увеличивающийся пропорционально размерам,
// а не фиксированными шагами.
type
  TObjectArray = array[0..MaxInt div SizeOf(TObject) - 1] of TObject;
  PObjectArray = ^TObjectArray;
  TWholeNumber = 0..MaxInt;
  TArray = class
  private
    fList: PObjectArray;
    fCapacity: TWholeNumber;
    fCount: TWholeNumber;
    procedure Grow;
  public
    constructor Create;
    procedure Add(Obj: TObject);
    procedure Remove(Obj: TObject);
    property Items[Index: TWholeNumber]: TObject
        read GetItem write SetItem;
    property Count: TWholeNumber read fCount;
    property Capacity: TWholeNumber read fCapacity;
  end;

procedure TArray.Add(Obj: TObject);
begin
  if Count = Capacity then
    Grow;
  fList[Count] := Obj;
  Inc(fCount);
end;

procedure TArray.Grow;
begin
  // Когда размер небольшой, увеличиваем на 100%, иначе – только на 50%.
```

```
    if Capacity < 64 then
      fCapacity := Capacity * 2
    else
      fCapacity := Capacity + Capacity div 2;
    ReallocMem(fList, Capacity * SizeOf(TObject));
  end;
```

Смотрите также

Процедуры FreeMem, GetMem, функции ReallocMemory, SysReallocMem.

ReallocMemory, функция

Синтаксис

```
function ReallocMemory(Ptr: Pointer; Size: Integer): Pointer; cdecl;
```

Описание

Функция ReallocMemory вызывает менеджер памяти для повторного выделения блока памяти другого размера. Функция ReallocMemory возвращает указатель на начало блока.

Функция ReallocMemory предназначена для использования в C++ Builder. Если вы пишете свой менеджер памяти на Delphi, вызывайте SysReallocMem.

ReallocMemory – настоящая функция.

Смотрите также

Функции FreeMemory, GetMemory, SysReallocMem, переменная IsMultiThread, процедура ReallocMem.

Record, ключевое слово

Синтаксис

```
type Имя = record Поля... end;
type Имя = packed record Поля... end;
type Имя = record Поля...
             case Селектор of Вариант, ...: (Поля...;);
             ...
           end;
type Имя = packed record Members...
             case Имя: Селектор of Вариант, ...: (Поля...;);
             ...
           end;
```

Описание

Запись (record) в Delphi аналогична записи стандартного языка Паскаль. Запись – это структурированный тип, содержащий ноль или более полей. Каждое поле, как и переменная, состоит из имени поля и

его типа. В переменной типа «запись» есть место для всех полей записи, что позволяет ссылаться на индивидуальные поля записи.

По умолчанию поля записи выравниваются, чтобы каждое поле начиналось с естественной границы. Упакованная (packed) запись не содержит дополнительных выравнивающих байтов между полями, что приводит к уменьшению размера записи, но за счет увеличения времени доступа. Подробности рассмотрены в описании ключевого слова packed.

Вариантные записи

Запись может быть вариантной (эквивалент объединения (union) в C и C++), в этом случае некоторые поля разделяют общее пространство внутри записи. Вариантная часть записи должна следовать после невариантных полей. *Селектор* может содержать только имя типа или объявлять отдельное поле записи. Поле-селектор сообщает, какой из вариантов следует использовать. Если поле-селектор не указано, то для определения корректного варианта структуры записи следует применять другие методы.

Для каждого из вариантов указывается одна и более констант, тип которых должен совпадать с типом селектора. После двоеточия в описании варианта внутри скобок указывается ноль или более полей. В вариантной части записи нельзя размещать длинные строки, динамические массивы, интерфейсы и значения типа Variant. На этапе компиляции Delphi не может определить, какие вариантные поля будут использоваться в конкретном случае, поэтому нельзя определить моменты для инициализации или завершения строк, динамических массивов и значений типа Variant.

Вариантные записи могут быть упакованными. Каждый из вариантов упаковывается независимо от других. Размер вариантной записи равен наибольшему из всех вариантов.

Советы и приемы

Записи требуют меньше накладных расходов, чем классы, поэтому они часто применяются в ситуациях, когда требуется создать много записей, но с ними не производятся сложные операции.

Пример

Дополнительные примеры приведены в описаниях TVarRec и TVarData.

```
type
  TEmployeeKind = (ekGrunt, ekBoss, ekSlacker);
  PEmployee = ^TEmployee;
  TEmployeeArray = array[0..MaxEmployee] of PEmployee;
  TEmploye e = record
    Name, ID: string;
    Salary: Currency;
    case Kind: TEmployeeKind of
```

```
    ekGrunt: (Boss: PEmployee;);
    ekBoss: (Group: TEmployeeArray;
             GroupSize: 0..MaxEmployee;);
    ekSlacker: ();
  end;
```

Смотрите также

Ключевые слова `Case`, `Packed`, `Type`, директивы компилятора `$A`, `$Align`.

RefAnyProc, переменная

Синтаксис

```
var RefAnyProc: Pointer;

procedure RefAny(var V: Variant);
RefAnyProc := @RefAny;
```

Описание

Переменная `RefAnyProc` указывает на процедуру, которая создает ссылку на вариантное значение `varAny`. Если требуется поддерживать счетчик ссылок, процедура *RefAny* увеличивает счетчик ссылок. Если каждое значения `varAny` должно содержать уникальную копию данных, можно скопировать данные `varAny` в процедуре `RefAnyProc`. По умолчанию в переменной хранится ссылка на процедуру, вызывающую ошибку времени выполнения 15 (`EVariantError`).

Советы и приемы

Модуль `CorbaObj` устанавливает значение переменной на процедуру, поддерживающую тип `Any` из CORBA. В программе, не работающей с CORBA, значения `varAny` можно использовать для собственных целей.

Пример

Пояснения по этому примеру смотрите в описании переменной `ChangeAnyProc`.

```
// Все значения Int64, сохраненные в varAny, содержат уникальное
// значение Int64. Когда Delphi создает ссылку, V устанавливается
// равным ссылке на копию оригинального значения. Эта процедура
// присваивает V уникальное значение типа Int64.
procedure RefVarInt64(var V: Variant);
var
  Value: Int64;
begin
  if TVarData(V).VType = varAny then
  begin
    Value := PInt64(TVarData(V).VAny)^;
    SetVarInt64(V, Value);
  end;
end;
```

```
...
RefAnyProc := @RefVarInt64;
```

Смотрите также

Переменные ChangeAnyProc, ClearAnyProc, типы TVarData, Variant.

Register, директива

Синтаксис

Объявление подпрограммы; register;

Описание

Директива register указывает компилятору на необходимость использования быстрого метода передачи параметров в функцию или процедуру через регистры, принятого в компиляторах фирмы Borland. Вызывающая сторона помещает первые три аргумента в регистры EAX, EDX и ECX, а остальные аргументы помещает в стек, начиная с самого левого аргумента. Параметры, которые не помещаются в 32-битовый регистр (например, Double) помещаются в стек, т. е. регистры содержат первые три аргумента, не находящиеся в стеке. Перед возвратом из подпрограммы аргументы выталкиваются из стека.

Функции возвращают порядковые значения, указатели, небольшие записи и множества в регистре EAX, вещественные – в стеке FPU. Строки, динамические массивы, значения типа Variant, большие записи и множества передаются как скрытый параметр var. Этот скрытый параметр является последним. Если подпрограмма является методом, переменная Self передается в EAX.

Советы и приемы

- Соглашение о вызове register фирмы Borland отличается от соглашения о вызове register фирмы Microsoft. Для максимальной совместимости с DLL- или COM-объектом используйте соглашения stdcall или safecall.
- Соглашение о вызове register применяется для подпрограмм по умолчанию.

Смотрите также

Директивы CDecl, Pascal, SafeCall, StdCall, ключевые слова Function, Procedure.

Register, процедура

Синтаксис

```
procedure Register;
begin
```

```
  ...
end;
```

Описание

Создавая пакет для разработки, программист обычно размещает процедуру Register в одном или нескольких модулях, содержащихся в пакете. Когда Delphi загружает пакет для разработки в IDE, она ищет процедуру по имени Register и, если процедура существует, вызывает ее. Процедуры вызываются в порядке их перечисления в исходном файле пакета.

Советы и приемы

- Имя процедуры должно соответствовать следующему формату: первая R – заглавная, egister – строчные. Это ограничение необычно для Паскаля, но обеспечивает совместимость с C++ Builder, в котором используется зависимый от регистра язык. Можно создавать пакет в Delphi и загружать его в C++ Builder и наоборот.
- Процедура Register не является обязательной.
- IDE Delphi вызывает процедуру Register при загрузке пакета для разработки. Если тот же самый пакет загружается из приложения, процедура Register автоматически не вызывается.

Смотрите также

Директива Package.

RegisterModule, процедура

Синтаксис

```
procedure RegisterModule(LibModule: PLibModule);
```

Описание

Процедура RegisterModule добавляет LibModule в начало списка исполняемых модулей. Delphi автоматически вызывает RegisterModule для пакетов, которые загружаются при запуске приложения. Delphi также регистрирует пакеты, которые программа загружает динамически.

RegisterModule – настоящая процедура.

Советы и приемы

Для просмотра списка исполняемых модулей используйте процедуру EnumModule, или, для модулей ресурсов, процедуру EnumResouceModules.

Смотрите также

Процедуры EnumModules, EnumResourceModules, UnregisterModule, переменная LibModuleList, типы PLibModule, TLibModule.

Reintroduce, директива

Синтаксис

```
Объявление метода; reintroduce;
```

Описание

Директива `reintroduce` применяется в производном классе для сокрытия имени виртуального или динамического метода, объявленного в базовом классе. Если эта директива не указывается, компилятор предупредит вас, что производный класс скрывает родительский метод. Сокрытие в данном контексте означает, что родительский метод не может быть вызван с помощью ссылки на объект, тип которого равен производному классу. Любая попытка вызова метода фактически вызывает метод производного класса.

Советы и приемы

Директива `reintroduce`, если она присутствует, должна быть первой директивой в описании метода.

Пример

```
type
  TVector = class
  public
    procedure Add(Item: Integer); virtual;
  end;
  TSingleVector = class
  public
    // Метод TSingleVector.Add скрывает TVector.Add, о чем вас
        // предупредит компилятор. Если вы намеренно скрываете метод,
        // обозначьте это директивой reintroduce.
    procedure Add(Item: Single); reintroduce;
  end;
```

Смотрите также

Директивы `Dynamic`, `Override`, `Virtual`, ключевые слова `Function`, `Procedure`.

RemoveModuleUnloadProc, процедура

Синтаксис

```
procedure RemoveModuleUnloadProc(Proc: TModuleUnloadProc);

procedure ВашаПроцедура(HInstance: Integer);
begin ... end;
RemoveModuleUnloadProc(ВашаПроцедура);
```

Описание

`RemoveModuleUnloadProc` удаляет вашу процедуру выгрузки исполняемого модуля из списка. Delphi поддерживает список пакетов, составля-

ющих приложение. Когда Delphi выгружает пакет, вызывается последовательность процедур выгрузки, передавая каждой из них дескриптор экземпляра DLL.

Для добавления процедуры в список служит процедура AddModuleUnloadProc.

RemoveModuleUnloadProc – настоящая процедура.

Смотрите также

Процедуры AddModuleUnloadProc, UnregisterModule, переменная ModuleUnloadList, типы PModuleUnloadRec, TModuleUnloadRec.

Rename, процедура

Синтаксис

```
procedure Rename(var F: File; const NewName: string);
procedure Rename(var F: TextFile; const NewName: string);
```

Описание

Процедура Rename переименовывает файл. Сначала следует присвоить файловой переменной текущее имя файла с помощью процедуры AssignFile. Если файл не может быть переименован, в качестве ошибки ввода/вывода выдается код ошибки Windows.

Смотрите также

Процедуры AssignFile, Erase, ключевое слово File, функция IOResult, тип TextFile.

Repeat, ключевое слово

Синтаксис

```
repeat
  Операторы...
until Логические выражения
```

Описание

Ключевое слово repeat работает так же, как в стандартном Паскале. *Операторы* выполняются многократно до тех пор, пока *Логическое выражение* не станет равным истине.

Советы и приемы

- *Операторы* всегда выполняются минимум один раз.
- Для выделения блока не требуются слова begin и end, т. к. список предложений ограничивается ключевыми словами repeat и until.

Пример

```
if FindFirst(Path, faAnyFile, Search) = 0 then
```

```
try
  repeat
    MungFile(Path + Search.Name);
  until FindNext(Search) <> 0;
finally
  FindClose(Search);
end;
```

Смотрите также

Тип `Boolean`, процедуры `Break`, `Continue`, ключевые слова `For`, `Until`, `While`.

Requires, директива

Синтаксис

```
package Имя;
requires
  ДругоеИмя, ...;
end.
```

Описание

Директива `requires` обозначает начало списка имен пакетов, которые требуются для сборки пакета *Имя*. Имена пакетов разделяются запятыми.

Советы и приемы

- Когда Delphi компилирует пакет, должны быть найдены все модули, содержащиеся в пакете, и все модули, используемые этими модулями. Все модули делятся на две группы: модули, содержащиеся в пакетах `requires`, и все остальные. Последние – это те, которые Delphi компилирует в новый пакет. Если вы не хотите, чтобы определенный модуль включался в пакет, этот модуль должен содержаться в другом пакете, указанном в директиве `requires` компилируемого пакета.
- Обычно для добавления и удаления пакетов `requires` используется редактор пакетов Delphi. Можно редактировать исходный файл пакета вручную, но при этом окно редактора пакета не обновляется. Закройте и откройте проект или группу проектов для того, чтобы среда разработки Delphi заново прочитала исходный файл пакета.
- Если при использовании IDE возникают проблемы, попробуйте создать новую группу проектов и добавьте пакет в эту группу.
- Пакеты – это DLL, и Windows выполняет поиск файлов пакетов так же, как и DLL. Нельзя указать полный путь в списке пакетов `requires` – только имя пакета.

Смотрите также

Директивы `Contains`, `Package`.

Reset, процедура

Синтаксис

```
procedure Reset(var F: TextFile);
procedure Reset(var F: File);
procedure Reset(var F: File; RecordSize: Integer);
```

Описание

Процедура Reset открывает существующий файл. Если файл текстовый, он открывается только для чтения. Если файл двоичный, он открывается так, что допускаются вызовы функций чтения и записи, однако режим открытия файла операционной системой определяется переменной FileMode. Reset не является настоящей процедурой.

Если файл – типизированный двоичный файл (file of *тип данных*), используется вторая форма Reset, т. е. передается только файловая переменная. Если файл – нетипизированный двоичный файл (просто file), можно дополнительно указать второй аргумент, указывающий размер записи. Если размер записи не указан, то он равен 128 байт.

Ошибки

- Если до Reset не была вызвана процедура AssignFile, генерируется ошибка 102.
- Если Reset не может открыть файл, в качестве ошибки ввода/вывода выдается код ошибки Windows.

Советы и приемы

- Если файл уже открыт, он сначала закрывается.
- По умолчанию Reset открывает двоичный файл для доступа на чтение и на запись. Для открытия только на чтение установите значение переменной FileMode в 0 перед вызовом процедуры Reset.
- Чтобы дописать двоичный файл с помощью процедуры Reset, откройте файл, переместитесь в конец файла и после этого начинайте запись.
- Если файл нетипизированный, проще всего при его открытии указать размер записи, равный 1, и затем использовать BlockRead и BlockWrite для чтения и записи требуемого количества данных. Как вариант можно создать тип File of Byte, и тогда при вызове Reset размер записи указывать не потребуется.

Пример

```
// Копирование файла.
procedure CopyFile(const Source, Dest: string);
const
  BufferSize = 8192;
var
```

```
  SourceFile, DestFile: File of Byte;
  Buffer: PByte;
  Count: LongInt;
begin
  AssignFile(SourceFile, Source);
  AssignFile(DestFile, Dest);
  Buffer := nil;
  try
    GetMem(Buffer, BufferSize);
    FileMode := 0;
    Reset(SourceFile);
    Rewrite(DestFile);
    repeat
      BlockRead(SourceFile, Buffer^, BufferSize, Count);
      BlockWrite(DestFile, Buffer^, Count);
    until Count = 0;
  finally
    CloseFile(SourceFile);
    CloseFile(DestFile);
    FreeMem(Buffer);
  end;
end;
```

Смотрите также

Процедуры `Append`, `CloseFile`, `Rewrite`, переменная `FileMode`, функция `IOResult`.

Resident, директива

Синтаксис

```
exports Имя resident;
```

Описание

Директива `resident` значения не имеет. Она существует только для обратной совместимости с Delphi 1.

Смотрите также

Директива `Exports`.

Resourcestring, ключевое слово

Синтаксис

```
resourcestring Имя = Строка; ... ;
```

Описание

Объявление `resourcestring` аналогично объявлению `const` для строки, за исключением того, что значение строки сохраняется в строковом табличном ресурсе. Далее в программе можно использовать *Имя* анало-

гично любой константе строкового типа. Компилятор заменяет строковую константу на вызов процедуры LoadResString для загрузки ресурса во время выполнения программы.

Советы и приемы

- Применяйте объявления resourcestring для всех строк, которые могут быть локализованы: в сообщениях об ошибках, подсказках и т. д. Это особенно важно при создании компонентов и другого программного обеспечения, предназначенного для сторонних разработчиков. Необходимо учитывать, что компонент может использоваться в приложении, требующем локализации, и запланировать это заранее. Delphi автоматически загружает строку из DLL с локализованными ресурсами, если таковая имеется.
- Создавая resourcestring, которая используется в качестве строки формата (предположим, для функции SysUtils.Format), всегда включайте в нее спецификаторы позиций. Это поможет переводчику, которому потребуется изменить порядок параметров при переводе строки на другой язык.
- Delphi автоматически присваивает ресурсам идентификаторы. Это означает, что можно разместить resourcestring в любом модуле, не беспокоясь о конфликтах идентификаторов ресурсов. Кроме того, это означает, что идентификаторы ресурсов могут измениться при каждой сборке проекта.
- Адрес resourcestring — это указатель типа PResStringRec, который можно использовать во время выполнения для получения идентификатора ресурса.

Пример

```
// Копирование файла.
procedure CopyFile(const SourceName, DestName: string);
resourcestring
  CantCopy = 'Невозможно скопировать %0:s в %1:s: %2:s';
var
  SourceStream, DestStream: TFileStream;
begin
  SourceStream := nil;
  DestStream := nil;
  try
    try
      SourceStream := TFileStream.Create(SourceName, fmOpenRead);
      DestStream := TFileStream.Create(DestName, fmCreate);
      SourceStream.CopyFrom(DestStream);
    finally
      SourceStream.Free;
      DestStream.Free;
    end;
  except
```

```
      on E: Exception do
        raise Exception.CreateFmt(CantCopy,
                                  [SourceName, DestName, E.Message]);
    end;
  end;
```

Смотрите также

Директива `Const`**, функция** `LoadResString`**, типы** `PResStringRec`**,** `TResStringRec`**.**

Result, переменная

Синтаксис

```
var Result: Тип возвращаемого функцией значения;
```

Описание

Delphi автоматически создает в каждой функции переменную с именем `Result`. Тип переменной совпадает с типом возвращаемого функцией значения. Значение, содержащееся в переменной `Result`, становится значением, возвращаемым функцией.

В стандартном Паскале функция указывает свое значение путем присваивания его псевдопеременной, имя которой совпадает с именем функции. Переменная `Result` работает аналогично, но поскольку она ведет себя как обычная переменная, ее можно использовать в выражениях, что нельзя делать с именем функции.

Функции возвращают порядковые значения, указатели, небольшие записи и множества в регистре `EAX`, вещественные значения – в стеке FPU. Строки, динамические массивы, значения типа `Variant`, большие записи и множества передаются через скрытый параметр `var`.

Советы и приемы

- Рассматривайте переменную `Result` как параметр `var`. Она не содержит корректного значения в начале работы функции, и ее необходимо инициализировать, либо как-нибудь иначе обеспечить, чтобы в ней было корректное значение.
- Delphi объявляет переменную `Result` даже в том случае, если она не используется. Если попытаться объявить локальную переменную с именем `Result`, компилятор выдаст сообщение об ошибке (Identifier redeclared).

Пример

```
// Вощвращает реверсированную копию строки S.
function Reverse(const S: string): string;
var
  I, J: Integer;
begin
  I := 1;
```

```
    J := Length(S);
    SetLength(Result, J);
    while I <= J do
    begin
      Result[I] := S[J];
      Result[J] := S[I];
      Inc(I);
      Dec(J);
    end;
  end;
```

Смотрите также

Ключевое слово `Function`, директива `Out`.

Rewrite, процедура

Синтаксис

```
procedure Rewrite(var F: TextFile);
procedure Rewrite(var F: File);
procedure Rewrite(var F: File; RecordSize: Integer);
```

Описание

Процедура `Rewrite` открывает новый файл для записи. Если файл уже существует, его содержимое уничтожается до открытия файла. Текстовый файл открывается только для записи. Двоичный файл открывается для чтения и для записи. `Rewrite` не является настоящей процедурой.

Если файл – типизированный двоичный файл (`file of` *тип данных*), используется вторая форма `Reset`, т. е. передается только файловая переменная. Если файл – нетипизированный двоичный файл (просто `file`), можно дополнительно указать второй аргумент, определяющий размер записи. Если размер записи не указан, то он равен 128 байт.

Ошибки

- Если до процедуры `Rewrite` не была вызвана `AssignFile`, генерируется ошибка 102.
- Если `Rewrite` не может открыть файл, в качестве ошибки ввода/вывода выдается код ошибки Windows.

Советы и приемы

- Если файл уже открыт, он сначала закрывается.
- Если файл нетипизированный, то проще всего при его открытии указать размер записи, равный 1, и затем использовать `BlockRead` и `BlockWrite` для чтения и записи требуемого количества данных. Как вариант можно создать тип `File of Byte`, и тогда не потребуется указывать размер записи при вызове `Rewrite`.

Пример

Пример рассмотрен в описании процедуры Reset.

Смотрите также

Процедуры Append, AssignFile, CloseFile, Reset, функция IOResult.

RmDir, процедура

Синтаксис

```
procedure RmDir(const Directory: string);
```

Описание

Процедура RmDir удаляет каталог. Если она не может удалить каталог, генерируется ошибка ввода/вывода и выдается код ошибки Windows. В частности, если каталог не пустой, код ошибки равен Error_Dir_Not_Empty (145). RmDir не является настоящей процедурой.

Пример

```
// Удаляет каталог и его содержимое.
procedure DestroyDir(const Directory: string);
var
  Path: string;
  Search: TSearchRec;
begin
  Path := IncludeTrailingBackslash(Directory);
  if FindFirst(Path + '*.*', faAnyFile, Search) = 0 then
  try
    repeat
      if (Search.Attr and faDirectory) <> 0 then
        DestroyDir(Path + Search.Name)
      else
        DeleteFile(Path + Search.Name);
    until FindNext(Search) <> 0;
  finally
    FindClose(Search);
  end;
  RmDir(Path);
end;
```

Смотрите также

Процедуры ChDir, GetDir, MkDir, функция IOResult.

Round, функция

Синтаксис

```
function Round(X: Вещественный тип): Int64;
```

Описание

Функция Round округляет вещественное значение до целого. Round не является настоящей функцией.

Советы и приемы

- Функция Round использует режим округления, определяемый управляющим состоянием FPU. По умолчанию Delphi устанавливает режим «ближе к четному», также называющийся «банковским округлением». Числа округляются к ближайшему целому, а значения, находящиеся точно между двумя целыми, округляются до четного числа.
- При изменении управляющего состояния FPU изменяется поведение Round и других функций, зависящих от Round. Для того чтобы использовать другой метод округления, измените управляющее состояние FPU временно, как показано ниже.

```
// Округление по направлению плюс бесконечности.
function RoundUp(X: Extended): Int64;
const
  RoundUpCW: Word = $1B32; // Аналогично Default8087CW, кроме режима
                           // округления
var
  OldCW: Word;
begin
  OldCW := Default8087CW;
  try
    Set8087CW(RoundUpCW);
    Result := Round(X);
  finally
    Set8087CW(OldCW);
  end;
end;
```

- Если X — целое, компилятор исключает вызов функции Round и просто возвращает X.
- Если X — Variant, Delphi автоматически преобразует его в вещественное число и выполняет округление.
- Компилятор не принимает аргумент Int64 для функции Round, однако использовать ее для Int64 все равно нет смысла.
- Если X равен бесконечности или NaN, Round выдает ошибку времени выполнения 6 (EInvalidOp).

Смотрите также

Функции Int, Trunc.

RunError, процедура

Синтаксис

```
procedure RunError(ErrorCode: Integer = 0);
```

Описание

Процедура RunError прерывает выполнение программы с определенным кодом ошибки. RunError не является настоящей процедурой.

Советы и приемы

- Когда возникает ошибка времени выполнения и ErrorProc равен nil, Delphi вызывает RunError для вывода сообщения об ошибке и остановки программы. В графическом приложении Delphi выдает сообщение об ошибке в диалоговом окне. В консольном приложении Delphi печатает сообщение об ошибке на консоль.
- Создавая собственную процедуру ErrorProc, можно использовать RunError для остановки программы и выдачи короткого сообщения об ошибке. Но процедура ErrorProc может выполнять и какое-нибудь другое действие, например, генерировать исключительную ситуацию или печатать другое сообщение об ошибке.
- Процедура ExitProc и разделы завершения модулей выполняются до завершения программы.
- Как и Halt, RunError это быстрый способ завершения программы, но он не является правильным для графического приложения, для завершения которого следует закрыть его главную форму.

Пример

```
// Выдача ошибки времени выполнения. Адрес вызова RunError
// становится адресом ошибки. Адрес, передаваемый в ErrorProc, -
// это адрес вызова ErrorProc, который близок, но не совсем
// совпадает с адресом вызова RunError. Можно изменить
// значение Caller, но этот метод сильно зависит от генератора
// кода.

procedure ReportError(ErrorCode: Integer);
  function Caller: Pointer;
  asm
    MOV EAX, [ESP]
  end;
type
  TErrorProc = procedure(Code: Integer; Addr: Pointer);
begin
  if ErrorProc <> nil then
    TErrorProc(ErrorProc)(ErrorCode, Caller);
  RunError(ErrorCode);
end;
```

Смотрите также

Переменные ErrorAddr, ErrorProc, ExitCode, ExitProc, NoErrMsg, ключевое слово Finalization, процедуры Halt, SetInOutRes.

SafeCall, директива

Синтаксис

Объявление подпрограммы; safecall;

Описание

Директива safecall уведомляет компилятор о необходимости использования безопасного метода передачи параметров функции или процедуре.

Как и для stdcall, вызывающая сторона помещает аргументы в стек, начиная с самого правого аргумента. Перед возвратом из подпрограммы аргументы выталкиваются из стека.

Если подпрограмма является функцией, Delphi генерирует дополнительный аргумент для сохранения возвращаемого значения. Функции и процедуры неявно преобразуются в функции, возвращающие значение HResult. Если подпрограмма является методом, Self и скрытый параметр (возвращаемое значение) оказываются последними и потому помещаются в стек в первую очередь. Компилятор автоматически помещает тело подпрограммы в обработчик исключительной ситуации, таким образом, Delphi перехватывает все исключительные ситуации, и safecall-метод никогда не генерирует исключительную ситуацию, видимую вызывающей стороне. Если подпрограмма завершается нормально, Delphi возвращает 0 в виде скрытого возвращаемого значения HResult. Если safecall-подпрограмма является методом и вызывает исключительную ситуацию, Delphi вызывает метод SafeCallException для преобразования типа исключительной ситуации в значение HResult. Если safecall-подпрограмма – это обычная функция или процедура, Delphi преобразует все исключительные ситуации в E_Unexpected. Схематично вызов safecall выглядит следующим образом:

```
type
  TSomething = class
    function Divide(I, J: Integer): Integer; safecall;
  end;

// Если Divide написан следующим образом:
function TSomething.Divide(I, J: Integer): Integer;
begin
  Result := I div J;
end;

// Delphi формирует примерно следующий код:
function TSomething.Divide(I, J: Integer; var Rtn: Integer): HResult;
```

```
begin
  try
    Rtn := I div J;
    Result := S_OK;
  except
    Result := Self.SafeCallException(ExceptObject, ExceptAddr);
  end;
end;
```

Компилятор генерирует вызов подпрограммы в соответствии с соглашением о вызовах stdcall, а также проверяет значение HResult, возвращаемое функцией. Если это значение сигнализирует об ошибке, Delphi вызывает процедуру из SafeCallErrorProc. Другими словами, вызов подпрограммы safecall выглядит следующим образом:

```
// Если написать следующий код:
X := Something.Divide(10, 2);

// Delphi фактически вызывает метод следующим образом:
ErrorCode := Something.Divide(X, 10, 2));
if Failed(ErrorCode) then
begin
  if SafeCallErrorProc <> nil then
    SafeCallErrorProc(ErrorCode, EIP); // EIP=указатель на текущую
                                       // инструкцию
  ReportError(24);
end;
```

Советы и приемы

- Следует использовать safecall для двойных диспетчерских интерфейсов.
- Delphi заботится об осуществлении безопасного вызова, поэтому программисту обычно не приходится заниматься деталями. Создавайте safecall-метод так же, как обычный метод.

Смотрите также

Директивы CDecl, Pascal, Register, StdCall, ключевые слова Function, Interface, Procedure, переменная SafeCallErrorProc.

SafeCallErrorProc, переменная

Синтаксис

```
var SafeCallErrorProc: Pointer;

procedure SafeCallError(ErrorCode: Integer; ErrorAddr: Pointer);
begin ... end;
SafeCallErrorProc := @SafeCallError;
```

Описание

Delphi вызывает процедуру, на которую указывает переменная SafeCallErrorProc, когда процедура с директивой safecall возвращает значение HResult, сигнализирующее об ошибке. Если SafeCallErrorProc равна nil, или данная процедура не вызвала исключительной ситуации и завершилась нормально, Delphi выдает ошибку времени выполнения 24 (ESafeCallException).

Советы и приемы

- Модуль ComObj устанавливает значение этой переменной на процедуру, которая пытается получить информацию через интерфейс IErrorInfo и вызвать исключительную ситуацию EOleException с использованием данных ErrorCode и IErrorInfo.
- При реализации COM-сервера в Delphi все исключительные ситуации Delphi, возникающие в safecall-методе, перехватываются методом SafeCallException и преобразуются в HResult. Процедура SafeCallErrorProc вызывается только для ошибок времени выполнения и исключительных ситуаций не связанных с Delphi.

Смотрите также

Ключевые слова Function, Interface, Procedure, директива SafeCall, тип TObject.

Seek, процедура

Синтаксис

```
procedure Seek(var F: File; RecordNumber: LongInt);
```

Описание

Процедура Seek перемещает позицию файла на запись номер RecordNumber. Seek не является настоящей процедурой.

Советы и приемы

- Для текстового файла процедура Seek неприменима. Если требуется изменить позицию в текстовом файле, используйте Windows API:

```
// Перемещение на позицию в текстовом файле, обозначенную байтовым
// значением. В описании процедуры SetInOutRes смотрите ту же
// процедуру с обработкой ошибок. Предполагаем, что буфер файла
// пуст.
procedure TextSeek(var F: TextFile; Pos: LongInt);
begin
  SetFilePointer(TTextRec(F).Handle, Pos, nil, File_Begin);
end;
```

- Начало файла имеет номер записи 0.

- Если F – нетипизированный файл, размер записи устанавливается при открытии файла с помощью процедур Reset или Rewrite.

Пример

```
// Переход на начало файла.
Seek(F, 0);

// Переход в конец файла.
Seek(F, FileSize(F));
```

Смотрите также

Ключевое слово File, функции FilePos, IOResult.

SeekEof, функция

```
function SeekEof(var F: TextFile = Input): Boolean
```

Описание

Функция SeekEof пропускает пустые символы и возвращает True, если достигнут конец текстового файла, и False, если это не так. SeekEof не является настоящей функцией.

Ошибки

- Если F не был открыт, SeekEof возвращает ошибку времени выполнения 103.
- Файл должен быть открыт процедурой Reset. Если файл F был открыт с помощью процедур Rewrite или Append, SeekEof генерирует ошибку времени выполнения 104.
- SeekEof работает только для текстовых файлов.

Пример

```
var
  CmdLine: string;
begin
  while not SeekEof do
  begin
    ReadLn(CmdLine);
    DoCommand(CmdLine);
  end;
end;
```

Смотрите также

Функции Eof, IOResult, процедуры Reset, SeekEoln.

SeekEoln, функция

Синтаксис

```
function SeekEoln(var F: Text = Input): Boolean
```

Описание

Функция SeekEoln пропускает пустые символы и возвращает True, если достигнут конец файла или конец строки. SeekEoln интерпретирует перевод каретки (#13) как конец строки. False возвращается в том случае, если текста не осталось или следующий символ – не перевод каретки. SeekEoln не является настоящей функцией.

Ошибки

- Если F не был открыт, SeekEoln возвращает ошибку времени выполнения 103.
- Файл должен быть открыт процедурой Reset. Если файл F был открыт с помощью процедур Rewrite или Append, SeekEoln генерирует ошибку времени выполнения 104.

Пример

```
// Читаем последовательность чисел из одной текстовой строки.
// Возвращает сумму этих чисел.
function SumLine: Double;
var
  X: Double;
begin
  Write('Number list: ');
  Result := 0.0;
  while not SeekEoln do
  begin
    Read(X);
    Result := Result + X;
  end;
end;
```

Смотрите также

Функции Eoln, IOResult, процедуры Reset, SeekEof, тип TextFile.

Self, переменная

Синтаксис

```
var Self: Тип класса;
```

Описание

В каждом методе Delphi объявляет переменную Self в качестве скрытого параметра. В методе объекта значение переменной равно ссылке на вызвавший объект. В методе класса Self – ссылке на класс.

Советы и приемы

- Конструктор обычно присваивает Self новое значение, которое затем возвращается конструктором. Однако вместо присваивания

другого значения в конструкторе, как правило, переопределяется метод NewInstance.

- В обычных методах присваивание значения Self не отражается за пределами метода.
- Метод также содержит неявную конструкцию with Self do для тела метода. Другими словами, все поля, методы и свойства находятся в своей области действия, и допускается обращение к ним без явной ссылки на Self.
- При использовании соглашения о вызовах register, Self передается как первый аргумент, т. е. через регистр EAX.
- Если установлены соглашения о вызовах pascal, Self является последним аргументом и потому помещается в стек последним, после остальных аргументов.
- Если используются соглашения о вызовах cdecl, safecall и stdcall, Self передается как последний аргумент процедуры и потому помещается в стек первым. Если функция должна вернуть строку, динамический массив, Variant или большую запись, указатель на результат помещается в стек после Self, как скрытый параметр var.

Пример

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Button: TButton;
begin
  Button := TButton.Create(Self);
  Button.Parent := Self;
  // Центрирование кнопки на форме. Заметьте, что Self.ClientWidth
  // и просто ClientWidth - это одно и то же.
  Button.Left := (Self.ClientWidth - Button.Width) div 2;
  Button.Top  := (ClientHeight - Button.Height) div 2;
  Button.Caption := 'Click me!';
  Button.OnClick := ButtonClick;   // or use Self.ButtonClick
end;
```

Смотрите также

Директивы CDecl, Pascal, Register, Safecall, StdCall, ключевое слово Class, тип TObject.

Set, ключевое слово

Синтаксис

```
type Имя = set of Порядковый тип;
```

Описание

Множественный тип определяется в терминах базового типа, который должен быть порядковым: перечислением, целым или символом. Del-

phi реализует множества с помощью битовой маски, где каждому элементу базового типа ставится в соответствие соответствующий бит множественного типа. Множественные типы ограничены 256 элементами, а порядковые значения базового типа соответственно ограничены диапазоном 0..255.

Битовая маска множества хранится как массив байт. Порядковое значение элемента множества определяет, какой бит какого байта занят этим элементом: порядковое значение 0 соответствует самому младшему биту первого байта, а 255 – самому старшему биту 32-го байта. Данные множества не содержат пустых байт, а если в байте есть хотя бы один элемент, то в множестве присутствует весь байт. Внутреннее представление множества обеспечивает постоянную битовую позицию для каждой порядковой величины, независимо от фактического типа множества.

Например, `set of 5..10` занимает 2 байта. Первый байт содержит элементы 5..7 в битах с 5 по 7, следующий – элементы 8..10 в битах с 0 по 2.

Советы и приемы

- Множества часто оказываются проще в применении, чем битовые маски.
- Программисты, перешедшие к Delphi из других языков программирования, и незнакомые с операторами Паскаля для множеств, могут ознакомиться с ними в главе 7 «Операторы».
- Существует возможность преобразования небольшого множества в целую битовую маску. При этом используйте целый тип с таким же размером, как размер множества. Например, для преобразования стиля шрифта, применяйте `Byte(Font.Style)`. Для множеств до 16 элементов подходит `Word`, до 32 – `LongWord`.

Пример

```
// Функция GetFileAttributes из Windows API возвращает битовую
// маску, но использование битовой маски выглядит неуклюже по
// сравнению с множеством. Определим функцию, возращающую
// те же атрибуты файлов в виде множества.
type
  TWindowsFileAttribute =
    (wfaReadonly, wfaHidden, wfaSystem, wfaReserved,  wfaDirectory,
     wfaArchive, wfaEncrypted, wfaNormal, wfaTemporary, wfaSparse,
     wfaReparsePoint, wfaCompressed, wfaOffline);
  TWindowsFileAttributes = set of TWindowsFileAttribute;

function FileAttributes(const FileName: string): TWindowsFileAttributes;
var
  Attributes: DWORD;
begin
  Attributes := GetFileAttributes(PChar(FileName));
```

```
  if Attributes = $FFFFFFFF then
    Result := []
  else
    Result := TWindowsFileAttributes(Word(Attributes));
end;
...
if wfaDirectory in FileAttributes(Path) then
  RmDir(Path);
...
if ([wfaHidden,wfaSystem] * FileAttributes(Path)) <> [] then
  ShowMessage(Path + ' - системный файл.');
```

Смотрите также

Ключевые слова `In`, `Type`.

Set8087CW, процедура

Синтаксис

```
procedure Set8087CW(ControlWord: Word);
```

Описание

Процедура `Set8087CW` устанавливает управляющее состояние FPU и сохраняет значение параметра `ControlWord` в переменной `Default8087CW`. `Set8087CW` – настоящая процедура.

Советы и приемы

- Дополнительную информацию об управляющем состоянии FPU можно получить в документации по архитектуре процессоров Intel.
- Чаще всего `Set8087CW` служит для изменения точности операций с плавающей точкой, маски исключительных ситуаций и режима округления. Применяя типы `Comp` и `Currency`, убедитесь, что не уменьшили точность операций с плавающей точкой.

Пример

```
type
  TRoundMode = (rmNearest, rmDown, rmUp, rmZero);
  TPrecisionMode = (pmSingle, pmReserved, pmDouble, pmExtended);
  TExceptionMask = (emInvalid, emDenormalized, emZeroDivide,
                    emOverflow, emUnderflow, emPrecision);
  TExceptionMasks = set of TExceptionMask;

  TFpuControl = record
    RoundMode: TRoundMode;
    Precision: TPrecisionMode;
    ExceptionMask: TExceptionMasks;
  end;
const
  RoundShift = 10;
  PrecisionShift = 8;
```

```
// Устанавливаем управляющее состояние FPU структурированным
// образом.
procedure SetFpuCW(const FpuCW: TFpuControl);
var
  CW: Word;
begin
  CW := Byte(FpuCW.ExceptionMask);
  CW := CW or (Ord(FpuCW.Precision) shl PrecisionShift);
  CW := CW or (Ord(FpuCW.RoundMode) shl RoundShift);
  Set8087CW(CW);
end;
```

Смотрите также

Типы Comp, Currency, Extended, переменная Default8087CW, функции Int, Round, Trunc.

SetInOutRes, процедура

Синтаксис

```
procedure SetInOutRes(Code: Integer);
```

Описание

Процедура SetInOutRes устанавливает код ошибки ввода/вывода равным Code. Вызов IOResult возвращает этот код ошибки и сбрасывает его в ноль.

Советы и приемы

- В каждом потоке поддерживается собственный код ошибок ввода/вывода. Вызывайте SetInOutRes в контексте требуемого потока.
- Delphi резервирует коды 100–106 для собственных ошибок ввода/вывода (список этих кодов приводится в описании функции IOResult). Для всех остальных ошибок используйте коды ошибок Windows.
- Вызов SetInOutRes не генерирует ошибки ввода/вывода, а лишь устанавливает код ошибки. Сообщать об ошибке ввода/вывода нужно отдельно. Соответствующие примеры рассмотрены в описаниях процедур ErrorProc и RunError.

Пример

```
// Реализация изменения позиции в текстовом файле.
procedure TextSeek(var F: TextFile; Position: Integer);
var
  Handle: Integer;
  NewPosition: DWORD;
begin
  // TextFile - это фактически запись, первое значение в которой -
  // дескриптор файла.
```

```
    Handle := PInteger(@F)^;
    NewPosition := SetFilePointer(Handle, Position, nil, File_Begin);
    if NewPosition = $FFFFFFFF then
    begin
      SetInOutRes(GetLastError);
      ReportError(0); // 0 - значит, произошла ошибка ввода/вывода
    end;
  end;
```

Смотрите также

Переменная ErrorProc, ключевое слово File, функция IOResult, процедура RunError, тип TextFile.

SetLength, процедура

Синтаксис

```
procedure SetLength(var S: string; Length: Integer);
procedure SetLength(var A: Тип массива; Length: Integer);
procedure SetLength(var A: Тип массива; Len1: Integer; Len2...);
```

Описание

Процедура SetLength изменяет размер строки или динамического массива. Если новая длина больше текущей длины, дополнительная область памяти не инициализируется, если только массив не состоит из строк, интерфейсов, других динамических массивов или Variant, для которых инициализация выполняется.

SetLength не является настоящей процедурой.

Советы и приемы

- Длина указывается в логических единицах (элементах массива, символах). В частности, если S является WideString, длина определяется в символах, а не в байтах.
- Если на строку или массив имеется несколько ссылок, SetLength всегда создает новый экземпляр данных и устанавливает его длину, даже если длина совпадает с текущей.
- Если A – многомерный динамический массив, то в процедуру SetLength можно передать одну или более значений длины. Первое число относится к самому левому измерению, последующие – устанавливают следующие измерения. По умолчанию для всех измерений, для которых не указано значение длины в SetLength, принимается 0.

Пример

```
// Возвращает строку длины Length, заполненную повторяющимся
// символом Fill.
function FillString(Length: Integer; Fill: Char): string;
begin
```

```
  SetLength(Result, Length);
  FillChar(Result[1], Length, Char);
end;

// Создает квадратную единичную матрицу размером N.
type
  TIntMatrix = array of array of integer;
procedure SetIdentityMatrix(var M: TIntMatrix; N: Integer);
var
  I, J: Integer;
begin
  SetLength(M, N, N);
  for I := 0 to N-1 do
  begin
    for J := 0 to N-1 do
      M[I, J] := 0;
    M[I, I] := 1;
  end;
end;
```

Смотрите также

Типы `AnsiString`, `ShortString`, `WideString`, ключевые слова `Array`, `String`, `Type`, процедуры `SetString`, `UniqueString`.

SetMemoryManager, процедура

Синтаксис

```
procedure SetMemoryManager(const NewMgr: TMemoryManager);
```

Описание

Процедура `SetMemoryManager` устанавливает новый менеджер памяти для обработки запросов на выделение памяти от процедур `GetMem`, `New`, операций создания объектов и других данных, включая длинные строки, динамические массивы и значения типа `Variant`.

`SetMemoryManager` — настоящая процедура.

Советы и приемы

- При установке нового менеджера памяти модуль, вызывающий `SetMemoryManager`, должен быть первым модулем проекта. Если это не так, первый модуль наверняка выделит память через встроенный менеджер памяти Delphi, и новому менеджеру после установки придется освобождать память, которую он не выделял.
- По той же причине не следует сохранять и восстанавливать исходный менеджер памяти. Delphi выделяет память в непредсказуемые моменты, и нельзя полагаться на порядок инициализации и завершения модулей, чтобы обеспечить освобождение памяти тем же менеджером, которым она выделялась. Единственный безопасный

подход – исключительное использование только одного менеджера памяти.

- Используя DLL, будьте осторожны с памятью, которую она выделяет и которая не освобождается при выгрузке библиотеки. Это легко может случиться, если через границы DLL передаются длинные строки, динамические массивы или значения типа Variant. Решение состоит в перенаправлении всех запросов к памяти в отдельную DLL. В Delphi модуль ShareMem использует для этого *borlndmm.dll*, и программист может подключить ту же библиотеку со своим менеджером памяти. Дополнительную информацию и примеры можно найти в главе 2.

Пример

```
// Отладочный менеджер памяти выделяет защитное слово вокруг
// каждого выделяемого участка памяти. Если защитное слово затирается,
// у вас возникли проблемы. Выделяемые и свободные блоки используют
// разные значения защитных слов для проверки на двойное освобождение.
// Когда блок освобождается, его содержимое переписывается с помощью
// FreeFill, что помогает отслеживать ошибки при обращении к
// освобожденному блоку.
const
  AllocatedGuard = $0AC0ACDC;
  FreeGuard      = $0FC0FCEE;
  FreeFill       = $42FE;
  GuardSize      = SizeOf(LongInt);
  MaxArraySize   = MaxInt div 4 - 1;
type
  TIntegerArray = array[0..MaxArraySize] of LongInt;
  PIntegerArray = ^TIntegerArray;

// Округление Size до значения, кратного размеру защитного
// слова
function RoundUpSize(Size: Integer): Integer;
begin
  Result := (Size + GuardSize - 1) div GuardSize * GuardSize
end;

procedure SetDebugManager;
var
  Mgr: TMemoryManager;
begin
  Mgr.GetMem := DebugGet;
  Mgr.FreeMem := DebugFree;
  Mgr.ReallocMem := DebugRealloc;
  SetMemoryManager(Mgr);
end;
```

Смотрите также

Функции FreeMemory, GetMemory, IsMemoryManagerSet, ReallocMemory, SysFreeMem, SysGetMem, SysReallocMem, процедура GetMemoryManager, тип TMemoryManager.

SetRaiseList, функция

Синтаксис

```
procedure SetRaiseList(RaiseList: Pointer);
```

Описание

Функция SetRaiseList устанавливает указатель на список исключительных ситуаций, возвращаемых RaiseList. Формат данных приведен в описании функции RaiseList.

SetRaiseList – настоящая процедура.

Советы и приемы

- В каждом потоке поддерживается собственный список исключительных ситуаций.
- Причина для вызова этой функции возникает редко. Delphi автоматически создает список при возникновении исключительной ситуации.

Смотрите также

Ключевое слово Raise, функция RaiseList.

SetString, процедура

Синтаксис

```
procedure SetString(var Str: string; Buffer: PChar; Length: Integer);
```

Описание

Процедура SetString устанавливает длину Str равной Length, затем копирует Length символов из Buffer в строку. SetString не является настоящей процедурой.

Советы и приемы

- Если Str – длинная строка, SetString всегда создает новую строку, таким образом, Str сохраняет свое уникальное значение.
- Str может иметь тип ShortString, в этом случае Length должна быть меньше 256.
- Buffer может быть равно nil, в этом случае длина строки меняется, а содержимое остается неинициализированным. В случае ShortString предыдущее содержимое строки остается нетронутым.

Пример

```
// Получаем дескриптор экземпляра или исполняемого модуля и
// возвращаем соответствующее имя файла. Заметьте, что
// Delphi автоматически преобразует PChar в строку, однако
// она должна просканировать PChar для определения длины
// строки. GetModuleFileName возвращает длину строки,
// поэтому необходимости сканировать строку нет.
function GetModuleName(Instance: THandle): string;
var
  Len: Integer;
  Buffer: array[0..MAX_PATH] of Char;
begin
  Len := GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  SetString(Result, Buffer, Len);
end;
```

Смотрите также

Типы AnsiString, ShortString, WideString, процедура SetLength, ключевое слово String.

SetTextBuf, процедура

Синтаксис

```
procedure SetTextBuf(var F: TextFile; var Buffer);
procedure SetTextBuf(var F: TextFile; var Buffer; Size: Integer);
```

Описание

Процедура SetTextBuf устанавливает новый текстовый буфер для последующих операций ввода/вывода. Buffer – это новый файловый буфер, Size – его размер в байтах. Если Size отсутствует, SetTextBuf использует SizeOf(Buffer).

Двоичные файлы не имеют буферов, поэтому SetTextBuf работает только для типа TextFile.

Можно установить буфер для любого открытого текстового файла. Не следует менять буфер, если уже выполнялись операции ввода/вывода, иначе можно потерять содержимое старого буфера. Если буфер выделен динамически, не освобождайте его до закрытия файла.

SetTextBuf не является настоящей процедурой.

Советы и приемы

Каждый текстовый файл имеет небольшой встроенный буфер. Для консольного ввода/вывода и других обычных применений достаточно небольшого буфера. Если вы обнаружите, что файловый ввод/вывод становится узким местом вашего приложения, попробуйте установить буфер большего размера.

Пример

```
// Читает список чисел из файла и возвращает их сумму.
function SumFile(const FileName: string): Double;
var
  Number: Double;
  F: TextFile;
  Buffer: array[0..8191] of Char;
begin
  Result := 0.0;
  AssignFile(F, FileName);
  Reset(F);
  try
    SetTextBuf(F, Buffer);
    while not SeekEof(F) do
    begin
      ReadLn(F, Number);
      Result := Result + Number;
    end;
  finally
    CloseFile(F);
  end;
end;
```

Смотрите также

Функция `IOResult`, процедуры `Reset`, `Rewrite`, тип `TextFile`.

Shl, ключевое слово

Синтаксис

Значение `shl` *Биты*

Описание

Оператор `shl` выполняет сдвиг влево целого *Значения* на *Биты*. Освобождаемые биты заполняются справа нулями.

Операнд *Биты* интерпретируется как целое без знака, по модулю размера *Значения*. То есть, если *Значение* – `LongInt`, операнд *Биты* интерпретируется по модулю 32, если *Значение* – `Int64`, то интерпретируется по модулю 64. Другими словами, величина сдвига определяется только несколькими младшими битами из операнда *Биты*.

Советы и приемы

- Помните, что размер типов `Integer` и `Cardinal` может измениться в новых версиях Delphi. Если необходимо определенное количество битов, используйте типы фиксированного размера, такие как 32-битовые `LongInt` и `LongWord`.
- Не применяйте оператор `shl` вместо умножения на степень двойки. Если подразумевается умножение, так и следует писать. Оператор

shl не выполняет проверку на переполнение. Дайте возможность компилятору Delphi самому выбрать наилучшую инструкцию.

Пример

```
// Определение размера целого, который может изменяться от
// версии к версии.
function BitsPerInteger: Integer;
var
  Test: Integer;
begin
  Test := 1;
  Result := 0;
  while Test <> 0 do
  begin
    Test := Test shl 1;
    Inc(Result);
  end;
end;
```

Смотрите также

Ключевые слова And, Not, Or, Shr, Xor.

ShortInt, тип

Синтаксис

```
type ShortInt = -128..127;
```

Описание

Тип ShortInt определяет 8-битовое целое со знаком.

Советы и приемы

- C, C++ и Java-программисты должны иметь в виду, что ShortInt не эквивалентен типу short в этих языках. Тип short ближе к типу SmallInt в Delphi.
- Дополнительную информацию о целых типах можно увидеть в описании типа Integer.

Смотрите также

Типы Byte, Integer.

ShortString, тип

Синтаксис

```
type ShortString;
type Имя = string[Константа];
```

Описание

Тип ShortString – это строка с максимальным размером в 255 символов. Можно объявить короткую строку с помощью типа ShortString или ключевого слова string с максимальной длиной строки. Максимальная длина должна быть представлена константным выражением в диапазоне от 0 до 255. Если применяются директивы компилятора $H или $LongStrings, можно просто употребить ключевое слово string для обозначения ShortString вместо AnsiString, принятого по умолчанию.

Delphi хранит короткую строку в виде массива символов с зарезервированным байтом для длины строки. При определении адреса короткой строки Delphi возвращает указатель на байт длины.

Советы и приемы

- Как и у всех строковых типов в Delphi, индексы символов строки начинаются с 1, но можно также ссылаться на нулевой элемент короткой строки, возвращающий длину строки в виде Char. Однако вместо ссылки на элемент 0 следует применять функцию Length и процедуру SetLength для получения и изменения длины строки.
- Большинство строк в новых Delphi-программах используют более удобный тип AnsiString, однако короткие строки полезны для чтения и записи данных. Пример приводится в описании ключевого слова File.

Смотрите также

Типы AnsiString, WideString, ключевые слова File, String, функция Length, процедуры SetLength, SetString, директивы компилятора $H, $LongStrings.

Shr, ключевое слово

Синтаксис

Значение shr *Биты*

Описание

Оператор shr выполняет сдвиг вправо целого *Значения* на *Биты*. Освобождаемые биты заполняются слева нулями.

Операнд *Биты* интерпретируется как целое без знака, по модулю размера *Значения*. То есть, если *Значение* – LongInt, операнд *Биты* интерпретируется по модулю 32, если *Значение* – Int64, то интерпретируются по модулю 64. Другими словами, величина сдвига определяется только несколькими младшими битами из *Биты*.

Советы и приемы

- Помните, что размер типов Integer и Cardinal может измениться в новых версиях Delphi. Если требуется определенное количество би-

тов, употребляйте типы фиксированного размера, такие как 32-разрядные LongInt и LongWord.

- Не используйте оператор shr вместо деления на степень двойки. Если подразумевается деление, так и следует писать. Оператор shr не выполняет проверок на переполнение и не учитывает знаковый бит. Дайте возможность компилятору Delphi самому выбрать наилучшую инструкцию.

Пример

```
// Определение размера целого, который может изменяться от
// версии к версии.
function BitsPerInteger: Integer;
var
  Test: Integer;
begin
  Test := -1;
  Result := 0;
  while Test <> 0 do
  begin
    Test := Test shr 1;
    Inc(Result);
  end;
end;
```

Смотрите также

Ключевые слова And, Not, Or, Shl, Xor.

Sin, функция

Синтаксис

```
function Sin(Number: Вещественный тип): Extended;
```

Описание

Функция Sin вычисляет и возвращает синус Number, являющийся значением угла в радианах. Функция Sin – встроенная.

Советы и приемы

- Delphi автоматически преобразует аргументы типа Integer и Variant в вещественное значение.
- Если Number равно «сигнализирующему» NaN, плюс бесконечности или минус бесконечности, Delphi генерирует ошибку времени выполнения 6 (EInvalidOp).
- Если Number равно «спокойному» NaN, результат равен Number.

Смотрите также

Функции ArcTan, Cos.

Single, тип

Синтаксис

```
type Single;
```

Описание

Тип `Single` — вещественный тип стандарта IEEE, использующий 4 байта для хранения знакового бита, 8-битовой экспоненты и 23-битовой мантиссы. Мантисса обычно нормализована, т. е. бит, предшествующий самому старшему значащему биту, содержит 1. Таким образом, числовое значение 0,0 представляется только нулевыми битами. Экспонента, составленная из единиц, представляет бесконечность (если мантисса равна нулю) или нечисловое значение (если мантисса не равна нулю).

Границы типа `Single` примерно равны от $1{,}18\times10^{-38}$ до $3{,}40\times10^{38}$ с точностью около семи десятичных знаков. В табл. 5-3 подробно показан формат конечных и специальных значений `Single`.

Таблица 5-1. Формат вещественных чисел Single

Класс значений	Знак	Биты экспоненты	Биты мантиссы
Положительные значения			
Нормализованные	0	От 0…1 до 1…10	От 0…0 до 1…1
Денормализованные	0	0…0	От 0…1 до 1…1
Ноль	0	0…0	0…0
Бесконечность	0	1…1	0…0
«Сигнализирующий» NaN	0	1…1	От 0…1 до 01…1
«Спокойный» NaN	0	1…1	От 1…0 до 1…1
Отрицательные значения			
Нормализованные	1	От 0…1 до 1…10	От 0…0 до 1…1
Денормализованные	1	0…0	От 0…1 до 1…1
Ноль	1	0…0	0…0
Бесконечность	1	1…1	0…0
«Сигнализирующий» NaN	1	1…1	От 0…1 до 01…1
«Спокойный» NaN	1	1…1	От 1…0 до 1…1

Советы и приемы

- Тип `Single` соответствует типу `float` в Java, C и C++.

- Тип `Single` обычно применяется, когда важнее всего экономия памяти. Он обычно не дает преимущества в производительности по сравнению с `Double` и характеризуется ограниченным диапазоном и меньшей точностью.
- В документации по архитектуре Intel (например, «Pentium Developer's Manual», том 3, «Architecture and Programming Manual») или стандарте IEEE 754 можно найти более подробную информацию о бесконечности и NaN (нечисловое значение). В Delphi использование NaN вызывает ошибку времени выполнения 6 (`EInvalidOp`).

Пример

```
type
  TSingle = packed record
    case Integer of
    0: (Float: Single;);
    1: (Bytes: array[0..3] of Byte;);
    2: (Words: array[0..1] of Word;);
    3: (LongWords: array[0..0] of LongWord;);
  end;
  TFloatClass = (fcPosNorm, fcNegNorm, fcPosDenorm, fcNegDenorm,
            fcPosZero, fcNegZero, fcPosInf, fcNegInf, fcQNaN, fcSNan);

// Возвращает класс вещественного числа: конечное, бесконечное,
// нечисловое; положительное или отрицательное, нормализованное или
// денормализованное. Класс определяется путем проверок
// экспоненты, знакового бита и мантиссы.
function fp_class(X: Single): TFloatClass; overload;
var
  XParts: TSingle absolute X;
  Negative: Boolean;
  Exponent: Byte;
  Mantissa: LongInt;
begin
  Negative := (XParts.LongWords[0] and $80000000) <> 0;
  Exponent := (XParts.LongWords[0] and $7F800000) shr 23;
  Mantissa :=  XParts.LongWords[0] and $007FFFFF;

  // Три первых варианта могут быть положительными или
  // отрицательными. Предположим, значение положительно и
  // проверим знаковый бит позднее.
  if (Mantissa = 0) and (Exponent = 0) then
    // Мантисса и экспонента равны нулю, следовательно, число равно
    // нулю.
    Result := fcPosZero
  else if Exponent = 0 then
    // Если экспонента равна нулю, а мантисса не равна нулю,
    // значит, число конечное, но денормализованное.
    Result := fcPosDenorm
  else if Exponent <> $FF then
    // Иначе, если экспонента не целиком состоит из единиц, число
```

```
      // нормализовано.
      Result := fcPosNorm
    else if Mantissa = 0 then
      // Экспонента состоит из 1, мантисса из 0, следовательно,
      // бесконечность.
      Result := fcPosInf

    else
    begin
      // Экспонента состоит из 1, мантисса - ненулевая, следовательно,
      // значение нечисловое. Проверяем, «спокойный» или
      // «сигнализирующий» NaN.
      if (Mantissa and $400000) <> 0 then
        Result := fcQNaN
      else
        Result := fcSNaN;
      Exit; // Не выделяем отрицательный NaN.
    end;

    if Negative then
      Inc(Result);
  end;
```

Смотрите также

Типы `Double`, `Extended`.

SizeOf, функция

Синтаксис

```
function SizeOf(const Значение или тип): Integer;
```

Описание

Функция `SizeOf` возвращает количество байт, занимаемое *Значением или типом*, которое может быть выражением или идентификатором типа. `SizeOf` не является настоящей функцией.

Советы и приемы

- Чаще всего `SizeOf` используется при вызове `GetMem` для определения объема запрашиваемой памяти.
- Размер указателя, ссылки на объект или ссылки на класс равен четырем, т. к. указатель занимает четыре байта (но в будущих версиях Delphi может требовать больше памяти для хранения указателя).
- Для определения количества байт, занимаемых объектом, вызывается метод объекта `InstanceSize`.

Пример

```
var
```

```
  P: ^Integer;
begin
  // Выделяет память для 32 целых.
  GetMem(P, 32 * SizeOf(Integer)); /
```

Смотрите также

Процедура GetMem, функция Length, тип TObject, ключевое слово Type.

Slice, функция

Синтаксис

```
function Slice(var A: array; Count: Integer): array;
```

Описание

Функция Slice возвращает первые Count элементов массива A в виде аргумента для параметра типа «открытый массив». Функция Slice может применяться только для передачи массива в подпрограмму. Она предоставляет удобный способ работы с динамически выделенным статическим массивом и возможность пользоваться при этом преимуществами открытых массивов Delphi.

Slice не является настоящей функцией.

Пример

```
// Пакет для рисования хранит данные о многоугольнике с числом вершин до MAX_
POINTS.
//Запись TPolygon хранит массив вершин и количество вершин
// многоугольника. DrawPolygon рисует многоугольник на виртуальном
// «холсте» Windows.
type
  TPolygon = record
    NumPoints: 0..MaxInt;
    Points: array[1..MAX_POINTS] of TPoint;
    end;
procedure DrawPolygon(Canvas: TCanvas; const Polygon: TPolygon);
begin
  Canvas.Polygon(Slice(Polygon.Points, Polygon.NumPoints));
end;
```

Смотрите также

Ключевые слова Array, Type, функция Copy.

SmallInt, тип

Синтаксис

```
type SmallInt = -32768..32767
```

Описание

Тип SmallInt – 16-битовое целое со знаком.

Дополнительная информация о целых типах приведена в описании типа Integer.

Смотрите также

Типы Integer, Word.

Sqr, функция

Синтаксис

```
function Sqr(X: Вещественный тип): Extended;
function Sqr(X: Integer): Integer;
function Sqr(X: Int64): Int64;
```

Описание

Функция Sqr возвращает квадрат своего аргумента. Она не является настоящей функцией, а преобразуется компилятором в inline-код.

Советы и приемы

- Если Number – минус или плюс бесконечность, результат равен плюс бесконечности.
- Если Number – минус ноль, результат равен плюс ноль.
- Если вещественный результат получается слишком большим, Delphi генерирует ошибку времени выполнения 8 (EOverFlow).
- Если целый результат слишком большой, Delphi 5 усекает его без проверки на переполнение.
- Если Number равен «сигнализирующему» NaN, Delphi вызывает ошибку времени выполнения 6 (EInvalidOp).
- Если Number равен «спокойному» NaN, результат равен Number.

Пример

```
function Hypotenuse(X, Y: Double): Double;
begin
  Result := Sqrt(Sqr(X) + Sqr(Y));
end;
```

Смотрите также

Типы Extended, Integer, функция Sqrt.

Sqrt, функция

Синтаксис

```
function Sqrt(Number: Вещественный тип): Extended;
```

Описание

Функция Sqrt возвращает квадратный корень своего аргумента. Она не является настоящей функцией, а преобразуется компилятором в inline-код.

Советы и приемы

- Delphi автоматически преобразует аргументы типа Integer и Variant в вещественные значения. Для преобразования Int64 в вещественное число, прибавьте к нему 0,0.
- Если Number равен плюс бесконечности, то и результат равен плюс бесконечности.
- Если Number равен минус ноль, то и результат равен минус ноль.
- Если Number равен отрицательному ненулевому значению, минус бесконечности или «сигнализирующему» NaN, Delphi генерирует ошибку времени выполнения 6 (EInvalidOp).
- Если Number равен «спокойному» NaN, результат равен Number.

Пример

```
function Hypotenuse(X, Y: Double): Double;
begin
  Result := Sqrt(Sqr(X) + Sqr(Y));
end;
```

Смотрите также

Тип Extended, функция Sqr.

StdCall, директива

Синтаксис

```
Объявление подпрограммы; stdcall;
```

Описание

Директива stdcall реализует стандартное соглашение о передаче аргументов, принятое в Windows: аргументы помещаются в стек, начиная с самого правого. Подпрограмма отвечает за выталкивание аргументов из стека.

Функции возвращают порядковые значения, указатели, небольшие записи и множества в регистре EAX, вещественные значения – в стеке FPU. Строки, динамические массивы, Variant, большие записи и множества передаются через скрытый параметр var. Если подпрограмма является методом, переменная Self помещается в стек непосредственно перед параметром var – результатом функции (если он существует).

Советы и приемы

- Используйте директиву stdcall для подпрограмм, которые вы экспортируете из DLL для вызова другими языками программирования.
- Применяйте stdcall для Windows-функций обратного вызова (callback functions).

Пример

```
// Создание списка окон верхнего уровня.
function Callback(Handle: HWND; List: TStrings): LongBool; stdcall;
var
  Caption: array[0..256] of Char;
  Len: Integer;
  Text: string;
begin
  Len := GetWindowText(Handle, Caption, SizeOf(Caption)-1);
  SetString(Text, Caption, Len);
  List.Add(Text);
  Result := True;
end;
...
EnumWindows(@Callback, LParam(ListBox1.Items));
```

Смотрите также

Директивы CDecl, Pascal, Register, SafeCall, ключевые слова Function, Procedure.

Stored, директива

Синтаксис

```
property Имя: Тип Директивы... stored Логическое выражение;
```

Описание

Директиве stored требуется параметр логического типа: метод, возвращающий результат логического типа, ссылка на поле типа Boolean или константное логическое выражение. В RTTI данного свойства записывается смещение поля, ссылка на метод или значение константы, и по этой информации Delphi определяет возможность исключения свойства из *dfm*-файла.

IDE вызывает метод, проверяет значение поля или использует значение логической константы, и, если это значение ложно, свойство не сохраняется в *dfm*-файле. Если значение stored истинно, выполняется действие по умолчанию, т. е. свойство сохраняется, если его значение отлично от значения по умолчанию.

Советы и приемы

- Директива stored часто понимается неправильно. Установка stored в True не обязывает Delphi сохранять значение свойства в *dfm*-файле. True – значение по умолчанию для параметра директивы stored. Наоборот, все, что можно сделать – это избежать запись свойства в *dfm*-файл, установив stored в False.
- Директива stored может быть использована со свойствами любого уровня доступа, однако она имеет значение только для публикуемых свойств.
- Если указывается имя метода, он может быть статическим или виртуальным, но не может быть динамическим методом или обработчиком сообщений.
- Ссылка на поле может быть именем поля, полем записи или элементом массива с константным индексом. Ссылка на поле должна иметь логический тип.

Пример

Класс TFont в Delphi позволяет указать размер шрифта в пунктах или пикселах. Свойство Height – это высота шрифта в пикселах, Size – размер шрифта в пунктах. В *dfm*-файле сохраняется только Height, Size вычисляется из величины Height. Поэтому свойство Size определяется с директивой stored False, как видно из примера:

```
type
  TFont = class(TGraphicsObject)
  ...
  published
    property Charset: TFontCharset read GetCharset write SetCharset;
    property Color: TColor read FColor write SetColor;
    property Height: Integer read GetHeight write SetHeight;
    property Name: TFontName read GetName write SetName;
    property Pitch: TFontPitch read GetPitch write SetPitch
        default fpDefault;
    property Size: Integer read GetSize write SetSize stored False;
    property Style: TFontStyles read GetStyle write SetStyle;
  end;
```

Смотрите также

Директивы Default, NoDefault, Published, ключевое слово Property.

Str, процедура

Синтаксис

```
procedure Str(const Value; var S: string);
procedure Str(const Value:Width; var S: string);
procedure Str(const Value:Width:Precision; var S: string);
```

Описание

Процедура Str форматирует числовую строку аналогично Write, за исключением того, что результат сохраняется в строке S. Value может быть любым числовым или логическим выражением. Строка S может быть длинной строкой, короткой строкой или символьным массивом с нулевым начальным индексом. Str не является настоящей процедурой.

Аргументы *Width* и *Precision* могут быть любыми целыми выражениями. *Width* определяет минимальный размер символьного представления Value, Precision – количество знаков после десятичной точки для вещественного числа.

Str использует столько символов, сколько требуется, поэтому *Width* – это лишь предлагаемая минимальная ширина. Если числу требуется меньше, чем *Width* символов, Str дополняет его пробелами слева. Если *Width* не указывается, Delphi использует 1 в качестве минимального значения для целых чисел и выводит вещественные числа с помощью 26 символов в следующем формате:

```
' -1.1234567890123456 7E+1234'
```

Если Value – вещественное значение, Str сокращает количество десятичных знаков, чтобы поместить значение в строку не более *Width* символов. Однако Str всегда оставляет как минимум одну цифру после десятичной точки. Можно указать значение *Precision*, которое сообщает процедуре Str, сколько знаков после десятичной точки нужно оставить. Если указывается *Precision*, Str использует форму записи числа с фиксированной точкой вместо экспоненциальной формы.

Советы и приемы

- Модуль SysUtils содержит несколько функций, обеспечивающих большую гибкость, чем Str.
- Если ShortString слишком мала для вывода всего числа Value, Str останавливается, когда строка заполнена, и не выдает сообщений об ошибке.

Пример

```
procedure ShowInfo;
var
  S: string;
begin
  Str(List.Count, S);
  ShowMessage('В списке ' + S + ' элементов.');
end;
```

Смотрите также

Тип String, процедуры Val, Write.

String, ключевое слово

Синтаксис

```
type string;
type Имя = string[Константа];
```

Описание

Ключевое слово string обозначает строковый тип, который равен либо AnsiString, либо ShortString, в зависимости от контекста и директивы компилятора $H или $LongStrings. Без указания максимальной длины в квадратных скобках, строка соответствует типу AnsiString (если только не использована директива компилятора $H или $LongStrings). С указанием максимальной длины строка интерпретируется как тип ShortString. Длина строки должна составлять от 0 до 255.

Подробности рассмотрены в описаниях типов AnsiString и ShortString.

Смотрите также

Типы AnsiChar, AnsiString, Char, PChar, PWideChar, ShortString, WideChar, WideString, функции Length, OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, SetLength, SetString, WideCharLenToStrVar, WideCharToStrVar, директивы компилятора $H, $LongStrings.

StringOfChar, функция

Синтаксис

```
function StringOfChar(Ch: Char; Count: Integer): string;
```

Описание

Функция StringOfChar создает и возвращает строку, состоящую из Count вхождений символа Ch. StringOfChar не является настоящей функцией.

Если Count = 0, StringOfChar возвращает пустую строку.

Смотрите также

Тип Char, процедура FillChar, ключевое слово String.

StringToOleStr, функция

Синтаксис

```
function StringToOleStr(const Str: string): PWideChar;
```

Описание

Функция StringToOleStr преобразует строку в «широкую» строку, пригодную для передачи подпрограммам OLE и COM.

StringToOleStr — настоящая функция.

Советы и приемы

«Широкая» строка создается динамически, и необходимо позднее освободить ее с помощью функции Windows API SysFreeString.

Пример

```
procedure Demo;
var
  OleStr: PWideChar;
begin
  OleStr := StringToOleStr(S);
  try
    SomeOLEProcedure(OleStr);
  finally
    SysFreeString(OleStr);
  end;
end;
```

Смотрите также

Функции OleStrToString, StringToWideChar, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

StringToWideChar, функция

Синтаксис

```
function StringToWideChar(const Str: string; Result: PWideChar;
                          Size: Integer): PWideChar;
```

Описание

Функция StringToWideChar преобразует строку Str в «широкую» строку. В Result копируется максимум Size-1 символов из Str, дополняя результирующую строку символом #0.

StringToWideChar — настоящая функция.

Пример

```
procedure Demo;
var
  OleStr: PWideChar;
begin
  GetMem(OleStr, (Length(S)+1) * SizeOf(WideChar));
  try
    StringToWideChar(S, OleStr, Length(S)+1);
    SomeOLEProcedure(OleStr);
  finally
    FreeMem(OleStr);
  end;
end;
```

Смотрите также

Функции OleStrToString, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

Succ, функция

Синтаксис

```
function Succ(const Value): Порядковое значение;
```

Описание

Функция Succ возвращает следующее значение порядковой величины, обычно перечисляемого значения. То есть возвращает перечисляемое значение, порядковая величина которого на единицу больше, чем Value. Succ не является настоящей функцией.

Советы и приемы

- Inc, Succ и операция сложения имеют почти одинаковую производительность, поэтому выбирайте для себя наиболее понятный вариант.
- Вызов Succ(High(SomeType)) генерирует ошибку времени выполнения 4 (ERangeError). Однако без проверки на переполнение Succ возвращает значение с соответствующей порядковой величиной, хотя оно и не является корректным для данного типа.

Пример

```
type TDay =
 (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
var
  Day: TDay;
begin
  Day := Succ(Day);
  ...
```

Смотрите также

Процедуры Dec, Inc, функции High, Low, Pred, директивы компилятора $OverflowChecks, $Q.

Swap, функция

Синтаксис

```
function Swap(Value: Integer): Integer;
```

Описание

Функция Swap переставляет байты самого младшего слова из Value. Старшие 16 бит не затрагиваются. Если Value имеет тип Int64, Swap иг-

норирует старшие 32 бита, оставляет неизменными следующие 16 бит и выполняет перестановку младших 16 бит. Функция возвращает результирующее 32-битовое слово (полностью отбрасывая исходные старшие 32 бита).

Swap не является настоящей функцией, а преобразуется в inline-код.

Советы и приемы

- Эта функция оставлена для обратной совместимости. В текущем воплощении она мало полезна.
- Перестановка байт – распространенная операция при пересылке файлов или данных по сети. Процессоры Intel используют прямой порядок байт для хранения данных (младшие биты хранятся в старших адресах), а некоторые другие процессоры – обратный (старшие биты в старших адресах).

Пример

```
// Перестановка байт 32-битового целого; подходит, например,
// для передачи данных между процессорами Intel и Sun.
function Swap32(Value: LongWord): LongWord;
begin
  Result := Swap(Value shr 16) or (Swap(Value) shl 16);
end;

// Если действительно требуется переставить байты в 32-битовом
// целом, например, для преобразования обратного порядка байт в
// в прямой, используйте не функцию Swap, а инструкцию
// BSWAP.
function FastSwap(Value: LongWord): LongWord; register; overload;
asm
  bswap eax
end;

// Для перестановки значения Int64 аргумент передается в стек,
// а возвращаемое значение - в EAX:EDX.
function FastSwap(Value: Int64): Int64; register; overload;
asm
  mov edx, [esp+8]
  bswap edx
  mov eax, [esp+12]
  bswap eax
end;
```

Смотрите также

Функции Hi, Lo, ключевые слова Shl, Shr.

SysFreeMem, функция

Синтаксис

```
function SysFreeMem(P: Pointer): Integer;
```

Описание

Функция SysFreeMem использует встроенный менеджер памяти Delphi для освобождения памяти, на которую ссылается указатель P. Возвращает ноль в случае успеха и ненулевой код ошибки, если P – некорректный указатель или nil.

SysFreeMem – настоящая функция.

Советы и приемы

- SysFreeMem применяется при разработке собственного менеджера памяти как фильтр для вызова из него подпрограмм встроенного менеджера памяти Delphi.
- Если вы не создаете менеджер памяти, вызывайте для освобождения памяти процедуры Dispose или FreeMem, а не функцию SysFreeMem.
- FreeMem и Dispose сами проверяют значение указателя на nil, вот почему SysFreeMem этого не делает.

Пример

```
// Пояснения по этому менеджеру памяти приведены в описании процедуры
// SetMemoryManager.

// Возвращает True, если память, на которую указывает PArray, является
// корректным блоком из «кучи» и защитные слова не были изменены.
// Возвращает False при любой ошибке.
function GuardsAreOkay(PArray: PIntegerArray; Fill: Boolean): Boolean;
const
  DelphiInUseFlag = 2;
  MemMgrFlags = 7;
var
  Size: LongWord;
begin
  // Сначала определяем размер блока. Размер представляет собой
  // длинное слово перед началом блока. Delphi устанавливает размер
  // и флаги в трех младших битах. Этот формат может измениться
  // в будущих версиях. Для выделенного блока средний бит равен 1.
  PArray := PIntegerArray(PChar(PArray) - SizeOf(Size));
  Size := PArray[0];
  PArray := PIntegerArray(PChar(PArray) + SizeOf(Size));

  if (Size and DelphiInUseFlag) <> DelphiInUseFlag then
  begin
    Result := False;
```

```
    Exit;
  end;

  // Удаляем флаг, выставленый в Delphi, и вычитаем размер
  // заголовка.
  Size := Size and not MemMgrFlags - SizeOf(Size);

  // Проверящем защитные слова.
  if PArray[0] <> AllocatedGuard then
    Result := False

  // Размер включает два защитных слова, поэтому конечное
  // защитное слово расположено по индексу size-1.
  else if PArray[Size div GuardSize - 1] <> AllocatedGuard then
    Result := False

  else
  begin
    // Если мы дошли до этого места, блок правильный.
    Result := True;

    // Если вызывающая сторона требует этого (как DebugFree), весь блок
    // заполняется специальным битовым шаблоном FreeFill, что
    // помогает обнаруживать проблемы, связанные со ссылками
    // на освобожденную память.
    if Fill then
      FillChar(PArray^, Size div 2, FreeFill);

    // Изменение значений защитных слов на FreeGuard, даже если
    // вызывающая сторона не хочет заполнять весь блок.
    PArray[0] := FreeGuard;
    PArray[Size div GuardSize - 1] := FreeGuard;
  end;
end;

// Возвращает ноль в случае успеха и ненулевое значение
// в случае неудачи.
function DebugFree(Mem: Pointer): Integer;
var
  PArray: PIntegerArray;
begin
  // Получение указателя на истинное начало блока памяти.
  PArray := PIntegerArray(PChar(Mem) - GuardSize);

  if not GuardsAreOkay(PArray, True) then
    Result := 1
  else
    Result := SysFreeMem(PArray);
end;
```

Смотрите также

Процедуры Dispose, FreeMem, GetMemoryManager, SetMemoryManager, функции FreeMemory, IsMemoryManagerSet, SysGetMem, SysReallocMem, переменная IsMultiThread, тип TMemoryManager.

SysGetMem, функция

Синтаксис

```
function SysGetMem(Size: Integer): Pointer;
```

Описание

Функция SysGetMem выделяет Size байт с помощью встроенного менеджера памяти Delphi. Функция возвращает указатель на выделенную память или nil в случае ошибки. Инициализация памяти не выполняется.

SysGetMem – настоящая функция.

Советы и приемы

- При создании собственного менеджера памяти функция SysGetMem может применяться для выделения памяти.
- Если вы не разрабатываете собственный менеджер памяти, вызывайте для выделения памяти процедуры New или GetMem, а не функцию SysGetMem.

Пример

```
// Пояснения по этому менеджеру памяти приведены в описании процедуры
// SetMemoryManager.

// Выделяет Size байт и возвращает указатель на новый участок
// памяти. В случае ошибки возвращает nil и генерирует
// исключительную ситуацию. Выделяет дополнительные байты для
// отладочных целей.
function DebugGet(Size: Integer): Pointer;
var
  PArray: PIntegerArray;
begin
  Size := RoundUpSize(Size);

  // Выделяет память с учетом двух защитных слов: в начале и в
  // конце блока.
  PArray := SysGetMem(Size + 2*GuardSize);
  if PArray = nil then
    Result := nil
  else
  begin
    // Запись защитных слов.
    PArray[0] := AllocatedGuard;
    PArray[Size div GuardSize + 1] := AllocatedGuard;
```

```
    // Возвращает указатель на память за первым защитным словом.
    Result := @PArray[1];
  end;
end;
```

Смотрите также

Процедуры GetMem, GetMemoryManager, New, SetMemoryManager, функции GetMemory, IsMemoryManagerSet, SysFreeMem, SysReallocMem, переменная IsMultiThread, тип TMemoryManager.

SysReallocMem, функция

Синтаксис

```
function SysReallocMem(P: Pointer; NewSize: Integer): Pointer;
```

Описание

Функция SysReallocMem вызывает встроенный менеджер памяти Delphi для повторного выделения блока памяти с новым размером. Новый блок может быть расположен по тому же адресу, что и старый, либо может быть скопирован по новому адресу. В любом случае SysReallocMem возвращает указатель на начало блока.

Если новый размер больше старого, новая область памяти не инициализируется. Если запрос на выделение памяти не может быть выполнен, SysReallocMem сохраняет старый блок в неизменном состоянии и возвращает nil. SysReallocMem — настоящая функция.

Пример

```
// Пояснения по этому менеджеру памяти приведены в описании процедуры
// SetMemoryManager.
// Функция GuardsAreOkay рассмотрена в описании функции SysFreeMem.
// Функция RoundUpSize рассмотрена в описании функции SetMemoryManager.


// Повторное выделение блока с новым размером. Возвращает указатель
// на новый блок памяти или nil в случае неудачи.
function DebugRealloc(Mem: Pointer; Size: Integer): Pointer;
var
  PArray: PIntegerArray;
begin
  // Получение указателя на истинное начало блока памяти.
  PArray := PIntegerArray(PChar(Mem) - GuardSize);

  if not GuardsAreOkay(PArray, False) then
    ReportError(2); // invalid pointer

  // Округление Size до числа, кратного размеру защитного
  // слова.
  Size := RoundUpSize(Size);
```

```
    PArray := SysReallocMem (PArray, Size);
    if PArray = nil then
      Result := nil
    else
    begin
      // Запись защитных слов.
      PArray[0] := AllocatedGuard;
      PArray[Size div GuardSize + 1] := AllocatedGuard;
      // Возвращаем указатель на область памяти, начинающуюся сразу
      // после первого защитного слова.
      Result := @PArray[1];
    end;
  end;
```

Смотрите также

Процедуры GetMemoryManager, ReallocMem, SetMemoryManager, функции IsMemoryManagerSet, ReallocMemory, SysFreeMem, SysGetMem, переменная IsMultiThread, тип TMemoryManager.

TClass, тип

Синтаксис

```
type TClass = class of TObject;
```

Описание

TClass – корневой тип для метаклассов. TClass можно использовать везде, где требуется обобщенная ссылка на класс.

Советы и приемы

- С помощью ссылки на класс можно вызывать любой метод класса.
- Операторы is и as нельзя применять со ссылкой на класс, но для проверки, является ли данный класс потомком другого класса, можно использовать метод класса InheritsFrom.
- В Delphi ссылка на класс представляет собой указатель на VMT класса. Следовательно, размер ссылки на класс равен размеру указателя.

Пример

```
// Преобразование имени класса в ссылку на класс.
type
  TClassMap = class
  private
    fList: TStrings;
    function GetClass(const Name: string): TClass;
    procedure PutClass(const Name: string; Value: TClass);
  public
    procedure Add(ClsRef: TClass);
    property Classes[const Name: string]: TClass
```

```
        read GetClass write SetClass; default;
  end;

procedure TClassMap.Add(ClsRef: TClass);
begin
  // TStrings требуется для связанных данных
  // значение типа TObject, поэтому преобразуем
  // TClass в TObject.
  fList.AddObject(ClsRef.ClassName, TObject(ClsRef));
end;

function TClassMap.GetClass(const Name: string): TClass;
begin
  Result := TClass(fList.Objects[fList.IndexOf(Name)]);
end;

procedure TClassMap.SetClass(const Name: string; Value: TClass);
var
  Index: Integer;
begin
  Index := fList.IndexOf(Name);
  if Index < 0 then
    fList.AddObject(Name, TObject(Value))
  else
    fList.Objects[Index] := TObject(Value);
end;
...
var
  Map: TClassMap;
  Cls: TClass;
...
Map.Add(TObject);
Map.Add(TComponent);
Map.Add(TForm);
Cls := Map['TButton'].ClassParent; // Cls := TButtonControl
```

Смотрите также

Ключевые слова `Class`, `Of`, `Type`, переменная `Self`, тип `TObject`.

TDateTime, тип

Синтаксис

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

Описание

Тип `TDateTime` хранит дату и время в виде количества дней с начала дня 30 декабря 1899 года, что соответствует стандарту, принятому в OLE-автоматизации. Целая часть – количество дней, дробная – время дня.

Советы и приемы

- TDateTime не содержит данных о часовом поясе, поэтому при записи даты и времени в файл или базу данных следует преобразовать локальное время в формат UTC (Universal Coordinated Time, всеобщее скоординированное время). При загрузке даты и времени из файла или базы данных выполните преобразование обратно в локальное время. Благодаря этому работа пользователей в разных часовых поясах не будет сопровождаться проблемами.
- Количество дней между значениями TDateTime можно вычислить с помощью вычитания и последующего преобразования результата в целое значение. В зависимости от обстоятельств используйте функции Round (округление) или Trunc (усечение). Например, расчет количества дней между 1:00 3 марта и 23:00 4 марта может дать один день (функция Trunc) или два дня (функция Round).
- В Unix дата и время представляются в виде количество секунд, прошедших с полуночи 1 января 1970 UTC. Если файл или база данных содержат время в формате Unix, оно может быть преобразовано в TDateTime делением на SecsPerDay и сложением с числом 25 569. Так будут получены дата и время UTC; для использования в большинстве программ Delphi их следует преобразовать в локальные дату и время.

Пример

```
// Преобразование локального времени в UTC путем добавления
// смещения часового пояса. При обратном преобразовании
// в локальное время смещение вычитается.
function LocalToUTC(DateTime: TDateTime): TDateTime;
var
  Info: TTimeZoneInformation;
  Bias: LongInt;
begin
  case GetTimeZoneInformation(Info) of
  Time_Zone_Id_Standard, Time_Zone_Id_Daylight:
  begin
    // Значение, возвращаемое GetTimeZoneInformation, соответствует
    // текущим дате и времени, а не хранящимся в DateTime. Определяем,
    // какое время хранится в DateTime – «зимнее» или «летнее».
    if IsDaylightSavingsTime(DateTime, Info) then
      Bias := Info.Bias + Info.DaylightBias
    else
      Bias := Info.Bias + Info.StandardBias;

    Result := DateTime + Bias / MinutesPerDay;
  end;
  Time_Zone_Id_Unknown:
    Result := DateTime + Info.Bias / MinutesPerDay;
  else
    RaiseLastWin32Error;
```

```
      Result := DateTime; // Выключаем предупреждение Delphi
    end;
  end;
```

Смотрите также

Тип Double, функции Frac, Int, Round, Trunc, ключевое слово Type.

TEnumModuleFuncLW, тип

Синтаксис

```
type TEnumModuleFuncLW =
    function (HInstance: LongWord; Data: Pointer): Boolean;
```

Описание

Функции EnumModules и EnumResouceModules вызывают пользовательскую функцию обратного вызова (callback function) для каждого исполняемого модуля. Сигнатура этой функции должна соответствовать типу TEnumModuleFuncLW.

Советы и приемы

Для обратной совместимости EnumModules и EnumResourceModules имеют перегруженные версии, которые используют callback-функции с первым аргументом LongInt вместо LongWord. В новых программах следует применять LongWord (или DWORD из модуля Windows, который является его аналогом).

Пример

Пример рассмотрен в описании функции EnumModules.

Смотрите также

Процедуры EnumModules, EnumResouceModules.

Test8086, переменная

Синтаксис

```
var Test8086: Byte;
```

Описание

Переменная Test8086 всегда имеет значение 2. Существует только для обратной совместимости.

Test8087, переменная

Синтаксис

```
var Test8087: Byte;
```

Описание

Значение переменной Test8087 всегда равно 3. Существует только для обратной совместимости.

TestFDIV, переменная

Синтаксис

```
var TestFDIV: ShortInt;
```

Описание

Значение переменной TestFDIV сообщает, содержит ли математический сопроцессор (FPU) корректную реализацию инструкции FDIV.

Если указана директива компилятора $U или $SafeDivide, все операции вещественного деления вызывают функцию, которая проверяет FPU на наличие печально известной ошибки FDIV процессора Pentium. Если ошибка присутствует, дальнейшие операции деления должны выполняться программной процедурой, выдающей корректный результат. Если ошибки нет, операции деления будут выполнятся аппаратурой FPU.

Переменная TestFDIV содержит одно из следующих значений:

–1 Процессор работает некорректно и будет выполнятся обход ошибки.

–0 Процессор еще не тестировался.

–1 Процессор протестирован и работает корректно.

Советы и приемы

TestFDIV существует только для обратной совместимости. Windows решает эту проблему на уровне операционной системы.

Смотрите также

Директивы компилятора $SaveDivide, $U.

Text, тип

Синтаксис

```
type Text;
```

Описание

Тип Text аналогичен типу TextFile. В новых программах следует использовать TextFile, чтобы избежать конфликтов с часто применяемым свойством Text.

Смотрите также

Тип TextFile.

TextFile, тип

Синтаксис

```
type TextFile;
```

Описание

Тип TextFile предназначен для работы с текстовыми файлами. Текстовый файл отличается от двоичного тем, что содержит строки текста, размер которых может быть различным. В отличие от стандартного Паскаля, тип TextFile не привязан к File of Char.

Фактический формат конца строки не имеет значения для программы на Delphi. Процедуры ReadLn и WriteLn, функции Eoln и SeekEoln автоматически обрабатывают завершение строк. Текстовый файл заканчивается физическим концом файла или первым вхождением символа #26 <Ctrl>+<Z>.

Советы и приемы

- По умолчанию текстовые файлы буферизуются через 128-байтовый буфер. С помощью процедуры SetTextBuf можно выбрать буфер другого размера.
- Тип TextFile эквивалентен следующей записи. В модуле SysUtils приводится описание этого типа, его можно скопировать в собственный модуль.

```
type
  PTextBuf = ^TTextBuf;
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;
    Mode: Integer;        // fmInput или fmOutput
    BufSize: Cardinal;    // Размер буфера.
    BufPos: Cardinal;     // Текущая позиция в буфере.
    BufEnd: Cardinal;     // Последняя позиция данных, хранящихся
                          // в буфере.
    BufPtr: PChar;        // Указатель на начало данных буфера.
    OpenFunc: Pointer;    // Указатели на функции, которые принимают
    InOutFunc: Pointer;   // параметр var типа TextFile и возвращают
    FlushFunc: Pointer;   // целый результат: ноль в случае успеха и
    CloseFunc: Pointer;   // ненулевой код в случае ошибки.
    UserData: array[1..32] of Byte;  // Может использоваться для данных
                                     //пользователя.
    Name: array[0..259] of Char;     // Имя файла.
    Buffer: TTextBuf;
  end;
const
  // Значения для Mode. Определены в SysUtils.
  fmClosed = $D7B0;
  fmInput  = $D7B1;
```

```
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

- Программист может создать собственный драйвер текстового файла и задать собственные значения для буфера и указателей на функции в записи TTextRec. Напишите собственную функцию назначения текстового файла (эквивалент AssignFile), инициализирующую запись TTextRec. Установите Mode в fmClosed, Name — в имя файла, или пустую строку (если «файл» не ставится в соответствие реальному файлу). Инициализируйте буфер, присвоив значения полям BufSize и BufPtr (обычно BufPtr равно @Buffer). Установите OpenFunc на собственную функцию открытия файла.

  Когда пользователь открывает файл, Delphi устанавливает поле Mode и вызывает функцию OpenFunc. Mode устанавливается по следующим правилам.

Пользователь вызвал...	Параметр Mode установлен в ...
Reset	FmInput
Rewrite	FmOutput
Append	FmInOut

  Если значение Mode равно fmInOut, измените его на fmOutput перед возвратом из собственной функции открытия файла. Установите поле InOutFunc на свою функцию ввода или вывода. Для определения типа файловых операций используйте Mode. Поле CloseFunc указывает на функцию, которая закрывает файл и устанавливает Mode обратно в fmClosed. Функция FlushFunc вызывается после каждой записи в файл. Можно либо сразу сбрасывать все выходные данные на диск, либо ждать заполнения буфера. Перед закрытием выходного файла Delphi автоматически вызывает InOutFunc для записи в файл содержимого последнего буфера.

Пример

```
// Создаем драйвер текстового файла, который ставит TextFile в
// соответствие потоку (Stream). Потоки реализуют гибкий, объектно-
// ориентированный подход к вводу/выводу, широко используемый в
// VCL.

const
  fmClosed = $D7B0;           // Константы режимов скопированы из
  fmInput  = $D7B1;           // SysUtils
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: THandle;
```

```
    Mode: Integer;
    BufSize: Integer;
    BufPos: Integer;
    BufEnd: Integer;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;

    // Пользовательские данные должны помещаться в 32 байта. В первой
    // части пользовательских данных хранится ссылка на поток.
    // Оставшаяся часть не используется.
    Stream: TStream;
    Unused: array[1..32-SizeOf(TStream)] of Byte;

    Name: array[0..259] of Char;
    Buffer: TTextBuf;
  end;

// Чтение буфера из потока.
function Input(var F: TTextRec): Integer;
begin
  F.BufPos := 0;
  F.BufEnd := F.Stream.Read(F.BufPtr^, F.BufSize);
  Result := 0;
end;

// Запись буфера в поток. Если запись не произошла,
// возвращается код ошибки Windows.
function Output(var F: TTextRec): Integer;
begin
  if F.BufPos = F.Stream.Write(F.BufPtr^, F.BufPos) then
    Result := 0
  else
    Result := GetLastError;
  F.BufPos := 0;
end;

// Не сбрасываем содержимое буфера после записи.
function Flush(var F: TTextRec): Integer;
begin
  Result := 0;
end;

// Закрытие файла ничего не делает.
function Close(var F: TTextRec): Integer;
begin
  Result := 0;
end;
```

```
// Открываем текстовый файл поверх потока.
function Open(var F: TTextRec): Integer;
begin
  case F.Mode of
  fmInput:
    begin
      // Читаем из существующего файла.
      F.Stream.Position := 0;
      F.BufEnd := 0;
      F.InOutFunc := @Input;
      Result := 0;
    end;
  fmInOut:
    begin
      // Дописываем в конец существующего файла.
      F.Mode := fmOutput;
      F.BufEnd := F.BufSize;
      F.Stream.Position := F.Stream.Size;
      F.InOutFunc := @Output;
      Result := 0;
    end;
  fmOutput:
    begin
      // Создаем новый файл.
      F.Stream.Position := 0;
      F.BufEnd := F.BufSize;
      F.InOutFunc := @Output;
      Result := 0;
    end;
  else
    Result := Error_Invalid_Parameter;
  end;
end;

// Присваивание потока текстовому файлу.
procedure AssignStream(var AFile: TextFile; Stream: TStream);
var
  F: TTextRec absolute AFile;
begin
  F.Handle := Invalid_Handle_Value;
  F.Mode := fmClosed;
  F.BufSize := SizeOf(F.Buffer);
  F.BufPtr := @F.Buffer;
  F.BufPos := 0;
  F.OpenFunc := @Open;
  F.CloseFunc := @Close;
  F.FlushFunc := @Flush;
  F.Stream := Stream;
  F.Name[0] := #0;
end;
...
```

```
// Для открытия текстового файла поверх потока вызываем
// AssignStream вместо AssignFile.
Stream := TMemoryStream.Create;
AssignStream(Output, Stream);
Rewrite(Output);
//... Для записи в стандартный выходной файл используется Write и WriteLn.
CloseFile(Output);

// Выводим содержимое потока.
Memo1.Lines.LoadFromStream(Stream);
```

Смотрите также

Процедуры Append, Read, ReadLn, Reset, Rewrite, SetTextBuf, Write, WriteLn, функции Eof, Eoln, IOResult, SeekEof, SeekEoln, ключевое слово File.

TGUID, тип

Синтаксис

```
type
  TGUID = packed record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Описание

Тип TGUID предназначен для хранения глобально уникального идентификатора (Globally Unique Identifier, GUID). При создании нового GUID Windows гарантирует, что его значение будет уникально среди всех GUID, созданных в мире. Delphi использует GUID для идентификации и поиска интерфейсов.

Советы и приемы

- Delphi автоматически преобразует имя интерфейса в его GUID, и поэтому имя интерфейса можно использовать для вызова, например, метода QueryInterface.
- В объявлении интерфейса GUID следует помещать в квадратные скобки, например:

```
type IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
```

- Для того чтобы сгенерировать новый GUID из IDE, применяется комбинация клавиш <Ctrl>+<Shift>+<G>.

- GUID не обязательно должен присутствовать в объявлении интерфейса, но обычно бывает нужен. Без GUID нельзя преобразовать интерфейс к другому типу.

Смотрите также

Ключевое слово `Interface`, интерфейс `IUnknown`, тип `PGUID`.

THeapStatus, тип

Синтаксис

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

Описание

Функция `GetHeapStatus` возвращает запись типа `THeapStatus`, в которой содержится информация о состоянии встроенного менеджера памяти Delphi. Если устанавливается собственный менеджер памяти, все значения состояния «кучи», перечисленные ниже, скорее всего, будут равны нулю.

TotalAddrSpace

Общее количество байт, управляемых менеджером памяти. Delphi начинает с 1 мегабайта виртуальной памяти. Адресное пространство увеличивается по необходимости следующим образом:

```
TotalAddrSpace = TotalUncommitted + TotalCommitted.
```

TotalUncommitted

Общее число управляемых менеджером памяти байт, которые были выделены не из файла подкачки.

TotalCommitted

Общее число управляемых менеджером памяти байт, которые были выделены из файла подкачки:

```
TotalCommitted = TotalAllocated + TotalFree + Overhead.
```

TotalAllocated

Общее число выделенных в данный момент и используемых программой байт памяти.

TotalFree

Общее количество байт, свободных для выделения программой. Если объем доступной памяти слишком мал для выполнения запроса, менеджер памяти получает от Windows дополнительный объем памяти.

```
TotalFree = FreeSmall + FreeBig + Unused.
```

FreeSmall

Общее количество байт в списке небольших свободных блоков. Менеджер памяти поддерживает списки свободных блоков определенного «небольшого» размера. Каждый из списков содержит список блоков определенного размера. Большинство запросов памяти в среднестатистической программе требуют небольшие блоки, и менеджер памяти может удовлетворять эти запросы быстрее, возвращая адрес блока из соответствующего списка. «Небольшие» в данном случае означает менее 4 Кбайт.

FreeBig

Для блоков более 4 Кбайт Delphi поддерживает единый список больших блоков. Запросы на выделение больших блоков удовлетворяются с помощью этого списка. Первый блок, достаточный для этого запроса, разбивается на два участка: выделяемую часть и остаток, который возвращается обратно в список свободных блоков.

Unused

Количество байт, управляемых менеджером памяти, но еще не выделенных пользовательской программой.

Overhead

Объем накладных расходов в байтах, использованных менеджером памяти для отслеживания выделяемых блоков.

HeapErrorCode

Внутренний код ошибки, должен быть равен нулю. Ненулевое значение означает, что внутренние структуры менеджера памяти нарушены. Поведение программы будет непредсказуемым, и можно ожидать ошибок нарушения доступа к памяти или чего-нибудь подобного. Конкретные коды ошибок при наличии профессиональной (или более мощной) версии Delphi можно найти в файле *Source\Rtl\GetMem.inc*.

Смотрите также

Функции `GetHeapStatus`, `IsMemoryManagerSet`, **процедура** `SetMemoryManager`.

Then, ключевое слово

Синтаксис

```
if Выражение then Предложение
```

Описание

Ключевое слово `then` является частью конструкции `if`. Подробности рассмотрены в описании оператора `if`.

Смотрите также

Ключевые слова `Else`, `If`.

ThreadVar, ключевое слово

Синтаксис

```
threadvar Имя = Тип; ...
```

Описание

Ключевое слово `threadvar` обозначает начало описания переменной аналогично ключевому слову `var` за исключением того, что `threadvar` может применяться только на уровне модуля. Переменные, объявленные с помощью ключевого слова `threadvar`, имеют различные значения в каждом потоке, в то время как обычные переменные разделяются между всеми потоками процесса.

Советы и приемы

- Если используется класс `TThread` (из модуля `Classes`), удобнее хранить данные конкретного потока в полях этого класса.
- Если создается подпрограмма или класс для многократного использования в многопоточных приложениях, можно с помощью переменных `threadvar` отслеживать данные конкретных потоков.
- Специфика объявления `threadvar` относится только к самой переменной. Если переменная является указателем или неявно содержит указатель (например, объект, длинную строку, динамический массив или `Variant`), данные могут быть доступны из нескольких потоков.
- `Threadvar` переменные используют технологию TLS (Thread Local Storage) Windows. Каждое обращение к переменной требует вызова функции, поэтому еще одной причиной применения класса `TThread` является то, что доступ к полям объекта выполняется быстрее, чем обращение к `threadvar`-переменным.
- Каждому исполняемому модулю, содержащему переменную `threadvar`, требуется отдельная ячейка TLS, но Windows предоставляет ограниченное число таких ячеек. В большом проекте следует ис-

пользовать пакеты, чтобы обеспечить доступность этого ограниченного ресурса.

Каждому модулю (приложению, DLL, пакету), который содержит переменную `threadvar`, требуется своя собственная, отдельная ячейка TLS. Если применяется статическое связывание с DLL, библиотека использует собственную копию модуля `System` и переменных `InOutRes` и `RaiseList`[1] (являющихся переменными `threadvar`). Проект с большим количеством DLL может исчерпать все доступные ячейки TLS, даже если не содержит собственных переменных `threadvar`.

Если применяются пакеты, используется разделяемый экземпляр модуля `System` из пакета *vcl50* и его переменные `threadvar`. Это значит, что проекту требуется только одна ячейка TLS, независимо от количества загружаемых им DLL и пакетов. Если какой-либо DLL или пакету требуются собственные переменные `threadvar`, дополнительная ячейка TLS требуется только этому исполняемому модулю.

- Не помещайте переменные `threadvar` в разделе завершения модуля, который может быть использован в составе DLL. Delphi освобождает память TLS при выгрузке DLL, но до запуска разделов завершения модулей.

Пример

```
// Каждый поток имеет собственную переменную List, однако программист
// должен сам позаботиться о присваивании каждой из переменных List
// уникального обьекта типа TList.
threadvar
  List: TList;

procedure ThreadFunc(Param: Pointer);
begin
  List := TList.Create;
  ...
end;

procedure StartThread;
begin
  Handle := BeginThread(nil, 0, ThreadFunc, nil, 0, Id);
end;
```

Смотрите также

Функция `BeginThread`, переменные `IsMultiThread`, `TlsIndex`, `TlsLast`.

---

[1] В Delphi 4 и последующих версиях в модуле `System` вместо переменной `RaiseList` определена переменная `RaiseListPtr`. – *Примеч. науч. ред.*

TInterfacedClass, тип

Синтаксис

```
type TInterfacedClass = class of TInterfacedObject;
```

Описание

Тип TInterfacedClass предназначен для удобства использования метакласса TInterfacedObject.

Смотрите также

Ключевое слово Class, тип TInterfacedObject.

TInterfacedObject, тип

Синтаксис

```
type
  TInterfacedObject = class(TObject, IUnknown)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Описание

Класс TInterfacedObject реализует интерфейс IUnknown с семантикой счетчика ссылок, принятой в Delphi и COM. Он также реализует метод QueryInterface для вызова GetInterface. Другими словами, TInterfacedObject предоставляет практическую реализацию методов интерфейса IUnknown и может служить в качестве базового для любых классов, реализующих один или более интерфейсов.

Советы и приемы

- При работе с COM можно использовать один из классов, производных от TInterfacedObject или TAggregatedObject, описанных в модулях ComObj и ComServ.
- Если COM не применяется, TInterfacedClass обеспечивает тот минимум функциональности, который необходим для того, чтобы получить от интерфейсов в Delphi максимальную пользу.
- Необходимости в использовании TInterfacedObject нет. Любой класс может сам реализовать методы интерфейса IUnknown. При реализа-

ции IUnknown в другом классе можно скопировать реализацию методов _AddRef, _Release и QueryInterface из класса TInterfacedObject.

Пример

```
type
  // Интерфейс для доступа к объекту
  IObject = interface
  ['{618977EA-1ADA-11D3-B1A8-00105AA9C2AD}']
    function GetValue: TObject;
    procedure SetValue(Value: TObject);
    property Value: TObject read GetValue write SetValue;
  end;

  TObjectWrapper = class(TInterfacedObject, IObject)
  private
    fObject: TObject;
    function GetValue: TObject;
    procedure SetValue(Value: TObject);
  public
    constructor Create(ObjRef: TObject);
    property Value: TObject read GetValue write SetValue;
  end;
```

Смотрите также

Ключевое слово Interface, тип TObject.

TInterfaceEntry, тип

Синтаксис

```
type
  TInterfaceEntry = packed record
    IID: TGUID;
    VTable: Pointer;
    IOffset: Integer;
    ImplGetter: Integer;
  end;
```

Описание

Все классы, реализующие интерфейсы, имеют таблицу своих интерфейсов в RTTI. Запись TInterfaceEntry содержит необходимую для реализации интерфейса информацию. Более подробно о RTTI рассказывается в главе 3.

Советы и приемы

- Объекты могут реализовывать интерфейсы с помощью свойств (директива implements). Значение свойства определяется полем или ссылкой на метод, которые хранятся в поле ImplGetter в виде смещения поля, смещения виртуального метода или указателя на стати-

ческий метод. Обычно класс создает интерфейс путем реализации всех его методов, и ImplGetter равно нулю.

- Если класс реализует интерфейс напрямую, компилятор генерирует таблицу виртуальных методов интерфейса и сохраняет ссылку на нее в поле VTable. Кроме того, компилятор генерирует скрытое поле для хранения интерфейса. Смещение поля в объекте хранится в поле IOffset. При создании нового объекта Delphi автоматически инициализирует скрытые поля интерфейса.
- Способ поиска интерфейсов, реализованный в Delphi, рассмотрен в описаниях типа TObject и его методов GetInterfaceTable и GetInterface.

Смотрите также

Директива Implements, ключевые слова Interface, Property, типы PInterfaceEntry, TGUID, TInterfaceTable, TObject.

TInterfaceTable, тип

Синтаксис

```
Type
  TInterfaceTable = packed record
    EntryCount: Integer;
    Entries: array[0..9999] of TInterfaceEntry;
  end;
```

Описание

Каждый класс, реализующий интерфейсы, имеет в RTTI таблицу интерфейсов, структура которой представлена записью TInterfaceTable. Дополнительную информацию можно получить в главе 3.

Советы и приемы

- EntryCount — это фактическое количество описаний интерфейсов в массиве Entries.
- Способ поиска интерфейсов, реализованный в Delphi, приводится в описаниях типа TObject и его методов GetInterfaceTable и GetInterface.

Смотрите также

Ключевое слово Interface, типы PInterfaceTable, TInterfaceEntry, TObject.

TLibModule, тип

Синтаксис

```
type
  TLibModule = record
```

```
    Next: PLibModule;
    Instance: LongWord;
    CodeInstance: LongWord;
    DataInstance: LongWord;
    ResInstance: LongWord;
    Reserved: Integer;
  end;
```

Описание

Тип TLibModule содержит информацию об исполняемом модуле проекта и обо всех пакетах, загружаемых проектом. Записи хранятся в виде односвязного списка, где поле Next указывает на следующий исполняемый модуль в списке.

Instance – это дескриптор экземпляра исполняемого модуля. ResInstance – дескриптор экземпляра соответствующего модуля ресурсов. Если исполняемый модуль имеет отдельную библиотеку ресурсов, Delphi автоматически загружает языковую библиотеку и сохраняет дескриптор ее экземпляра в ResInstance. В противном случае ResInstance равен Instance.

CodeInstance и DataInstance применяются в C++ Builder для библиотек.

Смотрите также

Переменная LibModuleList, тип PLibModule, процедуры RegisterModule, UnregisterModule.

TlsIndex, переменная

Синтаксис

```
unit SysInit;
var TlsIndex: Integer;
```

Описание

Каждый модуль отслеживает свою ячейку TLS (Thread Local Storage, локальные данные потока) для реализации переменных threadvar. Когда загружается исполняемый модуль, он проверяет TlsLast, чтобы узнать, есть ли в модуле переменные threadvar. Если они есть, он запрашивает ячейку TLS от Windows и записывает ее индекс в TlsIndex. Не изменяйте значение этой переменной. Если исполняемый модуль не может получить свободную ячейку TLS, вызывается ошибка времени выполнения 226.

Смотрите также

Ключевое слово ThreadVar, переменная TlsLast.

TlsLast, переменная

Синтаксис

```
unit SysInit;
var TlsLast: Byte;
```

Описание

Адрес `TlsLast` определяет объем памяти, необходимый для хранения переменных `threadvar`. Когда запускается поток, Delphi выделяет для этой цели указанный объем памяти в TLS (Thread Local Storage, локальные данные потока). Адрес переменной равен `nil`, если программа, библиотека или пакет не используют переменные `threadvar`, и этому исполняемому модулю не выделена ячейка TLS.

Смотрите также

Ключевое слово `ThreadVar`, переменная `Tlsindex`.

TMemoryManager, тип

Синтаксис

```
type
  TMemoryManager = record
    GetMem: function(Size: Integer): Pointer;
    FreeMem: function(P: Pointer): Integer;
    ReallocMem: function(P: Pointer; Size: Integer): Pointer;
  end;
```

Описание

Тип `TMemoryManager` содержит указатели на три функции, необходимые для реализации менеджера памяти.

Пример

Пример приведен в описании процедуры `SetMemoryManager`.

Смотрите также

Процедуры `GetMemoryManager`, `SetMemoryManager`, функция `IsMemoryManagerSet`, тип `PMemoryManager`.

TModuleUnloadProcLW, тип

Синтаксис

```
type
  TModuleUnloadProcLW = procedure(HInstance: LongWord);
```

Описание

Можно зарегистрировать функцию обратного вызова (callback function), которую Delphi вызывает при выгрузке пакета или исполняемого модуля. Сигнатура процедуры должна совпадать с типом TModuleUnloadProcLW, т.е. принимать в качестве аргумента дескриптор.

Советы и приемы

- Для обратной совместимости AddModuleUnloadProc и RemoveModuleUnloadProc имеют перегруженные версии, которые используют callback-функции с аргументом LongInt. В новом коде следует применять LongWord (или эквивалентный тип DWORD из модуля Windows).

Пример

Пример приведен в описании процедуры AddModuleUnloadProc

Смотрите также

Процедуры AddModuleUnloadProc, RemoveModuleUnloadProc, тип TModuleUnloadRec.

TModuleUnloadRec, тип

Синтаксис

```
type
  TModuleUnloadRec = record
    Next: PModuleUnloadRec;
    Proc: TModuleUnloadProcLW;
  end;
```

Описание

Delphi поддерживает односвязный список записей TModuleUnloadRec. Каждая запись содержит указатель на callback-функцию. Когда Delphi выгружает пакет или когда проект выгружает себя, Delphi вызывает все процедуры в списке, передавая им в качестве аргумента дескриптор экземпляра исполняемого модуля.

Описание

Пример приведен в описании процедуры AddModuleUnloadProc.

Смотрите также

Процедуры AddModuleUnloadProc, RemoveModuleUnloadProc, переменная ModuleUnloadList, типы PModuleUnloadRec, TModuleUnloadProcLW.

To, ключевое слово

Синтаксис

```
for Переменная := Выражение1 to Выражение2 do Оператор(ы)
```

Описание

Ключевое слово to является частью цикла for с увеличивающимся счетчиком. Подробности рассмотрены в описании цикла for.

Смотрите также

Ключевые слова Downto, For.

TObject, тип

Синтаксис

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: LongInt;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID):PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    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;
  end;
```

Описание

TObject – это корневой класс, от которого производятся все остальные классы. У него нет явных полей, однако имеется скрытое поле с указателем на VMT. В классе также определено несколько полезных методов, которые наследуются всеми классами.

Методы класса

В этом разделе описаны методы класса TObject.

ClassInfo, функция

Функция `ClassInfo` возвращает указатель на таблицу `TTypeInfo` класса. Ее вызов аналогичен `TypeInfo(ClassRef)`, за исключением того, что функция `TypeInfo` работает только для ссылки на класс, а `ClassInfo` можно вызвать с любой ссылкой на класс или объект, в том числе с переменной типа «метакласс».

ClassNameIs, функция

Функция `ClassNameIs` проверяет, соответствует ли данная строка имени класса. Имена классов не чувствительны к регистру букв, поэтому при сравнении различия в регистре не учитываются. Функция возвращает `True`, если `Name` совпадает с именем класса, и `False`, если отличается.

ClassParent, функция

Функция `ClassParent` возвращает метакласс непосредственного базового класса. `TObject.ClassParent` возвращает `nil`, т. к. `TObject` не имеет базового класса.

Create, конструктор

`Create` не выполняет никаких действий. Он существует для того, чтобы можно было вызвать унаследованный конструктор из конструктора при создании экземпляра любого другого класса. При создании производного класса и реализации его конструктора всегда вызывайте унаследованный конструктор.

Destroy, деструктор

Хотя Delphi позволяет объявлять несколько деструкторов, не следует это делать. Вместо этого переопределяйте `Destroy`. Деструктор `TObject.Destroy` не выполняет никаких действий, но всегда имеет смысл вызывать `inherited Destroy` из собственных деструкторов.

GetInterfaceEntry, функция

`GetInterfaceEntry` выполняет поиск по GUID и возвращает указатель на соответствующую запись `TInterfaceEntry` или `nil`, если сам класс и его родительские классы не поддерживают данный интерфейс. Если данный класс не поддерживает интерфейс, `GetInferfaceEntry` проверяет базовые классы на предмет поддержки интерфейса.

GetInterfaceTable, функция

`GetInterfaceTable` возвращает указатель на запись `TInterfaceTable`, содержащую информацию об интерфейсах, поддерживаемых классом. Если класс не поддерживает никаких интерфейсов, функция возвращает `nil`. Функция просто возвращает указатель, содержащийся в RTTI класса.

InheritsFrom, функция

InheritsFrom проверяет класс, совпадает ли он с типом AClass и является ли потомком AClass; если это так, возвращается True. Если AClass не является базовым для данного, функция возвращает False. Операторы is и as вызывают метод InheritsFrom для аналогичной проверки.

InitInstance, функция

Функция InitInstance получает указатель на область памяти, выделенную для объекта, и инициализирует ее. Она возвращает тот же указатель, но в виде ссылки на объект. Память, выделенная объекту, инициализируется нулями, инициализируются все интерфейсы, поддерживаемые классом, и указатель на VMT класса сохраняется в первом (скрытом) поле объекта.

Если метод NewInstance был переопределен для изменения способа выделения памяти для объектов, следует вызвать InitInstance для инициализации объекта.

InstanceSize, функция

Функция InstanceSize возвращает количество байт, которое требуется экземпляру класса. Не используйте SizeOf, т. к. ссылки на классы и объекты являются указателями, и потому SizeOf(AClass) и SizeOf(AnObject) возвращают то же значение, что и SizeOf(Pointer). Если метод NewInstance был переопределен для выделения памяти нестандартным образом, обязательно выделите как минимум InstanceSize байт.

MethodAddress, функция

Функция MethodAddress ищет имя метода в таблице публикуемых методов RTTI и возвращает указатель на код метода. Поиск выполняется без учета регистров букв. Если класс не реализует этот метод, MethodAddress выполняет поиск в таблицах RTTI родительских классов. Если ни один из родительских классов не реализует этот метод как публикуемый, функция возвращает nil. Можно сформировать запись TMethod (в модуле SysUtils), присвоив возвращаемое MethodAddress значение полю Code, а ссылку на объект – полю Data. Затем можно преобразовать TMethod к типу события, например, TNotifyEvent, как показано ниже:

```
var
  Notify: TNotifyEvent;
begin
  TMethod(Notify).Code := Form.MethodAddress('Button1Click');
  TMethod(Notify).Data := Form;
  Notify(Button1);
end;
```

MethodName, функция

Функция `MethodName` определяет имя публикуемого метода по указателю на его код. Если метод не найден, возвращается пустая строка. Delphi вызывает `MethodName` при формировании *dfm*-файла для сохранения имени метода в описании формы.

NewInstance, функция

Функция `NewInstance` создает новый экземпляр класса и инициализирует объект. Она возвращает новую ссылку на объект. Метод является виртуальным, поэтому можно переопределить его и изменить способ создания объектов определенного класса. Если вы сделали это, вызовите `InitInstance` для инициализации объекта и переопределите `FreeInstance` для освобождения памяти объекта.

Методы объекта

В этом разделе описаны статические и виртуальные методы `TObject`.

AfterConstruction, процедура

Delphi вызывает `AfterConstruction` после того, как конструкторы завершили свою работу. Можно переопределить этот метод для выполнения действий, которым требуется дождаться полноценного формирования объекта. Это может быть полезным, если создаются классы для использования в Delphi и C++ Builder. Эти два языка имеют разную семантику для конструкторов, но можно обойти это различие, переопределив `AfterConstruction`.

BeforeDestruction, процедура

Delphi вызывает `BeforeDestruction`, когда объект уничтожается, но до вызова всех деструкторов. Можно выполнить очистку объекта или другие действия, для выполнения которых требуется корректный объект. Это может быть полезным, если создаются классы для использования в Delphi и C++ Builder. Эти два языка имеют разную семантику для деструкторов, но можно обойти это различие, переопределив `BeforeDestruction`.

ClassType, функция

Функция `ClassType` возвращает ссылку на метакласс объекта, которая является просто указателем на VMT.

CleanupInstance, процедура

Процедура `CleanupInstance` выполняет очистку строк, переменных типа `Variant`, динамических массивов и интерфейсов. Переопределяя `FreeInstance`, вызовите `CleanupInstance` перед освобождением памяти объекта.

DefaultHandler, процедура

`DefaultHandler` — это обработчик сообщений по умолчанию. Delphi вызывает `DefaultHandler`, когда пытается обработать сообщение, но

не может найти для него обработчик. TObject реализует DefaultHandler, который не выполняет никаких действий. Производные классы могут переопределить и переопределяют этот метод для реализации других действий. Например, TWinControl перенаправляет сообщение на оконную процедуру управляющего элемента.

Dispatch, процедура

Процедура Dispatch выполняет диспетчеризацию сообщения. Dispatch проверяет первые два байта аргумента и интерпретирует их как номер сообщения. Далее выполняется поиск этого номера в RTTI класса, и если номер найден, Dispatch вызывает соответствующий метод-обработчик. Иначе выполняется поиск в RTTI родительских классов. Если Dispatch не может найти обработчик для сообщения, вызывается DefaultHandler.

FieldAddress, функция

Функция FieldAddress ищет имя поля в таблице публикуемых полей RTTI класса и возвращает указатель на поле объекта или nil, если класс и его родители не содержат полей с этим именем в секции published. Поиск не чувствителен к регистру символов. Публикуемые поля должны принадлежать к объектному типу, поэтому можно с уверенностью сказать, что в поле содержится ссылка на объект.

Free, процедура

Функция Free используется для уничтожения объекта. Не вызывайте напрямую деструктор объекта. Free проверяет, не равна ли ссылка на объект nil, и вызывает деструктор только для ненулевых ссылок.

FreeInstance, процедура

Процедура FreeInstance очищает объект (с помощью CleanupInstance) и освобождает память, занятую объектом. Можно переопределить этот метод, реализовав свои собственные действия. При переопределении FreeInstance обязательно вызовите CleanupInstance. Переопределяя NewInstance, необходимо также переопределить и FreeInstance.

GetInterface, функция

GetInterface выполняет поиск GUID и выбирает соответствующий интерфейс. Функция возвращает True, если интерфейс найден и присвоен параметру Obj, и False, если объект не поддерживает запрашиваемый интерфейс. Если интерфейс не поддерживается, Obj устанавливается в nil.

SafeCallException, функция

Delphi помещает любой вызов метода safecall в блок try-except для перехвата всех исключительных ситуаций Delphi. Затем вызывается метод SafeCallException для преобразования исключительной си-

туации в стандартный код ошибки Windows. TComObject более сложен и, по возможности, использует IErrorInfo для выдачи сообщения об ошибке. Все детали обрабатываются Delphi автоматически, и для того чтобы управлять выдачей клиентам ошибок COM-сервера, можно установить значение свойства ServerExceptionHandler.

Советы и приемы

Виртуальные методы класса TObject являются особым случаем в том смысле, что хранятся по отрицательным смещениям таблицы виртуальных методов. «Первым» методом в VMT является первый виртуальный метод производного класса. Дополнительную информацию о VMT можно найти в главе 3.

Смотрите также

Ключевые слова Class, Function, Interface, Procedure, Property, Type, процедуры FreeMem, GetMem, директивы Message, SafeCall, переменные SafeCallErrorProc, Self, тип TClass.

TResStringRec, тип

Синтаксис

```
type
  TResStringRec = packed record
    Module: ^LongInt;       // модуль ресурсов
    Identifier: Integer;    // идентификатор строкового ресурса
  end;
```

Описание

Delphi компилирует объявление resourcestring в запись TResStringRec и вызов процедуры LoadResString. Поле Module указывает на дескриптор модуля ресурсов, содержащего строковый ресурс, Identifier – идентификатор строкового ресурса. Компоновщик автоматически присваивает уникальный идентификатор каждой resourcestring.

Смотрите также

Функция LoadResString, тип PResStringRec, ключевое слово ResourceString.

Trunc, функция

Синтаксис

```
function Trunc(X: Вещественный тип): Int64;
```

Описание

Функция Trunc округляет вещественное значение путем отбрасывания дробной части (округляет в направлении нуля). В отличие от Int, Trunc возвращает целый результат. Trunc не является настоящей функцией.

Советы и приемы

- Функция Trunc временно устанавливает управляющее состояние FPU, выполняет округление и восстанавливает состояние FPU.
- Если X – целое, компилятор исключает вызов функции и просто возвращает значение X.
- Если X – значение типа Variant, Delphi автоматически преобразует его в вещественное число и выполняет округление.
- Компилятор не принимает аргументы типа Int64, однако вызов функции Trunc для Int64 не имеет смысла.
- Если X равно бесконечности или NaN, Trunc генерирует ошибку времени выполнения 6 (EInvalidOp).

Смотрите также

Функции Int, Round.

Truncate, процедура

Синтаксис

```
procedure Truncate(var F: file);
procedure Truncate(var F: TextFile);
```

Описание

Truncate выполняет усечение файла таким образом, что текущая позиция становится концом файла. Truncate не является настоящей процедурой.

Если файл не был открыт, Truncate выдает ошибку ввода/вывода 103.

Смотрите также

Функции Eof, FilePos, FileSize, IOResult, SeekEof, ключевое слово File, процедура Seek, тип TextFile.

Try, ключевое слово

Синтаксис

```
try Предложения... finally Предложения... end;

try Предложения... except Предложения... end;

try Предложения... except Обработчики... end;
```

Описание

Ключевое слово try обозначает начало конструкции try-except или try-finally. Эти две конструкции связаны, но служат для разных целей.

Конструкция try-finally применяется для управления памятью и другими ресурсами. Она сначала выполняет операторы секции try, а затем секции finally. Операторы секции finally выполняются независимо от того, каким образом было передано управление из секции try: исключительная ситуация, возврат из подпрограммы с помощью процедуры Exit или выход из цикла процедурой Break.

Конструкция try-except выполняет обработку ошибочных и исключительных ситуаций. Сначала выполняется секция try. Если возникла исключительная ситуация, управление передается секции except, в которой Delphi выполняет поиск соответствующего обработчика. Если секция except содержит только обычные операторы, Delphi выполняет их и затем продолжает выполнение программы после ключевого слова end конструкции try-except. Если секция except содержит один или более обработчиков исключительных ситуаций (обозначенных директивой on), Delphi ищет соответствующий обработчик. Если подходящий обработчик не может быть найден, управление передается следующей конструкции try-except в стеке вызовов.

Советы и приемы

- Используйте try-finally для уничтожения временных объектов, различных ресурсов и выполнения других действий по очистке. Обычно в подпрограмме или методе не бывает более одной конструкции try-finally.
- Конструкция try-except применяется для обработки исключительных случаев. В утилитах и общеупотребительных классах следует избегать использования try-except. С другой стороны, в приложении конструкция try-except предназначена для перехвата определенных исключительных ситуаций и выполнения с ними конкретных действий, например, записи данных в протокол ошибок, отправки разработчику сообщения по электронной почте или выдачи дружественного диалогового окна.
- Низкоуровневый код может использовать конструкцию try-except для перехвата низкоуровневых исключительных ситуаций, добавления новой информации и генерирования исключительной ситуации более высокого уровня.

Пример

```
// Копирование файла.
procedure CopyFile(const DestFile, SourceFile: string);
const
  ReadFlags = fmOpenRead or fmShareDenyWrite;
resourcestring
  sSourceError = 'Невозможно открыть исходный файл: %s';
  sDestError = 'Невозможно открыть целевой файл: %s';
  sCopyError = 'Невозможно скопировать %1:s to %2:s: %0:s';
var
  DestStream, SourceStream: TStream;
```

```
begin
  DestStream := nil;
  SourceStream := nil;
  try
    try
      SourceStream := TFileStream.Create(SourceFile, ReadFlags);
      DestStream := TFileStream.Create(DestFile, fmCreate);
      DestStream.CopyFrom(SourceStream);
    except
    // EFCreateError и EFOpenError являются производными от
    // EStreamError, поэтому сначала выполяем проверку производных
    // классов. Генерируем исключительную ситуацию, выдающую
    // дополнительную информацию. Если возникла другая исключительная
    // ситуация, она передается вызывающей стороне и далее, до
    // тех пор, пока Delphi не найдет другого обработчика
    // исключительной ситуации.
    on Ex: EFCreateError do
      raise EFCreateError.CreateFmt(sDestError, [Ex.Message]);
    on Ex: EFOpenError do
      raise EFOpenError.CreateFmt(sSourceError, [Ex.Message]);
    on Ex: EStreamError do
      raise EStreamError.CreateFmt(sCopyError,
                [Ex.Message, SourceFile, DestFile]);
    end;
  finally
    // Даже если в try-except возникла исключительная ситуация,
    // секция finally выполняется и объекты потоков освобождаются.
    DestStream.Free;
    SourceStream.Free;
  end;
end;
```

Смотрите также

Ключевые слова `Except`**,** `Raise`**, директивы** `Finally`**,** `On`**.**

TThreadFunc, тип

Синтаксис

```
type
  TThreadFunc = function(Parameter: Pointer): Integer;
```

Описание

Функция `BeginThread` **принимает в качестве одного из аргументов функцию потока. Сигнатура функции должна соответствовать типу** `TThreadFunc`**.** `BeginThread` **передает свой аргумент** `Param` **функции потока. Возвращаемое функцией потока значение становится кодом возврата потока.**

Пример

Пример рассмотрен в описании функции BeginThread.

Смотрите также

Функция BeginThread, процедура EndThread, переменная IsMultiThread, ключевое слово ThreadVar.

TVarArray, тип

Синтаксис

```
type
  TVarArray = packed record
    DimCount: Word;
    Flags: Word;
    ElementSize: Integer;
    LockCount: Integer;
    Data: Pointer;
    Bounds: array[0..255] of TVarArrayBound;
  end;
```

Описание

В записи TVarArray хранится информация, используемая для реализации массива Variant. Запись TVarData хранит указатель на запись TVarArray, если тип значения Variant содержит бит varArray.

DimCount

Содержит количество измерений, может принимать значения от 1 до 64.

Flags

Флаги, используемые Windows для отслеживания памяти, занимаемой массивом, и управления ею.

ElementSize

Размер каждого элемента массива в байтах. Размер элемента зависит от его типа, хранящегося в поле VType записи TVarData.

LockCount

Количество блокировок обычно равно нулю. Оно увеличивается при вызове VarArrayLock и уменьшается при вызове VarArrayUnlock. Изменить размер массива можно только в том случае, если LockCount равно нулю.

Data

Указатель на данные массива. Фактическое содержимое массива хранится в одномерном массиве, при этом самый левый индекс изменяется быстрее всего (развертка по строкам).

Bounds

Массив записей TVarArrayBound содержит границы каждого из размерностей массива. Поле Bounds выделяется динамически, поэтому фактически оно содержит DimCount записей.

Информацию о способе хранения указателя на запись TVarArray в Variant можно найти в описании типа TVarData.

Смотрите также

Типы PVarData, TVarArrayBound, TVarData, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim, тип Variant.

TVarArrayBound, тип

Синтаксис

```
type
  TVarArrayBound = packed record
    ElementCount: Integer;
    LowBound: Integer;
  end;
```

Описание

Границы для каждого измерения массива Variant сохраняются в записях TVarArrayBound. Запись содержит нижнюю границу массива и количество его элементов. Запись TVarArray массива Variant содержит массив записей TVarArrayBound.

Смотрите также

Типы TVarArray, TVarData, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

TVarData, тип

Синтаксис

```
type
  TVarData = packed record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
```

```
    varOleStr:   (VOleStr: PWideChar);
    varDispatch: (VDispatch: Pointer);
    varError:    (VError: LongWord);
    varBoolean:  (VBoolean: WordBool);
    varUnknown:  (VUnknown: Pointer);
    varByte:     (VByte: Byte);
    varString:   (VString: Pointer);
    varAny:      (VAny: Pointer);
    varArray:    (VArray: PVarArray);
    varByRef:    (VPointer: Pointer);
end;
```

Описание

Запись TVarData является реализацией в Delphi типа Variant. Можно преобразовать любое значение Variant к типу TVarData для обращения к внутренним данным Variant. Поле VType содержит тип вариантного значения (возвращаемый VarType). Другие поля содержат информацию для конкретных типов значений. Значения размером до 8 байт содержатся непосредственно в полях записи TVarData. Значения большего размера хранятся отдельно; в записи TVarData содержится указатель на фактические данные (например, VArray, указывающее на запись TVarArray).

Советы и приемы

- VString фактически является строковым типом, однако Delphi не позволяет хранить строки в записи с вариантами, т. к. при этом невозможно инициализировать и завершить запись. Код, управляющий типом Variant, управляет инициализацией и завершением строк.
- VUnknown фактически представляет собой интерфейс IUnknown, а VDispatch – интерфейс IDispatch. Как и в случае со строками, Delphi не разрешает использование интерфейсов в записях с вариантами.
- VAny – неявный указатель. Можно присвоить указатели на процедуры переменным ChangeAnyProc, ClearAnyProc и RefAnyProc для реализации типа varAny. В Delphi varAny присутствует для реализации типа Any в CORBA.
- VPointer – это указатель на вариантное значение. Если поле VType содержит бит varByRef, в Variant создается дополнительный уровень адресации, и устанавливается значение указателя на данные VPointer. Для того чтобы получить доступ к данным из поля VPointer, преобразуйте его к соответствующему типу Variant, например, PDataTime для varDate или PInteger для varInteger.

Пример

```
// Сохраняет значение типа Int64 в Variant с использованием
// типа varAny.
procedure SetVarInt64(var V: Variant; Value: Int64);
```

```
begin
  // Обеспечивает очистку старого значения.
  V := Unassigned;
  // Сохраняет значение Int64 в динамической памяти и
  // записывает указатель на него в Variant.
  TVarData(V).VType := varAny;
  GetMem(TVarData(V).VAny, SizeOf(Int64));
  PInt64(TVarData(V).VAny)^ := Value;
end;
```

Смотрите также

Переменные ChangeAnyProc, ClearAnyProc, RefAnyProc, **типы** Currency, Double, Integer, Pointer, PWideChar, Single, SmallInt, TDateTime, TVarArray, Variant, **ключевые слова** String, Type, **функция** VarType.

TVarRec, тип

Синтаксис

```
type
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
      vtInt64:      (VInt64: PInt64);
  end;
```

Описание

Параметр подпрограммы типа array of const фактически является массивом записей TVarRec. Delphi преобразует каждый элемент в запись TVarRec, записывая в поле VType код типа, а в соответствующее поле данных – фактическое значение элемента массива.

Советы и приемы

- Delphi использует открытые массивы и значения типа TVarRec для реализации подпрограмм, принимающих переменное число аргументов, сохраняя при этом строгую типизацию.
- TVarRec и TVarData никак не связаны между собой. Они выглядят похоже, т. к. выполняют сходные функции, однако не взаимозаменяемы.
- TVarRec поддерживает больше типов, чем Variant, но не все типы Delphi. В частности, не поддерживаются массивы, записи и перечисления.

Пример

```
// Вычисляет сумму произвольного количества значений.
function Sum(const Data: array of const): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := Low(Data) to High(Data) do
    case Data[I].VType of
    vtInt64:    Result := Result + Data[I].VInt64^;
    vtCurrency: Result := Result + Data[I].VCurrency^;
    vtInteger:  Result := Result + Data[I].VInteger;
    vtExtended: Result := Result + Data[I].VExtended^;
    vtVariant:  Result := Result + Data[I].VVariant^;
    else
      raise Exception.Create('В Sum передано нечисловое значение');
    end;
end;
```

Смотрите также

Ключевые слова Array, Const, типы TVarData, Variant.

Type, ключевое слово

Синтаксис

```
type Имя = Type declaration; ...
type Имя = type Type declaration;

type Имя1 = Имя2;
type Имя = (Идентификатор, ...);
type Имя = Expr1..Expr2;
type Имя = ^Тип;
type Имя1 = array[...] of Имя2;
type Имя = class ... end;
type Имя = class of ...;
type Имя = dispinterface ... end;
type Имя = file of Type;
```

```
type Имя = function ...;
type Имя = interface ... end;
type Имя = object ... end;
type Имя = procedure ...;
type Имя = record ... end;
type Имя = set of Порядковый тип;
```

Описание

Ключевое слово type обозначает, как и в стандартном Паскале, начало объявления типа. Если объявление типа снова начинается с ключевого слова type, Delphi генерирует уникальную RTTI для этого типа, даже если он является синонимом существующего. Кроме того, типы становятся различными по отношению к передаче параметров var.

- В программах на Delphi принято обозначать имена типов первой буквой T, кроме классов исключительных ситуаций (начинаются с буквы E), интерфейсов (начинаются с I) и типов указателей (начинаются с P).
- Объявление прототипа класса должно разрешаться в том же блоке type, где был объявлен класс. Например:

```
type
  TExample = class; // объявление прототипа
  TOther = class
    procedure Example(E: TExample);
  end;
  // Полное объявление класса без указания нового ключевого слова
  // «type», обозначающего новый блок type.
  TExample = class
    procedure Example(Other: TOther);
  end;
```

- За объявлением класса в разделе интерфейса модуля в его же разделе реализации должны следовать определения всех неабстрактных методов класса.

Пример

```
type
  TSuit = (Diamond, Club, Heart, Spade);
  PCard = ^TCard;  // pointer to TCard;
  TCard = record
    Suit: TSuit;
    Rank: 1..13;
  end;
  Deck = array[1..52] of TCard;
  TSuits = set of TSuit;

  TCardFunc = function(const Card: TCard): Integer;
  TCardProc = procedure(const Card: TCard);
  TCardMethodFunc = function(const Card: TCard): Integer of object;
  TCardMethodProc = procedure(var Card: TCard) of object;
```

```
IGame = interface
['{6164F471-5E41-11D3-B1B6-00105AA9C2AD}']
  procedure Play;
  procedure Draw(var Card: TCard);
end;
TGame = class(TInterfacedObject, IGame)
public
  procedure Play;
  procedure Draw(var Card: TCard);
end;
```

Смотрите также

Ключевые слова Array, Class, Dispinterface, File, Function, Interface, Object, Packed, Procedure, Record, Set, **типы** Integer, TClass, TextFile, TObject, **функция** TypeInfo.

TypeInfo, функция

Синтаксис

```
function TypeInfo(Имя типа): Pointer;
```

Описание

Функция TypeInfo возвращает указатель на информацию о типе времени выполнения для данного типа. Указатель фактически является указателем PTypeInfo на запись TTypeInfo. Эти типы объявляются в модуле TypInfo.

Дополнительная информация о RTTI приведена в главе 3.

Пример

```
// Преобразование множества в строковое представление,
// например '[fsBold,fsItalic]'.
function SetToString(Info: PTypeInfo; const Value): string;
var
  I: Integer;
  Data: PTypeData;        // данные о типе множества
  EnumInfo: PTypeInfo;    // данные о базовом типе множества
  EnumData: PTypeData;    // данные о базовом типе множества
begin
  if Info.Kind <> tkSet then
    Result := ''
  else
  begin
    Data := GetTypeData(Info);
    EnumInfo := Data^.CompType^;
    EnumData := GetTypeData(EnumInfo);

    Assert(EnumInfo.Kind in [tkEnumeration,tkInteger]);
```

```
    Result := '[';
    for I := EnumData.MinValue to EnumData.MaxValue do
      if I in TIntegerSet(Value) then
      begin
        // Элемент присутствует во множестве, поэтому добавляем его
        // имя к строке.
        if Length(Result) > 1 then
          Result := Result + ',';  // Разделяем элементы множества
                                   // запятыми.
        Result := Result + GetEnumName(EnumInfo, I);
      end;
    Result := Result + ']';
  end;
end;
...
S := SetToString(TypeInfo(TFontStyles), Control.Font.Style);
```

Unassigned, переменная

```
var Unassigned: Variant;
```

Синтаксис

Переменная `Unassigned` является значением типа `Variant`, обозначающим неинициализированную переменную. Delphi автоматически инициализирует каждую переменную типа `Variant` значением `Unassigned`. Если попытаться присвоить значение `Unassigned` выражению, Delphi выдаст ошибку времени выполнения 16 (`EVariantError`).

Советы и приемы

- Вызов `VarClear` аналогичен присваиванию переменной типа `Variant` значения `UnAssigned`.
- `VarIsEmpty` возвращает `True` для `Unassigned`.

Смотрите также

Переменная `Null`, процедура `VarClear`, тип `Variant`, функция `VarIsEmpty`.

UniqueString, процедура

Синтаксис

```
procedure UniqueString(var Str: string);
```

Описание

Процедура `UniqueString` обеспечивает равенство единице значения счетчика ссылок на `Str`. Если на `Str` нет других ссылок, `UniqueString` не выполняет никаких действий; в противном случае `UniqueString` создает новую копию строки и присваивает ее переменной `Str`.

`UniqueString` – настоящая процедура.

Советы и приемы

- Обычно в вызове `UniqueString` нет необходимости. Delphi использует для строк семантику «копирование при записи», т.е. при наличии нескольких ссылок на строку модификация одной из ссылок вызывает создание новой строки, так что другие ссылки продолжают указывать на оригинал.
- Если строка преобразуется к типу `PChar`, Delphi сделает так, что `PChar` будет указывать на уникальный экземпляр строки. Если строка модифицируется через указатель `PChar`, другие ссылки на исходную строку останутся неизменными.

Смотрите также

Ключевое слово `String`.

Unit, ключевое слово

Синтаксис

```
unit Имя;
interface
  Объявления...
implementation
  Объявления...
initialization
  Предложения...
finalization
  Предложения...
end.

unit Имя;
interface
  Объявления...
implementation
  Объявления...
begin
  Предложения...
end.
```

Описание

Ключевое слово `unit` обозначает начало модуля – основной единицы построения программ в Delphi. Имя модуля должно совпадать с именем файла (с расширением *.pas*). Дополнительная информация о модулях представлена в главе 2.

Советы и приемы

- И раздел интерфейса, и раздел реализации могут содержать объявление `uses`. В этом случае `uses` должен быть первым объявлением к разделе.

- Разделы инициализации и завершения не обязательны. Но если модуль содержит раздел завершения, он должен иметь и раздел инициализации (хотя бы и пустой).
- Вместо ключевого слова `initialization` можно использовать `begin`, но в этом случае модуль не может иметь раздела завершения.

Смотрите также

Ключевые слова `Begin`, `End`, `Finalization`, `Implementation`, `Initialization`, `Interface`, `Uses`.

UnitEntryTable, тип

Синтаксис

```
type UnitEntryTable = array[0..9999999] of PackageUnitEntry;
```

Описание

Каждый пакет содержит таблицу `UnitEntryTable`, в которой располагаются записи `PackageUnitEntry` для модулей, составляющих пакет.

Смотрите также

Функция `GetPackageInfoTable`, типы `PackageInfoTable`, `PackageUnitEntry`, `PUnitEntryTable`, ключевое слово `Unit`.

UnregisterModule, процедура

Синтаксис

```
procedure UnregisterModule(LibModule: PLibModule);
```

Описание

Процедура `UnregisterModule` удаляет библиотечный модуль из списка зарегистрированных в Delphi исполняемых модулей. Delphi автоматически вызывает `UnregisterModule` при выгрузке пакета или при завершении программы.

Советы и приемы

Можно вызвать `UnregisterModule` для удаления библиотечного модуля из списка, но при этом Delphi не выполнит действий по его завершению.

Смотрите также

Процедуры `EnumModules`, `EnumResourceModules`, `RegisterModule`, переменная `LibModuleList`, типы `PLibModule`, `TLibModule`.

Until, ключевое слово

Синтаксис

```
repeat
```

```
  Предложения...
until Логическое выражение
```

Описание

Ключевое слово until отмечает конец предложения repeat-until.

Смотрите также

Тип Boolean, процедуры Break, Continue, ключевое слово Repeat.

UpCase, функция

Синтаксис

```
function UpCase(C: Char): Char;
```

Описание

Функция UpCase преобразует ASCII символ к верхнему регистру. Если C не является символом нижнего регистра, UpCase возвращает C. UpCase — настоящая функция.

Советы и приемы

- Функция UpCase не обрабатывает международные символы — только от «a» до «z».
- Соответствующей функции DownCase в Delphi нет.

Пример

```
function DownCase(Ch: Char): Char;
begin
  if Ch in ['A'..'Z'] then
    Result := Chr(Ord(Ch) - Ord('A') + Ord('a'))
  else
    Result := Ch;
end;

// Преобразует идентификатор в каноническую форму, т.е. первый
// символ - в верхнем регистре, все остальные - в нижнем.
function Canonical(const S: string): string;
var
  I: Integer;
begin
  SetLength(Result, Length(S));
  if Length(S) > 0 then
    Result[1] := UpCase(S[1]);
  for I := 2 to Length(S) do
    Result[I] := DownCase(S[I]);
end;
```

Смотрите также

Типы AnsiChar, Char, WideChar.

Uses, ключевое слово

Синтаксис

```
uses Имя модуля, ...;
uses Имя модуля in Имя файла, ...
```

Описание

С помощью ключевого слова uses перечисляются имена модулей, импортируемых в данный модуль, программу или библиотеку. Объявление uses необязательно, но если оно есть, то должно быть первым объявлением в программе, библиотеке, разделе интерфейса или разделе реализации модуля.

Советы и приемы

- В исходном файле программы или библиотеки можно применять вторую форму синтаксиса uses, в которой *Имя файла* является строковым литералом, указывающим на имя файла, содержащего исходный текст модуля. Эта информация нужна компилятору для поиска компилируемого файла, а IDE – для управления проектом.
- Любые изменения в разделе интерфейса модуля могут привести к перекомпиляции всех зависимых от него модулей. Зависимый модуль перекомпилируется, если он использует одно из объявлений из раздела интерфейса, изменившихся по сравнению с исходным файлом. В большом проекте можно уменьшить время перекомпиляции, включая модули в объявление uses раздела реализации, когда это возможно.
- Идентификаторы, экспортируемые модулем в его разделе интерфейса, доступны всем другим модулям, использующим первый. Delphi выполняет просмотр модулей uses в порядке их перечисления, поэтому модули, указанные позднее, имеют преимущество по сравнению с модулями, перечисленными раньше. Если требуется использовать идентификатор из модуля, перечисленного раньше, можно уточнить имя идентификатора, указав имя модуля с точкой (.).

Пример

```
uses SysUtils, Windows;

// И в модуле Windows, и в модуле SysUtils определяется  функция DeleteFile.
// Т. к. Windows указан позже в списке uses, его DeleteFile имеет
// преимущество по сравнению с DeleteFile из SysUtils. Можно
// вызвать DeleteFile из SysUtils, указав перед именем функции
// имя модуля, например:
procedure RemoveFile(const FileName: string);
begin
  SysUtils.DeleteFile(FileName);
end;
```

Смотрите также

Ключевые слова `Implementation`, `Interface`, `Library`, `Unit`.

Val, процедура

Синтаксис

```
procedure Val(const S: string; var Result; var Code; Integer);
```

Описание

Процедура `Val` преобразует строку в числовое значение. Аргумент `Result` может быть целой, `Int64` или вещественной переменной. Если преобразование прошло успешно, `Code` равен нулю. Иначе значение `Code` обозначает позицию в строке, где `Val` впервые столкнулась с ошибкой формата. `Val` не является настоящей процедурой.

Советы и приемы

Для преобразования строки в вещественное число вызывайте функции преобразования строк из модуля `SysUtils` вместо `Val`. Проблема состоит в том, что `Val` не обращает внимание на локальные настройки разделителя десятичных знаков, делая процедуру бесполезной в интернациональной среде.

Пример

```
// Запрашиваем от пользователя число и возвращаем число, которое
// он ввел. Если пользователь ввел неверное значение, сообщаем,
// что произошла опечатка, и просим повторить попытку.
function GetNumber(const Prompt: string): Int64;
var
  S: string;
  Code: Integer;
begin
  repeat
    Write(Prompt);
    ReadLn(S);
    Val(S, Result, Code);
    if Code <> 0 then
    begin
      WriteLn(S);
      WriteLn('^':Code, ' неверное значение');
    end;
  until Code = 0;
end;
```

Смотрите также

Процедура `Str`.

Var, ключевое слово

Синтаксис

```
var
  Имя: Тип;
  Name: Тип = Выражение;
  ...
```

Описание

Ключевое слово `var` обозначает объявление одной или более переменных. Если ключевое слово `var` используется на уровне модуля, то объявляются глобальные переменные, жизненный цикл которых совпадает с жизненным циклом модуля в целом. Если ключевое слово `var` располагается в подпрограмме, то объявляются локальные переменные, область действия и жизненный цикл которых ограничен этой подпрограммой.

Можно также указать начальное значение глобальной переменной, но не локальной.

Советы и приемы

- Без указания начального значения глобальная переменная инициализируется нулем (пустой строкой, нулевым указателем и т. п.), локальная переменная не инициализируется.
- Локальные переменные обычно не инициализируются, но Delphi гарантирует, что переменные, требующие работы менеджера памяти, такие как строки, динамические массивы, переменные типа `Variant` и интерфейсы, работают корректно.
- Переменная `Result` представляет собой особый случай. Она инициализируется вызывающей стороной, а не подпрограммой, и начальное значение не всегда равно нулю.

Пример

```
unit Debug;
interface
var
  FileName: string = 'c:\debug.txt';
procedure Log(const Msg: string);
implementation
uses SysUtils;

procedure Log(const Msg: string);
var
  F: TextFile;
  TimeStamp: string;
begin
  AssignFile(F, FileName);
  if FileExists(FileName) then
```

```
    Append(F)
  else
    Rewrite(F);
  try
    TimeStamp := DateTimeToStr(Now);
    WriteLn(F, '[', TimeStamp, '] ', Msg);
  finally
    CloseFile(F);
  end;
end;
end.
```

Смотрите также

Ключевые слова `Const`, `Function`, `Library`, `Procedure`, `Program`, `ThreadVar`, `Unit`, переменные `Result`, `Self`.

VarArrayCreate, функция

Синтаксис

```
function VarArrayCreate(const Bounds: array of Integer;
    VarType: Integer): Variant;
```

Описание

Функция `VarArrayCreate` создает массив значений `Variant`. Первый аргумент определяет количество размерностей и границы каждой размерности. Второй аргумент – это тип значения элементов массива.

Границы массива указываются в виде пар целых чисел: нижней и верхней границы каждой размерности. Массив может иметь до 64 измерений. Если попытаться создать некорректный массив, `VarArrayCreate` генерирует ошибку времени выполнения 24 (`EVariantError`).

`VarArrayCreate` – настоящая функция.

Советы и приемы

- Неоднородный массив можно создать, указав в качестве типа элемента `varVariant`. В этом случае каждый из элементов массива может иметь значение `Variant` любого типа.
- Нельзя создать массив значений типов `varEmpty` или `varNull`.

Пример

```
// Возвращает массив Variant, содержащий 12 случайных значений.
function RandomDozen: Variant;
var
  I: Integer;
begin
  Result := VarArrayCreate([1, 12], varDouble);
  for I := VarArrayLowBound(Result) to VarArrayHighBound(Result) do
    Result[I] := Random;
```

```
end;
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayDimCount, функция

Синтаксис

```
function VarArrayDimCount(const V: Variant): Integer;
```

Описание

Функция VarArrayDimCount возвращает количество размерностей массива Variant. Если V не является массивом Variant, функция возвращает ноль.

VarArrayDimCount – настоящая функция.

Пример

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 12], varDouble);
  WriteLn(VarArrayDimCount(V)); // Печатает 1
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayHighBound, функция

Синтаксис

```
function VarArrayHighBound(const V: Variant; Dim: Integer): Integer;
```

Описание

Функция VarArrayHighBound возвращает верхнюю границу размерности Dim массива Variant V. Размерность Dim должна находиться в диапазоне от 1 до VarArrayDimCount. Если она выходит за этот диапазон или если V не является массивом Variant, функция выдает ошибку времени выполнения 20 (EVariantError).

VarArrayHighBound – настоящая функция.

Пример

Пример рассмотрен в описании функции VarArrayCreate.

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayLock, функция

Синтаксис

```
function VarArrayLock(var V: Variant): Pointer;
```

Описание

Функция VarArrayLock блокирует размерности массива Variant, чтобы их нельзя было изменить, и возвращает указатель на простой массив с содержимым массива Variant. Массив данных организован так, что самый левый индекс изменяется первым (развертывание по строкам). Заметьте, что обычные массивы Паскаля хранятся в порядке развертывания по столбцам, и первым изменяется самый правый индекс.

VarArrayLock не является настоящей функцией.

Советы и приемы

- Если требуется выполнить операцию над всеми элементами массива Variant, сначала заблокируйте массив для повышения производительности.
- Массив данных содержит фактические значения всех элементов массива Variant. Проверьте, что тип элемента массива совпадает с ожидаемым и преобразуйте указатель на данные в соответствующий тип.
- Пока массив заблокирован, нельзя изменить его размер, но можно изменить содержимое. Изменения в элементах массива отражаются на массиве данных, полученном от функции VarArrayLock.
- Можно заблокировать массив более одного раза. Для разблокирования массива необходимо вызвать VarArrayUnlock столько раз, сколько была вызвана VarArrayLock.

Пример

```
// Перестановка двух значений типа double.
procedure DSwap(var A, B: Double);
var
  Tmp: Double;
begin
  Tmp := A;
  A := B;
  B := Tmp;
end;

// Транспонирование матрицы значений varDouble. Для максимальной
```

```
// производительности блокируем массив Variant и обращаемся к данным
// в чистом виде.
procedure Transpose(var M: Variant);
type
  TDoubleArray = array[0..MaxInt div SizeOf(Double)-1] of Double;
  PDoubleArray = ^TDoubleArray;
var
  I, J: Integer;
  Data: PDoubleArray;
  Dim1: Integer;
  Low1, Low2: Integer;
  High1, High2: Integer;
begin
  Assert(VarType(M) = varArray or varDouble);
  Data := VarArrayLock(M);
  try
    Low1 := VarArrayLowBound(M, 1);
    Low2 := VarArrayLowBound(M, 2);
    High1 := VarArrayHighBound(M, 1);
    High2 := VarArrayHighBound(M, 2);
    Dim1 := High1 - Low1 + 1;

    // Массив Data представляет собой одномерный вектор, но
    // его значения все равно организованы в виде матрицы. Переставляем
    // элементы матрицы M[J, I] и M[I, J] с помощью массива Data.
    //
    for I := Low1 to High1 do
      for J := I to High2 do
          DSwap(Data[J-Low2 + (I-Low1)*Dim1],
                Data[I-Low2 + (J-Low1)*Dim1]);
  finally
    VarArrayUnlock(M);
  end;
end;
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, **функции** VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, **процедура** VarArrayRedim.

VarArrayLowBound, функция

Синтаксис

```
function VarArrayLowBound(const V: Variant; Dim: Integer): Integer;
```

Описание

Функция VarArrayLowBound **возвращает нижнюю границу размерности** Dim **массива** Variant V**. Размерность** Dim **должна находиться в диапазоне от 1 до** VarArrayDimCount**. Если** Dim **выходит за этот диапазон или если** V

не является массивом Variant, функция выдает ошибку времени выполнения 20 (EVariantError).

VarArrayLowBound – настоящая функция.

Пример

Пример рассмотрен в описании функции VarArrayCreate.

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayLock, VarArrayHighBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayOf, функция

Синтаксис

```
function VarArrayOf(const Values: array of Variant): Variant;
```

Описание

Функция VarArrayOf создает одномерный массив Variant, содержащий те же значения, что и открытый массив, переданный в качестве аргумента. Результирующий массив имеет нижнюю границу, равную нулю, и верхнюю границу, равную размеру массива минус 1.

VarArrayOf – настоящая функция.

Пример

```
// Следующие два примера выполняют одинаковые действия.
V := VarArrayOf([1.0, 20, 'Testing']);

V := VarArrayCreate([0, 2], varVariant);
V[0] := 1.0;
V[1] := 20;
V[2] := 'Testing';
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayRedim, процедура

```
procedure VarArrayRedim(var V: Variant; HighBound: Integer);
```

Описание

Процедура VarArrayRedim выполняет изменение размера самого правого измерения массива Variant V. Верхняя граница старшего измерения меняется на HighBound.

VarArrayRedim не является настоящей процедурой.

Советы и приемы

- Нельзя изменить размер массива Variant, пока массив заблокирован.
- Изменение размера массива сохраняет максимально возможное число его элементов. Если массив увеличивается, новые элементы инициализируются нулями, если тип элементов массива числовой, Unassigned – для массива varVariant и пустыми строками для массива со строковыми элементами.
- Невозможно изменить размер массива по его ссылке (т. е. по результату вызова VarArrayRef). В процедуру VarArrayRedim должен быть передан фактический массив.

Пример

```
// Читает числа, введенные пользователем, в саморасширяющийся
// массив. Эта функция - неэффективная, но ясная
// демонстрация функции VarArrayRedim.
function GetArray: Variant;
var
  Number: Integer;
begin
  Result := VarArrayCreate([1, 0], varInteger);
  while not Eof do
  begin
    ReadLn(Number);
    VarArrayRedim(Result, VarArrayHighBound(Result, 1) + 1);
    Result[VarArrayHighBound(Result, 1)] := Number;
  end;
end;
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType.

VarArrayRef, функция

Синтаксис

```
function VarArrayRef(const V: Variant): Variant;
```

Описание

Функция VarArrayRef создает новый массив Variant с установленным битом varByRef в поле VarType. Новый Variant непосредственно ссылается на данные массива V. Любые изменения в размерах или содержимом V также отражаются на новом массиве.

VarArrayRef – настоящая функция.

Советы и приемы

- Создав ссылку на массив, следует позаботиться, чтобы ссылка не могла «пережить» исходный массив. Если оба Variant имеют одинаковые области действия, все нормально, однако не возвращайте ссылку на массив из функции, для которой исходный массив является локальным.
- Нельзя изменить размеры массива по его ссылке.

Пример

```
var
  Orig, Ref: Variant;
begin
  Orig := VarArrayCreate([1, 10], varInteger);
  Ref := VarArrayRef(Orig);
  Orig[1] := 42;
  WriteLn(Ref[1]);                      // Печатает 42
  VarArrayRedim(Orig, 5);               // Отражается также и на Ref
  WriteLn(VarArrayHighBound(Ref, 1));   // Печатает 5
end;                                    // Ref и Orig очищаются
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarArrayUnlock, процедура

Синтаксис

```
procedure VarArrayUnlock(var V: Variant);
```

Описание

Процедура VarArrayUnlock разблокирует массив Variant, который ранее был заблокирован функцией VarArrayLock.

Массив может быть заблокирован более одного раза. Для разблокирования массива следует вызвать процедуру VarArrayUnlock столько же раз, сколько была вызвана VarArrayLock.

VarArrayUnlock — настоящая процедура.

Пример

Пример рассмотрен в функции VarArrayLock.

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarIsArray, VarType, процедура VarArrayRedim.

VarAsType, функция

Синтаксис

```
function VarAsType(const V: Variant; VarType: Integer): Variant;
```

Описание

Функция VarAsType выполняет преобразование V в другой тип Variant. VarType не может содержать битов varArray и varByRef. Если преобразование V к требуемому типу невозможно, VarAsType генерирует ошибку времени выполнения 15 (EVariantError).

VarAsType – настоящая функция.

Советы и приемы

- VarAsType – версия процедуры VarCast, возвращающая значение. Для того чтобы присвоить это значение переменной типа OleVariant, следует вызвать процедуру VarCast, т. к. она обеспечивает дополнительную проверку типов.
- Если Source уже имеет тип VarType, VarAsType выполняет простое копирование.

Пример

```
Int := 1;
Float := VarAsType(Int, varDouble);
```

Смотрите также

Типы OleVariant, TVarData, Variant, процедуры VarCast, VarCopy, функции VarIsArray, VarIsEmpty, VarIsNull.

VarCast, процедура

Синтаксис

```
procedure VarCast(var Dest: Variant; const Source: Variant;
                  VarType: Integer);
procedure VarCast(var Dest: OleVariant; const Source: Variant;
                  VarType: Integer);
```

Описание

Процедура VarCast выполняет преобразование Source в новый тип Variant, сохраняя результат в Dest. VarType не может содержать биты varArray и varByRef.

Если Dest имеет тип OleVariant, новый тип должен быть OLE-совместимым. В частности, нельзя преобразовать значение к типу varString, а необходимо использовать varOleStr. В противном случае VarCast выдаст ошибку времени выполнения 15 (EVariantError).

VarCast не является настоящей процедурой.

Советы и приемы

- Процедура VarCast аналогична функции VarAsType. Разница состоит в том, что VarCast выполняет дополнительные проверки при преобразовании в OleVariant.
- Если Source уже имеет тип VarType, VarCast выполняет простое копирование.

Пример

```
var
  V: Variant;
  O: OleVariant;
begin
  V := 'Это тест';     // VarType(V) = varString
  VarCast(O, V, varOleStr);
```

Смотрите также

Типы OleVariant, TVarData, Variant, функции VarAsType, VarIsArray, VarIsEmpty, VarIsNull, процедура VarCopy.

VarClear, процедура

Синтаксис

```
procedure VarClear(var V: Variant);
```

Описание

Процедура VarClear освобождает старое значение V и устанавливает тип V в varEmpty.

VarClear не является настоящей процедурой.

Пример

```
// Следующие два предложения выполняют одинаковые действия:
VarClear(V);
V := Unassigned;
```

Смотрите также

Переменные ClearAnyProc, Unassigned, тип Variant, функция VarIsEmpty.

VarCopy, процедура

Синтаксис

```
procedure VarCopy(var Dest: Variant; const Source: Variant);
```

Описание

Процедура VarCopy очищает Dest и копирует в нее значение Source.

VarCopy не является настоящей процедурой.

Советы и приемы

- Присваивание `Dest := Source` в конце концов приводит к вызову `VarCopy`, однако иногда оно может работать более эффективно.
- Если `Dest` имеет тип `OleVariant`, а `Source` — `Variant` со значением типа `varString`, `VarCopy` автоматически преобразует тип `Dest` к `varOleStr`.

Смотрите также

Типы `OleVariant`, `Variant`.

VarDispProc, переменная

Синтаксис

```
var VarDispProc: Pointer;

procedure Invoke(Result: PVariant; const Instance: Variant;
  CallDesc: PCallDesc; Params: Pointer); cdecl;
VarDispProc := @Invoke;
```

Описание

Переменная `VarDispProc` указывает на процедуру, вызывающую метод `IDispatch` по ссылке на значение типа `Variant`. Когда переменная типа `Variant` ссылается на COM-сервер, она фактически содержит интерфейс `IDispatch` и имеет тип `varDispatch`. При вызове программой метода или обращении к свойству COM-сервера объект, ссылка на который содержится в `Variant`, должен вызвать метод `IDispatch.Invoke`. Delphi делает это с помощью процедуры `VarDispProc`.

`Result` — это указатель на `Variant`, по которому должно быть записано возвращаемое функцией значение или `nil`, если результат не требуется. `Instance` — это `Variant`, содержащий интерфейс `IDispatch`, которым был вызван метод или свойство. `CallDesc` указывает на запись, описывающую вызываемый метод, `Params` указывает на массив, описывающий параметры метода.

Советы и приемы

- Не присваивайте `nil` переменной `VarDispProc`. В отличие от аналогичных процедурных указателей, если `VarDispProc` равна `nil`, Delphi не предусматривает на этот случай стандартного поведения, и произойдет ошибка нарушения доступа.
- Начальное значение `VarDispProc` указывает на процедуру, которая выдает ошибку времени выполнения 17 (`EVariantError`).
- В модуле `ComObj` для этого указателя определяется процедура, выполняющая все необходимые действия. Там же описан и тип `PCallDesc`.

Смотрите также

Переменная DispCallByIDProc, ключевые слова Dispinterface, Interface, интерфейс IDispatch, типы OleVariant, Variant.

VarFromDateTime, функция

Синтаксис

```
function VarFromDateTime(DateTime: TDateTime): Variant;
```

Описание

Функция VarFromDateTime создает Variant-значение типа varDate для данного значения TDateTime. VarFromDateTime — настоящая функция.

Советы и приемы

- VarFromDateTime служит для преобразования TDateTime в Variant. При попытке присвоить Variant-значение типа TDateTime Delphi создаст Variant типа varDouble, т. к. TDateTime фактически является вещественным типом.
- Формат TDateTime, принятый в Delphi, соответствует стандарту OLE, поэтому если создать Variant типа varDate, непосредственно присвоив значение полю VType записи TVarData, можно присвоить значение TDateTime полю VDate.
- Для получения дополнительного контроля над форматированием строк TDateTime, вызовите VarToDateTime и затем воспользуйтесь для форматирования функциями из модуля SysUtils. Преобразование значения varDate в varString всегда выполняется в коротком формате даты и времени.

Пример

```
// Следующие два примера выполняют одинаковые действия.
V := VarFromDateTime(Now);

VarClear(V);
TVarData(V).VType := varDate;
TVarData(V).VDate := Now;
```

Смотрите также

Типы TDateTime, Variant, процедура VarClear, функция VarToDateTime.

Variant, тип

Синтаксис

```
type Variant;
```

Описание

Тип Variant – динамический тип. Переменная Variant может изменять свой тип во время выполнения программы, получая иногда целое значение, иногда – строковое, а иногда – массив.

Delphi по необходимости автоматически преобразует числа, строки, интерфейсы и другие типы к типу Variant и обратно. Существует несколько функций для других манипуляций с типом Variant.

В главе 6 «Системные константы» перечислены все возможные типы значений Variant.

Советы и приемы

- Применение Variant дает гибкость, цена которой – производительность. Даже простая арифметика со значениями типа Variant значительно сложнее и занимает больше времени, чем арифметика с типизированными переменными. Любая ссылка на значение типа Variant должна проверяться во время выполнения программы, что может значительно увеличить накладные расходы, если Variant используется часто.
- Тип Variant прост в применении, однако требуется знать некоторые тонкости. Например, Unassigned и Null обозначают абсолютно разные значения и концепции в Variant. Проиллюстрировать эту разницу можно следующим образом: представьте себе Variant как ящик, в котором находится лист бумаги. На бумаге записано значение Variant: число, строка, время суток и т. д. Null – это чистый лист бумаги. Unassigned – пустой ящик.
- Для преобразования TDateTime в Variant и обратно используйте функции VarFromDateTime и VarToDateTime.
- Чаще всего тип Variant применяется для вызова сервера OLE-автоматизации, когда во время компиляции отсутствует диспетчерский интерфейс для этого сервера. Вызываются методы сервера, но Delphi не может найти эти методы на этапе компиляции и выполняет проверку во время выполнения программы. Обычно нельзя вызвать метод, не объявив в базовом классе или интерфейсе имя метода, его аргументы и возвращаемое значение, но при использовании типа Variant можно отложить все детали на время выполнения, когда Delphi сможет осуществить поиск метода и его сигнатуры с помощью интерфейса IDispatch.

Пример

```
var
  WordApp: Variant;
  A, X: Variant;
begin
  WordApp := CreateOleObject('Word.Basic');
  WordApp.FileNew;
```

```
  X := Pi;
  X := X / 2.0;   // Одновременное использование Variant и числа
                  // в выражениях.
  ShowMessage(X); // Delphi автоматически преобразует Variant
                  // в строку.

  // Создаем матрицу 3x3.
  A := VarArrayCreate([1, 3, 1, 3], varDouble);
  for I := 1 to 3 do
    for J := 1 to 3 do
      A[I, J] := I + J;
end;
```

Смотрите также

Переменные ChangeAnyProc, ClearAnyProc, EmptyParam, Null, RefAnyProc, Unassigned, VarDispProc, **типы** OleVariant, TVarData, **функции** VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarAsType, VarFromDateTime, VarIsArray, VarIsEmpty, VarIsNull, VarToDateTime, VarToStr, VarType, **процедуры** VarArrayRedim, VarArrayUnlock, VarCast, VarClear, VarCopy.

VarIsArray, функция

Синтаксис

```
function VarIsArray(const V: Variant): Boolean;
```

Описание

Функция VarIsArray **возвращает истину, если** Variant V **является массивом, и ложь – в противном случае. Массив** Variant **имеет установленный бит** varArray **в поле** VarType.

VarIsArray – **настоящая функция.**

Пример

```
// Возвращает сумму всех чисел одномерного массива, или, если
// аргумент не является массивом, возвращает его числовое
// значение.
function Sum(const V: Variant): Variant;
var
  I: Integer;
begin
  if VarIsArray(V) then
  begin
    Result := 0.0;
    Assert(VarArrayDimCount(V) = 1);
    for I := VarArrayLowBound(V, 1) to VarArrayHighBound(V, 1) do
      Result := Result + V[I];
  end
  else
```

```
    Result := V + 0.0; // Для того, чтобы результат был
                       // числовым.
end;
```

Смотрите также

Типы TVarArray, TVarArrayBound, Variant, функции VarArrayCreate, VarArrayDimCount, VarArrayHighBound, VarArrayLock, VarArrayLowBound, VarArrayOf, VarArrayRef, VarArrayUnlock, VarIsArray, VarType, процедура VarArrayRedim.

VarIsEmpty, функция

Синтаксис

```
function VarIsEmpty(const V: Variant): Boolean;
```

Описание

Функция VarIsEmpty возвращает True, если Variant V имеет тип varEmpty, и False для любого другого значения Variant. Начальный тип Variant — varEmpty.

VarIsEmpty — настоящая функция.

Пример

```
// Преобразует Variant в TDateTime, но если Variant не
// содержит значения, возвращает текущую дату и время.
function GetDate(V: Variant): TDateTime;
begin
  if VarIsEmpty(V) or VarIsNull(V) then
    Result := Now
  else
    Result := VarToDateTime(V);
end;
```

Смотрите также

Переменная Unassigned, процедура VarClear, тип Variant, функции VarIsNull, VarType.

VarIsNull, функция

Синтаксис

```
function VarIsNull(const V: Variant): Boolean;
```

Описание

VarIsNull возвращает True, если Variant V содержит значение типа varNull, и False — для любого другого значения. VarIsNull — настоящая функция.

Пример

Примеры приводятся в описании Null и VarIsEmpty.

Смотрите также

Переменная Null, тип Variant, функции VarIsEmpty, VarType.

VarToDateTime, функция

Синтаксис

```
function VarToDateTime(const V: Variant): TDateTime;
```

Описание

Функция VarToDateTime преобразует Variant в TDateTime. Если Variant не может быть преобразован в varDate, функция генерирует ошибку времени выполнения 15 (EVariantError).

Советы и приемы

- Присваивание значения типа Variant переменной типа TDateTime не работает: т. к. TDateTime – вещественный тип, произойдет попытка преобразовать Variant в varDouble, а не в varDate.
- VarToDateTime в конечном счете вызывает функцию VariantChangeTypeEx из Windows API для разбора строки и извлечения даты и времени. С подробностями можно ознакомиться в документации по Platform SDK. VarToDateTime понимает значительно больше форматов, чем StrToDateTime.

Пример

```
// Выделение даты и времени из строки.
function StringToDateTime(const S: string): TDateTime;
var
  V: Variant;
begin
  V := S;
  Result := VarToDateTime(V);
end;
```

Смотрите также

Типы TDateTime, Variant, функция VarFromDateTime.

VarToStr, функция

Синтаксис

```
function VarToStr(const V: Variant): string;
```

Описание

Функция VarToStr преобразует Variant в строку. Если V равно Null, VarToStr возвращает пустую строку. VarToStr – настоящая функция.

Советы и приемы

Единственное различие между вызовом VarToStr и простым присваиванием Variant строковой переменной обнаруживается, когда значение Variant равно Null. Присваивание Null строковой переменной приводит к ошибке времени выполнения 15 (EVariantError).

Смотрите также

Переменная Null, ключевое слово String, тип Variant, функция VarIsNull.

VarType, функция

Синтаксис

```
function VarType(const V: Variant): Integer;
```

Описание

Функция VarType возвращает код типа Variant. Код типа определяет фактический тип данных Variant, а также то, является Variant массивом или ссылкой.

Список кодов типов Variant приведен в главе 6.

VarType – настоящая функция.

Советы и приемы

Код типа – это битовая маска. Младшие 12 битов определяют код типа, в старших указывается дополнительная информация о типе. Для определения фактического кода типа используйте константу varTypeMask. Возможные модификаторы – это varArray для массива и varByRef для косвенных данных.

Пример

```
// Вызов VarType аналогичен обращению к полю VType записи TVarData.
// Следующие два примера служат для выполнения одной задачи.
// (Обращение к полю работает чуть быстрее, но текст программы труднее
// читать.)
I := VarType(V);

I := TVarData(V).VType;

if VarType(V) = varDateTime then
  TVarData(V).VDate := Now;
```

Смотрите также

Типы TVarData, Variant, функции VarIsArray, VarIsEmpty, VarIsNull.

Virtual, директива

Синтаксис

Объявление метода; virtual;

Описание

С помощью директивы virtual или директивы dynamic можно объявить виртуальный метод в базовом классе. Семантика обеих директив одинакова. Различие состоит лишь в реализации методов и способе их вызова. Подробности рассмотрены в главе 3.

Для переопределения метода производные классы должны использовать директиву override.

Советы и приемы

- Практически всегда директива virtual предпочтительнее, чем dynamic. В большинстве случаев виртуальные методы работают быстрее и занимают меньше памяти, чем динамические.
- Директива virtual должна следовать за директивами reintroduce и overload и предшествовать директивам соглашений о вызовах и директиве abstract (если эти директивы присутствуют в объявлении метода).

Пример

```
type
  WholeNumber = 0..MaxInt;
  TIterator = function(Item: Pointer): Boolean of object;
  // Абстрактный базовый класс для нескольких классов коллекций.
  // Производные классы должны переопределить ряд методов
  // для реализации конкретного поведения.
  TCollection = class
  protected
    constructor Create; virtual;
    function Add(Item: Pointer): WholeNumber; virtual; abstract;
    function GetCount: WholeNumber; virtual; abstract;
    function Delete(Index: WholeNumber): Pointer; virtual; abstract;
  public
    procedure Remove(Item: Pointer); virtual;
    procedure ForEach(Iterator: TIterator); virtual;
  end;
  // Абстрактные классы для нескольких видов двоичных деревьев.
  // Реализует методы двоичного дерева, однако оставляет
  // некоторые детали производным классам.
  TBinaryTree = class(TCollection)
  private
    fRoot: TBinaryNode;
    fCount: Integer;
    constructor Create; override;
  public
```

```
    function Add(Item: Pointer): WholeNumber; override; abstract;
    function GetCount: WholeNumber; override;
    function Delete(Index: WholeNumber): Pointer; override; abstract;
    procedure ForEach(Iterator: TIterator); override;
  end;
  // Сбалансированное двоичное дерево. Обратите внимание на
  // открытый конструктор. Все абстрактные методы реализованы.
  TBalancedBinaryTree = class(TBinaryTree)
  public
    constructor Create; override;
    function Add(Item: Pointer): WholeNumber; override;
    function Delete(Index: WholeNumber): Pointer; override;
  end;
```

Смотрите также

Директивы Abstract, Dynamic, Overload, Override, Reintroduce, тип Class.

While, ключевое слово

Синтаксис

```
while Выражение do Оператор
```

Описание

Предложение while работает так же, как в стандартном Паскале. Пока *Выражение* равно истине, Delphi многократно выполняет *Оператор*. Если *Выражение* ложно при первой проверке, *Оператор* не будет исполнен.

Смотрите также

Тип Boolean, процедуры Break, Continue, ключевые слова Do, For, Repeat.

WideChar, тип

Синтаксис

```
type WideChar = #0..#65535;
```

Описание

Тип WideChar представляет 16-битовый символ в формате Unicode. Unicode – надмножество типа AnsiChar: значения между #0 и #255 совпадают в обоих наборах символов, однако в Unicode присутствует много дополнительных символов для поддержки разнообразных языков.

Смотрите также

Типы AnsiChar, Char, PWideChar, WideString, функции Chr, OleStrToString, Ord, StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, ключевое слово String.

WideCharLenToString, функция

Синтаксис

```
function WideCharLenToString(Source: PWideChar; Len: Integer): string;
```

Описание

Функция WideCharLenToString преобразует строку Unicode из Len символов (не байт) в строку многобайтовых символов. Дополнительную информацию по Unicode и многобайтовым символам можно увидеть в описании типа WideString.

Смотрите также

Функции OleStrToString, StringToOleStr, StringToWideChar, WideCharToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

WideCharLenToStrVar, процедура

Синтаксис

```
procedure WideCharLenToStrVar(Source: PWideChar; Len: Integer;
   var Dest: string);
```

Описание

Процедура WideCharLenToStrVar преобразует строку Unicode из Len символов (не байт) в строку многобайтовых символов, которая записывается в Dest. Дополнительную информацию о Unicode и многобайтовых символах можно найти в описании типа WideString.

Смотрите также

Функции OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

WideCharToString, функция

Синтаксис

```
function WideCharToString(Source: PWideChar): string;
```

Описание

Функция WideCharToString преобразует строку Unicode, ограниченную #0, в строку многобайтовых символов. Дополнительную информацию о Unicode и многобайтовых символах можно найти в описании типа WideString.

Смотрите также

Функции OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, процедуры OleStrToStrVar, WideCharLenToStrVar, WideCharToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

WideCharToStrVar, процедура

Синтаксис

```
procedure WideCharToStrVar(Source: PWideChar; var Dest: string);
```

Описание

Процедура WideCharToString преобразует строку Unicode, ограниченную #0, в строку многобайтовых символов и сохраняет ее в Dest. Дополнительная информация о Unicode и многобайтовых символах приведена в описании типа WideString.

Смотрите также

Функции OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, WideCharLenToStrVar, типы PWideChar, WideChar, WideString, ключевое слово String.

WideString, тип

Синтаксис

```
type WideString;
```

Описание

Тип WideString аналогичен AnsiString, но вместо строки символов AnsiChar в WideString хранится строка в формате Unicode, состоящая из 16-битовых символов WideChar.

WideString отслеживает собственную длину и автоматически дописывает символ #0 к концу строки, чтобы она могла быть легко преобразоваа в PWideChar.

Внутренний формат хранения WideString – это указатель на запись; только указатель фактически указывает на поле Data, а Length хранится в четырех байтах, предшествующих указателю WideString.

```
type
  // Приводится логическая структура WideString, однако это объявление
  // лишь описательное и не предназначено для компиляции.
  TWideString = record
    Length: LongWord;
    Data: array[1..Length+1] of WideChar;
  end;
```

Советы и приемы

- Delphi автоматически управляет памятью переменных типа WideString, как и AnsiString. В отличие от AnsiString, WideString не имеет счетчика ссылок, и любое присваивание значения WideString приводит к созданию полной копии строки. Поэтому использование типа WideString менее эффективно, чем AnsiString.
- Delphi автоматически выполняет преобразование между AnsiString и WideString, из PWideChar в WideString и из массива с нулевым нижним индексом и элементами типа WideChar в WideString.
- Можно преобразовать WideString в PWideChar, что часто требуется при вызове функций Windows API, но необходимо предпринимать те же меры предосторожности, что и при преобразовании AnsiString в PChar. Delphi автоматически уничтожает строку, когда она больше не нужна, и при этом значение указателя PChar или PWideChar становится некорректным. Это не страшно, если указатель больше нигде не используется, поэтому применяйте PWideChar при вызове функции API, но не сохраняйте указатель PWideChar для дальнейшего употребления.
- При преобразовании WideString в AnsiString Delphi использует кодовую страницу ANSI и преобразование не-ANSI-символов, принятое по умолчанию в Windows. Можно управлять процессом преобразования, вызывая функцию WideCharToMultiByte из Windows API напрямую.
- После преобразования «широкой» строки в «узкую» последняя может содержать многобайтовые символы. В модуле SysUtils есть несколько функций для работы с многобайтовыми строками. Подробности приводятся в приложении В.
- VCL не поддерживает Unicode. Для того чтобы использовать управляющие элементы Unicode в Windows NT, приходится создавать собственные компоненты или искать решение от стороннего разработчика.

Смотрите также

Типы AnsiString, PWideChar, WideChar, функции Length, OleStrToString, StringToOleStr, StringToWideChar, WideCharLenToString, WideCharToString, процедуры OleStrToStrVar, SetLength, SetString, WideCharLenToStrVar, WideCharToStrVar, ключевое слово String.

With, ключевое слово

Синтаксис

```
with Выражение do Оператор
```

Описание

Конструкция `with` добавляет запись, объект, класс или интерфейс к диапазону символьных имен. Delphi выполняет поиск имен в следующем порядке:

- Поля записей, объектов, классов или интерфейсов, перечисленных в конструкции `with`, начиная с последней или самой внутренней и продолжая более ранними или внешними конструкциями `with`.
- Локальные переменные и параметры подпрограммы, в том числе неявно определенные переменные, такие как `Result` и `Self`.
- Поля переменной `Self` (если подпрограмма является методом). Можно рассматривать каждый метод как неявное предложение `with Self do`, содержащее тело метода.
- Глобальные переменные того же модуля.
- Глобальные переменные других модулей, начиная с последнего перечисленного в объявлении `uses`.

Советы и приемы

- Будьте осторожны с конструкцией `with`. Беспорядочное ее применение скрывает смысл кода и усложняет идентификацию объектов, к которым относятся перечисленные методы и свойства. Изменения в записи, объекте, классе или интерфейсе может привести к иной интерпретации идентификатора. Если повезет, это приведет к синтаксической ошибке; если нет – изменения останутся незамеченными до тех пор, пока программа не станет вести себя некорректно.
- Тем не менее, конструкция `with` имеет свои области применения. Она может оказаться удобным средством, позволяющим обойтись без создания временной переменной, когда такая переменная не добавляет ясности в код.

Пример

```
// Когда пользователь нажимает кнопку, элемент добавляется
// в список и вызывается процедура редактирования, чтобы
// пользователь мог присвоить этому элементу понятное имя.
procedure TMyForm.Button1Click(Sender: TObject);
begin
  with ListView1.Items.Add do
  begin
    Caption := 'Новое имя'; // Обращается к заголовку нового элемента
    EditCaption;            // списка, а не к Self.Caption.
  end;
end;
```

Смотрите также

Ключевое слово `Do`, переменная `Self`.

Word, тип

Синтаксис

```
type Word = 0..65535;
```

Описание

Тип Word представляет собой 16-битовые целые значения без знака.

Дополнительную информацию по целым типам можно найти в описании типа Integer.

Смотрите также

Типы Integer, SmallInt.

WordBool, тип

Синтаксис

```
type WordBool;
```

Описание

Тип WordBool – это логический тип, размер которого равен размеру Word. Значение WordBool ложно, если его порядковое значение равно нулю, и истинно в противном случае. WordBool использует –1 в качестве порядкового значения для констант истинности (WordBool(True)).

Советы и приемы

- Значения типа WordBool применимы везде, где и Boolean. Больше всего они подходят для интерфейса с C и C++, где любое ненулевое целое рассматривается как истина.
- ByteBool и LongBool аналогичны WordBool, но имеют другой размер.

Смотрите также

Ключевые слова And, Not, Or, Xor, типы Boolean, ByteBool, LongBool.

Write, директива

Синтаксис

```
property Имя: Тип ... write Setter;
```

Описание

Директива write сообщает Delphi о способе изменения значения свойства. *Setter* может быть ссылкой на поле, именем метода текущего или родительского класса.

Если *Setter* – поле, его тип должен совпадать с типом свойства. Применяются обычные правила доступа, поэтому поле не может быть из сек-

ции private родительского класса, если только родительский класс не определяется в том же модуле.

Поле может быть составного типа (записью или массивом), и тогда в *Setter* должно быть указано имя записи или элемент массива соответствующего типа (с константным индексом). Записи и массивы могут быть вложенными.

Если *Setter* – метод, он должен быть процедурой, тип аргумента которой совпадает с типом свойства. Метод может быть статическим или виртуальным, но не может быть динамическим или обработчиком сообщения.

Если свойство индексированное или является массивом, *Setter* должен быть методом. Первым его параметром должно быть значение индекса типа Integer. Последующие аргументы являются индексами массива. Тип каждого из аргументов *Setter* должен совпадать с соответствующим индексом массива. Последний аргумент – новое значение свойства.

Когда пользователь присваивает значение свойству, Delphi присваивает значение полю *Setter* или вызывает метод *Setter*.

Советы и приемы

- Если *Setter* – поле, Delphi компилирует все ссылки на свойство в прямое обращение к полю, так что падение производительности при использовании свойства вместо поля отсутствует.
- Если публикуемое свойство имеет объектный тип (отличный от TComponent и его потомков), следует определить метод *Setter*, вызывающий метод Assign данного объекта:

```
type
  TDemo = class(TComponent)
  private
    Font: TFont;
    procedure SetFont(NewFont: TFont);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
  published
    property Font: TFont read fFont write SetFont;
  end;

procedure TDemo.SetFont(NewFont: TFont);
begin
  Font.Assign(NewFont);
end;
```

- В хорошем стиле программирования принято определять все поля закрытыми и создавать защищенные, открытые и публикуемые свойства для обращения к этим полям. Если позднее потребуется

изменить класс, можно будет заменить поле на метод, не влияя на код, зависящий от класса и его свойства.

Пример

Примеры рассмотрены в описании ключевого слова property.

Смотрите также

Ключевые слова Class, Property, директивы Index, Read.

Write, процедура

Синтаксис

```
procedure Write(var F: File; var Value; ...);
procedure Write(Expr:Width:Precision; ...);
procedure Write(var F: TextFile; Expr:Width:Precision; ...);
```

Описание

Процедура Write предназначена для записи текста и других данных в файл. Записывая двоичный файл, необходимо указать переменную того же типа, что и базовый тип файла. Можно сохранить несколько записей, перечислив соответствующие переменные в аргументах процедуры Write.

При записи текстового файла можно использовать строковые, числовые, символьные и логические значения. С каждым значением могут быть указаны любые целые выражения *Width* и *Precision*. *Width* определяет минимальный размер символьного представления Value, *Precision* – количество знаков после десятичной точки для вещественного числа.

Процедура Write записывает столько символов, сколько требуется, поэтому *Width* – лишь предлагаемая минимальная ширина. Если числу требуется меньше чем *Width* символов, Write дополняет его пробелами слева. Если *Width* не указывается, Delphi использует 1 в качестве минимальной длины для целых чисел, а вещественные числа выводит в следующем 26-символьном формате:

```
' -1.1234567890123456E+1234'
```

Если *Expr* – вещественное значение, Write сокращает количество десятичных знаков, чтобы поместить значение в строку не более *Width* символов. Однако Write всегда использует, как минимум, одну цифру после десятичной точки. Можно задать значение *Precision*, которое сообщает процедуре Write, сколько знаков после десятичной точки следует указать. Если указывается *Precision*, Write использует форму записи числа с фиксированной точкой вместо экспоненциальной формы.

Советы и приемы

- Если файл не был назначен процедурой AssignFile, Write генерирует ошибку ввода/вывода.

- Если файл не был открыт для записи, Write выдает ошибку ввода/вывода 104.
- Если запись была прервана из-за того, что диск переполнен, Write иногда выдает ошибку Windows Error_Disk_Full (112); иногда – ошибку ввода/вывода 101.
- Если в первом аргументе не указана файловая переменная, Write выполняет запись в текстовый файл Output.
- Процедура Str выполняет те же действия, что и Write, кроме того, что «записывает» одиночную величину в строку, а не в файл.

Пример

```
var
  D1, D2: TSomeRecord;
  F: File of TSomeRecord;
begin
  ...
  Write(F, D1, D2);
```

Смотрите также

Процедуры BlockWrite, Read, Str, WriteLn, ключевое слово File, функция IOResult, переменная Output, тип TextFile.

WriteLn, процедура

Синтаксис

```
procedure WriteLn(Expr:Width:Precision; ...);
procedure WriteLn(var F: TextFile; Expr:Width:Precision; ...);
```

Описание

WriteLn действует аналогично процедуре Write за исключением того, что печатает конец строки после вывода своих аргументов. Конец строки – это перевод каретки, за которым следует перевод строки (#13#10). Без указания других аргументов, кроме текстового файла, WriteLn печатает только конец строки.

Советы и приемы

- Если файл не был назначен процедурой AssignFile, Write генерирует ошибку ввода/вывода 102.
- Если файл не был открыт для записи, Write выдает ошибку ввода/вывода 104.
- Если запись была прервана из-за того, что диск переполнен, Write иногда выдает ошибку Windows Error_Disk_Full (112), иногда – ошибку ввода/вывода 101.
- Если в первом аргументе не указана файловая переменная, Write выполняет запись в текстовый файл Output.

- Информация о форматировании выходных данных приведена в описании процедуры Write.

Пример

```
// Печать таблицы квадратных корней
WriteLn('Число Квадратный корень');
for I := 1 to 10 do
  WriteLn(I:3, '':2, Sqrt(I):18:15);
```

Смотрите также

Процедуры BlockWrite, ReadLn, Write, функция IOResult, переменная Output, тип TextFile.

Writeonly, директива

Синтаксис

```
property Name: Type writeonly;
```

Описание

Директива writeonly применима только к свойствам в объявлении диспетчерского интерфейса. Подробности рассмотрены в описании ключевого слова dispinterface.

Смотрите также

Ключевые слова Dispinterface, Property, директива Readonly.

Xor, ключевое слово

Синтаксис

Логическое выражение xor *Логическое выражение*
Целое выражение xor *Целое выражение*

Описание

Оператор xor выполняет над своими операндами операцию исключающего «ИЛИ». Если операнды имеют тип Boolean, оператор возвращает результат типа Boolean: True – если операнды различны, и False, если они одинаковы.

Целый xor оперирует над каждым битом своих операндов, устанавливая бит результата в 1, если соответствующие биты в обоих операндах различны, и в 0 – если оба операнда имеют идентичные биты. Если один операнд меньше другого, Delphi расширяет меньший операнд слева нулевыми битами. Результат имеет размер большего операнда.

Советы и приемы

- Для операндов логического типа xor работает как сравнение на неравенство. В большинстве случаев оператор <> выглядит понятнее, чем xor.

- Операция xor обратима, поэтому многие используют ее для шифрования паролей. Такое применение xor аналогично запиранию двери при помощи жевательной резинки. Этот вариант не годится, т. к. легко взламывается. Поищите в Интернете, и вы найдете подходящие бесплатные решения для более безопасного шифрования паролей.

Пример

```
var
  I, J: Integer;
begin
  I := $25;
  J := $11;
  WriteLn(I xor J); // Writes 52 (which is $34)
...
```

Смотрите также

Ключевые слова And, Not, Or, Shr, Shr, типы Boolean, ByteBool, LongBool, WordBool, директива компилятора $B, директива $BoolEval.

Глава 6

Системные константы

Материал данной главы выведен за рамки главы 5 «Справочник по языку» с целью облегчения поиска необходимой информации. Для того чтобы не загромождать предыдущую главу всеми идентификаторами по отдельности, ниже все системные константы представлены в логическом порядке. Все описанные в данной главе константные литералы определены в модуле `System` и поэтому всегда доступны.

Коды типов значений Variant

Функция `VarType` возвращает код типа значения `Variant`. Код типа – это значение `SmallInt`, которое содержит идентификатор типа с необязательными модификаторами `varArray` и `varByRef`. Любой тип, кроме `varEmpty` и `varNull`, может иметь модификатор `varArray`. Delphi автоматически заботится о модификаторе `varByRef`. Дополнительная информация приведена в описании типа `Variant` в главе 5. В табл. 6-1 перечислены идентификаторы типов, а в табл. 6-2 – модификаторы.

Таблица 6-1. Идентификаторы типов Variant

Литерал	Значение	Описание
`varEmpty`	$0000	Значение не присвоено
`varNull`	$0001	Нулевое значение
`varSmallint`	$0002	16-битовое целое со знаком
`varInteger`	$0003	32-битовое целое со знаком
`varSingle`	$0004	32-битовое число с плавающей точкой

Таблица 6-1 (продолжение)

Литерал	Значение	Описание
varDouble	$0005	64-битовое число с плавающей точкой
varCurrency	$0006	64-битовое число с фиксированной точкой и четырьмя десятичными знаками
varDate	$0007	64-битовое число с плавающей точкой, обозначающее дату и время
varOleStr	$0008	«Широкая» строка, ограниченная нулем
varDispatch	$0009	Интерфейс IDispatch
varError	$000A	32-битовый код ошибки
varBoolean	$000B	Логическая «истина» или «ложь»
varVariant	$000C	Указатель на другое значание типа Variant
varUnknown	$000D	Интерфейс IUnknown
varByte	$0011	8-битовое целое без знака
varStrArg	$0048	Не используется в Variant, однако описывает строковый параметр метода IDispatch
varString	$0100	Тип AnsiString Delphi
varAny	$0101	Скрытый указатель, используется для значений Any в CORBA
varTypeMask	$00FF	Маска для извлечения идентификатора из VarType

Таблица 6-2. Модификаторы типов Variant

Литерал	Значение	Описание
VarArray	$2000	Variant является массивом
VarByRef	$4000	Дополнительный уровень адресации для доступа к данным

Типы открытых массивов

Если параметр подпрограммы является вариантным открытым массивом (array of const), Delphi передает аргумент-массив путем преобразования каждого элемента массива в запись TVarRec. Поле VType каждой записи идентифицирует тип элемента. Дополнительную информацию можно найти в описании ключевого слова array. В табл. 6-3 перечислены значения VType для записи типа TVarRec.

Таблица 6-3. Возможные значения TVarRec.VType

Литерал	Значение	Тип элемента
VtInteger	0	Integer
VtBoolean	1	Boolean
VtChar	2	Char
VtExtended	3	Extended
VtString	4	ShortString
VtPointer	5	Pointer
VtPChar	6	PChar
VtObject	7	TObject
VtClass	8	TClass
VtWideChar	9	WideChar
VtPWideChar	10	PWideChar
VtAnsiString	11	AnsiString
VtCurrency	12	Currency
VtVariant	13	Variant
VtInterface	14	IUnknown
VtWideString	15	WideString
vtInt64	16	Int64

Смещения в таблице виртуальных методов

В главе 3 «Информация времени выполнения» описан формат таблицы виртуальных методов класса (VMT). Delphi не предоставляет тип записи для доступа к VMT, однако определены смещения (в байтах) различных частей VMT относительно ссылки на класс (`TClass`). Заметьте, что смещения могут изменяться от одной версии Delphi к другой. В табл. 6-4 приведены имена и значения смещений.

Таблица 6-4. Смещения в таблице виртуальных методов класса

Литерал	Значение	Описание
vmtSelfPtr	−76	Указатель на начало VMT
vmtIntfTable	−72	Указатель на таблицу интерфейсов
vmtAutoTable	−68	Указатель на таблицу автоматизации

Таблица 6-4 (продолжение)

Литерал	Значение	Описание
vmtInitTable	−64	Указатель на таблицу инициализации и завершения
vmtTypeInfo	−60	Указатель на запись TTypeInfo
vmtFieldTable	−56	Указатель на таблицу публикуемых полей
vmtMethodTable	−52	Указатель на таблицу публикуемых методов
vmtDynamicTable	−48	Указатель на таблицу динамических методов и сообщений
vmtClassName	−44	Указатель на имя класса в формате ShortString
vmtInstanceSize	−40	Размер экземпляра класса в байтах
vmtParent	−36	Указатель на VMT базового класса
vmtSafeCallException	−32	Адрес метода SafeCallException
vmtAfterConstruction	−28	Адрес метода AfterConstruction
vmtBeforeDestruction	−24	Адрес метода BeforeDestruction
vmtDispatch	−20	Адрес метода Dispatch
vmtDefaultHandler	−16	Адрес метода DefaultHandler
vmtNewInstance	−12	Адрес метода NewInstance
vmtFreeInstance	−8	Адрес метода FreeInstance
vmtDestroy	−4	Адрес деструктора Destroy

Коды ошибок времени выполнения

Delphi не экспортирует литералы для своих ошибок времени выполнения. Вместо этого определено несколько литералов в разделах реализации модулей System и SysInit, другие коды ошибок определены явно в коде модуля System. Для удобства в следующих таблицах перечислены все коды ошибок времени выполнения, встроенные в модуль System.

Внутренние коды ошибок используются самим модулем System. Программистам, реализующим процедуру ErrorProc, придется иметь дело именно с этими кодами.

Процедура ErrorProc модуля SysUtils преобразует внутренние коды ошибок в исключительные ситуации. В табл. 6-5 перечислены внутренние коды ошибок и классы исключительных ситуаций, с которыми работает модуль SysUtils для каждого вида ошибок.

Если модуль SysUtils не используется или если ошибка возникла до инициализации модуля SysUtils либо после его завершения, процедуры ErrorProc модуля System преобразуют внутренние коды ошибок во внешние. Некоторые ошибки работы с памятью и указателями не генерируют немедленной ошибки нарушения доступа, а лишь повреждают данные в памяти таким образом, что не позволяют Delphi корректно завершиться. Эти ошибки часто проявляются в виде сообщений об ошибках времени выполнения при завершении программы. В табл. 6-6 перечислены внешние коды ошибок.

В табл. 6-7 перечислены коды ошибок ввода/вывода, которые выдаются с внутренним кодом 0. Заметьте, что любой код ошибки Windows также может быть кодом ошибки ввода/вывода. (Модуль SysUtils генерирует для всех ошибок ввода/вывода исключительную ситуацию EInOutError.)

Таблица 6-5. Внутренние коды ошибок

Номер ошибки	Описание	Класс исключительной ситуации из SysUtils
0	Ошибка ввода/вывода; см. табл 6-7	EInOutError
1	Переполнение памяти	EOutOfMemory
2	Неверный указатель	EInvalidPointer
3	Деление целого числа на ноль	EdivByZero
4	Нарушение границ массива	ErangeError
5	Ошибка переполнения целого или перечисления	EintOverflow
6	Неверная вещественная операция	EinvalidOp
7	Деление числа с плавающей точкой на ноль	EzeroDivide
8	Вещественное переполнение	Eoverflow
9	Потеря значимости в вещественном числе	Eunderflow
10	Неверное преобразование объектного типа	EinvalidCast
11	Нарушение доступа	EaccessViolation
12	Переполнение стека	EStackOverflow
13	Прерывание по <Ctrl>+<Break>	EcontrolC
14	Привилегированная инструкция	EPrivilege
15	Неверное преобразование типа Variant	EVariantError
16	Неверная операция с типом Variant	EVariantError

Таблица 6-5 (продолжение)

Номер ошибки	Описание	Класс исключительной ситуации из SysUtils
17	Отсутствует диспетчер OLE для метода Variant	EVariantError
18	Невозможно создать массив Variant	EVariantError
19	Операция с массивом для значения типа Variant, не являющегося массивом	EVariantError
20	Нарушение границ массива Variant	EVariantError
21	Допущение нарушено	EAssertionFailed
22	Внешняя исключительная ситуация SysUtils	EExternalException
23	Неверное преобразование типа интерфейса	EIntfCastError
24	Ошибка в методе safecall	ESafeCallException

Таблица 6-6. Внешние коды ошибок

Внешний код ошибки	Описание
200	Деление на ноль
201	Нарушение границ массива
202	Неверный указатель
203	Переполнение памяти
204	Неверный указатель
205	Вещественное переполнение
206	Потеря значимости вещественного числа
207	Неверная вещественная операция
210	Вызов абстрактного метода
215	Нарушение границ целого или перечисления
216	Нарушение доступа
217	Необработанная исключительная ситуация
218	Привилегированная инструкция
219	Неверное преобразование типа класса
220	Неверное преобразование типа Variant

Таблица 6-6 (продолжение)

Внешний код ошибки	Описание
221	Неверная операция с `Variant`
222	Отсутствует диспетчер для метода `Variant`
223	Невозможно создать массив `Variant`
224	Операция с массивом для значения типа `Variant`, не являющегося массивом
225	Нарушение границ массива `Variant`
226	Невозможно инициализировать локальные данные потока (TLS)
227	Ошибка при вызове процедуры `Assert`
228	Неверное преобразование типа интерфейса
229	Ошибка из `safecall`-метода

Таблица 6-7. Коды ошибок ввода/вывода

Код ошибки	Описание
100	Чтение за концом файла
101	Диск переполнен
102	Не была вызвана процедура `AssignFile`
103	Файл закрыт
104	Файл не был открыт для чтения
105	Файл не был открыт для записи
106	Неверно форматированные данные для процедуры `Read`

Глава 7

Операторы

Эта глава описывает символьные операторы, их приоритеты и семантику. В главе 5 «Справочник по языку» подробно описаны именованные операторы, однако не приведены символьные, поскольку их трудно расположить в алфавитном порядке. В этой главе подробно рассказано о символьных операторах.

В Delphi определены перечисленные ниже операторы. В каждой строке приведены операторы, имеющие одинаковый приоритет. Самый высокий приоритет имеют операторы в начале списка, наименьший – указанные в конце:

```
@ not ^ + - (унарные операторы)
* / div mod and shl shr as
+ - or xor
> < >= <= <> = in is
```

`Variant` может быть строкой, числом, логическим или другим значением. Если в выражении присутствуют операнды типа `Variant` и не-`Variant`, Delphi преобразует все операнды в `Variant`. Если типы значений `Variant` не совпадают, Delphi преобразует один или оба операнда в тип, необходимый для данной операции, и выдает результат типа `Variant`. Дополнительную информацию о преобразовании типов значений `Variant` можно найти в главе 5.

Унарные операторы

`@`, *оператор*

Возвращает адрес своего операнда. Адрес переменной или обычной подпрограммы имеет тип `Pointer` или, если используется директива

компилятора $T или $TypedAddress, представляет собой типизированный указатель (например PInteger).

Адрес метода состоит из двух частей: указателя на код и указателя на данные, поэтому можно присвоить адрес метода только переменной соответствующего типа, но не переменной обобщенного типа Pointer. Если получение адреса метода производится по ссылке на класс, а не на объект, оператор @ возвращает только указатель на код. Например:

```
Ptr := @TObject.Free;
```

Если при присваивании адреса подпрограммы переменной процедурного типа сигнатура подпрограммы совпадает с типом переменной, применение оператора @ не требуется. Delphi сможет отличить присваивание адреса подпрограммы. Если типы не совпадают, например, в случае присваивания процедуры обработки ошибок переменной ErrorProc, которая является нетипизированным указателем, следует прибегнуть к оператору @:

```
function GetHeight: Integer;
begin
  Result := 42;
end;

procedure HandleError(Code: Integer; Addr: Pointer);
begin
  ...
end;

type
  TIntFunction = function: Integer;
var
  F: TIntFunction;
  I: Integer;
begin
  // Оператор @ не требуется. Присваиваем адрес GetHeight переменной F.
  F := GetHeight;
  // Вызываем GetHeight.
  I := F;

  // Требуется оператор @, т. к. тип ErrorProc - Pointer.
  ErrorProc := @HandleError;

  // Вызываем GetHeight через F и сравниваем целые результаты.
  if F = GetHeight then
    ShowMessage('Значения высоты равны');
  // Проверяем указатели, чтобы определить, указывает ли F на GetHeight.
  if @F = @GetHeight then
    ShowMessage('Указатели на функции равны');
```

Кроме того, оператор @ можно размещать в левой части оператора присваивания, чтобы присвоить переменной процедурный указатель, имеющий некорректный тип. Например, значение, возвращаемое из функции GetProcAddress Windows API является адресом процедуры, но имеет тип Pointer. Обычно преобразование типов – это наилучший способ решения проблемы, однако некоторые предпочитают использовать оператор @ в левой части присваивания, например:

```
type
  TIntFunction = function: Integer;
var
  F: TIntFunction;
begin
  // Следующие два оператора делают одно и то же.
  @F := GetProcAddress(DllHandle, 'GetHeight');
  F := TIntFunction(GetProcAddress(DllHandle, 'GetHeight'));
```

Чтобы получить адрес переменной процедурного типа, повторите оператор @, например @@Variable:

```
var
  F: TIntPointer;
  P: Pointer;
begin
  F := GetHeight;       // F указывает на GetHeight.
  P := @@F;             // Записываем адрес переменной F в P.
```

Not, *оператор*

Логическое отрицание. Подробности рассмотрены в описании ключевого слова not в главе 5.

^, *оператор*

Разыменование указателя. Этот постфиксный оператор указывается после выражения типа «указатель» и возвращает значение, на которое он указывает. Тип результата является базовым типом указателя.

При обращении к полю записи или элементу массива через указатель Delphi автоматически разыменовывает его. Например:

```
type
  TRecord = record
    Member: PInteger;
  end;
  PRecord = ^TRecord;
var
  P: PRecord;
  I: Integer;
begin
```

```
New(P);
P.Member := @I;      // Неявно подразумевается P^.Member
P.Member^ := 10;     // Требуется явное указание
                     // оператора ^
```

+, оператор

Знаковое тождество. Оператор «+» может использоваться перед любым числовым выражением. Эффекта не имеет, т. е. `+7 = 7`.

–, оператор

Арифметическое отрицание. Оператор «–» может использоваться перед любым числовым выражением для изменения его знака, например `–(X + Y)`. Заметьте, что Delphi рассматривает отрицательную константу как оператор «-», примененный к положительной константе. Это означает, что нельзя записать в явном десятичном виде самое маленькое отрицательное целое. Вместо этого применяется функция `Low(Integer)`.

Мультипликативные операторы

**, оператор*

Мультипликативный оператор также используется как оператор пересечения для множества. Если операнды являются множествами одного типа, в результате получается множество, содержащее элементы, присутствующие в обоих множествах-операндах. Например:

```
var
  A, B, C: set of 0..7;
begin
  A := [0, 1, 2, 3];
  B := [2, 3, 4, 5, 6];
  C := A * B;           // C := [2,3]
  A := [1, 0];
  C := A * B;           // C := []
```

/, оператор

Вещественное деление. При делении двух целых операнды преобразуются в вещественные. Деление на ноль генерирует ошибку времени выполнения 7.

`Div`, *оператор*

Целочисленное деление. Подробности приведены в главе 5.

`Mod`, *оператор*

Модуль (остаток) от деления. Подробности рассмотрены в главе 5.

And, *оператор*

Логическое или побитовое «И». Подробности можно найти в главе 5.

Shl, *оператор*

Сдвиг влево. Подробности – в главе 5.

Shr, *оператор*

Сдвиг вправо. Более подробные сведения приведены в главе 5.

As, *оператор*

Проверка типа и преобразование ссылки на объект и интерфейс. Подробности смотрите в главе 5.

Аддитивные операторы

+, *оператор*

Сложение, сцепление строк, объединение множества и смещение указателя PChar. Если оба операнда строковые, результат получается сцеплением этих двух строк. Если операнды – множества одинакового типа, результатом является множество, содержащее все элементы обоих операндов-множеств.

Если один операнд имеет тип PChar или PWideChar, а другой операнд целый, результатом является указатель, значение которого смещено на целое количество символов (не байт) от данного указателя. Целое смещение может быть положительным или отрицательным, однако результирующий указатель должен находиться в пределах того же массива символов, что и указанный операндом типа PChar или PWideChar. Например:

```
var
  A, B, C: set of 0..7;
  Text: array[0..5] of Char;
  P, Q: PChar;
begin
  A := [0, 1, 2];
  B := [5, 7, 4];
  C := A + B;          // C := [0, 1, 2, 4, 5, 7];

  Text := 'Hello';
  P := Text;
  Q := P + 3;
  WriteLn(Q);          // Печатает 'lo'
  WriteLn('Hi' + Q);   // Печатает 'Hilo'
```

–, оператор

Отрицание, разница массивов, смещение указателя PChar или разница двух значений PChar. Если операнды являются множествами одинакового типа, результат является множеством, содержащим элементы, присутствующие в левом операнде и отсутствующие в правом.

Если один операнд имеет тип PChar или PWideChar, а другой операнд целый, результатом является указатель, значение которого смещено на целое количество символов (не байт) от данного указателя. Целое смещение может быть положительным или отрицательным, однако результирующий указатель должен находиться в пределах того же массива символов, что и указанный операндом типа PChar или PWideChar. Например:

```
var
  A, B, C: set of 0..7;
  Text: array[0..5] of Char;
  P, Q: PChar;
begin
  A := [0, 1, 2, 4, 5, 6];
  B := [5, 7, 4];
  C := A - B;           // C := [0, 1, 2, 6];
  C := B - A;           // C := [7];

  Text := 'Hello';
  P := Text;
  Q := P + 3;
  WriteLn(Q);           // Печатает 'lo'
  P := Q - 2;
  WriteLn(P);           // Печатает 'ello'
  WriteLn(Q - P);       // Печатает 2
```

Or, *оператор*

Логическое или побитовое «ИЛИ». Смотрите подробнее в главе 5.

Xor, *оператор*

Логическое или побитовое исключающее «ИЛИ». Смотрите подробнее в главе 5.

Операторы сравнения

=, оператор

Равенство. Сравнение чисел, строк, указателей и множеств на точное равенство.

Заметьте, что сравнение вещественных чисел на точное равенство редко работает именно так, как предполагает программист. Следу-

ет использовать приближенное сравнение, учитывающее приблизительность вещественных значений. В книге «Получисленные алгоритмы» Дональда Кнута (D. E. Knuth «Seminumerical Algorithms», 3 издание, Addison Wesley Longman, 1998) в разделе 4.2.2 приведено отличное обсуждение этой проблемы и предлагаются ее решения.

<>, ператор

Неравенство. Сравнивает числа, строки, символьные указатели и множества. Возвращает логическое отрицание оператора «=».

>, оператор

Больше чем; сравнивает числа, строки или символьные указатели. Когда сравниваются значения типа `PChar` или `PWideChar`, выполняется сравнение указателей. Оператор возвращает `True`, если левый операнд указывает на более дальнюю позицию, чем правый. Оба операнда должны указывать на один и тот же символьный массив. Сравниваются порядковые значения символов, составляющих строки. (Модуль `SysUtils` содержит функции, которые позволяют сравнивать строки с учетом региональных настроек Windows. В большинстве приложений следует использовать эти функции вместо простого сравнения строк. Подробности рассмотрены в приложении В «Модуль SysUtils».) Например:

```
var
  Text: array[0..5] of Char;
  P, Q: PChar;
begin
  Text := 'Hello';
  P := Text;
  Q := P + 3;
  WriteLn(Q > P);     // True, поскольку Q указывает на более
                      // поздний элемент, чем P
  if Text > 'Howdy' then
    // Строковое сравнение возвращает False
```

<, оператор

Меньше чем; сравнивает числа, строки или символьные указатели. Когда сравниваются значения типа `PChar` или `PWideChar`, выполняется сравнение указателей. Оператор возвращает `True`, если левый операнд указывает на более ближнюю позицию, чем правый. Оба операнда должны указывать на один и тот же символьный массив. Сравниваются порядковые значения символов, составляющих строки. (Модуль `SysUtils` содержит функции, которые позволяют сравнивать строки с учетом региональных настроек Windows. В большинстве приложений следует использовать эти функции вместо простого сравнения строк. Подробности приведены в приложении В «Модуль SysUtils».) Например:

```
var
  Text: array[0..5] of Char;
  P, Q: PChar;
begin
  Text := 'Hello';
  P := Text;
  Q := P + 3;
  WriteLn(Q < P);    // False, поскольку Q указывает на более
                     // поздний элемент, чем P
  if Text < 'Howdy' then
    // Строковое сравнение возвращает True
```

>=, *оператор*

Больше или равно либо надмножество. Для чисел, строк, указателей `PChar` **или** `PWideChar` **оператор «>=» равен** `True`**, когда либо «>» – истина, либо «=» – истина. Для множеств оператор «>=» проверяет, являются ли все элементы правого операнда элементами левого множества-операнда. Например:**

```
var
  A, B: set of 0..7;
begin
  A := [0, 1, 2, 4, 5, 6];
  B := [5, 7, 4];
  if A >= B then
    // False, так как (7 in A) - False
  B := [1, 2, 0]
  if A >= B then
    // True, т. к. все элементы B присутствуют в A
```

<=, *оператор*

Меньше или равно либо подмножество. Для чисел, строк, указателей `PChar` **или** `PWideChar` **оператор «<=» равен** `True`**, когда либо «<» – истина, либо «=» – истина. Для множеств оператор «<=» проверяет, являются ли все элементы левого операнда элементами правого множества-операнда. Например:**

```
var
  A, B: set of 0..7;
begin
  A := [4, 5, 6];
  B := [5, 7, 4];
  if A <= B then
    // False, так как (6 in A) равен False
  B := [5, 7, 6, 4]
  if A <= B then
    // True, поскольку все элементы A присутствуют в B
```

`In`, *оператор*

Проверяет, является ли элемент членом множества. Подробности рассмотрены в главе 5.

`Is`, *оператор*

Проверяет тип ссылки на объект. Подробности приведены в главе 5. Заметьте, что оператор не работает для интерфейсов; используйте вместо него метод `QueryInterface`. (Смотрите также описание функции `Supports` в приложении B.

Глава 8

Директивы компилятора

Директивы компилятора – это специальные комментарии, которые управляют его поведением, аналогично директивам `#pragma` в C++. В этой главе дано общее описание директив компилятора и приведен их полный список.

Комментарий-директива – это комментарий, первым символом которого является знак доллара (`$`). Допускается любой из двух вариантов комментария, принятых в Паскале:

```
{$AppType GUI}
(*$AppType GUI*)
```

Нельзя использовать комментарии в стиле C++ (`//$AppType`) для директив компилятора.

Если первый символ комментария не является символом доллара, комментарий не является директивой компилятора. Обычный прием для отключения или «закомментирования» директивы компилятора – это вставка пробела или другого символа непосредственно перед символом доллара, например:

```
{ $AppType GUI}
(**$AppType GUI*)
```

Delphi поддерживает три варианта директив компилятора: переключатели, параметры и директивы условной компиляции. Переключатель – это логический флаг: средство языка или компилятора, которое может быть включено или выключено. Параметр предоставляет информацию, например имя файла или размер стека. Директива условной компиляции позволяет определить условия и выборочно компилировать части исходного файла в зависимости от выставленных усло-

вий. Условия являются логическими (установлено или не установлено).

Имена и параметры директив компилятора не чувствительны к регистру. Нельзя сократить имя директивы компилятора. Delphi игнорирует дополнительный текст комментария после директивы компилятора и ее параметров (если они есть). Однако следует воздержаться от включения дополнительного текста в комментарий, так как в будущие версии Delphi могут быть введены дополнительные параметры директив компилятора. Вместо включения другого комментария в комментарий, содержащий директиву компилятора, используйте отдельный комментарий.

Допускается объединение нескольких директив в одном комментарии, разделенных запятыми, причем ни перед, ни после запятой не должно быть пробелов. Директива с длинным именем должна быть последней в комментарии. При наличии нескольких директив с длинными именами или директивы с параметром следует использовать отдельные комментарии для каждой из них:

```
{$R+,Q+,C+,Align On}
{$O+,M 1024,40980}
```

Некоторые директивы являются глобальными для проекта. Эти директивы обычно указываются в исходном файле проекта (*dpr*-файл). Некоторые директивы применимы только к пакетам и должны находиться в исходном файле пакета (*dpk*-файл). Другие директивы применяются ко всему файлу, в котором они присутствуют. Такой файл может быть исходным файлом проекта, пакета или модуля. Если директива встречается более одного раза в файле проекта или пакета, компилятор Delphi исполняет последнюю из них.

Некоторые директивы имеют локальную область действия. Их можно включать и выключать несколько раз в одном файле. Некоторые локальные директивы применяются к подпрограмме в целом, при этом используется та директива, которая действует на момент окончания подпрограммы.

Некоторые директивы имеют два имени – односимвольное и длинное. Длинные имена легче читаются и, кроме того, полезны в тех случаях, когда один и тот же символ играет роль и переключателя, и параметра. Например, переключатель `$I` эквивалентен `$IOChecks`, а параметр `$I` обозначает `$Include`. Применение длинного имени позволяет избежать возможных ошибок, например, директива `{$I -}` включит в исходный текст файл с именем «-» вместо того, чтобы отменить действие переключателя `$I` (из-за пробела между `I` и `-`).

Обычно директивы компилятора указываются в исходном файле, но их можно указать также и в командной строке компилятора. (Подробности смотрите в приложении А «Утилиты командной строки».) Большинство настроек компилятора имеют свои аналоги в настройках

проекта, поэтому можно установить настройки в IDE. Эти установки применяются ко всем модулям проекта, но вы можете переопределить настройки проекта директивами в исходных файлах модуля. Очень удобно, что IDE также создает файл конфигурации компилятора (с расширением *cfg*), который содержит практически ту же информацию о конфигурации проекта, но в формате, пригодном для использования компилятором командной строки.

Установка настроек проекта в IDE – удобное средство. Если вы поступаете именно так, то для гарантии сохраните файл настроек (*dof*) вместе с исходными файлами проекта в системе контроля версий. Если *dof*-файл не сохраняется вместе с исходными файлами или вы продаете исходный код и не хотите включать *dof*-файлы, включите настройки компилятора в сами исходные файлы. Не все программисты используют одинаковые настройки, а они могут влиять на синтаксис и семантику компилятора. Без знания конкретных настроек невозможно перекомпилировать большинство программ на Delphi.

Некоторые директивы компилятора применяются в C++ Builder, и Delphi игнорирует эти директивы. C++ Builder использует их для создания *hpp*-файлов для модулей языка Паскаль.

$A, директива компилятора

Смотрите

Директива компилятора `$Align`.

$Align, директива компилятора

Синтаксис

```
{$A+}        // по умолчанию
{$Align On}  // по умолчанию
{$A-}
{$Align Off}
```

Диапазон

Локальный.

Описание

Когда директива `$A` или `$Align` включена, выполняется выравнивание полей классов, объектов и записей. Когда директива выключена, все структурированные типы упаковываются (`packed`). Выравнивание обеспечивает более быструю работу программы, но ценой дополнительного расхода памяти.

Директива `$Align` в начале файла действует на все объявления этого файла. Для отдельных объявлений лучше использовать модификатор `packed`.

Пример

```
type
{$Align On}
  TPadded = record
    A: Byte;
    B: Integer; // Delphi вставляет выравнивающие байты между A и B
  end;
  TPacked = packed record
    A: Byte;
    B: Integer; // Выравнивание не выполняется, так как запись объявлена
                // как упакованная.
  end;
{$Align Off}
  TUnaligned = record
    A: Byte;
    B: Integer; // выравнивание не выполняется
  end;
```

Смотрите также

Ключевое слово `Packed`.

$AppType, директива компилятора

Синтаксис

```
{$AppType GUI}      // по умолчанию
{$AppType Console}
```

Диапазон

Проект.

Описание

Директива компилятора `$AppType` сообщает компоновщику о типе создаваемого приложения: графическое или консольное. В консольном приложении Delphi открывает файлы `Input` и `Output` для чтения с консоли и записи на консоль. При компиляции консольного приложения Delphi определяет символ условной компиляции `CONSOLE` во всех модулях, составляющих проект. По умолчанию создается графическое приложение.

Смотрите также

Переменные `Input`, `IsConsole` и `Output`, ключевое слово `Program`.

$Assertions, директива компилятора

Синтаксис

```
{$C+}            // по умолчанию
{$Assertions On} // по умолчанию
{$C-}
{$Assertions Off}
```

Диапазон

Локальный.

Описание

Директива $C или $Assertions включает или выключает процедуру Assert. Когда она включена (по умолчанию), компилятор генерирует код для проверки допущений. Когда директива выключена, компилятор игнорирует все вызовы процедуры Assert и не генерирует для них код.

Советы и приемы

Обычным заблуждением является то, что допущения предназначены в основном для отладки и что в окончательной версии они должны быть отключены. Допущения обеспечивают правильное функционирование программы. Если допущение нарушено, это означает, что состояние программы отлично от предполагаемого. Если бы программа продолжила работу, ее поведение стало бы непредсказуемым. Результатом может быть разрушение пользовательских данных и другие мрачные последствия. Лучше сообщить о нарушении допущения и прервать программу.

Если допущение во внутреннем цикле приводит к понижению производительности, его можно отключить при помощи условной компиляции и директивы компилятора $Assertions.

Пример

```
Assert(List.Count > 0, 'Внутренняя ошибка: Список должен быть пуст');
for I := 0 to List.Count-1 do
begin
{$ifndef DEBUG}
 {$Assertions Off}
{$endif}
  Assert(IsValid(List[I]));
{$ifndef DEBUG}
 {$Assertions On}
{$endif}
  DoSomething(List[I]);
end;
```

Смотрите также

Процедура Assert, директивы компилятора $IfDef и $IfNDef.

$B, директива компилятора

Смотрите

Директива компилятора $BoolEval.

$BoolEval, директива компилятора

Синтаксис

```
{$B-}          // по умолчанию
{$BoolEval Off} // по умолчанию
{$B+}
{$BoolEval On}
```

Диапазон

Локальный.

Описание

По умолчанию в операторах `and` и `or` правый операнд не вычисляется, если результат выражения ясен из левого операнда. Это называется «сокращенная схема вычислений». Программисты C, C++ и Java знакомы с сокращенной схемой вычисления логических операторов.

Программисты, предпочитающие традиционный метод вычисления операторов `and` и `or`, принятый в языке Паскаль, могут отключить сокращенную схему вычисления операторов с помощью директивы компилятора `$BoolEval`. Заметьте, что Delphi-программисты, как правило, предпочитают сокращенную схему, так как это позволяет создавать более читабельный код.

Большинство Delphi-программ работают корректно только с включенной опцией `$BoolEval`. Не выключайте ее, если не уверены, что код не полагается на сокращенную схему вычисления операторов.

Пример

```
// Гарантирует, что путь заканчивается косой чертой.
// Первый подход использует сокращенную схему
// вычисления оператора AND.
{$BoolEval Off}
if (Length(Path) > 0) and (Path[Length(Path)] <> '\') then
  Path := Path + '\';

// Так следует записать предыдущее выражение без использования
// сокращенного вычисления оператора.
{$BoolEval On}
if Length(Path) > 0 then
  if Path[Length(Path)] <> '\' then
    Path := Path + '\';
```

Смотрите также

Ключевое слово `And`, тип `Boolean`, ключевое слово `Or`.

$C, директива компилятора

Смотрите

Директива компилятора `$Assertions`.

$D

Смотрите

Директивы компилятора `$DebugInfo` и `$Description`.

$DebugInfo, директива компилятора

Синтаксис

```
{$D+}           // по умолчанию
{$DebugInfo On} // по умолчанию
{$D-}
{$DebugInfo Off}
```

Диапазон

Файл.

Описание

Когда включена опция `$DebugInfo`, компилятор генерирует информацию о номерах строк, что облегчает последующую отладку модуля. Без информации о номерах строк единственный возможный способ отладки – это окно CPU.

Чтобы максимально воспользоваться преимуществами отладчика, следует также включить директиву `$LocalSymbols` для сохранения информации о типах, переменных и подпрограммах модуля.

Отладочная информация записывается в *dcu*-файл модуля и не влияет на производительность и размер конечного *exe*- или *dll*-файла. Чаще всего `$DebugInfo` отключают, когда *dcu*-файлы передаются без исходного текста. В очень больших проектах можно немного сэкономить время и объем файлов, выключив `$DebugInfo`, но в большинстве случаев экономия ничтожна.

Смотрите также

Директивы компилятора `$DefinitionInfo`, `$LocalSymbols` и `$ReferenceInfo`.

$Define, директива компилятора

Синтаксис

{$Define *Name*}

Диапазон

Файл.

Описание

Директива `$Define` определяет *`Name`* как символ для использования в условной компиляции. Последующие проверки *`Name`* с помощью директивы `$IfDef` выдают положительный результат. Вы можете определить одно и то же имя в файле несколько раз. Пример смотрите в описании директивы `$IfDef`. Имена условных символов не чувствительны к регистру.

В отличие от макроопределений C и C++, определение Delphi является просто флагом. Имя либо определено, либо нет. Проверка этого факта выполняется директивами компилятора `$IfDef` и `$IfNDef`. В одном файле один и тот же символ может выступать и как символ условной компиляции, и как идентификатор.

Смотрите также

Директивы компилятора `$Else`, `$Endif`, `$IfDef`, `$IfNDef`, `$IfOpt` и `$Undef`.

$DefinitionInfo, директива компилятора

Синтаксис

```
{$YD}                 // по умолчанию
{$DefinitionInfo On} // по умолчанию
{$Y+}
{$Y-}
{$DefinitionInfo Off}
```

Диапазон

Файл.

Описание

По умолчанию компилятор сохраняет информацию об определениях всех идентификаторов – типов, переменных и подпрограмм – определенных в файле модуля. Обозреватель проекта (Project Browser) и редактор кода в IDE позволяют просматривать эту информацию, отслеживать ссылки на идентификаторы, используемые в модуле и непосредственно переходить к определениям этих идентификаторов.

Когда переключатель `$DefinitionInfo` выключен, невозможно воспользоваться средствами навигации обозревателя проекта и редактора кода, но при этом сокращается время компиляции и уменьшаются размеры *dcu*-файлов. Генерируемая ссылочная информация не влияет на производительность и размер окончательного *exe*- или *dll*-файла.

Директивы $DefinitionInfo и $ReferenceInfo определяют три возможных варианта создания ссылочной информации. Директива $Y предоставляет сокращенное определение для этих вариантов:

- Директива $YD обозначает включенную информацию об определениях идентификаторов и выключенную информацию о ссылках на них. Это состояние по умолчанию.
- Директива $Y+ обозначает включенную информацию об определениях и ссылках.
- Директива $Y- обозначает выключенную информацию об определениях и ссылках.
- Нельзя включить информацию о ссылках, не включив автоматически информацию об определениях идентификаторов, поэтому четвертое состояние отсутствует.

Для того чтобы воспользоваться информацией об определениях и ссылках, следует также включить директивы $DebugInfo и $LocalSymbols.

Смотрите также

Директивы компилятора $DebugInfo, $LocalSymbols и $ReferenceInfo.

$DenyPackageUnit, директива компилятора

Синтаксис

```
{$DenyPackageUnit Off} // по умолчанию
{$DenyPackageUnit On}
```

Диапазон

Файл.

Описание

Если директива $DenyPackageUnit включена, Delphi отключает возможность использования модуля в пакете. Некоторые модули могут выполнять специальные действия по инициализации и завершению работы приложения и не могут использоваться в пакетах (которые фактически являются DLL). Например, модуль ComServ Delphi содержит директиву $DenyPackageUnit. В модуле реализованы несколько методов управления DLL COM-сервера, и эти методы должны находиться непосредственно в самой DLL, а не в пакете.

Смотрите также

Директива Package, директива компилятора $WeakPackageUnit.

$Description, директива компилятора

Синтаксис

{$Description *Строка*}

```
{$D Строка}
```

Диапазон

Проект.

Описание

Строка описания, указанная в директиве `$Description`, может иметь длину до 256 символов и ограничивается одинарными кавычками. Delphi сохраняет строку описания в *exe*- или *dll*-файле.

В большинстве приложений следует использовать ресурсы информации о версии вместо строки описания. В ресурсе можно указать значительно более подробную информацию. Строки описаний полезнее всего для пакетов разработки, так как Delphi показывает эти строки в виде имен пакетов в диалоговом окне «Install Packages».

Смотрите также

Директивы компилятора `$D` и `$E`.

$DesignOnly, директива компилятора

Синтаксис

```
{$DesignOnly Off} // по умолчанию
{$DesignOnly On}
```

Описание

Директива `$DesignOnly` используется в *dpk*-файле для пометки пакета как пакета только для разработки. Такой пакет нельзя присоединить к приложению или библиотеке. Его можно только установить в IDE Delphi или C++ Builder.

При создании компонента поместите модуль компонента в пакет времени выполнения, а редакторы свойств и компонентов – в пакет для разработки. При этом приложение может присоединить пакет времени выполнения и избежать накладных расходов, связанных с присоединением кода разработки, который приложению не требуется.

Смотрите также

Директива `Package`, директива компилятора `$RunOnly`.

$E, директива компилятора

Синтаксис

```
{$E Расширение}
```

Диапазон

Проект.

Описание

С помощью директивы $E указывают расширение файла для библиотеки. По умолчанию используется расширение *dll*. Можете включить или опустить начальную точку (.), но не помещайте расширение в кавычки.

Используйте директиву $E при компиляции хранителя экрана (screen saver), приложения для панели управления Windows и других специальных DLL.

Пример

```
{$e CPL}  { приложение для панели управления }
{$e .scr} { хранитель экрана }
```

Смотрите также

Директива компилятора $Description, ключевое слово Library.

$Else, директива компилятора

Синтаксис

```
$Else
```

Диапазон

Локальный.

Описание

Директива компилятора $Else обозначает else-секцию условия. Если условие удовлетворено, компилятор опускает код между директивами компилятора $Else и $EndIf. Если условие не выполнено, компилятор компилирует исходный код else-секции.

Пример смотрите в описании $IfDef.

Смотрите также

Директивы компилятора $Define, $EndIf, $IfDef, $IfNDef, $IfOpt и $Undef.

$EndIf, директива компилятора

Синтаксис

```
$Endif
```

Диапазон

Локальный.

Описание

Директива $EndIf обозначает конец секции условной компиляции. Пример смотрите в описании $IfDef.

Смотрите также

Директивы компилятора `$Else`, `$IfDef`, `$IfNDef`, `$IfOpt` и `$Undef`.

$ExtendedSyntax, директива компилятора

Синтаксис

```
{$X+}                 // по умолчанию
{$ExtendedSyntax On} // по умолчанию
{$X-}
{$ExtendedSyntax Off}
```

Диапазон

Локальный.

Если переключатель `$ExtendedSyntax` включен, компилятор расширяет синтаксис стандартного Паскаля несколькими дополнительными возможностями:

- Значение, возвращаемое при вызове функции, может не присваиваться переменной. Иногда результат, возвращаемый функцией, не является важным. Расширенный синтаксис Delphi позволяет вызвать функцию так, как будто она является процедурой, и игнорировать ее результат.
- Функция может присваивать возвращаемое значение неявно объявленной переменной `Result`.
- Массив `Char`, индекс которого имеет целый тип и начинается с нуля, может рассматриваться как строка. Строка должна оканчиваться символом `#0`. Можно также использовать такой массив в любом контексте, где требуется тип `PChar`, и компилятор автоматически подставит адрес первого элемента массива.

Для большинства других расширений языка Паскаль, присутствующих в Delphi, нет соответствующих директив компилятора. Об исключении смотрите описание директивы `$BoolEval`.

Смотрите также

Директива компилятора `$BoolEval`, ключевое слово `Function`, тип `PChar`.

$ExternalSym, директива компилятора

Синтаксис

{$ExternalSym *Идентификатор*}

Диапазон

Локальный.

Описание

Директива компилятора `$ExternalSym` сообщает компилятору C++ Builder, что *Идентификатор* является внешним символом, который не требуется определять в заголовочном файле C++ (*hpp*), генерируемом для модуля языка Паскаль.

Иногда символ, который требуется определить в модуле языка Паскаль, уже определен где-то в C++ Builder. Директива компилятора `$ExternalSym` применяется для сообщения C++ Builder о том, что данный символ не требуется записывать в файл *hpp*. Заметьте, что для имени пространства имен используется имя модуля. Директивой `$ExternalSym` вы сообщаете компилятору, что символ определен вне пространства имен. Этот вариант является противоположностью директивы `$NoDefine`, которая определяет символ в пределах пространства имен модуля.

Пример

```
type
  size_t = LongWord;
  {ExternalSym size_t} {этот тип уже определен в C++. }
var
  S: size_t;
```

Смотрите также

Директивы компилятора `$HppEmit`, `$NoDefine`, `$NoInclude`.

$G, директива компилятора

Смотрите

Директива компилятора `$ImportedData`.

$H, директива компилятора

Смотрите

Директива компилятора `$LongStrings`.

$Hints, директива компилятора

Синтаксис

```
{$Hints On} // по умолчанию
{$Hints Off}
```

Диапазон

Локальный, применяется к подпрограмме в целом.

Описание

Директива $Hints включает или выключает подсказки компилятора. Компилятор выдает подсказки для кода, который кажется подозрительным или возможно содержащим ошибку. Например, компилятор выдает подсказку, если в классе имеется неиспользуемый закрытый (private) метод. Возможно, вы создали метод, зная, что он в данный момент не используется, однако планируете реализовать его в будущей версии. Можно подавить выдачу таких подсказок директивой $Hints.

Советы и приемы

Не выключайте подсказки компилятора глобально для файла или проекта. Подсказки компилятора полезны и могут предупредить многие часто возникающие ошибки. Выключайте подсказки только в том случае, если знаете, что это безопасно, и только для конкретной подпрограммы.

Пример

```
type
  TExample = class
  private
  {$Hints off}  { Я знаю, что сейчас эта подпрограмма не работает,
однако воспользуюсь ею позднее. }
    procedure Unused;
  {$Hints on}
    ...
  end;
```

Смотрите также

Директива компилятора $Warnings.

$HppEmit, директива компилятора

Синтаксис

```
(*$HppEmit Строка*)
```

Диапазон

Локальный.

Описание

Директива компилятора $HppEmit предназначена для записи *Строки* в файл *hpp* для C++ Builder. Строка записывается в указанную пользователем секцию в начале файла. Все строки, указанные в $HppEmit, записываются в том же порядке, в котором они указаны в *pas*-файле.

Директива $HppEmit позволяет включить в *hpp*-файл любой текст, начиная с дополнительных директив #include и кончая объявлениями типов.

Пример

```
type
  TComplex = record
    Re, Im: Double;
  end;

// Определяем структуру для C++ по-другому.
// Заметьте, что в коде HppEmit применяются символы { и }, которые
// нельзя указать в комментариях языка Паскаль, использующих символы
// { и }, так как эти комментарии не допускают строк в кавычках.
// Вместо этого указываются символы (* и *).

{$ExternalSym TComplex}
(*$HppEmit 'struct TComplex {'*)
(*$HppEmit '  double Re;'*)
(*$HppEmit '  double Im;'*)
(*$HppEmit '  TComplex(double r=0, double i=0) : Re(r), Im(i) {}'*)
(*$HppEmit '};'*)
```

Смотрите также

Директивы компилятора $ExternalSyms, $NoDefine и $NoInclude.

$I, директива компилятора

Смотрите

Директивы компилятора $Include и $IOChecks.

$IfDef, директива компилятора

Синтаксис

```
{$IfDef Имя}
   ...
{$Else}
   ...
{$Endif}
```

Диапазон

Локальный.

Описание

Директива $IfDef обозначает начало секции условной компиляции. Если *Имя* определено (предшествующей директивой $Define), компилируется исходный код после директивы $IfDef, вплоть до соответствующей директивы $Else или $EndIf. Использование $Else не обязательно. Секции условной компиляции могут быть вложенными.

Часто условная компиляция применяется для проверки версии Delphi. В табл. 8-1 перечислены символы, предопределенные компилятором, с указанием версии Delphi, в которой определяется каждый из них.

Таблица 8-1. Предопределенные символы

Символ	Версия компилятора
CONSOLE	Определен, если {$AppType Console}
CPU386	Все версии
VER80	Delphi 1
VER90	Delphi 2
VER93	C++ Builder 1
VER100	Delphi 3
VER110	C++ Builder 3
VER120	Delphi 4
VER125	C++ Builder 4
VER130	Delphi 5
WINDOWS	Delphi 1
WIN32	Delphi 2, 3, 4, 5; C++ Builder 1, 3, 4

Пример

```
{$IfOpt D+} // Если отладка включена.
  {$define DEBUG}
  {$Assertions On}
  {$RangeChecks On}
  {$OverflowChecks On}
{$else}
  {$undef DEBUG} // Если символ определен в настройках проекта
{$endif}
...
{$ifdef DEBUG}
  Log('Отладочное сообщение');
{$endif}
```

Смотрите также

Директивы компилятора `$Define`, `$Else`, `$Endif`, `$IfNDef`, `$IfOpt` и `$Undef`.

$IfNDef, директива компилятора

Синтаксис

```
{$IfNDef Имя}
   ...
{$Endif}
```

Диапазон

Локальный.

Описание

Директива $IfNDef аналогична $IfDef, но, в отличие от последней, проверяет, что *Имя не* определено. Пояснения и пример смотрите в описании директивы $IfDef.

Смотрите также

Директивы компилятора $Define, $Else, $Endif, $IfDef, $IfOpt и $Undef.

$IfOpt, директива компилятора

Синтаксис

```
{$IfOpt Переключатель+}
   ...
{$Endif}
{$IfOpt Переключатель-}
   ...
{$Endif}
```

Диапазон

Локальный.

Описание

Директива $IfOpt схожа с $IfDef, но проверяет состояние директив-переключателей компилятора. Могут проверяться только однобуквенные переключатели. Если директива включена, проверка *Переключатель+* возвращает истину, *Переключатель-* – ложь. Если директива выключена, *Переключатель+* ложен, *Переключатель-* возвращает истину.

Дополнительную информацию смотрите в описании директивы компилятора $IfDef.

Пример

```
// Временно отключаем проверку границ диапазонов, затем возвращаемся
// к исходному состоянию переключателя.
{$IfOpt R+}
  {$define RangeChecks}
{$else}
  {$undef RangeChecks}
{$endif}
{$R-}
... // код, который требует выключения проверки границ диапазонов

{$ifdef RangeChecks}
  {$R+}
{$endif}
```

Смотрите также

Директивы компилятора $Define, $Else, $Endif, $IfDef, $IfNDef и $Undef.

$ImageSize, директива компилятора

Синтаксис

```
{$ImageBase Целое}
```

Диапазон

Проект.

Описание

Директива `$ImageBase` определяет адрес, по которому Windows загружает исполняемый модуль. По умолчанию базовый адрес равен $00400000 – стандартное значение Windows. Для программ этот адрес менять не нужно, однако следует изменить его для библиотек и пакетов.

Каждый исполняемый модуль в ОС Windows должен загружаться по уникальному адресу. Windows пытается загрузить модуль по указанному в директиве базовому адресу, но если адрес конфликтует с другим исполняемым модулем того же процесса, Windows перемещает модуль на другой виртуальный адрес. Перемещение увеличивает время загрузки модуля.

Стандартные пакеты Delphi используют базовые адреса, начинающиеся с $40000000. Системные DLL Microsoft начинаются с адреса $70000000. Для пакетов и DLL сторонних разработчиков нет подобных стандартов, поэтому следует выбирать адреса, которые не конфликтуют с другими.

$ImplicitBuild, директива компилятора

Синтаксис

```
{$ImplicitBuild On}   // по умолчанию
{$ImplicitBuild Off}
```

Диапазон

Файл.

Описание

По умолчанию Delphi автоматически перекомпилирует пакеты, когда в этом возникает необходимость. Выключайте директиву `$ImplicitBuild`, чтобы избежать проверки на необходимость повторной компиляции пакета.

Во время разработки пакета имеет смысл включить `$ImplicitBuild`, особенно если вы работаете с несколькими взаимосвязанными пакетами. Легко забыть, какие модули были изменены и какие пакеты следует перекомпилировать, поэтому пусть Delphi разбирается с этим за вас.

Завершив разработку пакета, можно отключить $ImplicitBuild, чтобы сократить накладные расходы на определение необходимости перекомпиляции.

Советы и приемы

- Значение по умолчанию применяется только в тех случаях, когда директива отсутствует в исходном файле пакета. Редактор пакетов Delphi IDE имеет собственное значение по умолчанию, и когда новый пакет создается в редакторе, в исходный текст пакета вставляется директива $ImplicitBuild Off.
- Заметьте также, что неявная перекомпиляция происходит, только если изменился исходный файл. Если поменялась только форма (т. е. *dfm*-файл), пакет не перекомпилируется, даже если директива $ImplicitBuild включена.

Смотрите также

Директива Package.

$ImportedData, директива компилятора

Синтаксис

```
{$G+}                // по умолчанию
{$ImportedData On}   // по умолчанию
{$G-}
{$ImportedData Off}
```

Диапазон

Локальный.

Описание

Директива компилятора $ImportedData не реализована.

Ссылка на глобальную переменную в другом модуле обычно требует дополнительного уровня адресации, если модуль находится в отдельном пакете. Теоретически директива $ImportedData исключает эту адресацию. Практически ее использование не дает эффекта. Компилятор автоматически определяет, требуется ли адресация, и компилирует ссылки на глобальные переменные соответствующим образом.

$Include, директива компилятора

Синтаксис

{$Include *Имя файла*}
{$I *Имя файла*}

Диапазон

Локальный.

Описание

Директива компилятора `$Include` указывает компилятору на необходимость чтения файла с именем *Имя файла* и включает все его содержимое в место, где указана директива `$Include`.

Наилучший способ применения `$Include` – импорт условных определений и директив компилятора. Для импорта объявлений типов, переменных и подпрограмм следует использовать модули.

Пробелы и другие специальные символы включаются в *Имя файла* при помощи кавычек.

Смотрите также

Ключевое слово `Unit`.

$IOChecks, директива компилятора

Синтаксис

```
{$I+}           // по умолчанию
{$IOChecks On}  // по умолчанию
{$I-}
{$IOChecks Off}
```

Диапазон

Локальный.

Описание

Когда `$IOChecks` включена, Delphi проверяет результаты всех функций ввода/вывода и сообщает об ошибке времени выполнения в случае неудачного вызова. Когда `$IOChecks` выключена, для определения успешности завершения операции ввода/вывода нужно использовать функцию `IOResult`.

Пример

```
// Возвращает истину, если указанный файл имеет формат PostScript,
// и ложь, если файл не существует или не является файлом в
// в формате PostScript.
// Начало файла в формате PostScript обозначается символами '%!'.
function IsPostScript(const FileName: string): Boolean;
type
  TPair = array[0..1] of Char;
const
  PostScript: TPair = '%!';
var
  F: File of TPair;
  Pair: TPair;
begin
  AssignFile(F, FileName);
  FileMode := 0;
```

```
{$IOChecks Off}
  Reset(F);
  Read(F, Pair);
{$IOChecks On}
  Result := (IOResult = 0) and (Pair = PostScript);
  CloseFile(F);
end;
```

Смотрите также

Функция `IOResult`.

$J, директива компилятора

Смотрите

Директива компилятора `$WriteableConst`.

$L, директива компилятора

Смотрите

Директивы компилятора `$Link` и `$LocalSymbols`.

$Link, директива компилятора

Синтаксис

{`$L` *Имя файла*}
{`$Link` *Имя файла* }

Диапазон

Локальный.

Директива компилятора `$Link` присоединяет перемещаемый объектный файл (*obj*) к проекту Delphi. Объектный файл может предоставлять реализацию подпрограмм, объявленных с директивой `external`. Объектный файл должен иметь формат объектных файлов Intel, нельзя использовать файлы в формате COFF, генерируемые Visual C++ и большинством других компиляторов C и C++. В то же время, поддерживаются объектные файлы, генерируемые Borland C++ и Borland TASM (Turbo Assembler), но не C++ Builder.

Для имен файлов, содержащих пробелы и другие необычные символы, используйте кавычки. По умолчанию файл, указываемый параметром *Имя файла*, имеет расширение *obj*.

Смотрите также

Директива `External`.

$LocalSymbols, директива компилятора

Синтаксис

```
{$L+}              // по умолчанию
{$LocalSymbols On} // по умолчанию
{$L-}
{$LocalSymbols Off}
```

Диапазон

Файл.

Описание

Если включена директива `$DebugInfo`, можно также включить директиву `$LocalSymbols` для сохранения в модуле информации о подпрограммах, типах, переменных и номерах строк исходного файла. Информация об идентификаторах записывается в *dcu*-файл и не влияет на производительность или размер окончательного *exe*- или *dll*-файла. Delphi игнорирует директиву компилятора `$LocalSymbols`, когда выключена директива `$DebugInfo`.

В очень больших проектах можно немного сэкономить время и объем файлов, выключив `$LocalInfo`, но в большинстве случаев экономия ничтожна и не стоит прилагаемых усилий.

Смотрите также

Директивы компилятора `$DebugInfo`, `$DefinitionInfo` и `$ReferenceInfo`.

$LongStrings, директива компилятора

Синтаксис

```
{$H+}              // по умолчанию
{$LongStrings On} // по умолчанию
{$H-}
{$LongStrings Off}
```

Диапазон

Локальный.

Описание

По умолчанию Delphi интерпретирует тип `string` как `AnsiString`. Если директива компилятора `$LongStrings` выключена, Delphi возвращается к стилю поведения Delphi 1, в котором `string` интерпретируется как `ShortString`.

Большинство новых Delphi-программ оперируют длинными строками, работать с которыми значительно проще, чем с короткими. Опцию `$LongStrings` следует выключать только для обеспечения совместимости с Delphi 1.

Смотрите также

Типы `AnsiString` и `ShortString`, ключевое слово `String`.

$M, директива компилятора

Синтаксис

{$M *MinStackSize*,*MaxStackSize*}

Диапазон

Проект.

Описание

Директива компилятора `$M` устанавливает минимальный и максимальный размер стека. По умолчанию минимальный размер равен 16 Кбайт, максимальный – 1 Мбайт. Для ясности вместо директивы `$M` используйте `$MinStackSize` и `$MaxStackSize`.

Пример

```
// Размеры стека по умолчанию в шестнадцатеричном виде
{$M $4000,$100000}
```

Смотрите также

Директивы компилятора `$MaxStackSize`, `$MinStackSize` и `$TypeInfo`.

$MaxStackSize, директива компилятора

Синтаксис

{$MaxStackSize *Целое*}

Диапазон

Проект.

Описание

Директива `$MaxStackSize` устанавливает максимальный размер стека программы в байтах. Для библиотеки эта директива компилятора игнорируется, т. к. DLL не имеет своего собственного стека. По умолчанию максимальный размер стека равен 1 Мбайт (1 048 576 байт), которого более чем достаточно для большинства программ.

Windows создает стек, минимальный размер которого определяется директивой `$MinStackSize`, и он по необходимости растет до максимального размера, указанного в директиве `$MaxStackSize`. Если стек достигает максимального размера, Delphi выдает ошибку времени выполнения 12 (`EStackOverflow`).

В большинстве случаев ошибки переполнения стека бывают вызваны каким-либо дефектом программы, таким как неограниченная рекур-

сия. Внимательно поищите ошибку в коде перед тем, как увеличивать максимальный размер стека.

Смотрите также

Директивы компилятора `$M` и `$MinStackSize`.

$MinEnumSize, директива компилятора

Синтаксис

```
{$Z1}            // по умолчанию
{$MinEnumSize 1} // по умолчанию
{$Z2}
{$Z4}
{$MinEnumSize 2}
{$MinEnumSize 4}
{$Z-} // устаревшее, означает {$Z1}
{$Z+} // устаревшее, означает {$Z4}
```

Диапазон

Локальный.

Описание

Директива компилятора `$MinEnumSize` задает минимальный размер (в байтах), устанавливаемый Delphi для перечисляемого типа. По умолчанию он равен одному байту, т. е. Delphi использует наиболее подходящий для конкретного типа размер. Перечисления, содержащие до 256 литералов, помещаются в один байт, перечисления до 65 568 литералов – в два байта.

Для обеспечения совместимости перечисляемого типа с типом `enum` в программе C или C++ следует настроить размер перечисляемого типа. Обычно требуется установить минимальный размер перечисляемого типа равным 4 байтам.

Пример

```
{$MinEnumSize 4}  // Увеличиваем размер для совместимости с C.
type
  TCEnum = (cRed, cBlack);
{$MinEnumSize 1} // Восстанавливает значение по умолчанию.

function GetColor: TCEnum; external 'cdemo.dll';
```

Смотрите также

Ключевое слово `Type`.

$MinStackSize, директива компилятора

Синтаксис

{$MinStackSize *Целое*}

Диапазон

Проект.

Описание

Директива $MinStackSize устанавливает начальный размер стека в байтах. Для библиотеки эта директива игнорируется, т. к. DLL не имеет своего собственного стека. По умолчанию минимальный размер стека равен 16 Кбайт (16 384 байт), что подходит для большинства программ.

Windows увеличивает размер стека по необходимости вплоть до достижения максимальной величины, указанной в директиве компилятора $MaxStackSize.

Смотрите также

Директивы компилятора $M и $MaxStackSize.

$NoDefine, директива компилятора

Синтаксис

{$NoDefine *Идентификатор*}

Диапазон

Локальный.

Описание

Если директива компилятора $NoDefine включена, то C++ Builder не записывает определение *Идентификатора* в *hpp*-файл. В отличие от $ExternalSym, компилятор предполагает, что будет дано альтернативное определение *Идентификатора* с помощью директивы $HppEmit. Сделав это, убедитесь, что символ определяется в пространстве имен модуля.

Пример

```
unit Complex;
interface
type
  TComplex = record
    Real, Imag:  Double;
  end;
  {NoDefine   TComplex}
  (*$HppEmit  'namespace Complex  {*)
  (*$HppEmit  '  class TComplex  {' *)
  (*$HppEmit  '  public: ' *)
```

```
(*$HppEmit  '      TComplex() : m_real(0), m_imag(0) {} ' *)
(*$HppEmit  '    private: ' *)
(*$HppEmit  '      double m_real, m_imag; ' *)
(*$HppEmit  '    }; ' *)
(*$HppEmit  '};' *)
```

Смотрите также

Директивы компилятора `$ExternalSym`, `$HppEmit` и `$NoInclude`.

$NoInclude, директива компилятора

Синтаксис

{`$NoInclude` *Имя Модуля*}

Диапазон

Локальный.

Описание

Если директива компилятора `$NoInclude` включена, то C++ Builder не помещает предложение `#include "UnitName.hpp"` в *hpp*-файл модуля. По умолчанию C++ Builder подключает заголовочные файлы всех пользовательских модулей, указанных в объявлении `uses`.

Используйте `$NoInclude`, если известно, что объявления модуля присутствуют в другом *hpp*-файле. Возможно, придется совместить директиву `$NoInclude` с директивой `$HppEdit` для подключения другого *hpp*-файла.

Смотрите также

Директивы компилятора `$ExternalSym`, `$HppEmit` и `$NoDefine`.

$O, директива компилятора

Смотрите

Директива компилятора `$Optimization`.

$ObjExportAll, директива компилятора

Синтаксис

```
{$ObjExportAll Off} // по умолчанию
{$ObjExportAll On}
```

Диапазон

Файл.

Описание

Директива компилятора `$ObjExportAll` сообщает C++ Builder о необходимости экспорта всех символов при компиляции модуля в *obj*-файл. Эта директива включается при использовании модуля языка Паскаль в DLL или пакете, написанном на C++, чтобы обойтись без создания *def*-файла.

Смотрите также

Ключевое слово `Exports`.

$OpenStrings, директива компилятора

Синтаксис

```
{$P+}              // по умолчанию
{$OpenStrings On} // по умолчанию
{$P-}
{$OpenStrings Off}
```

Диапазон

Локальный.

Описание

Директива компилятора `$OpenStrings` позволяет рассматривать все параметры типа `string` как параметры `OpenString`. Если включена директива `$LongStrings`, Delphi игнорирует `$OpenStrings`.

Директива `$OpenStrings` влияет только на параметры, объявленные с типом `string`, и только когда выключена директива `$LongStrings`. Даже если `$OpenStrings` включена, тип `OpenString` может использоваться для объявления параметра «открытой строки».

`$OpenStrings` существует для обратной совместимости. В новых программах на Delphi следует применять длинные строки вместо коротких.

Смотрите также

Тип `OpenString`, ключевое слово `String`.

$Optimization, директива компилятора

Синтаксис

```
{$O+}              // по умолчанию
{$Optimization On} // по умолчанию
{$O-}
{$Optimization Off}
```

Диапазон

Локальный, применяется к подпрограмме в целом.

Описание

Когда директива `$Optimization` включена, компилятор производит более эффективный код. Оптимизации, выполняемые Delphi, безопасны и не влияют на семантику кода. Иногда при включенной оптимизации осложняется отладка, но если нет специфической проблемы с оптимизатором, следует оставлять директиву `$Optimization` включенной.

Советы и приемы

Некоторые виды оптимизации могут ввести программиста в заблуждение при запуске программы в отладчике. Наиболее распространенные трудности при отладке возникают в связи со следующими видами оптимизации:

- Исключение избыточного или ненужного кода. Нельзя установить точку останова на предложении, если оптимизатор посчитал это предложение не служащим никакой цели.
- Циклы `for` с обратным отсчетом. Если нет обращения к управляющей переменной цикла `for`, Delphi оптимизирует переменную для того, чтобы ее минимальное значение было равно нулю.
- Переупорядочение веток кода. Предложения `Break`, `Continue`, `Exit` и `goto` могут быть оптимизированы и переупорядочены. Чаще всего это выглядит как оптимизация предложения `Break` в подпрограмме в немедленный выход из нее.
- Иногда предложения `case` компилируются в таблицу переходов. Таблица выглядит как код, однако фактически является смещениями кода для разных значений `case`. Таблицы переходов усложняют чтение кода. Компилятор может генерировать таблицу переходов, даже если оптимизация выключена.

Если из-за оптимизатора возникли сложности при установке точки останова в определенном месте кода, временно вставьте инструкцию ассемблера для точки останова. Жестко запрограммированная точка останова предпочтительнее отключения оптимизатора, поскольку действует только на одну конкретную точку кода. Другая возможность состоит в вызове подпрограммы, не выполняющей ничего. Например:

```
procedure Example;
var
  I: Integer;
begin
  for I := 1 to 10 do
    if SomeCondition then
      Break;          // Оптимизатор исключает Break, так что на
end;                  // это предложение нельзя установить точку останова.

...
    if SomeCondition then
    begin
```

```
      asm int 3 end;  // Один из обходных способов жесткого кодирования
      Break;          // точки останова.
    end;

procedure Empty;
begin
end;

...
    if SomeCondition then
    begin
      Empty;  // Другой способ – вызов ничего не делающей процедуры и
      Break;  // установить точку останова отладчиком.
    end;
```

Смотрите также

Директива компилятора `$StackFrames`.

$OverflowChecks, директива компилятора

Синтаксис

```
{$Q-}                 // по умолчанию
{$OverflowChecks Off} // по умолчанию
{$Q+}
{$OverflowChecks On}
```

Диапазон

Локальный.

Описание

Директива `$OverflowChecks` сообщает компилятору о необходимости вставки кода проверки результатов операций с целыми и перечисляемыми значениями, гарантируя, что результат останется в границах целого или перечисляемого типа. Если результат операции выходит за пределы диапазона, Delphi выдает ошибку времени выполнения 5 (`EIntOverflow`).

Советы и приемы

Директиву `$OverflowChecks` следует включать всегда, т. к. она позволяет обнаруживать многие ошибки в программе. Частое заблуждение состоит в том, что проверки следует отключать в окончательных версиях программ. Если возникает переполнение целого или перечисления, желательно найти эту ошибку как можно быстрее, не дожидаясь, пока она распространится непредсказуемым образом по всей программе, возможно, разрушив пользовательские данные. Не выключайте проверку на переполнение, если только не возникла конкретная проблема с производительностью, и даже в этом случае выключайте ее только для критических частей кода.

Смотрите также

Директива компилятора $RangeChecks.

$P, директива компилятора

Смотрите

Директива компилятора $OpenStrings.

$Q, директива компилятора

Смотрите

Директива компилятора $OverflowChecks.

$R, директива компилятора

Смотрите

Директивы компилятора $RangeChecks и $Resource.

$RangeChecks, директива компилятора

Синтаксис

```
{$R-}              // по умолчанию
{$RangeChecks Off} // по умолчанию
{$R+}
{$RangeChecks On}
```

Диапазон

Локальный.

Описание

Директива $RangeCheck уведомляет компилятор о необходимости вставки кода проверки операций с массивами на предмет соответствия значения индекса массива его границам. Если индекс не попадает в границы, Delphi выдает ошибку времени выполнения 4 (ERangeError).

Советы и приемы

- Директиву $RangeChecks следует включать всегда, т. к. она позволяет отлавливать многие ошибки в программе. Частое заблуждение состоит в том, что проверки следует отключать в окончательных версиях программ. Если возникает ошибка выхода за границы массива, желательно найти эту ошибку как можно быстрее, не дожидаясь, пока она распространится непредсказуемым образом по всей программе, возможно, разрушив пользовательские данные. Не выключайте проверку на границы диапазона, если только не воз-

никла конкретная проблема с производительностью, и даже в этом случае выключайте ее только для критических частей кода.

- Многие проблемы с нарушением безопасности в сети Интернет связаны с ошибками границ массивов. Программы чаще всего пишутся на языке C, в котором проверка выхода за границы массивов не выполняется. При написании программ в Delphi непростительно не воспользоваться преимуществами средств безопасности, предоставляемыми языком программирования.
- Если директива `$RangeChecks` включена, Delphi также проверяет обращения к строкам, чтобы значение индекса символа находилось между 1 и длиной строки, однако если длинная строка пуста, Delphi генерирует ошибку нарушения доступа вместо ошибки времени выполнения (поскольку в Delphi пустая строка представлена нулевым указателем). Поэтому перед использованием индекса строки всегда следует проверять, что строка не пустая:

```
if (Length(S) > 0) and (S[1] = '#') then
  ShowMessage('S starts with #');
```

Смотрите также

Директива компилятора `$OverflowChecks`.

$RealCompatibility, директива компилятора

Синтаксис

```
{$RealCompatibility Off} // по умолчанию
{$RealCompatibility On}
```

Диапазон

Локальный.

Описание

Когда директива `$RealCompatibility` включена, компилятор делает тип `Real` эквивалентным `Real48`. По умолчанию тип `Real` эквивалентен `Double`. Эта директива сохранена для обеспечения обратной совместимости. В новых программах следует использовать новые вещественные типы (`Single`, `Double` и `Extended`). Можно воспользоваться типом `Real48`, если требуется прочитать значение, записанное в формате старого типа `Real`.

Смотрите также

Типы `Double`, `Extended`, `Real`, `Real48`, `Single`.

$ReferenceInfo, директива компилятора

Синтаксис

```
{$YD}                 // по умолчанию
{$Y+}
{$Y-}

{$ReferenceInfo Off} // по умолчанию
{$ReferenceInfo On}
```

Диапазон

Локальный.

Описание

По умолчанию отладочная информация модуля содержит определения всех идентификаторов, объявленных в модуле. При использовании директивы `$ReferenceInfo` также можно сохранить в файле ссылки на идентификаторы. При наличии информации о ссылках обозреватель проекта (Project Browser) и редактор кода могут указать на все обращения к типу, переменной, подпрограмме или другому идентификатору.

Для использования `$ReferenceInfo` должна быть включена директива `$DefinitionInfo`. Директивы `$DefinitionInfo` и `$ReferenceInfo` определяют три возможных варианта создания ссылочной информации. Директива `$Y` предоставляет сокращенное обозначение этих состояний:

- Директива `$YD` обозначает включенную информацию об определениях идентификаторов и выключенную информацию о ссылках на них. Это состояние по умолчанию.
- Директива `$Y+` обозначает включенную информацию об определениях и ссылках.
- Директива `$Y-` обозначает выключенную информацию об определениях и ссылках.
- Нельзя включить информацию о ссылках, не включив автоматически информацию об определениях идентификаторов, поэтому четвертое состояние отсутствует.

Для создания информации об определениях и ссылках требуется, чтобы были включены директивы `$DebugInfo` и `$LocalSymbols`.

$Resource, директива компилятора

Синтаксис

```
{$R Имя файла}
{$Resource Имя файла}
{$R 'Имя файла.res' 'Имя файла.rc'}
```

```
{$Resource 'Имя файла.res' 'Имя файла.rc'}
```

Диапазон

Локальный.

Описание

Директива `$Resource` включает в проект или пакет внешний файл ресурсов. При сборке проекта или пакета Delphi включает файл ресурса в окончательный *exe*-, *dll*- или *bpl*-файл. Файл ресурса не должен существовать при компиляции модуля – только при его сборке. Исходный файл может загружать любое количество файлов ресурсов.

Если в качестве параметра *Имя файла* указан символ (`*`, звездочка), Delphi использует базовое имя исходного файла, подставляя расширение, указанное в директиве. Например, модули форм должны содержать директиву `{$R *.DFM}`, которая обеспечивает загрузку описания формы как файла ресурса. (Для совместимости с Delphi 1 двоичные *dfm*-файлы являются 16-разрядными файлами ресурсов. Delphi преобразует их при сборке проекта в 32-разрядные ресурсы.)

Форма директивы `$Resource` с двумя параметрами применяется только для *dpr*-файлов. Если директива присутствует в начале *dpr*-файла проекта и содержит два имени файла, первый файл присоединяется к проекту, второй – отображается в менеджере проекта. Когда IDE Delphi компилирует проект, скрипт ресурсов компилируется в *res*-файл, который может быть присоединен к проекту. Компиляция скрипта ресурсов – уникальное средство IDE. Компилятор командной строки не выполняет компиляцию *rc*-файла. (Дополнительную информацию о компиляторе командной строки можно найти в приложении А.)

Помещайте имена файлов в одинарные кавычки, если имя содержит пробелы или другие необычные символы или если используется форма директивы с двумя именами файлов.

$RunOnly, директива компилятора

Синтаксис

```
{$RunOnly Off}  // по умолчанию
{$RunOnly On}
```

Диапазон

Проект.

Описание

Директива `$RunOnly` помещается в *dpk*-файл пакета, для того чтобы пометить его как пакет времени выполнения. Такой пакет нельзя загрузить в IDE Delphi. Вы можете только присоединить его к приложению или библиотеке.

При создании компонента помещайте его модули в пакет времени выполнения, а редакторы свойств и компонентов – в пакет для разработки. Таким образом приложение может присоединить только пакет времени выполнения и исключить код разработки, который приложению не требуется.

Смотрите также

Директива компилятора `$DesignOnly`.

$SafeDivide, директива компилятора

Синтаксис

```
{$U-}              // по умолчанию
{$SafeDivide Off} // по умолчанию
{$U+}
{$SafeDivide On}
```

Диапазон

Локальный.

Описание

Когда директива `$SafeDivide` включена, Delphi гарантирует, что печально известная ошибка FDIV процессоров Pentium не окажет влияния на вашу программу. При наличии директивы `$SafeDivide` деление с плавающей точкой выполняется специальной функцией. Эта функция, при первом ее вызове вашей программой, проверяет наличие в процессоре ошибки FDIV. Если аппаратура FPU имеет дефект, все последующие операции деления осуществляются программным способом. Если процессор работает правильно, последующие деления выполняются аппаратно, а проверяющая подпрограмма устанавливает переменную `TestFDIV` в –1, если процессор работает некорректно, и в 1 – при правильной работе.

Ошибка FDIV влияет только на старые процессоры Pentium и только на определенные операции деления. Сегодня большинство программ могут игнорировать эту проблему.

Смотрите также

Переменная `TestFDIV`.

$StackFrames, директива компилятора

Синтаксис

```
{$W-}              // по умолчанию
{$StackFrames Off} // по умолчанию
{$W+}
{$StackFrames On}
```

Диапазон

Локальный.

Описание

Директива $StackFrames уведомляет компилятор о необходимости генерировать стековый фрейм для всех вызовов подпрограмм (кроме подпрограмм, написанных на языке ассемблера). По умолчанию стековый фрейм создается только тогда, когда это требуется для локальных переменных.

Некоторые средства отладки требуют наличия стековых фреймов. Если вы не используете подобное средство, можно сохранить установку по умолчанию, которая дает несколько более эффективный код.

$T, директива компилятора

Смотрите

Директива компилятора $TypedAddress.

$TypedAddress, директива компилятора

Синтаксис

```
{$T-}               // по умолчанию
{$TypedAddress Off} // по умолчанию
{$T+}
{$TypedAddress On}
```

Диапазон

Локальный.

Описание

Если директива $TypedAddress включена, оператор @ возвращает типизированный адрес переменной или подпрограммы. По умолчанию оператор @ возвращает обобщенный тип Pointer. Даже если директива $TypedAddress включена, присваивания типизированного указателя переменной типа Pointer допускаются, так же как и присваивания значения типа Pointer конкретному типу указателя.

Советы и приемы

- Включайте директиву $TypedAddress, т. к. это способствует хорошей практике программирования и аккуратному использованию указателей. Ненадежные присваивания указателей смогут быть выявлены на этапе компиляции.
- Функция Addr всегда возвращает нетипизированный указатель Pointer. При включенной директиве $TypedAddress используйте Addr,

если требуется нетипизированный указатель, и @ – если требуется типизированный адрес.

Пример

```
var
  P: PInteger;
  Q: ^Double;
  I: Integer;
begin
{$TypedAddress On}
  P := @I;
  Q := @I; // не допускается, так как типы не совпадают
{$TypedAddress Off}
  Q := @I; // допускается, хотя типы и не совпадают
```

Смотрите также

Оператор @, функция Addr, тип Pointer.

$TypeInfo, директива компилятора

Синтаксис

```
{$M-}           // по умолчанию
{$TypeInfo Off} // по умолчанию
{$M+}
{$TypeInfo On}
```

Диапазон

Локальный.

Описание

Если директива $TypeInfo включена, классы компилируются с информацией времени выполнения. Это означает, что класс может иметь секцию published, и компилятор сохраняет дополнительную информацию о классе в таблице виртуальных методов класса.

Любой класс, который является наследником класса с RTTI, также имеет RTTI, поэтому директива $TypeInfo требуется только для базового класса. В Delphi класс TPersistent компилирован с директивой $TypeInfo, поэтому все производные классы (в том числе TComponent, TForm, все компоненты VCL и связанные с ними классы) имеют RTTI.

Смотрите также

Ключевое слово Class.

$U, директива компилятора

Смотрите

Директива компилятора $SafeDivide.

$Undef, директива компилятора

Синтаксис

```
{$Undef Имя}
```

Диапазон

Локальный.

Описание

Директива $Undef аннулирует определение *Имени*, так что последующие директивы {$IfDef *Имя*} возвращают ложь.

Смотрите также

Директивы компилятора $Define, $Else, $Endif, $IfDef, $IfNDef и $IfOpt.

$V, директива компилятора

Смотрите

Директива компилятора $VarStringChecks.

$VarStringChecks, директива компилятора

Синтаксис

```
{$V+}                  // по умолчанию
{$VarStringChecks On} // по умолчанию
{$V-}
{$VarStringChecks Off}
```

Диапазон

Локальный.

Описание

Обычно тип var параметра должен точно совпадать с типом фактического параметра. Это ограничение гарантирует безопасность типов при изменении подпрограммой значения var параметра. Если директива $VarStringChecks выключена, Delphi ослабляет ограничения для коротких строк, позволяя передать в подпрограмму, требующую параметр-переменную типа short string, любой аргумент типа short string.

Если директива $VarStringChecks выключена, необходимо позаботиться о том, чтобы подпрограмма не превысила размер аргумента типа short string. Один из способов состоит в том, чтобы передавать только аргументы максимально возможной длины (например, типа string[255] или ShortString). Еще лучше использовать вместо коротких строк длинные.

Советы и приемы

Выключайте директиву $VarStringChecks только тогда, когда это требуется для обратной совместимости. В современных программах рекомендуется применять длинные строки, однако если короткие строки необходимы, старайтесь использовать тип OpenString вместо отключения $VarStringChecks.

Смотрите также

Типы AnsiString, OpenString и ShortString, ключевое слово String.

$W, директива компилятора

Смотрите

Директива компилятора $StackFrames.

$Warnings, директива компилятора

Синтаксис

```
{$Warnings On}  // по умолчанию
{$Warnings Off}
```

Диапазон

Локальный, применяется к подпрограмме в целом.

Описание

Директива компилятора $Warnings включает или отключает предупреждения компилятора. Компилятор выдает предупреждения для кода, который производит впечатление некорректного, не нарушая при этом ни одного из синтаксических правил Delphi Pascal. Например, компилятор генерирует предупреждение, если подпрограмма использует переменную до того, как ей было присвоено какое-либо значение, или предупреждает о том, что подпрограмма может не во всех ситуациях вернуть значение. Если вы знаете, что данное предупреждение безосновательно, можете подавить его директивой $Warnings.

Советы и приемы

- Не следует отключать предупреждения компилятора для всего файла или проекта в целом. Предупреждения компилятора полезны и предупреждают о многих распространенных ошибках. Отключайте предупреждения только тогда, когда вы уверены в безопасности этого, и в любом случае делайте это только для конкретной подпрограммы.
- Вместо отключения предупреждений попытайтесь переписать код таким образом, чтобы компилятор не генерировал предупреждение. Иногда это потребует переписывания кода, который вы счита-

ете «мертвым» и никогда не используемым, однако альтернатива может быть хуже. Отключая предупреждения, можно пропустить ошибку.

Пример

```
// Отключение предупреждения о неопределенности переменной Result.
{$Warnings off}
function RunProgram(const Path: string): DWORD;
var
  Code: DWORD;
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
begin
  FillChar(StartupInfo, SizeOf(StartupInfo), 0);
  StartupInfo.cb := SizeOf(StartupInfo);
  if not CreateProcess(PChar(Path), nil, nil, nil, False, 0, nil, nil,
                       StartupInfo, ProcessInfo)
  then
    RaiseLastWin32Error
    // Вызывает исключительную ситуацию, поэтому функция никогда не
    // завершается, и результат не имеет значения.
    // Использование директивы $Warnings – это одно из возможных решений;
    // другой вариант – установка значения Result здесь. Мертвый код
    // вполне удовлетворяет компилятор и не создает проблемы отключения
    // предупреждений для всей функции.
  else
  begin
    WaitForSingleObject(ProcessInfo.hProcess, Infinite);
    GetExitCodeProcess(ProcessInfo.hProcess, Code);
    Result := Code;
  end;
end;
{$Warnings on}
```

Смотрите также

Директива компилятора `$Hints`.

$WeakPackageUnit, директива компилятора

Синтаксис

```
{$WeakPackageUnit Off}
{$WeakPackageUnit On}
```

Диапазон

Файл.

Описание

Помещайте директиву `$WeakPackageUnit` в модуль, включенный в пакет, если выполняются следующие условия:

- Пакет будет использоваться в большом количестве приложений.
- Модуль не вызывается большинством этих приложений.
- Модулю требуется DLL, которая еще отсутствует у большинства пользователей.

Обычно пакету требуется DLL, потому что она требуется модулю. Это значит, что приложению, которое использует пакет, также требуется эта DLL. Другими словами, приложение должно поставляться вместе с DLL, даже если она приложением не используется.

Директива `$WeakPackageUnit` решает эту проблему путем «слабого связывания» модуля с пакетом. А именно – модуль фактически не присоединяется к *bpl*-файлу. Вместо этого модуль полностью помещается в *dcp*-файл, в котором обычно содержится только интерфейсная часть модуля. Если приложение не обращается к модулю, он не присоединяется к приложению, и ему не требуется связанная с модулем DLL. Если приложение использует модуль, он статически присоединяется к приложению и ему требуется связанная с модулем DLL.

Если в пакет включен другой модуль (`Unit2`), который использует «слабо связанный» модуль (`Unit1`), `Unit2` разрушает ограничения «слабого» связывания. Приложение, которое использует пакет, всегда присоединяет `Unit2` (поскольку `Unit2` всегда находится в пакете), которому требуется `Unit1`, в свою очередь требующий DLL. Поэтому «слабосвязанный» модуль должен стоять особняком в пакете, чтобы «слабое связывание» могло работать корректно.

«Слабо связанный» модуль не может содержать глобальные переменные, раздел инициализации или раздел завершения.

Директива `$WeakPackageUnit` накладывает множество ограничений, и потому применяется редких случаях. Однако иногда эта директива абсолютно необходима. Если вы не создаете пакеты для продажи другим Delphi-разработчикам, не занимайтесь ее изучением.

Смотрите также

Директива `Package`.

$WriteableConst, директива компилятора

Синтаксис

```
{$J+}                // по умолчанию
{$WriteableConst On} // по умолчанию
{$J-}
{$WriteableConst Off}
```

Диапазон

Локальный.

Описание

Когда определяется константа с указанием типа, Delphi выделяет для этой константы область памяти. Программа может изменить значение такой «константы», как если бы она была переменной. Разница между типизированной константой и локальной переменной состоит в том, что типизированная константа, определенная в подпрограмме, сохраняет свое значение при различных вызовах подпрограммы.

Если директива `$WriteableConst` отключена, предотвращается возможность присваивания значения типизированной константе, таким образом, она становится истинной константой. Однако для типизированных констант в подпрограмме замены нет, поэтому отключайте `$WriteableConst` осторожно. Отключайте эту возможность только для определенных констант, значение которых вы хотите защитить, и не отключайте директиву глобально в проекте или в целом файле.

```
const
     {$WriteableConst off}
     BoolNames: array[Boolean] of string = ('Никак', 'Как хотите');
     {$WriteableConst on}
```

Смотрите также

Ключевое слово `Const`.

$X, директива компилятора

Смотрите

Директива компилятора `$ExtendedSyntax`.

$Y, директива компилятора

Смотрите

Директивы компилятора `$DefinitionInfo` и `$ReferenceInfo`.

$Z, директива компилятора

Смотрите

Директива компилятора `$MinEnumSize`.

Приложение А

Утилиты командной строки

Интерактивные среды разработки – это хорошо, но и командную строку пока не стоит сбрасывать со счетов. Часто легче выполнить компиляцию большого проекта с помощью компилятора, чем в IDE. В этой главе рассказано об эффективном применении компилятора и других утилит командной строки.

dcc32.exe, компилятор

Формат вызова

```
dcc32 [-Aunit=alias] [-B] [-CC] [-CG] [-Dsyms] [-Edirectory]
  [-Faddress] [-GD] [-GP] [-GS] [-H] [-Ipaths] [-J] [-JP] [-Kaddr]
  [-LEdirectory] [-LNdirectory]
  [-LUpackage] [-M] [-Ndirectory] [-Opaths] [-P] [-Q] [-Rpaths] [-TXext]
  [-Upaths] [-V] [-VN] [-VR] [-W] [-Z] [-$A+|-] [-$B+|-] [-$C+|-]
  [-$D+|-] [-$G+|-] [-$H+|-] [-$I+|-] [-$J+|-] [-$L+|-]
  [-$M+|-|minStackSize[,maxStackSize]] [-$O+|-] [-$P+|-]
  [-$Q+|-] [-$R+|-] [-$T+|-] [-$U+|-] [-$V+|-] [-$W+|-]
  [-$X+|-] [-$Y+|-YD] [-$Z+|-|1|2|4] file... [options...]
```

Описание

Запускаемый из командной строки компилятор Delphi *dcc32.exe* используется также и в IDE. Для управления компилятором требуется указать ключи в командной строке или в конфигурационном (*.cfg*) файле. IDE автоматически создает конфигурационный файл для каждого проекта, чтобы облегчить компиляцию проекта или модуля из командной строки с теми же настройками, что и в IDE.

Можно располагать ключи и имена файлов в командной строке в любом порядке. Компилятор читает все ключи до начала компиляции

файлов. Имена файлов могут относиться к любым исходным файлам программ, библиотек, модулей или пакетов. В отличие от IDE, с помощью компилятора командной строки можно скомпилировать одиночный исходный файл модуля (*.pas*). Если имя файла соответствует проекту, библиотеке или пакету, компилятор также выполняет сборку необходимых модулей в конечный *exe*-, *dll*- или *bpl*-файл. Если опущено расширение в имени исходного файла, Delphi сначала пытается скомпилировать файл с расширением *.pas*, а если таковой не найден, то файл с расширением *.dpr*. Для компиляции пакета должно быть указано расширение *.dpk*.

Если в командной строке не указано ни одного имени файла, компилятор выдает краткую сводку возможных ключей и формата вызова.

Компилятор извлекает настройки из четырех источников. Проверяются все четыре источника в порядке, указанном ниже, поэтому последующие источники могут переопределять настройки, указанные ранее:

- Глобальный конфигурационный файл *dcc32.cfg*, расположенный в каталоге *bin* Delphi. Можно изменить этот файл, указав в нем настройки, применяемые для всех проектов, например, ключ `-u` для перечисления каталогов, в которых Delphi будет выполнять поиск *dcu*-файлов.

- Локальный конфигурационный файл *dcc32.cfg*, находящийся в текущем каталоге.

- Файл конфигурации проекта, расположенный в том же каталоге, что и исходный файл проекта. Его имя совпадает с именем файла проекта и имеет расширение *.cfg*. Заметьте, что IDE автоматически создает *cfg*-файл для каждого проекта, сохраняя в нем настройки проекта.

- Можно переопределить настройки конфигурационных файлов, добавив или изменив ключи в командной строке. Ключи могут быть указаны до или после имени исходного файла.

Конфигурационный файл может содержать любое количество ключей и строк, в которых они расположены. Для большей ясности лучше помещать отдельные ключи в отдельных строках файла.

Каждый ключ может начинаться с дефиса (`-`) или косой черты (`/`). Если употребляется косая черта, можно объединять ключи в одном аргументе командной строки, а если дефис – ключи должны быть разделены пробелами. В обоих случаях можно применять однобуквенные директивы компилятора в качестве переключателей командной строки, например `$A+`. Директивы компилятора объединяются запятыми, например `$A+, B+, M-`. Переключатели не чувствительны к регистру, поэтому `/H` аналогичен `/h`.

Будьте аккуратны при использовании директив-переключателей с компилятором командной строки. В отличие от директив компилятора, указанных в исходном коде, директивы-переключатели в командной строке и *cfg*-файлах не проверяются на наличие ошибок. В качестве имени переключателя может выступать любая буква, а неверные переключатели игнорируются. После буквы допускается любой символ, и большинство из них интерпретируется как +.

В командной строке может быть перечислено произвольное количество имен файлов. Компилятор компилирует исходные файлы по одному. Если исходный файл использует другие модули, компилятор проверяет актуальность этих модулей и по необходимости сначала компилирует их.

В отличие от IDE, компилятор командной строки не компилирует скрипты ресурсов, перечисленные в директивах компилятора `$R` и `$Resource`. Приходится запускать компилятор ресурсов отдельно (компилятор ресурсов описан в следующем разделе.)

Ниже перечислены все ключи командной строки:

`-A`*`unit=alias;...`*

Создать псевдоним модуля *unit* так, что все ссылки на модуль в объявлении `uses` фактически становятся ссылками на псевдоним (*alias*). Этот ключ чаще всего используется, когда новая версия модуля имеет имя, отличное от более ранней. Можно перечислить несколько пар модуль–псевдоним, разделив их точками с запятой, например `AWinTypes=Windows;WinProcs=Windows`.

`-B`

Компилировать все используемые модули, даже если *dcu*-файлы актуальны (up to date). Другой способ заставить Delphi компилировать все модули проекта – удалить предварительно его *dcu*-файлы.

`-CC`

Создать консольное приложение (директива компилятора `$AppType Console`).

`-CG`

Создать графическое приложение (директива компилятора `$AppType GUI`).

`-D`*`sym1;sym2;...`*

Определить *sym1*, *sym2* и т. д. как символы условной компиляции. Можно определить несколько символов, разделив их точкой с запятой (`;`). Ключ `-D` эквивалентен директиве компилятора `$Define`.

`-E`*`directory`*

Сохранять выходной *exe*- или *dll*-файл в каталоге *directory*. По умолчанию это текущий каталог.

`-F`*address*

Искать строку исходного текста, соответствующую адресу, полученному в диалоговом окне ошибки времени выполнения. Если приложение выдает ошибку по определенному адресу, и нужно найти источник ошибки, перекомпилируйте проект с теми же ключами, которые использовались ранее, указав переключатель `-F`. Компилятору требуется отладочная информация (`-$D+`) для нахождения номера строки исходного текста.

`-GD`

Создавать подробный *map*-файл, в котором перечислены все сегменты, общие символы и адреса кода каждой строки каждого модуля, для которого включена отладочная информация (директива компилятора `$DebugInfo`). *Map*-файл сохраняется в выходном каталоге проекта (ключ `-E` или `-LE`), его имя совпадает с именем проекта, и он имеет расширение *.map*.

`-GP`

Создавать *map*-файл среднего размера, в котором перечислены все сегменты и общие символы. *Map*-файл сохраняется в выходном каталоге проекта (ключ `-E` или `-LE`), его имя совпадает с именем проекта, и он имеет расширение *.map*.

`-GS`

Создавать краткий *map*-файл, в котором перечислены только сегменты, т. е. начальный адрес и размер сегмента кода и данных каждого модуля. *Map*-файл сохраняется в выходном каталоге проекта (ключ `-E` или `-LE`), он имеет расширение *.map*, а его имя совпадает с именем проекта.

`-H`

Генерировать подсказки компилятора (директива компилятора `$Hints`).

`-I`*paths*

Искать исходные файлы в каталогах, указанных в *paths*. Имена каталогов разделяются точкой с запятой.

`-J`

Генерировать для каждого модуля *obj*-файл языка C в дополнение к *dcu*-файлу.

`-JP`

Генерировать для каждого модуля *obj*-файл языка C++ в дополнение к *dcu*-файлу.

`-K`*addr*

Установить базовый адрес, равный целому значению *addr* (директива компилятора `$ImageBase`). Например `-K$400000`.

`-LE`*`directory`*

Сохранить выходной *bpl*-файл в указанном каталоге *directory*. По умолчанию это текущий каталог.

`-LN`*`directory`*

Сохранить выходной *dcp*-файл в указанном каталоге *directory*. По умолчанию используется текущий каталог.

`-LU`*`package`*

Выполнить связывание с пакетом времени выполнения *package*. Проект динамически связывается с модулями пакета и не включает копии модулей в *exe*- или *dll*-файл проекта.

`-M`

Создать файлы модулей, компилируя их только в том случае, если это необходимо.

`-N`*`directory`*

Сохранить выходные *dcu*-файлы в указанном каталоге *directory*. По умолчанию это текущий каталог.

`-O`*`paths`*

Искать *obj*-файлы в каталогах, указанных в *paths*. Имена каталогов разделяются точкой с запятой.

`-P`

Искать модуль по короткому (8.3) имени файла в дополнение к длинному имени. Используйте этот ключ, если файлы хранятся в файловой системе, не поддерживающей длинные имена, например, на старом файл-сервере NetWare.

`-Q`

Выполнить «тихую» компиляцию и не перечислять имена файлов в процессе компиляции.

`-R`*`paths`*

Искать *res*-файлы в каталогах, указанных в *paths*. Имена каталогов разделяются точкой с запятой.

`-TX`*`ext`*

Установить расширение выходного файла в *ext*. В отличие от директивы компилятора `$E` (которая работает только для библиотек), ключ `-TX` применим и к программам, и к библиотекам. Если ведущая точка (`.`) опущена в *ext*, компилятор вставляет ее автоматически. Например `-TXscr`.

`-U`*`paths`*

Искать *dcu*-файлы в каталогах, указанных в *paths*. Имена каталогов разделяются точкой с запятой.

`-V`

Сохранить в *exe*-, *dll*- или *bpl*-файле информацию для Turbo Debugger. Встроенному отладчику Delphi не нужна эта информация – ее использует только Turbo Debugger. Применение этого ключа значительно увеличивает размер файла, но не влияет на производительность.

`-VN`

Генерировать пространства имен C++. Указывайте этот ключ только в том случае, если модуль Delphi будет включен в проект C++, т. к. это увеличивает размер выходного файла и время компиляции.

`-VR`

Генерировать символы для удаленной отладки (*rsm*-файл). Применение этого ключа не влияет на *exe*- или *dll*-файл проекта, однако *rsm*-файл может быть большим. Включайте этот ключ, только если используется удаленный отладчик Borland.

`-W`

Генерировать предупреждения компилятора (директива компилятора `$Warnings`).

`-Z`

Отключить скрытую перекомпиляцию пакетов нижнего уровня (директива компилятора `$ImplicitBuild`).

`-$A-`
`-$A+`

Выровнять поля записей, классов и объектов (директива компилятора `$Align`). По умолчанию `-$A+`.

`-$B-`
`-$B+`

Вычислять логические выражения без использования сокращенной схемы (директива компилятора `$BoolEval`). По умолчанию `-$B+`.

`-$C-`
`-$C+`

Компилировать допущения (директива компилятора `$Assertions`). По умолчанию `-$C+`.

`-$D-`
`-$D+`

Генерировать отладочную информацию для записи в *dcu*-файл (директива компилятора `$DebugInfo`). По умолчанию `-$D+`.

`-$G-`
`-$G+`

Этот ключ не реализован (разъяснения приведены в описании директивы компилятора `$ImportedData`). По умолчанию `-$G+`.

`-$H-`

`-$H+`

Тип `string` обозначает `AnsiString`. Выключите эту настройку, чтобы сделать `string` эквивалентом `ShortString` (директива компилятора `$LongStrings`). По умолчанию `-$H+`.

`-$I-`

`-$I+`

Разрешить проверку операций ввода/вывода, в результате чего сбои в подпрограммах ввода и вывода вызывают ошибки времени выполнения (директива компилятора `$IOChecks`). По умолчанию `-$I+`.

`-$J-`

`-$J+`

Константы, объявленные как типизированные, доступны для записи во время выполнения программы (директива компилятора `$WriteableConst`). По умолчанию `-$J+`.

`-$L-`

`-$L+`

Сохранить информацию о локальных символах как часть отладочной информации *dcu*-файла (директива компилятора `$LocalSymbols`). По умолчанию `-$L+`.

`-$M-`

`-$M+`

Генерировать информацию о типе времени выполнения для всех объявлений классов (директива компилятора `$TypeInfo`). По умолчанию `-$M-`.

`-$M`*`minStackSize`*

`-$M`*`minStackSize`*`,`*`maxStackSize`*

Установить минимальный размер стека (директива компилятора `$MinStackSize`) и, если он указан, максимальный размер стека (директива компилятора `$MaxStackSize`). По умолчанию это 16384 и 1047576 байт (`-$M$4000`, `$100000`).

`-$O-`

`-$O+`

Включить оптимизацию компилятора (директива компилятора `$Optimizations`). По умолчанию `-$O+`.

`-$P-`

`-$P+`

Рассматривать параметры подпрограмм `ShortString` как `OpenString` (директива компилятора `$OpenStrings`). По умолчанию `-$P+`.

`-$Q-`

`-$Q+`

Включить проверку на переполнение целого или перечисления (директива компилятора `$OverflowChecks`). По умолчанию `-$Q-`.

`-$R-`

`-$R+`

Включить проверку границ массивов (директива компилятора `$RangeChecks`). По умолчанию `-$R-`.

`-$T-`

`-$T+`

Оператор `@` возвращает типизированный указатель (директива компилятора `$TypedAddress`). По умолчанию `-$T-`.

`-$U-`

`-$U+`

Выполнять безопасное вещественное деление, обходя печально известную ошибку FDIV процессора Pentium (директива компилятора `$SafeDivide`). По умолчанию `-$U-`.

`-$V-`

`-$V+`

Если данная настройка отключена, ослабляются ограничения на строковые параметры `var`, допуская аргументы типа `string`, максимальный размер которых не совпадает с максимальным размером параметра (директива компилятора `$VarStringChecks`). По умолчанию `-$V+`.

`-$W-`

`-$W+`

Генерировать стековые фреймы (директива компилятора `$StackFrames`). По умолчанию `-$W-`.

`-$X-`

`-$X+`

Включить расширения синтаксиса языка Паскаль (директива компилятора `$ExtendedSyntax`). По умолчанию `-$X+`.

`-$Y-`

`-$Y+`

`-$YD`

Генерировать отладочную информацию и информацию о ссылках (директива компилятора `$ReferenceInfo` и `$DebugInfo`). По умолчанию `-$Y+`.

-$Z-
-$Z+
-$Z1
-$Z2
-$Z4

Установить минимальный размер перечисляемых типов, равный 1, 2 или 4 байтам (директива компилятора $MinEnumSize). -$Z- эквивалентно -$Z1, а -$Z+ эквивалентно -$Z4. По умолчанию -$Z1.

brcc32.exe, компилятор ресурсов

Формат вызова

```
brcc32 [-16] [-31] [-32] [-w32] [-ccodepage] [-dname[=string]
  [-fofilename] [-h] [-?] [-ipaths] [-llanguage] [-m] [-r]
  [-v] [-x] file
```

Описание

Компилятор ресурсов компилирует скрипт ресурсов (файл с расширением *.rc*) в файл ресурсов (*.res*). Компилятор ресурсов командной строки может оказаться удобнее, чем менеджер проектов из IDE, когда требуется автоматизировать сборку проектов.

Формат скрипта ресурсов рассмотрен в документации Microsoft Platform SDK. Эта информация доступна на сайте *msdn.microsoft.com* и на компакт-дисках Microsoft Software Development Network.

Ключи компилятора не чувствительны к регистру. Каждый ключ может начинаться с дефиса (-) или с косой черты (/). В отличие от компилятора командной строки, объединять ключи нельзя даже при использовании косой черты.

Остальная часть раздела описывает ключи командной строки компилятора ресурсов.

-16
-31

Формировать 16-битовый *res*-файл (для совместимости с Windows 3.x).

-32
-w32

Формировать 32-битовый *res*-файл (для Windows 9x, Windows NT, Windows 2000). Это установка по умолчанию.

-c*codepage*

Установить кодовую страницу по умолчанию, равную *codepage*. Кодовая страница определяет интерпретацию многобайтовых символов. Список поддерживаемых кодовых страниц приведен в документации по Windows API.

`-d`*`name`*`[=`*`string`*`]`

Определить *name*, как в директиве `#define` исходного текста скрипта. По умолчанию *name* присваивается значение 1. Можно указать другое значение для имени после знака равенства.

`-fo`*`filename`*

Установить имя выходного файла в *filename*. По умолчанию файл имеет то же имя, что и входной файл, но с расширением *.res*. Можно указать другое расширение.

`-h`

`-?`

Выдать краткое сообщение о ключах командной строки.

`-i`*`paths`*

Искать файлы, указанные в директиве `#include` исходного текста скрипта, в каталогах *paths*. Имена каталогов разделяются точкой с запятой.

`-l`*`language`*

Установить язык по умолчанию *language*. Список поддерживаемых языков приведен в документации по Windows API.

`-m`

Включить поддержку многобайтовых символов. Скрипт ресурсов состоит из 8-байтовых символов ANSI, однако некоторым ресурсам требуется Unicode. Для создания символов Unicode можно использовать многобайтовые символы.

`-r`

Игнорируется. Оставлена для совместимости с предыдущими версиями компилятора ресурсов.

`-v`

Наглядный режим: выдает имена файлов и всех ресурсов в процессе их компиляции. В отличие от других ключей, `-V` в верхнем регистре игнорируется и не совпадает с ключом `-v` в нижнем регистре.

`-x`

Игнорировать переменную окружения `INCLUDE`. По умолчанию компилятор ресурсов ищет в переменной окружения `INCLUDE` пути для файлов, указанных в директиве `#include` (как и для ключа `-i`).

convert.exe, конвертор файлов DFM

Формат вызова

```
convert [-b] [-i] [-s | -t] имена файлов... | @filelist
```

Описание

Delphi может сохранять описания форм (*dfm*-файлы) в двоичном или текстовом формате. IDE позволяет программисту выбрать предпочтительный формат. Кроме того, можно использовать программу *convert.exe* для преобразования двоичного файла в текстовый или текстового в двоичный.

В Windows NT также имеется программа с именем *convert.exe*. Убедитесь, что запускаете «правильную» утилиту *convert*.

Утилита *convert.exe* читает каждый из указанных файлов, преобразует его и записывает результат в новый файл. Если входной файл имеет расширение *.dfm*, он должен иметь двоичный формат, и *convert.exe* создает текстовый файл с расширением *.txt*. Если входной файл имеет расширение *.txt*, он должен иметь текстовый формат, преобразуется в двоичный формат и записывается в файл с расширением *.dfm* и тем же именем.

Если *dfm*-файл имеет текстовый формат, или *txt*-файл является двоичным файлом, *convert.exe* печатает сообщение об ошибке и пропускает файл.

Можно перечислить любое количество файлов и масок, и *convert.exe* обработает их все. Если требуется преобразовать несколько файлов, перечислите их или задайте маски в отдельном текстовом файле и укажите имя файла, содержащего список, в командной строке после знака `@`.

Без указания ключей *convert.exe* работает так же, как в Delphi 4 и более ранних версиях. Новое в Delphi 5 – это переключатели командной строки для управления процессом преобразования. Ключи не чувствительны к регистру. Каждый ключ может начинаться с дефиса (`-`) или с косой черты (`/`). В отличие от компилятора командной строки, объединять ключи нельзя, даже при использовании косой черты.

В остальной части раздела описаны ключи командной строки:

`-b`

Всегда преобразовывать в двоичный формат, независимо от расширения файла.

`-i`

Преобразование на месте. Вместо копирования *dfm*-файла в *txt*-файл или наоборот каждый файл преобразуется на месте, и его содержимое заменяется результатом преобразования. Без ключей `-b` или `-t` ключ `-i` переводит все двоичные файлы в текстовые, а все текстовые файлы в двоичные.

`-s`

Преобразование файлов в подкаталогах. Для каждого файла, указанного в командной строке или в списке файлов, утилита *convert* выполняет поиск файлов с тем же именем в подкаталогах каталога, содержащего данный файл. Если имя файла не содержит каталога или имени логического диска, поиск выполняется в текущем рабочем каталоге.

`-t`

Всегда преобразовывать в текстовый формат, независимо от расширения файла.

tdump.exe, утилита создания дампа

Формат вызова

```
tdump [-a] [-a7] [-boffset] [-C] [-d] [-e] [-ea[:v]] [-ed]
  [-ee[=symbol]] [-eiid] [-el] [-em[=symbol] [-em.[module]]
  [-ep] [-er] [-ex] [-h] [-iid] [-l] [-le[=symbol]] [-li[=symbol]]
  [-m][-o] [-oiid] [-oxid] [-q] [-r] [-R]
  [-s[number | boffset | eoffset | f | s | u ] [-?] [=str]]
  [-um] [-v] [-xid]
  [inputfile [listfile]] [options...]
```

Описание

Программа *tdump.exe* выводит полезную информацию о двоичных файлах, в частности о файлах с расширениями *.exe*, *.dll*, *.bpl*, *.obj* и *.lib*. Первое имя файла, указанное в командной строке, – это входной файл. (Нельзя использовать маски для указания нескольких файлов.) Второе имя файла – выходной файл. Если не указан файл листинга *listfile*, *tdump* выводит информацию на стандартное устройство вывода. Вывод *tdump* обычно довольно объемист, поэтому есть смысл использовать *listfile*. Ключи и имена файлов могут располагаться произвольным образом.

В отличие от переключателей других утилит командной строки, переключатели *tdump* чувствительны к регистру. Можно обозначать переключатель дефисом (-) или косой чертой (/). В отличие от компилятора, переключатели нельзя объединять, даже если используется косая черта.

Объектные файлы могут быть созданы либо в формате COFF (Common Object File Format, универсальный формат объектных файлов), либо OMF (Object Module Format, формат объектных модулей фирмы Intel). COFF используется в Visual C++ и в других компиляторах, но ни в одном из компиляторов фирмы Borland. OMF применяется в Delphi, Borland C++ и других компиляторах. Если задать неверные ключи для данного формата файла, *tdump* печатает сообщение об ошибке и завершает работу.

Дополнительную информацию о формате OMF можно найти на сайте для разработчиков фирмы Intel (*http://developer.intel.com/*). Там же доступна информация о формате PE (Portable Executable, переносимый исполняемый файл), который используется для программ и DLL Win32. Подробное описание формата файлов PE для Win32 и формата объектных файлов COFF приведено в книге Мэта Петрика (Matt Pietrek) «Windows 95 System Programming Secrets» (IDG books, 1995).

Без указания переключателей *tdump* выводит содержимое входного файла *inputfile* в формате, соответствующем типу файла, т. е. исполняемого файла DOS, исполняемого файла PE (Win32) или DLL, объектного или библиотечного файла OMF, объектного или библиотечного файла COFF. Если *tdump* не может определить тип файла, выводится содержимое файла в шестнадцатеричном формате (ключ `-h`).

`-a`

Вывести содержимое *inputfile* в виде 8-битовых символов ANSI. Вместо управляющих символов печатаются точки (.).

`-a7`

Вывести содержимое *inputfile* в виде 7-битовых символов ANSI. Вместо неотображаемых и управляющих символов печатаются точки (.).

`-b`*offset*

Начинать чтение *inputfile* с байтового смещения *offset*.

`-C`

inputfile имеет формат COFF. Имя файла должно заканчиваться расширением *.obj* или *.lib*. Этот ключ можно применять для изучения объектных и библиотечных файлов из Visual C++ и других компиляторов.

`-d`

Вывести 32-битовую отладочную информацию из объектного или библиотечного файла OMF. Имя файла должно заканчиваться расширением *.obj* или *.lib*.

`-e`

Вывести информацию об *inputfile* как об исполняемом файле. Файл может быть исполняемым файлом или DLL для Windows, DOS или OS/2. Выводится заголовок файла, заголовки секций, импортируемые символы, списки экспортируемых символов и ресурсов.

`-ea[:v]`

Включить в выходную информацию список всех символов, экспортируемых *exe*- или *dll*-файлом. По умолчанию выводятся только именованные символы, отсортированные в алфавитном порядке. При указании переключателя `-ea` список не сортируется; с пере-

ключателем `-ea:v` – сортируется в порядке относительных виртуальных адресов.

`-ed`

Не выводить отладочную информацию *exe*- или *dll*-файла.

`-ee[=`*`symbol`*`]`

Выводить только список символов, экспортируемых *exe*- или *dll*-файлом. При указании *symbol* выводится только этот символ, если он экспортируется данным файлом.

`-ei`*`id`*

Выводить только таблицу *id* файла форматов LE или LX (*exe*-файл OS/2 или *vxd*-файл Windows 3.x). Параметр *id* может быть равен `HDR`, `FIX`, `NAM` или `ENT` (*id* чувствителен к регистру).

`-el`

Не печатать номера строк из *exe*-файла.

`-em[=`*`symbol`*`]`

Выводить только список символов, импортируемых *exe*- или *dll*-файлом. При указании *symbol* выводится только этот символ, если он импортируется данным файлом.

`-em.[=`*`module`*`]`

Выводить только список модулей, импортируемых *exe*- или *dll*-файлом. При указании *module* выводится только этот модуль, если он импортируется данным файлом.

`-ep`

Не выводить PE-заголовок *exe*- или *dll*-файла.

`-er`

Не выводить запись перемещений *exe*- или *dll*-файла.

`-ex`

Не выводить новый заголовок *exe*- или *dll*-файла.

`-h`

Вывести содержимое *inputfile* в шестнадцатеричном формате.

`-i`*`id`*

Включить в выходную информацию таблицы 16-разрядного отладчика Turbo Debugger, обозначенные *id*. Для вывода на стандартное устройство вывода (не в *listfile*) списка обозначений отладочных таблиц укажите вопросительный знак (`?`). Каждая таблица идентифицируется одной буквой (например, A – таблица символов, B – таблица модулей, C – таблица файлов исходных текстов и т. д.). Перечислив несколько букв, можно получить несколько таблиц.

`-l`

Интерпретировать *inputfile* как библиотеку OMF. Файл должен иметь расширение *.lib*. Компиляторы C и C++ фирмы Borland генерируют библиотеки в формате OMF, однако не все компиляторы поддерживают этот формат.

`-le[=symbol]`

Вывести запись определения экспорта (EXPDEF) для всех символов, имя которых содержит *symbol*. Если имя символа не указано, *tdump* печатает все записи EXPDEF.

`-li[=symbol]`

Вывести запись определения импорта (IMPDEF) для всех символов, имя которых содержит *symbol*. Если имя символа не указано, *tdump* печатает все записи IMPDEF.

`-m`

Отключить обратную коррекцию имен для C++. Delphi корректирует имена для совместимости с C++.

`-o`

Интерпретировать *inputfile* как объектный файл в формате OMF. Файл должен иметь расширение *.obj*. Delphi может генерировать *obj*-файлы в формате OMF, как и компиляторы C и C++ фирмы Borland.

`-oi`*id*

Выводить запись OMF, обозначенную *id*. Для печати списка имен записей OMF на стандартном устройстве вывода (не в *listfile*) укажите вопросительный знак (?) вместо *id*.

`-ox`*id*

Исключить запись OMF, обозначенную *id*. Для печати списка имен записей OMF на стандартном устройстве вывода (не в *listfile*) укажите вопросительный знак (?) вместо *id*.

`-q`

Не печатать сообщение об авторских правах, которое *tdump* обычно помещает в начале выходных данных.

`-r`

Печатать шестнадцатеричный дамп содержимого записей OMF.

`-R`

Выводить таблицу перемещений PE *exe*- или *dll*-файла.

`-s[`*number* | b*offset* | e*offset* | f | s | u `] [=`*str*`]`

Поиск читаемых строк. Если указана строка поиска *str*, печатаются только строки, содержащие ее. В противном случае печатаются все строки, содержащие не менее *number* печатных символов. Можно совмещать несколько ключей поиска, например `-s8ufb100f1000=unit`.

`-snumber[=str]`

Поиск строк, содержащих не менее *number* печатных символов. По умолчанию 4 символа.

`-sboffset[=str]`

Начинать поиск с байтового смещения *offset*.

`-seoffset[=str]`

Закончить поиск на байтовом смещении *offset*.

`-sf[=str]`

Форматировать длинные строки путем переноса.

`-ss[=str]`

Выполнять регистро-зависимый поиск.

`-su[=str]`

Печатать результаты поиска без ведущего байта смещения (в стиле Unix).

`-s[=str]`

Выполнять регистро-независимый поиск.

`-um`

Читать *inputfile* и искать все скорректированные имена символов. Выполнить обратную коррекцию и вывести имена по одному в строке в выходной файл.

`-v`

Печатать более наглядную информацию для ресурсов, файлов в формате COFF, файлов в формате PE и строковых дампов.

`-xid`

Исключить таблицы 16-разрядного отладчика Turbo Debugger, обозначенные *id*. Для вывода на стандартное устройство вывода (не в *listfile*) списка обозначений отладочных таблиц указывайте вопросительный знак (?). Каждая таблица идентифицируется одной буквой (например, A – таблица символов, B – таблица модулей, C – таблица файлов исходных текстов и т. д.). Перечислив несколько букв, можно исключить несколько таблиц.

`-?`

Вывод справки по переключателям командной строки.

delphi32.exe, IDE

Формат вызова

```
delphi32 [-attach:process;event] [-b] [-d file arguments] [-hhostname] [-hm]
[-hv] [-m] [-np] [-ns] [-opath] [-sdpaths] [-td] [file]
```

Описание

IDE не является утилитой командной строки, однако при указании переключателей `-b` и `-m` (описанных в этом разделе) она может рассматриваться как компилятор командной строки. Она также понимает переключатели, позволяющие запустить сеанс отладки или вывести отладочную информацию самой среды Delphi.

Ключи не чувствительны к регистру. Каждый ключ может начинаться с дефиса (`-`) или с косой черты (`/`). В отличие от компилятора командной строки, нельзя объединять ключи, даже при использовании косой черты.

В остальной части раздела описываются ключи командной строки.

`-attach:`*`process`*`;`*`event`*

Запустить Delphi для отладки работающего процесса, идентификатор которого равен *process*. Для присоединения к процессу нужно указать идентификатор события *event*. В Delphi этот ключ применяется для запуска оперативного отладчика. Идентификаторы для параметра *event* нельзя генерировать вручную, поэтому данный ключ не указывается в других ситуациях.

`-b`

Выполнить полную сборку проекта, указанного в командной строке, и завершить работу. Ошибки, предупреждения и подсказки записываются в файл с именем, совпадающим с именем проекта, и расширением *.err*. Файл ошибок удаляется, если проект скомпилирован без ошибок.

`-d`*`file arguments...`*

`-d` *`file arguments...`*

Загрузить файл и запустить отладчик. Все аргументы командной строки, следующие за именем файла, передаются запускаемой программе.

`-h`*`hostname`*

Запустить сеанс удаленной отладки на компьютере *hostname*.

`-hm`

Выводить статистику «кучи» в заголовке главного окна Delphi. Статистика «кучи» показывает, сколько памяти занимает IDE. Информация об этих значениях приведена в описаниях функций `AllocMemCount` и `AllocMemSize`. Числа обновляются во время простоя IDE.

`-hv`

Проверить «кучу» на корректность и выводить ее статус в заголовке главного окна Delphi. Корректность «кучи» проверяется во время простоя Delphi.

`-m`

Выполнить сборку проекта, указанного в командной строке, и завершить работу. Ошибки, предупреждения и подсказки записываются в файл с именем, совпадающим с именем проекта, и расширением *.err*. Файл ошибок удаляется, если проект скомпилирован без ошибок.

`-np`

При запуске не загружать проект и настройки рабочего стола.

`-ns`

Не выводить заставку при запуске.

`-o`*`path`*

При компиляции с переключателями `-b` и `-m` записать сообщения об ошибках, предупреждениях и подсказках в файл, путь к которому отличен от стандартного (имя файла проекта и расширение *.err*).

`-sd`*`paths`*

Искать исходные тексты для отладки в каталогах, перечисленных в *paths*. Каталоги разделяются точкой с запятой.

`-td`

Включить некоторые возможности Turbo Debugger в отладчике IDE:

- Окна CPU и FPU остаются открытыми после завершения процесса.
- Команда Run → Program Reset сбрасывает счетчик отлаживаемой программы и заново загружает ее в отладчик. Если процесс не запущен, отладчик загружает последнюю отлаживаемую программу.
- Если проект не открыт, точки останова и контрольные переменные сохраняются в настройках рабочего стола.

Приложение B

Модуль SysUtils

В модуле SysUtils содержится много полезных классов, типов, переменных и подпрограмм. Они настолько важны для программирования в Delphi, что многие пользователи забывают, что модуль SysUtils не является встроенным и обязательным. Поскольку этот модуль не является частью языка Delphi Pascal, он не рассматривается в основном тексте книги. С другой стороны, модуль SysUtils играет настолько важную роль в большинстве Delphi-программ, пакетов и библиотек, что не описать его было бы ошибкой – поэтому данное приложение и включено в эту книгу.

Ошибки и исключительные ситуации

В этом разделе перечислены классы и подпрограммы, которые помогают в обработке ошибок и исключительных ситуаций. Самая важная роль модуля SysUtils состоит в том, что в нем присваиваются значения указателям на процедуры обработки ошибок, объявленным в модуле System, а именно:

- AbstractErrorProc вызывает исключительную ситуацию EAbstractError.
- AssertErrorProc вызывает исключительную ситуацию EAssertionFailed.
- ErrorProc преобразует ошибки времени выполнения в исключительные ситуации.
- ExceptClsProc преобразует исключительные ситуации Windows и аппаратные исключительные ситуации в классы исключительных ситуаций Delphi.

- Переменная ExceptionClass устанавливается равной Exception.
- ExceptObjProc преобразует исключительные ситуации Windows и аппаратные исключительные ситуации в объекты исключительных ситуаций Delphi.
- ExceptProc выводит информацию об исключительной ситуации и вызывает Halt.

Если ошибка или исключительная ситуация возникла до инициализации модуля SysUtils или после его завершения, программист не будет иметь преимуществ, предоставляемых этими процедурами обработки ошибок, и ему придется рассчитывать только на упрощенный подход модуля System. Это чаще всего случается в процессе завершения программы: приложение разрушает память при освобождении уже освобожденной памяти, переполнении буфера и т. д., однако проблема не была распознана сразу. После завершения модуля SysUtils выявляется разрушение структуры «кучи», и модуль System выдает ошибку времени выполнения 217. Иногда модуль генерирует исключительную ситуацию, в этом случае возникает ошибка времени выполнения 216. Можно исправить эту ситуацию путем перемещения модуля SysUtils в начало объявления uses проекта. Попробуйте переместить его в начало основного исходного файла (но после ShareMem). Это заставит другие модули завершиться до отмены обработчиков событий SysUtils.

Классы исключительных ситуаций

Другой важной задачей, которую выполняет модуль SysUtils при обработке ошибок, является объявление иерархии классов исключительных ситуаций. Класс Exception является корневым классом в иерархии классов исключительных ситуаций. Модуль SysUtils устанавливает переменную ExceptionClass равной классу Exception, поэтому IDE останавливается при возникновении любой исключительной ситуации, которая является наследником Exception.

Все объекты исключительных ситуаций имеют связанное с ними строковое сообщение. При возникновении исключительной ситуации сообщение задается в конструкторе класса исключительной ситуации. В классе Exception объявлено несколько конструкторов, как показано в примере В-1, где приведены открытые методы и свойства этого класса. Перегруженные конструкторы позволяют указать статическую строку сообщения, загрузить строковый ресурс по указанному идентификатору или загрузить resourcestring по ее адресу (PResStringRec). Можно использовать простую строку или строку формата с соответствующими аргументами. Допускается также указание номера контекста помощи (который по умолчанию равен нулю). Вместо форм конструктора с указанием ресурса лучше использовать resourcestring и вызывать Create, CreateFmt, CreateHelp или CreateFmtHelp. Формы конст-

руктора с ресурсами оставлены для совместимости с ранними версиями Delphi.

Пример B-1. Открытый (public) интерфейс классов Exception

```
type
  Exception = class(TObject)
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg:string; const Args:array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec: PResStringRec); overload;
    constructor CreateResFmt(Ident: Integer;
        const Args: array of const); overload;
    constructor CreateResFmt(ResStringRec: PResStringRec;
        const Args: array of const); overload;
    constructor CreateHelp(const Msg: string; AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string;
        const Args: array of const; AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer; AHelpContext: Integer);
        overload;
    constructor CreateResHelp(ResStringRec: PResStringRec;
        AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(ResStringRec: PResStringRec;
        const Args: array of const; AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(Ident: Integer;
        const Args: array of const; AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext write FHelpContext;
    property Message: string read FMessage write FMessage;
  end;
```

Все классы исключительных ситуаций наследуются от `Exception`. В табл. B-1 приведена иерархия классов. При создании приложения следует произвести собственные классы исключительных ситуаций от стандартных. Это позволяет легко настроить и расширить исключительные ситуации в будущих версиях приложения.

Таблица B-1. Классы исключительных ситуаций

Классы исключительных ситуаций	Описание
`Exception`	Базовый класс
`Eabort`	«Тихая» исключительная ситуация, вызываемая процедурой `Abort`
`EabstractError`	Вызов абстрактного метода
`Eexternal`	Аппаратная исключительная ситуация или исключительная ситуация Windows
`EAccessViolation`	Нарушение доступа

Классы исключительных ситуаций	Описание
EcontrolC	Пользовательское прерывание
EExternalException	Другая внешняя исключительная ситуация[a]
EintError	Целая математическая ошибка
EDivByZero	Целое деление на ноль
EIntOverflow	Переполнение целого или перечисления
ERangeError	Ошибка границ массива
EmathError	Вещественная математическая ошибка
EInvalidOp	Неверный вещественный операнд или операция
EOverflow	Вещественное переполнение
EUnderflow	Потеря значимости в вещественном числе
EZeroDivide	Вещественное деление на ноль
Eprivilege	Ошибка при попытке выполнения привилегированной инструкции
EStackOverflow	Переполнение стека
EAssertionFailed	Ошибка при вызове процедуры Assert
EconvertError	Ошибка преобразования данных
EheapException	Ошибка динамической памяти
EInvalidPointer	Неверный указатель в менеджере памяти
EOutOfMemory	Нехватка памяти для менеджера памяти
EinOutError	Ошибка ввода/вывода
EintfCastError	Ошибка преобразования интерфейса (оператор as)
EinvalidCast	Ошибка преобразования типа объекта (оператор as)
EInvalidContainer	(Не используется в VCL)
EinvalidInsert	(Не используется в VCL)
EpackageError	Невозможно загрузить или выгрузить пакет
EpropReadOnly	(Не используется в VCL)
EpropWriteOnly	(Не используется в VCL)
ESafeCallException	В методе safecall произошел сбой
EvariantError	Ошибка с типами данных Variant или OleVariant
EWin32Error	Код ошибки Windows

a. Delphi распознает целый ряд кодов внешних исключительных ситуаций, которым соответствуют специальные классы, приведенные в этой таблице. `EExternalException` генерируется в тех случаях, когда Delphi сталкивается с неопознанным кодом внешней ошибки. – *Примеч. науч. ред.*

Большинство исключительных ситуаций являются тривиальными производными базового класса. Объявление `EOverflow`, в частности, приведено в примере B-2.

Пример B-2. Тривиальный класс исключительной ситуации EOverflow

```
type
  EOverflow = class(EMathError);
```

В некоторых классах объявляются дополнительные методы или поля:

EExternal, класс

```
type
  EExternal = class(Exception)
  public
    ExceptionRecord: PExceptionRecord;
  end;
```

Содержит указатель на запись об исключительной ситуации Windows. Подробности о формате записи исключительной ситуации рассмотрены в документации по Windows Platform SDK.

EHeapException, класс

```
type
  EHeapException = class(Exception)
  public
    procedure FreeInstance; override;
  end;
```

Переопределяет `FreeInstance` для управления фактическим освобождением объекта исключительной ситуации. В модуле `SysUtils` создается единичный объект исключительной ситуации каждого класса-наследника `EHeapException`. Таким образом, после переполнения памяти Windows или разрушения структуры «кучи» новый объект не создается.

EInOutError, класс

```
type
  EInOutError = class(Exception)
  public
    ErrorCode: Integer;
  end;
```

Сохраняет код ошибки ввода/вывода в поле `ErrorCode`. Код ошибки может быть равен одному из встроенных в Delphi значений или коду ошибки Windows. Коды ошибок Delphi описаны в конце главы 6

«Системные константы». Для того чтобы получить текст ошибки ввода/вывода Windows, вызовите функцию SysErrorMessage, описанную далее в этом разделе.

EWin32Error, класс

```
type
  EWin32Error = class(Exception)
  public
    ErrorCode: DWORD;
  end;
```

Сохраняет код ошибки Windows в ErrorCode и устанавливает сообщение об исключительной ситуации равным тексту сообщения об ошибке Windows.

Поддержка обработки ошибок

В модуле SysUtils также объявлены некоторые типы и подпрограммы для обеспечения дополнительной поддержки обработки ошибок и исключительных ситуаций:

Abort, процедура

```
procedure Abort;
```

Генерирует исключительную ситуацию EAbort. Графическое приложение игнорирует ее, поэтому процедуру Abort можно применять для вызова исключительной ситуации, не видимой пользователю.

AddExitProc, процедура

```
procedure AddExitProc(Proc: TProcedure);
```

Вместо процедур выхода следует использовать секции завершения, однако в старых программах еще могут встречаться процедуры выхода. В модуле System имеется одна переменная, ExitProc, для хранения указателя на процедуру выхода. Модуль SysUtils идет дальше и поддерживает цепочку процедур выхода. Вместо присваивания значения ExitProc используйте AddExitProc. SysUtils устанавливает ExitProc на собственный обработчик, который вызывает все процедуры выхода в порядке, обратном порядку их добавления.

AddTerminateProc, процедура

```
type TTerminateProc = function: Boolean;
procedure AddTerminateProc(TermProc: TTerminateProc);
```

Добавляет процедуру в список процедур завершения программы. Графическое приложение вызывает процедуры завершения, когда программа должна завершиться. Если какая-то из процедур возвращает False, приложение не завершается.

Beep, процедура

```
procedure Beep;
```

Вызывает функцию `MessageBeep(0)` из Windows API, которая воспроизводит системный звук по умолчанию.

CallTerminateProcs, функция

```
function CallTerminateProcs: Boolean;
```

Графическое приложение вызывает функцию `CallTerminateProcs`, когда получает сообщение `WM_QUERYENDSESSION`. `CallTerminateProcs` вызывает каждую из процедур завершения и, если какая-то из функций возвращает `False`, `CallTerminateProcs` сразу возвращает `False` и приложение не завершается. Если все функции завершения вернули `True`, `CallTerminateProcs` также возвращает `True`.

ExceptAddr, функция

```
function ExceptAddr: Pointer;
```

Когда возникает исключительная ситуация, в переменную `ExceptAddr` записывается адрес кода. Ее значение может использоваться для повторного вызова исключительной ситуации, для протоколирования или выдачи на экран сообщения об ошибке.

ExceptionErrorMessage, функция

```
function ExceptionErrorMessage(ExceptObject: TObject;
    ExceptAddr: Pointer; Buffer: PChar; Size: Integer): Integer;
```

Функция `ExceptionErrorMessage` предназначена для форматирования строки сообщения об исключительной ситуации. В первых двух аргументах функции передаются объект исключительной ситуации и адрес ее возникновения. Строковый буфер и его длина передаются в следующих двух аргументах. Функция возвращает длину форматированной строки. Сообщение содержит имя исполняемого модуля, адрес возникновения исключительной ситуации, имя класса и сообщение (если класс исключительной ситуации является наследником класса `Exception`).

ExceptObject, функция

```
function ExceptObject: TObject;
```

Функция `ExceptObject` возвращает ссылку на объект исключительной ситуации. Вызывайте эту функцию только при обработке исключительной ситуации, т. к. Delphi уничтожает объект после завершения обработчика.

OutOfMemoryError, процедура

```
procedure OutOfMemoryError;
```

Функция OutOfMemoryError вызывается для формирования исключительной ситуации EOutOfMemory. SysUtils заранее создает одиночный объект EOutOfMemory, который может многократно использоваться функцией OutOfMemoryError. Это позволяет избежать попытки создания нового объекта исключительной ситуации при нехватке памяти. При создании собственного менеджера памяти не следует применять OutOfMemoryError для выдачи сообщения об ошибке. Вместо этого проверьте, чтобы функции выделения и повторного выделения памяти возвращали nil, а Delphi позаботится обо всем остальном.

RaiseLastWin32Error, процедура

```
procedure RaiseLastWin32Error;
```

Процедура RaiseLastWin32Error генерирует исключительную ситуацию EWin32Error, используя самый последний код ошибки Windows (полученный при вызове функции GetLastError) в качестве кода ошибки для исключительной ситуации. Вызывайте RaiseLastWin32-Error в тех случаях, когда происходит сбой при вызове функции Windows API и требуется сообщить об этом как об исключительной ситуации. Смотрите также описание Win32Check далее в этом разделе.

ShowException, процедура

```
procedure ShowException(ExceptObject: TObject; ExceptAddr: Pointer);
```

Функция ShowException выводит сообщение об исключительной ситуации. В консольном приложении сообщение об исключительной ситуации записывается в файл Output. В графическом приложении сообщение выводится в диалоговом окне. Текст сообщения получается с помощью функции ExceptionErrorMessage.

SysErrorMessage, функция

```
function SysErrorMessage(ErrorCode: Integer): string;
```

SysErrorMessage возвращает текст по коду ошибки Windows. Приложение всегда должно сообщать об ошибке Windows текстовым сообщением, а не номером ошибки. Класс EWin32Error автоматически включает текст ошибки в сообщение об исключительной ситуации, однако EInOutError не форматирует большинство кодов ошибок Windows. Кроме того, обязательно вызовите SysErrorMessage, реализуя собственную обработку ошибки Windows.

Win32Check, функция

```
function Win32Check(RetVal: LongBool): LongBool;
```

Каждый раз, когда вызывается функция Windows API, следует проверять возвращаемый ею статус, чтобы убедиться, что функция завершилась успешно. Сбой в функции API может произойти по

многим неочевидным причинам. Простейший способ проверки – это вызов функции Win32Check, принимающей логический аргумент и вызывающей RaiseLastWin32Error, если он равен False.

Работа с файлами

В модуле SysUtils объявлено несколько полезных типов, констант и подпрограмм для работы с файлами и каталогами, а также разбора имен файлов. Эти средства описаны ниже.

Файловый ввод и вывод

Стандартные процедуры ввода/вывода языка Паскаль для выполнения фактических операций обращаются к Windows API. Допускаются также прямые вызовы Windows API, однако многим, возможно, будет проще воспользоваться функциями файлового ввода/вывода из модуля SysUtils. Эти функции – тонкая оболочка для функций Windows API, предоставляющая возможность, например, работать со строками Delphi вместо строк PChar.

FileClose, процедура

```
procedure FileClose(Handle: Integer);
```

Процедура FileClose закрывает открытый файл. Если файл не открыт, ошибка игнорируется.

FileCreate, функция

```
function FileCreate(const FileName: string): Integer;
```

Функция FileCreate создает новый файл и открывает его для чтения и записи без учета разделяемых ресурсов. Если файл уже существует, он создается заново. Результатом является дескриптор файла или –1 в случае ошибки.

FileGetDate, функция

```
function FileGetDate(Handle: Integer): Integer;
```

FileGetDate возвращает дату изменения открытого файла. Дата изменения обновляется при закрытии файла, поэтому, возможно, следует сначала закрыть файл, а затем вызвать FileAge для получения наиболее точного значения даты изменения. Функция FileGetDate возвращает –1 в случае ошибки.

Дата и время файла упаковываются в битовые поля 32-битового целого. Точность соответствует двум секундам, год может принимать значения от 1980 до 2099. На рис. В-1 показан формат даты и времени файла. Преобразование даты файла в TDateTime рассмотрено в описании функции FileDateToDateTime.

Рис. В-1. Формат даты и времени файла

FileOpen, функция

```
function FileOpen(const FileName: string; Mode: LongWord): Integer;
```

Функция `FileOpen` открывает существующий файл. В `Mode` объединяются флаги доступа и совместного использования файла, как показано в табл.В-2. Выберите один режим доступа и один режим совместного использования и объедините их операцией сложения или включающего «ИЛИ». Возвращаемое значение – дескриптор файла или –1 в случае ошибки.

Табл. В-2. Режимы доступа и совместного использования файлов

Литерал	Значение	Описание
Режимы доступа		
`fmOpenRead`	$0000	Доступ только для чтения
`fmOpenWrite`	$0001	Доступ только для записи
`fmOpenReadWrite`	$0002	Доступ для чтения и для записи
Режимы совместного использования		
`fmShareCompat`	$0000	Монопольный доступ
`fmShareExclusive`	$0010	Монопольный доступ
`fmShareDenyWrite`	$0020	Разрешается совместное чтения
`fmShareDenyRead`	$0030	Разрешается совместная запись
`fmShareDenyNone`	$0040	Разрешается совместное чтение и совместная запись

FileRead, функция

```
function FileRead(Handle: Integer; var Buffer; Count: LongWord): integer;
```

Функция `FileRead` читает до `Count` байт в `Buffer`. Заметьте, что `Buffer` является фактическим буфером, а не указателем. Возвращается реально прочитанное количество байт или –1 в случае ошибки.

FileSeek, функция

```
function FileSeek(Handle, Offset, Origin: Integer): Integer; overload;
function FileSeek(Handle: Integer; const Offset: Int64;
```

```
    Origin: Integer): Int64; overload;
```

Функция FileSeek устанавливает позицию для чтения и записи (в байтах) для открытого файла и возвращает новую позицию в файле (байтовое смещение от начала файла). В табл. В-3 перечислены возможные значения параметра Origin. Чтобы определить текущую позицию в файле, передайте Origin значение File_Current и Offset, равное нулю. Перегруженная версия функции позволяет менять позицию в файлах, больших 2 Гбайт.

Табл. В-3. Значения параметра Origin для функции FileSeek

Литерал	Значение	Описание
File_Begin	0	Смещение (offset) рассчитывается относительно начала файла
File_Current	1	Смещение рассчитывается относительно текущей позиции в файле
File_End	2	Смещение рассчитывается относительно конца файла

Если операция изменения позиции в файле не была выполнена успешно, 32-разрядная версия функции возвращает −1, а 64-разрядная версия возвращает −1 в младших 32 битах результата. Поскольку это значение может соответствовать корректному значению смещения, используйте функцию GetLastError для получения кода ошибки, который равен нулю в случае успешного изменения позиции в файле.

FileSetDate, функция

```
function FileSetDate(Handle: Integer; Age: Integer): Integer;
```

FileSetDate устанавливает время изменения открытого файла. Время доступа и создания файла в Windows NT не изменяются. В случае успеха возвращается ноль, в случае неудачи – код ошибки.

FileWrite, функция

```
function FileWrite(Handle: Integer; const Buffer; Count: LongWord):
    Integer;
```

Процедура FileWrite записывает в файл Count байт из Buffer. Процедура возвращает количество байт или −1 в случае ошибки. Заметьте, что Buffer фактически содержит записываемые данные, а не является указателем.

TFileRec, тип

```
const fmClosed = $D7B0;
const fmInput  = $D7B1;
const fmOutput = $D7B2;
const fmInOut  = $D7B3;
```

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: Integer;
    RecSize: Cardinal;
    Private: array[1..28] of Byte;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
  end;
```

Стандартный тип `file` **языка Паскаль фактически является записью** `TFileRec`**. Преобразуйте файловую переменную к типу** `TFileRec` **для получения доступа к дескриптору файла и другим полям. Например, дату изменения файла можно определить следующим образом:**

```
function GetFileDate(var F: file): TDateTime;
var
  Date: Integer;
begin
  Date := FileGetDate(TFileRec(F).Handle);
  if Date = -1 then
    raise Exception.Create('Невозможно определить дату изменения файла');
  Result := FileDateToDateTime(Date);
end;
```

TTextRec, тип

```
type
  PTextBuf = ^TTextBuf;
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;
    Mode: Integer;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
    Buffer: TTextBuf;
  end;
```

Стандартный тип `Text` **или** `TextFile` **языка Паскаль фактически является записью** `TTextRec`**. Можно преобразовать** `TextFile` **к типу** `TTextRec` **для получения доступа к дескриптору файла и другим полям записи. Кроме того, можно определить собственный драйвер текс-**

тового файла, заполнив поля записи TTextRec. Подробности приведены в описании типа TextFile в главе 5 «Справочник по языку».

Работа с файлами

В этом разделе описаны другие функции для работы с файлами:

DeleteFile, функция

```
function DeleteFile(const FileName: string): Boolean;
```

Функция DeleteFile удаляет файл, возвращая True в случае успеха и False в случае неудачи. Для получения кода ошибки используется функция GetLastError. Заметьте, что в модуле Windows также имеется функция DeleteFile, но ей требуется аргумент типа PChar. Получение ошибки несоответствия типов аргументов связано, скорее всего, с тем, что модуль Windows указан в объявлении uses позднее, чем модуль SysUtils. Расположите SysUtils после модуля Windows или явно укажите имя модуля – SysUtils.DeleteFile.

FileAge, функция

```
function FileAge(const FileName: string): Integer;
```

Функция FileAge возвращает дату модификации файла. Возвращается дата файла или –1 в случае ошибки. Преобразование даты файла в TDateTime рассмотрено в описании функции FileDateToDateTime, а на рис. B-1 (ранее в этом приложении) приведен формат даты и времени файла.

FileExists, функция

```
function FileExists(const FileName: string): Boolean;
```

Функция FileExists возвращает True, если файл существует, и False, если файл не существует или пользователь не имеет достаточно прав для определения существования файла.

FileGetAttr, функция

```
function FileGetAttr(const FileName: string): Integer;
```

Функция FileGetAttr возвращает атрибуты файла или –1 в случае ошибки. Атрибуты файла возвращаются в виде битовой маски. В табл. B-4 описаны атрибуты файла, которые могут быть возвращены функцией. В модуле SysUtils объявляются константы для некоторых из этих атрибутов, однако для других придется использовать литералы, объявленные в модуле Windows. Дополнительную информацию об атрибутах файлов Windows можно найти в документации Microsoft Platform SDK, в частности, в описании функции GetFileAttributes. (Некоторые из атрибутов определены только в Windows 2000.)

Табл. В-4. Атрибуты файлов

Литерал модуля Windows	Литерал SysUtils	Описание
File_Attribute_Archive	FaArchive	Файл помечен для архивирования
File_Attribute_Compressed	Нет	Содержимое файла сжато
File_Attribute_Directory	FaDirectory	Файл является каталогом
File_Attribute_Encrypted	Нет	Содержимое файла зашифровано
File_Attribute_Hidden	FaHidden	Файл является скрытым и не отображается в списке файлов каталога
File_Attribute_Normal	Нет	Атрибуты не установлены
File_Attribute_Offline	Нет	Файл в данный момент не доступен
File_Attribute_Readonly	FaReadonly	Файл только для чтения
File_Attribute_Reparse_Point	Нет	Файл является точкой подсоединения
File_Attribute_Sparse_File	Нет	Разреженный файл
File_Attribute_System	FaSysFile	Системный файл
File_Attribute_Temporary	Нет	Временный файл

FileSearch, функция

```
function FileSearch(const Name, DirList: string): string;
```

Функция FileSearch выполняет поиск файла Name в каталогах, перечисленных в DirList. Поиск всегда начинается с текущего каталога и затем продолжается в каталогах, перечисленных в DirList. Имена каталогов разделяются точками с запятой. Name может быть простым именем файла или относительным путем. FileSearch возвращает полный путь или пустую строку, если файл не найден.

FileSetAttr, функция

```
function FileSetAttr(const FileName: string; Attr: Integer): Integer;
```

Функция FileSetAttr изменяет атрибуты файла и возвращает ноль в случае успеха или код ошибки в случае неудачи. В табл. В-4 перечислены атрибуты файлов, поддерживаемые Windows.

FileClose, процедура

```
procedure FindClose(var F: TSearchRec);
```

Процедура FindClose завершает поиск по каталогам, начатый функцией FindFirst. Если функция FindFirst вернула код успешного за-

вершения, программа должна вызвать FindClose после завершения поиска.

FindFirst, функция

```
type
  TFileName = type string;
  TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
  end;
function FindFirst(const Path: string; Attr: Integer;
    var Rec: TSearchRec): Integer;
```

Функция FindFirst начинает поиск файлов в каталоге. Выполняется поиск файлов, соответствующих спецификатору Path, который может содержать групповые символы (* и ?). Если Attr равен нулю, выполняется поиск всех обыкновенных файлов. Также можно задать битовую маску из атрибутов faHidden, faSysFile и faDirectory для поиска специальных файлов и каталогов. Для поиска всех каталогов и файлов используйте faAnyFile. Функция FindFirst возвращает ноль в случае успеха и код ошибки в случае неудачи. Если возвращается ноль, необходимо позднее вызвать FindClose для освобождения ресурсов, выделенных при вызове FindFirst.

Если FindFirst возвращает ноль, Rec заполняется информацией о первом найденном файле. Ниже описываются поля, предназначенные для программиста. Остальные поля предназначены для подпрограмм FindFirst, FindNext и FindClose.

Attr

Атрибуты файла. Список атрибутов файлов Windows приведен в табл. В-4.

FindData

Более подробная информация содержится в записи Windows, задокументированной как WIN32_FIND_DATA в Platform SDK. В частности, можно получить короткое (8.3) имя файла, размер файла более 2 Гбайт или определить время и дату последнего доступа и создания файла.

Name

Длинное имя файла. Короткое имя файла рассмотрено в поле записи FindData.

Size

Размер файла в байтах. Если размер файла превышает 2 Гбайт, Size содержит младшие 32 бита истинного размера файла; полный размер файла рассмотрен в поле записи FindData.

Time

Дата и время изменения файла в формате даты файла. Преобразование этого значения в TDateTime разобрано в описании функции FileDateToDateTime. На рис. B-1 (ранее в этой главе) приведен формат даты и времени файла.

FindNext, функция

```
function FindNext(var F: TSearchRec): Integer;
```

После того как функция FindFirst вернула нулевое значение, с помощью FindNext можно получить информацию о следующем файле, соответствующем критериям поиска. В качестве единственного аргумента FindNext передается та же запись TSearchRec, что была использована для FindFirst. Возвращаемое значение равно нулю в случае успеха или коду ошибки в случае возникновения ошибки. Если FindNext больше не нашла файлов, код ошибки равен Error_No_More_Files. Обязательно вызовите FindClose после завершения работы функций FindFirst и FindNext.

RenameFile, функция

```
function RenameFile(const OldName, NewName: string): Boolean;
```

Функция RenameFile изменяет имя файла или каталога и возвращает True в случае успеха и False в случае неудачи. Для получения кода ошибки вызовите GetLastError. Можно переименовать файл на другой каталог или диск. Каталог переименовывается только в пределах диска.

Работа с каталогами

В этом разделе перечислены функции для работы с каталогами. Если какая-то из этих функций возвращает False, вызовите GetLastError для получения кода ошибки.

CreateDir, функция

```
function CreateDir(const Dir: string): Boolean;
```

Функция CreateDir создает каталог и возвращает True в случае успеха и False в случае неудачи. Используйте CreateDir вместо MkDir, если результат логического типа представляется более удобным, чем коды ошибок ввода/вывода.

GetCurrentDir, функция

```
function GetCurrentDir: string;
```

Функция GetCurrentDir возвращает текущий рабочий диск и каталог. Возвращаемое значение совпадает со строкой, получаемой при вызове функции GetDir с первым аргументом, равным нулю, однако GetCurrentDir удобнее в использовании.

RemoveDir, функция

```
function RemoveDir(const Dir: string): Boolean;
```

Функция RemoveDir удаляет пустой каталог и возвращает True в случае успеха и False в случае неудачи. Используйте RemoveDir вместо RmDir, если не хотите работать с кодами ошибок ввода/вывода.

SetCurrentDir, функция

```
function SetCurrentDir(const Dir: string): Boolean;
```

Функция SetCurrentDir устанавливает текущий рабочий диск и каталог. Функция возвращает True в случае успеха и False в случае неудачи. Используйте SetCurrentDir вместо ChDir, если не хотите работать с кодами ошибок ввода/вывода.

Работа с дисками

DiskFree, функция

```
function DiskFree(Drive: Byte): Int64;
```

Функция DiskFree возвращает количество свободных байт на диске. Параметр Drive равен 1 для диска A:, 2 для диска B: и т. д. Если значение Drive равно нулю, DiskFree извлекает информацию для диска по умолчанию. DiskFree корректно работает с дисками размером более 2 Гбайт, за исключением Windows 95a, которая не поддерживает большие дисковые разделы. DiskFree возвращает −1, если значение Drive некорректно.

DiskSize, функция

```
function DiskSize(Drive: Byte): Int64;
```

Функция DiskSize возвращает размер диска в байтах. Параметр Drive равен 1 для диска A:, 2 – для диска B: и т. д. Если значение Drive равно нулю, DiskSize извлекает информацию для диска по умолчанию. DiskSize корректно работает с дисками размером более 2 Гбайт, за исключением Windows 95a, которая не поддерживает большие дисковые разделы. DiskSize возвращает −1, если значение Drive некорректно.

Работа с именами файлов

В модуле SysUtils имеется несколько функций, облегчающих разбор и обработку имен файлов. Все эти функции корректно работают с многобайтовыми строками.

ChangeFileExt, функция

```
function ChangeFileExt(const FileName, Extension: string): string;
```

Функция ChangeFileExt меняет расширение файла, т. е. часть имени после последней точки (.). Если в имени файла нет расширения, Extension дописывается в конец имени файла. Возвращается новое имя файла. Extension может быть любой строкой, в том числе пустой. По соглашению непустые расширения начинаются с точки (.).

ExcludeTrailingBackSlash, функция

```
function ExcludeTrailingBackslash(const S: string): string;
```

Функция ExcludeTrailingBackslash возвращает копию S, но если последний символ S равен обратной косой черте (\), этот символ удаляется из результирующей строки. Чаще всего с помощью этой функции подготавливают имя каталога для использования с другими функциями, такими как SetCurrentDir.

ExpandFileName, функция

```
function ExpandFileName(const FileName: string): string;
```

Функция ExpandFileName раскрывает абсолютное имя, добавляя букву диска и каталог.

ExpandUNCFileName, функция

```
function ExpandUNCFileName(const FileName: string): string;
```

Функция ExpandUNCFileName возвращает полностью раскрытый, абсолютный путь файла. Буква диска раскрывается в путь UNC (универсальное соглашение об именах), например \\server\sharename.

ExtractFileDir, функция

```
function ExtractFileDir(const FileName: string): string;
```

Функция ExtractFileDir возвращает часть имени файла, содержащую путь, без завершающей обратной косой черты. Результат может быть использован в качестве аргумента любой из функций работы с каталогами (описанных ранее в этом приложении).

ExtractFileDrive, функция

```
function ExtractFileDrive(const FileName: string): string;
```

Функция ExtractFileDrive возвращает букву диска или хост UNC и имя разделяемого ресурса из имени файла. Если имя файла не содержит спецификатор диска, функция возвращает пустую строку.

ExtractFileExt, функция

```
function ExtractFileExt(const FileName: string): string;
```

Функция ExtractFileExt возвращает расширение имени файла, включая ведущую точку (.). Если имя файла не содержит расширение, функция возвращает пустую строку.

ExtractFileName, функция

```
function ExtractFileName(const FileName: string): string;
```

Функция ExtractFileName возвращает собственно имя файла, включая расширение, без указания буквы диска и каталога. Если FileName завершается двоеточием или обратной косой чертой, функция возвращает пустую строку.

ExtractFilePath, функция

```
function ExtractFilePath(const FileName: string): string;
```

ExtractFilePath возвращает путь из имени файла – все, что не возвращает функция ExtractFileName. Заметьте, что возвращаемый путь может завершаться символом (\), в отличие от функции ExtractFileDir, которая извлекает имя каталога без завершающей косой черты (что обычно бывает полезнее).

ExtractRelativePath, функция

```
function ExtractRelativePath(const BaseName, DestName: string): string;
```

Функция ExtractFilePath возвращает путь, идентифицирующий файл DestName относительно каталога BaseName.

ExtractShortPathName, функция

```
function ExtractShortPathName(const FileName: string): string;
```

Функция ExtractShortPathName преобразует FileName в короткое имя (8.3). Если FileName уже является коротким именем, оно возвращается без изменений. Если файл не существует, возвращается пустая строка.

Заметьте, что функции ExtractLongPathName не существует, т. к. большинство операционных систем семейства Windows не имеют функций API для получения длинного имени. Вместо этого можно использовать функцию GetLongPathName из примера B-3.

Пример B-3. Функция GetLongPathName

```
// Разделяет путь на часть, содержащую букву диска и каталог, и
// часть, содержащую имя файла. Если путь является именем UNC, имя
// разделяемого ресурса остается в первой части.
procedure ExtractFileParts(const Path: string;
                           var Directory, Name: string);
var
  I: Integer;
begin
  // Получаем букву диска или имя хоста UNC и имя
  // разделяемого ресурса. В SysUtils нет подходящей
```

```
  // для этого функции.
  I := LastDelimiter('\:', Path);
  Name := Copy(Path, I + 1, MaxInt);
  if (I > 1) and
     (Path[I] = '\') and
     (not (Path[I - 1] in ['\', ':']) or
         (ByteType(Path, I - 1) = mbTrailByte))
  then
    Dec(I);
  Directory := Copy(Path, 1, I);

  // Если Directory состоит только из хоста UNC, значит, мы
  // в Name извлекли имя разделяемого ресурса.
  if (Length(Directory) > 2) and
     (Directory[1] = '\') and
     (LastDelimiter('\', Directory) = 2) then
  begin
    Directory := Path;
    Name := '';
  end;
end;

function GetLongPathName(const PathName: string): string;
var
  Directory, FileName, FullName: string;
  LongName: string;
  Info: TShFileInfo;
begin
  FullName := ExcludeTrailingBackslash(PathName);
  repeat
    ExtractFileParts(FullName, Directory, FileName);
    if FileName = '' then
      // Если путь состоит только из буквы диска, дальнейшее
      // раскрытие не требуется.
      LongName := IncludeTrailingBackslash(Directory)
    else if ShGetFileInfo(PChar(FullName), 0, Info, SizeOf(Info),
                          Shgfi_DisplayName) = 0 then
    begin
      // Невозможно развернуть имя файла.
      Result := '';
      Exit;
    end
    else
      LongName := Info.szDisplayName;

    // Убедимся, что обратные косые черты включены в результат.
    if Result = '' then
      Result := LongName
    else
      Result := IncludeTrailingBackslash(LongName) + Result;
    FullName := Directory;
```

```
  until FileName = '';
end;
```

IncludeTrailingBackSlash, функция

```
function IncludeTrailingBackslash(const S: string): string;
```

Функция `IncludeTrailingBackslash` возвращает копию `S`, но гарантирует, что последним символом `S` является обратная косая черта (`\`). Если косая черта отсутствует, она добавляется. `IncludeTrailingBackslash` применяется при формировании пути из каталога и имен файлов.

Работа со строками

В этом разделе перечислены подпрограммы для работы со строками `AnsiString` и `PChar`. `AnsiString` поддерживает многобайтовые строки, т. е. строки, в которых одиночный символ может занимать более одного байта. Функции, перечисленные в последующих разделах, могут корректно обрабатывать многобайтовые строки. В отличие от строк `PChar`, в `AnsiString` могут содержаться любые символы, в том числе `#0`. Однако некоторые строковые функции предполагают, что в строке нет символов `#0` (кроме `#0`, присутствующего в конце строки). Такие функции отмечены в описаниях.

Образец работы со строками, которые могут быть многобайтовыми, приведен в примере B-4 в конце этого раздела.

Функции для работы со строками ANSI

Функции работы с строками ANSI используют аргументы и результаты типа `AnsiString`, но, что более важно, учитывают региональные установки Windows для работы с символами ANSI, такими как буквы со знаком ударения. Кроме того, эти функции могут работать с многобайтовыми строками.

AdjustLineBreaks, функция

```
function AdjustLineBreaks(const S: string): string;
```

Функция `AdjustLineBreaks` преобразует строку `S` в формат DOS, заменяя одиночные символы возврата каретки (`#13`) и перевода строки (`#10`) на пары CR-LF. Существующие символы CR-LF остаются неизменными. Файлы, копируемые из систем Unix и Macintosh, содержат иные символы окончания строк, чем могут «ввести в заблуждение» многие программы DOS и Windows. (В Macintosh используется только символ возврата каретки; в Unix – только символ перевода строки.)

AnsiCompareFileName, функция

```
function AnsiCompareFileName(const S1, S2: string): Integer;
```

Функция AnsiCompareFileName аналогична AnsiCompareText, однако решает проблему полноценных японских имен файлов (Zenkaku), где символы ASCII занимают два байта вместо одного. Для сравнения двух имен файлов всегда применяйте AnsiCompareFileName.

AnsiCompareStr, функция

```
function AnsiCompareStr(const S1, S2: string): Integer;
```

Функция AnsiCompareStr сравнивает две строки и возвращает целый результат в соответствии с табл. В-5.

Таблица В-5. Результат функции AnsiCompareStr

Результат	Описание
-1	S1 < S2
0	S1 = S2
+1	S1 > S2

Сравнение является чувствительным к регистру и принимает во внимание региональные настройки Windows и многобайтовые строки.

AnsiCompareText, функция

```
function AnsiCompareText(const S1, S2: string): Integer;
```

Функция AnsiCompareText выполняет сравнение двух строк и возвращает целый результат в том же виде, что и AnsiCompareStr. Сравнение *не* является чувствительным к регистру и принимает во внимание региональные настройки Windows и многобайтовые строки.

AnsiLastChar, функция

```
function AnsiLastChar(const S: string): PChar;
```

Функция AnsiLastChar возвращает указатель на последний символ строки S, который может не быть последним байтом многобайтовой строки. Если строка пустая, возвращается nil.

AnsiLowerCase, функция

```
function AnsiLowerCase(const S: string): string;
```

Функция AnsiLowerCase преобразует строку S в нижний регистр, принимая во внимание региональные настройки Windows для преобразования символов с ударением и многобайтовых символов.

AnsiLowerCaseFileName, функция

```
function AnsiLowerCaseFileName(const S: string): string;
```

Функция AnsiLowerCaseFileName преобразует имя файла S в нижний регистр, учитывая региональные настройки Windows для преобразования символов с ударением и многобайтовых символов. AnsiLo-

werCaseFilename решает проблему полноценных японских имен файлов (Zenkaku), где символы ASCII занимают два байта вместо одного.

AnsiPos, функция

```
function AnsiPos(const Substr, S: string): Integer;
```

Функция AnsiPos возвращает позицию первого вхождения SubStr в строке S или ноль, если S не содержит SubStr. Эта функция предполагает, что S не содержит символы #0 (кроме #0, находящегося за концом строки).

AnsiQuotedStr, функция

```
function AnsiQuotedStr(const S: string; Quote: Char): string;
```

Функция AnsiQuotedStr возвращает копию S, заключенную в кавычки. Символ кавычек указан в параметре Quote. Вхождения Quote в S повторяются и в результирующей строке. Эта функция предполагает, что в S нет символов #0 (кроме #0, находящегося за концом строки).

AnsiSameStr, функция

```
function AnsiSameStr(const S1, S2: string): Boolean;
```

Функция AnsiSameStr возвращает True, если строки S1 и S2 идентичны, и False – в противном случае. Сравнение выполняется с учетом регистра.

AnsiSameText, функция

```
function AnsiSameText(const S1, S2: string): Boolean;
```

Функция AnsiSameText возвращает True, если строки S1 и S2 идентичны, и False – в противном случае. Сравнение выполняется без учета регистра.

AnsiUpperCase, функция

```
function AnsiUpperCase(const S: string): string;
```

Функция AnsiUpperCase преобразует строку S в верхний регистр, учитывая региональные настройки Windows для преобразования символов с ударением и многобайтовых символов.

AnsiUpperCaseFileName, функция

```
function AnsiUpperCaseFileName(const S: string): string;
```

Функция AnsiUpperCaseFileName преобразует имя файла S в верхний регистр, учитывая региональные настройки Windows для преобразования символов с ударением и многобайтовых символов. AnsiUpperCaseFileName решает проблему полноценных японских имен файлов (Zenkaku), где символы ASCII занимают два байта вместо одного.

ByteToCharIndex, функция

```
function ByteToCharIndex(const S: string; Index: Integer): Integer;
```

Функция ByteToCharIndex возвращает индекс символа, соответствующий позиции байта Index в строке S. В многобайтовой строке индекс символа может быть меньше, чем индекс байта. Если индекс байта не является корректным, возвращается ноль.

ByteToCharLen, функция

```
function ByteToCharLen(const S: string; MaxLen: Integer): Integer;
```

Функция ByteToCharLen возвращает количество символов в строке S, рассматривая максимум MaxLen байт. В многобайтовой строке символьная длина может быть меньше байтовой.

ByteType, функция

```
type TMbcsByteType = (mbSingleByte, mbLeadByte, mbTrailByte);
function ByteType(const S: string; Index: Integer): TMbcsByteType;
```

Функция ByteType возвращает тип байта в позиции Index строки S. Возможные типы перечислены в табл. В-6.

Таблица В-6. Значения ByteType

Литерал	Описание
mbSingleByte	Одиночный однобайтовый символ
mbLeadByte	Первый байт многобайтового символа
mbTrailByte	Второй байт многобайтового символа

В отличие от других строковых функций, ByteType не выполняет каких-либо проверок на границы диапазонов своих аргументов. Если Index выходит за границы, функция ByteType не идентифицирует ошибку, и результат может быть непредсказуемым.

CharToByteIndex, функция

```
function CharToByteIndex(const S: string; Index: Integer): Integer;
```

Функция CharToByteIndex возвращает индекс байта для символа в логической позиции Index строки S. В многобайтовой строке индекс байта может быть больше, чем индекс символа. Если индекс символа имеет некорректное значение, возвращается ноль.

CharToByteLen, функция

```
function CharToByteLen(const S: string; MaxLen: Integer): Integer;
```

Функция CharToByteLen возвращает количество байт в строке S, рассматривая максимум MaxLen символов строки S. В многобайтовой строке символьная длина может быть меньше байтовой.

FindCmdLineSwitch, функция

```
function FindCmdLineSwitch(const Switch: string;
    SwitchChars: TSysCharSet; IgnoreCase: Boolean): Boolean;
```

FindCmdLineSwitch возвращает True, если командная строка содержит переключатель Switch. Параметр SwitchChars указывает, какие символы следует использовать для отметки начала переключателя (обычно «-/»). Если IgnoreCase равно False, имя переключателя должно точно совпадать со Switch; если IgnoreCase равно True, переключатель командной строки может быть в нижнем, в верхнем или в смешанном регистре.

IsDelimiter, функция

```
function IsDelimiter(const Delimiters, S: string; Index: Integer):
    Boolean;
```

Функция IsDelimiter возвращает True, если символ, начинающийся с позиции байта Index в строке S, является одним из символов Delimiters. False возвращается, если символ не содержится в Delimiters, если Index находится вне границ строки или если символ в позиции Index является первым или вторым байтом многобайтового символа. Delimiters не может быть многобайтовой строкой, и нельзя использовать #0 в качестве одного из разделителей.

IsPathDelimiter, функция

```
function IsPathDelimiter(const S: string; Index: Integer): Boolean;
```

Функция IsPathDelimiter возвращает True, если символ, начинающийся в позиции Index строки S, является обратной косой чертой (\) – разделителем каталогов в Windows.

IsValidIdent, функция

```
function IsValidIdent(const Ident: string): Boolean;
```

Функция IsValidIdent возвращает True, если Ident содержит корректный идентификатор Delphi Pascal, т. е. строку, первый символ которой является буквой или подчеркиванием, последующие символы – буквами, цифрами или символами подчеркивания.

LastDelimiter, функция

```
function LastDelimiter(const Delimiters, S: string): Integer;
```

Функция LastDelimiter возвращает индекс последнего (самого правого) вхождения любого из символов Delimiters в строке S. Если ни один из символов Delimiters не присутствует в S, LastDelimiter возвращает ноль. Delimiters не может быть многобайтовой строкой, и нельзя использовать #0 в качестве одного из разделителей.

StringReplace, функция

```
type TReplaceFlags = set of (rfReplaceAll, rfIgnoreCase);
```

```
function StringReplace(const S, OldSubStr, NewSubStr: string;
    Flags: TReplaceFlags): string;
```

StringReplace возвращает копию S, где OldSubStr заменена на NewSubStr. Если Flags содержит rfReplaceAll, заменяются все вхождения OldSubStr; иначе заменяется только первое вхождение. Поиск OldSubStr выполняется с учетом регистра, если только не включен флаг rfIgnoreCase.

WrapText, функция

```
function WrapText(const Line, BreakStr: string;
    BreakChars: TSysCharSet;  MaxCol: Integer): string; overload;
function WrapText(const Line: string; MaxCol: Integer = 45): string;
    overload;
```

Функция WrapText возвращает копию Line, разбитую на несколько строк шириной MaxCol столбцов. Каждая строка разбивается, когда ее длина доходит до MaxCol символов. Для разбиения строки после последовательности символов BreakChars в конец строки вставляется BreakStr. (В Delphi 5 как существующие переводы строк рассматриваются символы #13 и #10, независимо от BreakStr.)

Вторая форма WrapText использует [' ', '-', #9] (пробел, дефис, табуляция) в качестве BreakSet и #13#10 (возврат каретки, перевод строки) в качестве BreakStr.

Функции для работы с PChar

Delphi может автоматически преобразовывать строку PChar в AnsiString, однако иногда требуется избежать накладных расходов на преобразование. Для этого можно использовать многочисленные функции для работы со строками PChar в их исходном формате. Заметьте, что операции со строками PChar могут работать медленнее, чем операции с AnsiString, т. к. длина AnsiString известна заранее, а для PChar требуется сканировать строку, чтобы найти, где она завершается.

При работе со строками PChar постарайтесь не вызвать переполнения памяти, связанной с этой строкой. Эта задача ложится на плечи программиста, т. к. PChar – это лишь указатель, не настоящий массив, поэтому Delphi не знает размер памяти, выделенной строке. Когда это возможно, используйте L-версии функций, например AnsiStrLComp вместо AnsiStrComp.

Там, где это не указано отдельно, все функции, описанные в данном разделе, корректно работают с многобайтовыми строками.

AnsiExtractQuotedStr, функция

```
function AnsiExtractQuotedStr(var Src: PChar; Quote: Char): string;
```

Функция AnsiExtractQuotedStr выбирает из Src строку, заключенную в кавычки. Первый символ Src должен быть равен Quote, иначе

функция сразу завершается, возвращая пустую строку. Если это условие выполняется, строка, заключенная в символы кавычек, копируется и возвращается в качестве результата функции. Повторяющиеся символы кавычек преобразуются в одиночные кавычки. Src изменяется, указывая теперь на символ сразу за закрывающей кавычкой. Если в Src нет закрывающей кавычки, то Src присваивается указатель на завершающий байт #0, и функция возвращает весь текст до конца строки.

AnsiStrComp, функция

```
function AnsiStrComp(S1, S2: PChar): Integer;
```

Функция AnsiStrComp сравнивает строки S1 и S2 и возвращает целый результат аналогично AnsiCompareStr. Сравнение выполняется с учетом регистра и использует региональные настройки Windows.

AnsiStrIComp, функция

```
function AnsiStrIComp(S1, S2: PChar): Integer;
```

Функция AnsiStrIComp сравнивает строки S1 и S2 и возвращает целый результат аналогично AnsiCompareStr. Сравнение выполняется без учета регистра и использует региональные настройки Windows.

AnsiStrLastChar, функция

```
function AnsiStrLastChar(S: PChar): PChar;
```

Функция AnsiStrLastChar возвращает указатель на последний символ строки S перед завершающим нулевым байтом. В многобайтовой строке последний символ – это не всегда последний байт.

AnsiStrLComp, функция

```
function AnsiStrLComp(S1, S2: PChar; MaxLen: Cardinal): Integer;
```

Функция AnsiStrLComp сравнивает вплоть до MaxLen символов строк S1 и S2, возвращая целый результат аналогично AnsiCompareStr. Сравнение выполняется с учетом регистра и использует региональные настройки Windows.

AnsiStrLIComp, функция

```
function AnsiStrLIComp(S1, S2: PChar; MaxLen: Cardinal): Integer;
```

Функция AnsiStrLIComp выполняет сравнение до MaxLen символов строк S1 и S2, возвращая целый результат аналогично AnsiCompareStr. Сравнение выполняется без учета регистра и использует региональные настройки Windows.

AnsiStrLower, функция

```
function AnsiStrLower(Str: PChar): PChar;
```

Функция AnsiStrLower на месте преобразует Str в нижний регистр. Функция возвращает значение Str. Преобразование использует ре-

гиональные настройки Windows для корректного преобразования не-ASCII символов.

AnsiStrPos, функция

```
function AnsiStrPos(Str, SubStr: PChar): PChar;
```

Функция AnsiStrPos ищет первое вхождение SubStr в Str и возвращает либо указатель на начало подстроки в Str, либо nil, если подстрока не найдена.

AnsiStrRScan, функция

```
function AnsiStrRScan(Str: PChar; Chr: Char): PChar;
```

Функция AnsiStrRScan выполняет поиск последнего (самого правого) вхождения символа Chr в Str и возвращает указатель на символ в строке Str или nil, если символ не найден.

AnsiStrScan, функция

```
function AnsiStrScan(Str: PChar; Chr: Char): PChar;
```

Функция AnsiStrScan выполняет поиск первого вхождения символа Chr в Str и возвращает указатель на символ в Str или nil, если символ не найден.

AnsiStrUpper, функция

```
function AnsiStrUpper(Str: PChar): PChar;
```

Функция AnsiStrUpper на месте преобразует Str в верхний регистр. Функция возвращает значение Str. Преобразование использует региональные настройки Windows для корректного преобразования не-ASCII символов.

StrAlloc, функция

```
function StrAlloc(Size: Cardinal): PChar;
```

Функция StrAlloc выделяет память для строки PChar, которая может содержать до Size - 1 байт. Память не инициализируется. Для освобождения выделенной памяти применяйте процедуру StrDispose.

StrBufSize, функция

```
function StrBufSize(const Str: PChar): Cardinal;
```

Функция StrBufSize возвращает количество байт, выделенных функцией StrAlloc или StrNew. Для любых других строк PChar эта функция не работает.

StrByteType, функция

```
function StrByteType(Str: PChar; Index: Cardinal): TMbcsByteType;
```

Функция StrByteType возвращает тип байта в позиции Index строки Str. Программист должен позаботиться, чтобы Index был коррект-

ным индексом для строки. Байт может быть однобайтовым символом, первым или вторым байтом многобайтового символа. Перечисляемый тип TMbcsByteType подробно рассмотрен в описании функции ByteType ранее в этом приложении.

StrCat, функция

```
function StrCat(Dest: PChar; const Source: PChar): PChar;
```

Функция StrCat копирует Source в конец Dest. Необходимо убедиться, что в Dest достаточно места для копии Source. Рассмотрите возможность применения процедуры StrLCat для того, чтобы предотвратить переполнение буфера Dest. StrCat возвращает указатель на начало целевой строки (Dest).

StrCopy, функция

```
function StrCopy(Dest: PChar; const Source: PChar): PChar;
```

Функция StrCopy копирует Source в Dest. Убедитесь, что в Dest достаточно места для Source. В большинстве программ следует использовать процедуру StrLCopy, чтобы не вызвать переполнение буфера Dest. StrCopy возвращает указатель на начало целевой строки (Dest).

StrDispose, процедура

```
procedure StrDispose(Str: PChar);
```

Процедура StrDispose освобождает строку PChar, выделенную функциями StrNew и StrAlloc. Если Str равно nil, StrDispose завершается, не выполняя никаких действий. Не путайте эту процедуру с DisposeStr.

StrECopy, функция

```
function StrECopy(Dest: PChar; const Source: PChar): PChar;
```

Функция StrECopy копирует Source в Dest аналогично StrCopy, однако StrECopy возвращает указатель на символ #0 в конце Dest.

StrEnd, функция

```
function StrEnd(const Str: PChar): PChar;
```

Функция StrEnd возвращает указатель на символ #0 в конце строки Str.

StrLCat, функция

```
function StrLCat(Dest: PChar; const Source: PChar; MaxLen: Cardinal):
    PChar;
```

StrLCat копирует (дописывает) до MaxLen байт из Source в конец Dest. StrLCat возвращает указатель на начало целевой строки (Dest). Если Source – многобайтовая строка, следует гарантировать, чтобы MaxLen не попадал в середину многобайтового символа.

StrLCopy, функция

```
function StrLCopy(Dest: PChar; const Source: PChar; MaxLen: Cardinal):
    PChar;
```

Функция StrLCopy копирует до MaxLen байт из Source в Dest. Если Source является многобайтовой строкой, следует гарантировать, чтобы MaxLen не попадал в середину многобайтового символа.

StrLen, функция

```
function StrLen(const Str: PChar): Cardinal;
```

Функция StrLen возвращает длину строки Str в байтах. Заметьте, что количество символов в многобайтовой строке может превышать количество байт.

StrMove, функция

```
function StrMove(Dest: PChar; const Source: PChar; Count: Cardinal):
    PChar;
```

Функция StrMove копирует точно Count байтов из Source в Dest. В отличие от StrCopy, StrMove не останавливается в конце строки Source. Буфер-источник и целевой буфер могут перекрываться. StrMove возвращает значение Dest. Процедура Move (описанная в главе 5) и StrMove делают одно и то же; выберите наиболее удобную функцию для конкретного применения.

StrNew, функция

```
function StrNew(const Str: PChar): PChar;
```

Функция StrNew создает новую копию Str и возвращает указатель на начало новой строки. Для освобождения строки следует использовать StrDispose. Не путайте эту функцию с NewStr.

StrPCopy, функция

```
function StrPCopy(Dest: PChar; const Source: string): PChar;
```

Функция StrPCopy копирует строку AnsiString в строку PChar. Необходимо гарантировать, что буфер Dest достаточен для строки Source. Можно воспользоваться функцией StrPLCopy, чтобы не допустить переполнения буфера Dest. StrPCopy возвращает указатель на начало строки Dest.

StrPLCopy, функция

```
function StrPLCopy(Dest: PChar; const Source: string;
    MaxLen: Cardinal): PChar;
```

Функция StrPLCopy копирует до MaxLen байт из ANSI-строки Source в Dest. Возвращает указатель на начало Dest. Если Source – многобайтовая строка, следует гарантировать, что MaxLen не попадает в середину многобайтового символа.

Устаревшие строковые подпрограммы

Строковые подпрограммы и переменные, описанные в этом разделе, устарели и не должны использоваться в новых программах. Они оставлены только для обратной совместимости.

AppendStr, процедура

```
procedure AppendStr(var Dest: string; const S: string);
```

Процедура `AppendStr` добавляет строку `S` в конец `Dest`. В новом коде следует использовать обычный оператор присваивания.

```
Dest := Dest + S;
```

AssignStr, процедура

```
procedure AssignStr(var P: PString; const S: string);
```

Процедура `AssignStr` выполняет обычное присваивание строки, т. е. `P^ := S`.

CompareStr, функция

```
function CompareStr(const S1, S2: string): Integer;
```

Функция `CompareStr` выполняет сравнение двух строк и возвращает целый результат из табл. В-7. Сравнение выполняется с учетом регистра и не учитывает региональных настроек Windows. В новых программах следует использовать `AnsiCompareStr`.

Таблица В-7. Результаты функции CompareStr

Результат	Описание
Отрицательный	`S1 < S2`
Ноль	`S1 = S2`
Положительный	`S1 > S2`

CompareText, функция

```
function CompareText(const S1, S2: string): Integer;
```

Функция `CompareText` сравнивает две строки и возвращает целый результат аналогично `CompareStr`, однако сравнение выполняется без учета регистра. Региональные настройки Windows не используются, поэтому только символы ASCII сравниваются без учета регистра. В новых программах следует вызывать `AnsiCompareText`.

DisposeStr, процедура

```
procedure DisposeStr(P: PString);
```

Процедура `DisposeStr` освобождает строку, выделенную функцией `NewStr`. Не путайте эту процедуру с `StrDispose`.

EmptyStr, переменная

```
var EmptyStr: string = '';
```

EmptyStr — это глобальная переменная, содержащая пустую строку. Не меняйте значение этой переменной. Она оставлена исключительно для обратной совместимости. В Delphi пустая строка представлена нулевым указателем, и лучше следует предпочесть явное указание пустой строки (`''`) ссылке на глобальную переменную.

LowerCase, функция

```
function LowerCase(const S: string): string;
```

Функция LowerCase преобразует строку в нижний регистр, однако изменяются только символы ASCII. В новых программах следует использовать AnsiLowerCase.

NewStr, функция

```
function NewStr(const S: string): PString;
```

Функция NewStr создает новую строку как копию строки S и возвращает указатель на новую строку. Длинные строки теперь реализованы как указатели, поэтому нет смысла использовать дополнительный уровень адресации. Для освобождения строки предназначена DisposeStr. Не путайте эту функцию с StrNew.

NullStr, переменная

```
var NullStr: PString = @EmptyStr;
```

NullStr является указателем на пустую строку. Не изменяйте значение этой переменной.

QuotedStr, функция

```
function QuotedStr(const S: string): string;
```

Функция QuotedStr возвращает копию строки, заключенную в одинарные кавычки. Одинарные кавычки, присутствующие в S, повторяются, как принято и в языке Паскаль. QuotedStr работает медленно, если в S содержится много одинарных кавычек. В новых программах следует применять AnsiQuotedStr.

SameText, функция

```
function SameText(const S1, S2: string): Boolean;
```

Функция SameText возвращает True, если S1 и S2 имеют одинаковое содержимое без учета регистра, т. е. если CompareText возвращает ноль. В новых программах следует использовать AnsiSameText.

StrComp, функция

```
function StrComp(const Str1, Str2: PChar): Integer;
```

Функция StrComp выполняет сравнение строк Str1 и Str2, возвращая целый результат аналогично CompareStr. В новых программах следует применять AnsiStrComp.

StrIComp, функция

```
function StrIComp(const Str1, Str2: PChar): Integer;
```

Функция StrIComp выполняет сравнение строк Str1 и Str2 без учета регистра и региональных настроек Windows, возвращая целый результат аналогично CompareStr. В новых программах следует использовать AnsiStrIComp.

StrLComp, функция

```
function StrLComp(const Str1, Str2: PChar; MaxLen: Cardinal): Integer;
```

Функция StrLComp выполняет сравнение до MaxLen символов в строках Str1 и Str2 , возвращая целый результат аналогично CompareStr. Регистр имеет значение, однако региональные настройки Windows не учитываются. В новых программах следует использовать AnsiStrLComp.

StrLIComp, функция

```
function StrLIComp(const Str1, Str2: PChar; MaxLen: Cardinal): Integer;
```

Функция StrLIComp выполняет сравнение до MaxLen символов в строках Str1 и Str2, возвращая целый результат аналогично CompareStr. Регистр значения не имеет, региональные настройки Windows не учитываются. В новых программах следует использовать AnsiStrLIComp.

StrLower, функция

```
function StrLower(Str: PChar): PChar;
```

Функция StrLower преобразует Str в нижний регистр и возвращает указатель на начало строки. Преобразуются только символы ASCII, региональные настройки Windows не учитываются.

StrPas, функция

```
function StrPas(const Str: PChar): string;
```

Функция StrPas преобразует строку PChar в AnsiString. Delphi делает это автоматически, поэтому в новых программах нет смысла вызывать StrPas.

StrPos, функция

```
function StrPos(const Str, SubStr: PChar): PChar;
```

Функция StrPos выполняет поиск первого вхождения SubStr в Str и возвращает указатель на первый байт подстроки в Str. Если подстрока не найдена, возвращается nil. В новых программах следует использовать AnsiStrPos.

StrRScan, функция

```
function StrRScan(const Str: PChar; Chr: Char): PChar;
```

Функция StrRScan выполняет поиск последнего (крайнего правого) вхождения символа Chr в строке Str и возвращает указатель на символ Str или nil, если символ не найден. В новых программах следует использовать AnsiStrRScan.

StrScan, функция

```
function StrScan(const Str: PChar; Chr: Char): PChar;
```

Функция StrScan выполняет поиск первого вхождения символа Chr в строке Str и возвращает указатель на символ Str или nil, если символ не найден. В новых программах следует использовать AnsiStrScan.

StrUpper, функция

```
function StrUpper(Str: PChar): PChar;
```

Функция StrUpper на месте преобразует Str в верхний регистр и возвращает указатель на начало строки. Преобразуются только символы ASCII, региональные настройки Windows не учитываются. В новых программах следует использовать AnsiStrUpper.

Trim, функция

```
function Trim(const S: string): string;
```

Функция Trim возвращает копию строки S без ведущих и завершающих управляющих символов и пробелов. Управляющие символы – это любые символы, порядковая величина которых меньше значения символа «пробел» в ANSI (#32). В отличие от других устаревших функций, в Delphi нет соответствующей ей функции AnsiTrim. В примере В-4 приведен один из способов реализации такой функции.

Пример В-4. ANSI-версия функции Trim

```
type
  TTrimType = (ttLeft, ttRight);
  TTrimTypes = set of TTrimType;
// Отбрасывает пробелы и управляющие символы.
// Корректно обрабатывает многобайтовые строки.
function AnsiTrim(const S: string;
    TrimType: TTrimTypes = [ttLeft, ttRight]): string;
var
  Left, Right: 1..MaxInt;
  I: 1..MaxInt;
begin
  Left := 1;
  Right := Length(S);
  if ttLeft in TrimType then
```

```
  begin
    I := 1;
    while I <= Length(S) do
    begin
      if S[I] in LeadBytes then
        Inc(I)
      else if Ord(S[I]) > Ord(' ') then
        Break;
      Inc(I);
    end;
    Left := I;
  end;
  if ttRight in TrimType then
  begin
    I := Length(S);
    while I > Left do
    begin
      if ByteType(S, I) = mbTrailByte then
        Dec(I)
      else if Ord(S[I]) > Ord(' ') then
        Break;
      Dec(I);
    end;
    Right := I;
  end;
  Result := Copy(S, Left, Right);
end;

function AnsiTrimLeft(const S: string): string;
begin
  Result := AnsiTrim(S, [ttLeft]);
end;

function AnsiTrimRight(const S: string): string;
begin
  Result := AnsiTrim(S, [ttRight]);
end;
```

TrimLeft, функция

```
function TrimLeft(const S: string): string;
```

Функция `TrimLeft` **возвращает копию строки** `S` **без ведущих управляющих символов и пробелов. Управляющие символы – это любые символы, порядковая величина которых меньше значения символа «пробел» в ANSI (**`#32`**).**

TrimRight, функция

```
function TrimRight(const S: string): string;
```

Функция `TrimLeft` **возвращает копию строки** `S` **без завершающих управляющих символов и пробелов. Управляющие символы – это лю-**

бые символы, код которых меньше значения символа «пробел» в ANSI (#32).

UpperCase, функция

```
function UpperCase(const S: string): string;
```

Функция UpperCase возвращает копию строки S, в которой ASCII-символы нижнего регистра преобразованы в верхний регистр. Для обработки ANSI-символов с учетом региональных настроек Windows используйте AnsiUpperCase.

Преобразование чисел

В этом разделе перечислены типы, константы и подпрограммы, которые облегчают преобразование целых, денежных значений и чисел с плавающей точкой в текст, и преобразование текста в числа.

CurrencyDecimals, переменная

```
var CurrencyDecimals: Byte;
```

В переменной CurrencyDecimals указано значение по умолчанию для количества цифр справа от десятичной точки, используемое функцией Format и другими подобными функциями при форматировании денежных значений. Значение по умолчанию берется из региональных настроек Windows.

CurrencyFormat, переменная

```
var CurrencyFormat: Byte;
```

Переменная CurrencyFormat определяет, где расположение символа валюты относительно денежной суммы при форматировании положительного значения типа Currency. Значение по умолчанию берется из региональных настроек Windows. Возможные значения перечислены в табл. B-8.

Таблица B-8. Значения CurrencyFormat

Значение	Описание	Пример
0	<символ><сумма>	$1.00
1	<сумма><символ>	1.00$
2	<символ><пробел><сумма>	$ 1.00
3	<сумма><пробел><символ>	1.00 $

CurrencyString, переменная

```
var CurrencyString: string;
```

CurrencyString – это символ, идентифицирующий денежное значение, например «$» для долларов США. Значение по умолчанию берется из региональных настроек Windows.

CurrToStr, функция

```
function CurrToStr(Value: Currency): string;
```

Функция CurrToStr преобразует значение типа Currency в строку, используя локальный символ валюты и общий формат вещественных чисел. Минимальное количество цифр после десятичной точки для форматированной строки не установлено (т. е. значение CurrencyDecimals не учитывается).

CurrToStrF, функция

```
function CurrToStrF(Value: Currency; Format: TFloatFormat;
    Digits: Integer): string;
```

Функция CurrToStrF преобразует значение типа Currency в строку, используя данный Format (обычно ffCurrency) и Digits цифр после десятичной точки. Дополнительная информация о параметре Format приведена в описании функции FloatToStrF.

DecimalSeparator, переменная

```
var DecimalSeparator: Char;
```

DecimalSeparator – это символ, который отделяет целую часть вещественного числа от дробной. Значение по умолчанию берется из региональных настроек Windows.

FloatToDecimal, процедура

```
type
  TFloatRec = packed record
    Exponent: Smallint;
    Negative: Boolean;
    Digits: array[0..20] of Char;
  end;
  TFloatValue = (fvExtended, fvCurrency);
procedure FloatToDecimal(var Result: TFloatRec; const Value;
    ValueType: TFloatValue; Precision, Decimals: Integer);
```

Функция FloatToDecimal – это низкоуровневая процедура форматирования. Другие функции форматирования вызывают ее, однако в приложениях она применяется редко. FloatToDecimal записывает экспоненту и знак в запись Result и форматирует мантиссу в виде строки десятичных цифр. Precision – это минимальное количество значащих цифр, Decimals – количество цифр перед десятичной точкой. Значение Value должно иметь тип Currency или Extended, и необходимо указать соответствующий тип в ValueType (ftCurrency или ftExtended). Если ValueType и тип Value не совпадают, результат может быть непредсказуемым.

FloatToStr, функция

```
function FloatToStr(Value: Extended): string;
```

Функция FloatToStr преобразует вещественное число в строку, используя общий формат с 15 цифрами точности.

FloatToStrF, функция

```
type TFloatFormat = (ffGeneral, ffExponent, ffFixed, ffNumber,
                     ffCurrency);
function FloatToStrF(Value: Extended; Format: TFloatFormat;
    Precision, Digits: Integer): string;
```

Функция FloatToStrF возвращает вещественное значение Value в виде форматированной строки. Аргумент Format определяет, как форматируется число. Precision – это количество значащих цифр, Digits – обычно количество цифр после десятичной точки. Форматированная строка поможет содержать разделители DecimalSeparator и ThousandSeparator.

Максимальная точность функции FloatToStrF – 18 десятичных знаков, однако максимальная точность типов Extended, Currency и Comp – 19 десятичных знаков. Вычисления с этими типами выполняются корректно с полной точностью, однако если попытаться вывести на экран или на печать максимальное или минимальное значение Currency или Comp, последние десятичные знаки не будут корректными, а значения типа Extended не могут быть отформатированы с максимальной точностью.

Независимо от значения Format плюс бесконечность всегда форматируется как строка INF, минус бесконечность – как -INF. «Спокойный» NaN форматируется как NAN. (Строки INF и NAN не могут быть локализованы.) Для конечных значений параметр Format работает следующим образом:

ffCurrency

Число форматируется с использованием значений CurrencyFormat или NegCurrFormat (например, $31 415,00). Digits определяет количество знаков после десятичной точки. Если Digits равно нулю, десятичный разделитель не выводится.

ffExponent

Форматирует число в соответствии с научной нотацией (например -3,14E+01). Всегда присутствует минимум одна цифра перед десятичным разделителем, Precision определяет общее число форматируемых символов. Digits определяет количество цифр в экспоненте, которое может быть в диапазоне от 1 до 4. Экспонента всегда начинается со знака «плюс» или «минус». При необходимости перед всем числом указывается знак «минус».

ffFixed

Число форматируется в соответствии с фиксированной десятичной нотацией (например -3141,59). Как минимум одна цифра всегда присутствует перед десятичным разделителем. Если слева от десятичного разделителя требуется более чем Precision цифр, формат автоматически меняется на ffExponent.

ffGeneral

Форматирует число в соответствии с фиксированной или экспоненциальной нотацией: по возможности применяется фиксированная нотация, в противном случае – экспоненциальная. Для чисел менее 0,00001 всегда используется экспоненциальная нотация. Ведущие нули удаляются, и если перед десятичной точкой нет цифр, символ разделителя также удаляется.

ffNumber

Форматирует строку в фиксированном формате, но использует символ ThousandSeparator для разделения тысяч, миллионов и т. д. (например –314 159,26535).

FloatToText, функция

```
function FloatToText(Buffer: PChar; const Value; ValueType:TFloatValue;
    Format: TFloatFormat; Precision, Digits: Integer): Integer;
```

Функция FloatToText – низкоуровневая, преобразует число с плавающей точкой в массив символов, сохраняя результат в Buffer. Возвращает длину форматированной строки. Заметьте, что строка *не* дополняется символом #0. Необходимо гарантировать, что Buffer имеет достаточный размер для хранения форматированной строки. Аргументы Value, ValueType, Format, Precision и Digits интерпретируются так же, как и в функции FloatToDecimal.

FloatToTextFmt, функция

```
function FloatToTextFmt(Buffer: PChar; const Value;
    ValueType: TFloatValue; Format: PChar): Integer;
```

FloatToTextFmt – низкоуровневая, форматирует вещественное число в соответствии со строкой формата Format. Результат записывается в Buffer, который должен иметь размер, достаточный для форматированной строки. Возвращает длину строки. Заметьте, что строка *не* дописывается символом #0. Value может иметь тип Currency или Extended, ValueType указывает на тип Value, как в функции FloatToDecimal.

Строка Format указывает способ форматирования строки. В ней может содержаться до трех частей, разделенных точкой с запятой. Первая часть определяет формат положительных чисел, вторая – отрицательных, третья – формат отображения нуля. Если какая-либо из частей отсутствует, используется формат положительных чисел. Если отсутствует положительная часть, применяется общий

формат FloatToStrF. Каждая часть состоит из спецификаторов формата. Спецификаторы описывают формат или маску для одной числовой величины. Значение округляется до того количества десятичных знаков, которое указано после десятичного разделителя. Количество и положение спецификаторов 0 определяет, сколько десятичных знаков до и после десятичного разделителя всегда будет присутствовать в форматированной строке.

#

Указывает на цифру, если в данной позиции числа должна быть цифра.

0

Указывает на цифру, если в данной позиции числа должна быть цифра; иначе используется '0'.

. *(точка)*

Указывает на десятичный разделитель (переменная DecimalSeparator). Второе и последующие указания этого спецификатора в одной части формата игнорируются. Если формат не содержит спецификатора десятичной точки, значение округляется до целого.

, *(запятая)*

Указывает на разделитель тысяч (переменная ThousandSeparator) между каждой группой из трех цифр слева от десятичного разделителя. Этот спецификатор может присутствовать в любом месте строки формата.

E+
E-
e+
e-

Форматирует число с использованием научной нотации. Регистр спецификатора определяет регистр символа экспоненты (E) в форматированной строке. Спецификаторы E+ и e+ генерируют знак «плюс» для положительных экспонент. Спецификаторы E- и e- опускают знак «плюс» для положительных экспонент.

'xxx'
"xxx"

Символы в кавычках копируются в форматированную строку буквально (без самих кавычек). Можно указывать в кавычках другие символы форматирования.

;

Точка с запятой разделяет положительную, отрицательную и нулевую части строки формата.

(все остальное)

Все остальные символы копируются в форматированную строку.

В табл. В-9 приведено несколько примеров строк формата.

Таблица В-9. Примеры строк формата для FloatToTextFmt

Строка формата	Значение	Результат
#.00	−31,456	-31,46
,0.##	3 141 592	3 141 592
#;(#);'0.'	0	0.
#;(#)	−31,456	(31)

FmtLoadStr, функция

```
function FmtLoadStr(Ident: Integer; const Args: array of const):string;
```

Функция FmtLoadStr загружает строковый ресурс и определяет строку из него как строку формата при вызове Format. В новых программах следует использовать resourcestring и вызывать Format напрямую.

FmtStr, процедура

```
procedure FmtStr(var Result: string; const Format: string;
    const Args: array of const);
```

Функция FmtStr форматирует строку аналогично Format, однако вместо передачи ее в качестве результата функции записывает ее в Result.

Format, функция

```
function Format(const Format: string; const Args: array of const):
    string;
```

Функция Format возвращает форматированную строку. Аргумент Format указывает, как формировать строку, используя аргументы из вариантного открытого массива Args. Строка Format копируется в результирующую строку, при этом разворачиваются спецификаторы формата, обозначенные знаком процента (%). Для того чтобы определить знак процента как литерал, повторите его: %%. Все спецификаторы формата имеют следующий синтаксис:

```
% [index :] [-] [width] [.precision] type
```

index – это индекс в массиве Args, где первый аргумент обозначается нулем. Без указания индекса каждый спецификатор формата соответствует следующему элементу массива Args. Если указан индекс, можно несколько раз повторять один и тот же аргумент в одной строке формата. Планируя локализовать приложение, следует использовать спецификатор *индекса* для всех аргументов, т. к. в другом языке может потребоваться другой порядок аргументов.

width определяет минимальное количество символов для форматируемого поля. Если фактическая форматируемая величина меньше, чем *width* символов, она дополняется пробелами слева. Если перед *width* указан знак «минус», число выравнивается слева и дополняется пробелами справа.

Разные типы форматов по-разному используют *precision*. Подробности рассмотрены в описаниях каждого из форматов.

Можно жестко указать числовые значения *index*, *width* и *precision* в строке формата (например `%1:10.2f`) или использовать звездочку для любого или всех этих значений (например `%*:*.*f`). Тогда для данного значения будет использован следующий аргумент из массива `Args`. Значение `Args` должно быть целым.

Тип формата (*type*) обозначается одним символом в верхнем или нижнем регистре:

`d` Форматирует десятичное целое со знаком. *Precision* определяет минимальное число выводимых цифр; по необходимости число дополняется нулями. Значение `Args` должно быть целым.

`e` Форматирует вещественное число в соответствии с экспоненциальной нотацией. *Precision* определяет общее число цифр в форматированной строке. По умолчанию равно 15. Значение `Args` должно быть числом с плавающей точкой или `Currency`.

`f` Форматирует вещественное число в соответствии с фиксированной нотацией. *Precision* определяет количество цифр после десятичного разделителя. По умолчанию равно 2. Значение `Args` должно быть числом с плавающей точкой или `Currency`.

`g` Форматирует вещественное число в соответствии с общей нотацией. *Precision* определяет количество значащих цифр. Значение `Args` должно быть числом с плавающей точкой или `Currency`.

`m` Форматирует значение типа `Currency`. *Precision* определяет количество знаков после десятичной точки. Значение по умолчанию определяется переменной `CurrencyDecimals`, а формат валюты – переменными `CurrencyFormat` и `NegCurrFormat`. Значение `Args` должно быть числом с плавающей точкой или `Currency`.

`n` Форматирует вещественное число в соответствии с числовой нотацией, т. е. по правилам фиксированной нотации с разделителем `ThousandSeparator` для групп из трех цифр. Значение `Args` должно быть числом с плавающей точкой или `Currency`.

`p` Форматирует указатель как шестнадцатеричное число. Значение `Args` должно быть указателем.

- s Форматирует символ или строку. *Precision* определяет максимальное число выводимых символов. Значение Args должно иметь тип AnsiChar, AnsiString, PChar, PWideChar, ShortString, Variant или WideString. Тип WideChar не допускается. Variant преобразуется в строку и затем форматируется.
- u Форматирует целое десятичное число без знака. *Precision* определяет минимальное число выводимых десятичных цифр; число дополняется по необходимости нулями. Значение Args должно быть целым.
- x Форматирует шестнадцатеричное целое без знака. Регистр спецификатора типа определяет регистр символов шестнадцатеричного числа. *Precision* определяет минимальное число выводимых цифр, при необходимости дополняется нулями. Значение Args должно быть целым.

Если строка формата содержит ошибки или если *type* не совпадает с типом соответствующего аргумента массива Args, Format генерирует исключительную ситуацию EConvertError.

FormatBuf, функция

```
function FormatBuf(var Buffer; BufLen: Cardinal; const Format;
    FmtLen: Cardinal; const Args: array of const): Cardinal;
```

Функция FormatBuf форматирует строку и записывает результат в Buffer. В Buffer записывается не более BufLen символов, включая завершающий байт #0. Длина форматированной строки задается в FmtLen. Возвращается длина результирующей строки. Дополнительную информацию можно найти в описании функции Format.

FormatCurr, функция

```
function FormatCurr(const Format: string; Value: Currency): string;
```

Функция FormatCurr форматирует величину типа Currency в строку в соответствии с аргументом Format. Дополнительная информация приведена в описании функции FloatToTextFmt.

FormatFloat, функция

```
function FormatFloat(const Format: string; Value: Extended): string;
```

Функция FormatFloat форматирует число с плавающей точкой в строку в соответствии с аргументом Format. Дополнительные сведения содержатся в описании функции FloatToTextFmt.

HexDisplayPrefix, переменная

```
var HexDisplayPrefix: string;
```

HexDisplayPrefix – это строка, указываемая перед шестнадцатеричными целыми. В программах на Delphi это '$'. В программах на C++

Builder это '0x'. Не изменяйте значение этой переменной – несколько модулей VCL проверяют ее, чтобы определить, является приложение проектом Delphi или C++ Builder. (Правильный способ выполнения этой проверки рассмотрен в описании переменной ModuleIsCpp в главе 5.)

IntToHex, функция

```
function IntToHex(Value: Int64; Digits: Integer): string; overload;
function IntToHex(Value: Integer; Digits: Integer): string; overload;
```

Функция IntToHex форматирует целое в шестнадцатеричную строку из Digits знаков. В начало результирующей строки HexDisplayPrefix *не* добавляется.

IntToStr, функция

```
function IntToStr(Value: Int64): string; overload;
function IntToStr(Value: Integer): string; overload;
```

Функция IntToStr форматирует целое в виде строки.

NegCurrFormat, переменная

```
var NegCurrFormat: Byte;
```

NegCurrFormat определяет, как форматируются отрицательные значения типа Currency. Эта переменная похожа на CurrencyFormat, но имеет значительно больше возможных значений (табл. B-10). Значение по умолчанию берется из региональных настроек Windows.

Таблица B-10. Значения NegCurrFormat

Значение	Описание	Пример
0	(<символ><сумма>)	($1,00)
1	–<символ><сумма>	-$1,00
2	<символ>–<сумма>	$-1,00
3	<символ><сумма>–	$1,00-
4	(<сумма><символ>)	(1,00$)
5	–<сумма><символ>	-1,00$
6	<сумма>–<символ>	1,00-$
7	<сумма><символ>–	1,00$-
8	–<сумма><пробел><символ>	-1,00 $
9	–<символ><пробел><сумма>	-$ 1,00
10	<сумма><пробел><символ>–	1,00 $-
11	<символ><пробел><сумма>–	$ 1,00-

Значение	Описание	Пример
12	<символ><пробел>–<сумма>	$ -1,00
13	<сумма>–<пробел><символ>	1,00- $
14	(<символ><пробел><сумма>)	($ 1,00)
15	(<сумма><пробел><символ>)	(1,00 $)

StrFmt, функция

```
function StrFmt(Buffer, Format: PChar; const Args: array of const):
    PChar;
```

Функция `StrFmt` форматирует строку и записывает результат в `Buffer`. Необходимо гарантировать, что размер буфера достаточен для записи результата. Возвращаемым значением является `Buffer`. Чтобы избежать переполнения буфера, используйте вместо этой функции `StrLFmt`.

StrLFmt, функция

```
function StrLFmt(Buffer: PChar; MaxLen: Cardinal; Format: PChar;
    const Args: array of const): PChar;
```

Функция `StrLFmt` форматирует строку и записывает результат в `Buffer`. В `Buffer` записывается не более `MaxLen` байт, не считая завершающего байта `#0`, который всегда дописывается к форматированной строке. Возвращаемым значением является `Buffer`.

StrToCurr, функция

```
function StrToCurr(const S: string): Currency;
```

Функция `StrToCurr` преобразует строку в значение типа `Currency`. Если строка не может быть преобразована, генерируется исключительная ситуация `EConvertError`. Дополнительная информация приведена в описании функции `TextToFloat`.

StrToFloat, функция

```
function StrToFloat(const S: string): Extended;
```

Функция `StrToFloat` преобразует строку в вещественное значение. Если строка не может быть преобразована, генерируется исключительная ситуация `EConvertError`. Дополнительную информацию можно найти в описании функции `TextToFloat`.

StrToInt64, функция

```
function StrToInt64(const S: string): Int64;
```

Функция `StrToInt64` преобразует строку в целое `Int64`. Ведущие и завершающие пустые символы игнорируются. Строка может начинаться со знака (‘+’ или ‘-‘) и может быть либо десятичным, либо

шестнадцатеричным представлением числа (обозначенным '$' или в стиле C++ – '0x' или '0X', т. е. за нулем следует буква *x* в любом регистре). Если строка пустая или не является корректным целым, генерируется исключительная ситуация EConvertError.

StrToInt, функция

```
function StrToInt(const S: string): Integer;
```

Функция StrToInt преобразует строку в целое число. Ведущие и завершающие пустые символы игнорируются. Строка может начинаться со знака ('+' или '-') и может быть либо десятичным, либо шестнадцатеричным представлением числа (обозначенным '$' или в стиле C++ – '0x'). Если строка пустая или не является корректным целым, генерируется исключительная ситуация EConvertError.

StrToInt64Def, функция

```
function StrToInt64Def(const S: string; Default: Int64): Int64;
```

Функция StrToInt64Def преобразует строку в целое Int64. Ведущие и завершающие пустые символы игнорируются. Строка может начинаться со знака ('+' или '-') и может быть либо десятичным, либо шестнадцатеричным представлением числа (обозначенным '$' или в стиле C++ – '0x'). Если строка пустая или не является корректным целым, возвращается Default.

StrToIntDef, функция

```
function StrToIntDef(const S: string; Default: Integer): Integer;
```

Функция StrToIntDef преобразует строку в целое число. Ведущие и завершающие пустые символы игнорируются. Строка может начинаться со знака ('+' или '-') и может быть либо десятичным, либо шестнадцатеричным представлением числа (обозначенным '$' или в стиле C++ – '0x'). Если строка пустая или не является корректным целым, возвращается Default.

TextToFloat, функция

```
function TextToFloat(Buffer: PChar; var Value; ValueType: TFloatValue):
    Boolean;
```

Функция TextToFloat выполняет разбор строки для поиска вещественного числа. Значение ValueType должно быть равно ftCurrency или ftExtended и совпадать с типом Value. В строке не может быть символов валюты и разделителей тысяч. Символ десятичного разделителя (если он присутствует в строке) должен совпадать со значением переменной DecimalSeparator. Число может быть записано в фиксированной или экспоненциальной нотации и иметь ведущий знак «плюс» или «минус». Функция TextToFloat возвращает True в случае успеха или False, если строка не может быть преобразована в число.

ThousandSeparator, переменная

```
var ThousandSeparator: Char;
```

ThousandSeparator – это символ, используемый для разделения тысяч в форматированном числовом или денежном значении. Значение по умолчанию берется из региональных настроек Windows.

Дата и время

В этом разделе перечислены типы, константы и подпрограммы, относящиеся к работе с датами и временем. В Delphi имеется пять способов хранения даты и времени:

- Тип TDateTime – наиболее распространенный способ хранения даты и времени. Подробности рассмотрены в описании этого типа в главе 5.
- DOS хранит дату изменения файла в простом формате, совпадающем по размеру с целым. Дату файла можно преобразовывать в тип TDateTime и обратно.
- Тип TTimeStamp – это запись, хранящая дату и время в отдельных полях, поэтому с ней сложнее работать, чем с TDateTime. Между этими типами можно выполнять преобразования.
- Год, месяц, день, час, минуту, секунду и миллисекунду можно хранить как отдельные значения, преобразуемые в TDateTime и обратно с помощью подпрограмм кодирования и декодирования (Encode... и Decode...).
- Windows хранит отдельные компоненты даты и времени в записи TSystemTime. В Delphi имеются функции, облегчающие преобразование между TSystemTime и TDateTime.

Date, функция

```
function Date: TDateTime;
```

Функция Date возвращает текущую дату в локальном часовом поясе. Время устанавливается на полночь (ноль).

DateDelta, константа

```
const DateDelta = 693594;
```

DateDelta – это количество дней между 1 января 0001 года и 31 декабря 1899. Первое значение – это начальная дата, используемая для TTimeStamp, второе – начальная дата для TDateTime.

DateSeparator, переменная

```
var DateSeparator: Char;
```

DateSeparator – это символ, применяемый для разделения месяца, дня и года в кратком формате даты. Значение по умолчанию берется из региональных настроек Windows.

DateTimeToFileDate, функция

```
function DateTimeToFileDate(DateTime: TDateTime): Integer;
```

Функция DateTimeToFileDate преобразует значение TDateTime в дату файла, которую можно использовать при вызове функции FileSetDate. На рис. B-1 (ранее в этом приложении) показан формат даты и времени файла. Время обрезается до двухсекундной точности. Если год не попадает в диапазон даты файла (от 1980 до 2099), возвращается ноль.

DateTimeToStr, функция

```
function DateTimeToStr(DateTime: TDateTime): string;
```

Функция DateTimeToStr форматирует значение TDateTime в строку в следующем порядке: сначала дату в соответствии с ShortDateFormat, затем вставляя пробел, затем время в соответствии с LongTimeFormat. Время опускается, если оно равно нулю (полночь).

DateTimeToString, процедура

```
procedure DateTimeToString(var Result: string; const Format: string;
    DateTime: TDateTime);
```

Процедура DateTimeToString преобразует значение TDateTime в строку, используя Format для управления форматированием. Полученная строка записывается в Result. Если попытаться сформировать строку длиннее 256 символов, DateTimeToString обрезает ее.

Формат даты и времени определяется путем последовательной интерпретации символов строки Format. Результирующая строка формируется из соответствующим образом форматированных даты и времени в порядке, указанном в строке формата. Символы, которые не являются спецификаторами формата, копируются в результирующую строку. Повторное указание символа форматирования большее число раз, чем указано в этом разделе, обычно не опасно, но может привести к непредсказуемым результатам. Строка Format не чувствительна к регистру (кроме спецификаторов AM/PM и A/P). Если строка Format является пустой, по умолчанию задается формат 'C'. Ниже приведены спецификаторы формата даты и времени:

c — Используется ShortDateFormat, затем пробел, затем LongTimeFormat. Время опускается, если оно равно нулю (полночь).

d — Обозначает день месяца (1-31).

dd — Обозначает день месяца, дополненный до двух символов (01-31).

`aaa` `ddd` — Обозначает день недели из `ShortDayNames`.

`aaaa` `dddd` — Обозначает день недели из `LongDayNames`.

`ddddd` — Обозначает дату в формате `ShortDateFormat`.

`dddddd` — Обозначает дату в формате `LongDateFormat`.

`g` — Обозначает краткое имя эры для японских и тайваньских региональных настроек.

`gg` — Обозначает полное имя эры для азиатских региональных настроек.

`e` — Обозначает год текущей эры без ведущего нуля.

`ee` — Обозначает год текущей эры, дополненный ведущими нулями до минимум двух цифр.

`m` — Обозначает номер месяца (`1-12`). Если `m` следует сразу за спецификатором `h` или `hh`, обозначает минуты (`0-59`).

`mm` — Обозначает номер месяца, дополненный до двух цифр (`01-12`). Если `mm` следует сразу за спецификатором `h` или `hh`, обозначает минуты (`00-59`).

`mmm` — Обозначает месяц из `ShortMonthNames`.

`mmmm` — Обозначает месяц из `LongMonthNames`.

`yy` — Обозначает год в виде двух цифр (`00-99`).

`yyyy` — Обозначает год в виде четырех цифр (`0000-9999`).

`h` — Обозначает час (`0-23`). Используется 12-часовой формат, если в формате присутствуют спецификаторы `AM/PM`.

`hh` — Обозначает час, дополненный до двух цифр (`00-23`). Используется 12-часовой формат, если в формате присутствуют спецификаторы `AM/PM`.

`n` — Обозначает минуты (`0-59`).

`nn` — Обозначает минуты, дополненные до двух цифр (`00-59`).

`s` — Обозначает секунды (`0-59`).

`ss` — Обозначает секунды, дополненные до двух цифр (`00-59`).

`z` Обозначает миллисекунды (`0-999`).

`zzz` Обозначает миллисекунды, дополненные до трех цифр (`000-999`).

`t` Обозначает время в формате `ShortTimeFormat`.

`tt` Обозначает время в формате `LongTimeFormat`.

`am/pm` Обозначает `'am'` для полуночи и значений времени до полудня и `'pm'` для значений времени после полудня. Для указанного перед этим спецификатора `h` или `hh` используется 12-часовой формат (`0-11`). Регистр букв в точности совпадает с регистром спецификатора.

`a/p` Обозначает `'a'` для полуночи и значений времени до полудня и `'p'` для значений времени после полудня. Для указанного перед этим спецификатора `h` или `hh` задается 12-часовой формат (`0-11`). Регистр букв в точности совпадает с регистром спецификатора.

`ampm` Обозначает `TimeAMString` для полуночи и значений времени до полудня и `TimePMString` для полудня и значений времени после полудня. Указанный перед этим спецификатор `h` или `hh` использует 12-часовой формат (`0-11`).

`/` Обозначает символ `DateSeparator`.

`:` Обозначает символ `TimeSeparator`.

`'xxx'` `"xxx"` Обозначает точное содержимое строки в кавычках (без кавычек).

DateTimeToSystemTime, процедура

```
procedure DateTimeToSystemTime(DateTime: TDateTime; var SystemTime: TSystem-
Time);
```

Функция `DateTimeToSystemTime` преобразует значение `TDateTime` в системное время Windows.

DateTimeToTimeStamp, функция

```
function DateTimeToTimeStamp(DateTime: TDateTime): TTimeStamp;
```

Функция `DateTimeToTimeStamp` преобразует значение типа `TDateTime` в запись `TTimeStamp`.

DateToStr, функция

```
function DateToStr(Date: TDateTime): string;
```

Функция DateToStr форматирует дату из TDateTime в виде строки, используя ShortDateFormat.

DayOfWeek, функция

```
function DayOfWeek(Date: TDateTime): Integer;
```

Функция DayOfWeek возвращает день недели даты, где 1 обозначает воскресенье, 7 – субботу.

DecodeDate, процедура

```
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
```

Процедура DecodeDate извлекает год, месяц и день месяца из TDateTime. Значение месяца находится в диапазоне 1–12 (январь–декабрь).

DecodeTime, процедура

```
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
```

Процедура DecodeTime извлекает время в часах, минутах, секундах и миллисекундах из TDateTime.

EncodeDate, функция

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

Функция EncodeDate создает TDateTime из данных значений года, месяца и дня месяца. Время в результирующем значении равно нулю. Значение месяца должно находиться в диапазоне 1–12. Если какой-либо из аргументов имеет некорректное значение, генерируется исключительная ситуация EConvertError.

EncodeTime, функция

```
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
```

Функция EncodeTime создает TDateTime из данных компонентов времени. Дата в результирующем значении равна нулю. Если какой-либо из аргументов имеет некорректное значение, генерируется исключительная ситуация EConvertError.

EraNames, переменная

```
var EraNames: array[1..7] of string;
```

В EraNames хранится до семи имен эр, соответствующих эрам в EraYearOffsets, необходимых для форматирования дат в некоторых азиатских региональных настройках.

EraYearOffsets, переменная

```
var EraYearOffsets: array[1..7] of Integer;
```

В EraYearOffsets хранятся смещения годов для семи или менее эр. Каждая эра связана с соответствующим именем из EraNames.

FileDateToDateTime, функция

```
function FileDateToDateTime(FileDate: Integer): TDateTime;
```

Функция FileDateToDateTime преобразует дату и время файла в TDateTime. На рис. В-1 (ранее в этом примечании) приведен формат даты и времени файла.

FormatDateTime, функция

```
function FormatDateTime(const Format: string; DateTime: TDateTime):
  string;
```

Функция FormatDateTime форматирует дату и время в виде строки. Параметр Format подробно рассмотрен в описании функции DateTimeToString.

IncMonth, функция

```
function IncMonth(const Date: TDateTime; NumberOfMonths: Integer):
  TDateTime;
```

Функция IncMonth добавляет NumberOfMonths к месяцу в Date и возвращает полученное значение TDateTime. Если день месяца больше, чем количество дней в результирующем месяце, результат получается равным последнему дню месяца.

IsLeapYear, функция

```
function IsLeapYear(Year: Word): Boolean;
```

Функция IsLeapYear возвращает True, если год является високосным, или False – для невисокосного года.

LongDateFormat, переменная

```
var LongDateFormat: string;
```

LongDateFormat – это формат, используемый функцией DateTimeToString со спецификатором ddddd. Значение по умолчанию берется из региональных настроек Windows.

LongDayNames, переменная

```
var LongDayNames: array[1..7] of string;
```

В LongDayNames содержатся полные имена дней недели. Первый день в массиве соответствует воскресенью, последний – субботе. Значения по умолчанию берутся из региональных настроек Windows.

LongMonthNames, переменная

```
var LongMonthNames: array[1..12] of string;
```

В LongMonthNames содержатся полные имена месяцев (по порядку от января до декабря). Значения по умолчанию берутся из региональных настроек Windows.

LongTimeFormat, переменная

```
var LongTimeFormat: string;
```

LongTimeFormat – это формат, используемый функцией DateTimeToString со спецификаторами c и tt. Значение по умолчанию берется из региональных настроек Windows.

MonthDays, константа

```
type PDayTable = ^TDayTable;
type TDayTable = array[1..12] of Word;
const MonthDays: array [Boolean] of TDayTable;
```

В MonthDays хранится количество дней в месяце. Массив двумерный, первое измерение имеет тип Boolean: True для високосного года и False для невисокосных лет. Второе измерение – номер месяца в диапазоне 1-12 (от января до декабря).

MSecsPerDay, константа

```
const MSecsPerDay = SecsPerDay * 1000;
```

MSecsPerDay – это константа, облегчающая некоторые операции со временем, содержит количество миллисекунд в сутках.

MSecsToTimeStamp, функция

```
function MSecsToTimeStamp(MSecs: Comp): TTimeStamp;
```

Функция MSecsToTimeStamp преобразует количество миллисекунд в запись TTimeStamp.

Now, функция

```
function Now: TDateTime;
```

Функция Now возвращает текущую дату и время в локальном часовом поясе.

ReplaceDate, процедура

```
procedure ReplaceDate(var DateTime: TDateTime; const NewDate: TDateTime);
```

Процедура ReplaceDate заменяет дату в DateTime на NewDate, не влияя на компоненту времени.

ReplaceTime, процедура

```
procedure ReplaceTime(var DateTime: TDateTime; const NewTime: TDateTime);
```

Процедура ReplaceTime изменяет время в DateTime на NewTime, не влияя на компоненту даты.

SecsPerDay, константа

```
const SecsPerDay = 24 * 60 * 60;
```

SecsPerDay – это константа, облегчающая некоторые операции со временем, содержит количество секунд в сутках.

ShortDateFormat, переменная

```
var ShortDateFormat: string;
```

ShortDateFormat – это формат, используемый функцией DateTimeToString со спецификаторами c и ddddd. Значение по умолчанию берется из региональных настроек Windows.

ShortDayNames, переменная

```
var ShortDayNames: array[1..7] of string;
```

ShortDayNames содержит сокращенные имена дней недели. Первый день массива – это воскресенье, последний – суббота. Значения по умолчанию берутся из региональных настроек Windows.

ShortMonthNames, переменная

```
var ShortMonthNames: array[1..12] of string;
```

ShortMonthNames содержит сокращенные имена месяцев (по порядку, от января до декабря). Значения по умолчанию берутся из региональных настроек Windows.

ShortTimeFormat, переменная

```
var ShortTimeFormat: string;
```

ShortTimeFormat – это формат, используемый функцией DateTimeToString со спецификатором формата t.

StrToDate, функция

```
function StrToDate(const S: string): TDateTime;
```

Функция StrToDate выполняет разбор строки для извлечения даты. Строка должна содержать дату в формате, определенном в ShortDateFormat, компоненты даты должны разделяться символами DateSeparator. Если в строке содержится только два числа, они интерпретируются как месяц и день текущего года. Если формат строки некорректный, генерируется исключительная ситуация EConvertError. Если год указан только двумя цифрами, он разворачивается в четыре цифры в соответствии со значением переменной TwoDigitsYearCenturyWindow.

StrToDateTime, функция

```
function StrToDateTime(const S: string): TDateTime;
```

Функция StrToDateTime выполняет разбор строки для извлечения даты и времени. Строка может начинаться с даты в формате функции StrToDate. Если дата отсутствует, StrToDateTime использует текущую локальную дату. Дата и время разделяются одним и более пробелами. Время должно храниться в формате, требуемом функцией StrToTime. Если строка имеет некорректный формат, генерируется исключительная ситуация EConvertError.

StrToTime, функция

```
function StrToTime(const S: string): TDateTime;
```

Функция StrToTime выполняет разбор строки для извлечения времени. Строка должна содержать минимум одно число (часы). Минуты, секунды и миллисекунды могут отсутствовать. Компоненты времени должны разделяться символами TimeSeparator. Строка времени может начинаться или заканчиваться спецификатором времени до и после полудня (AM и PM), который может быть одним из следующих: TimeAMString, TimePMString, 'AM' или 'PM' (регистр не имеет значения). Если строка имеет некорректный формат, генерируется исключительная ситуация EConvertError.

SystemTimeToDateTime, функция

```
function SystemTimeToDateTime(const SystemTime: TSystemTime): TDateTime;
```

Функция SystemTimeToDateTime преобразует системное время в формате Windows в эквивалентное значение TDateTime.

Time, функция

```
function Time: TDateTime;
```

Функция Time возвращает текущее время в локальном часовом поясе с нулевым значением даты.

TimeAMString, переменная

```
var TimeAMString: string;
```

TimeAMString – это строка, которая содержит значения времени до полудня. Значение по умолчанию берется из региональных настроек Windows.

TimePMString, переменная

```
var TimePMString: string;
```

TimePMString – это строка, которая содержит значения времени после полудня. Значение по умолчанию берется из региональных настроек Windows.

TimeSeparator, переменная

```
var TimeSeparator: Char;
```

TimeSeparator – это символ, разделяющий компоненты времени. Значение по умолчанию берется из региональных настроек Windows.

TimeStampToDateTime, функция

```
function TimeStampToDateTime(const TimeStamp: TTimeStamp): TDateTime;
```

Функция TimeStampToDateTime преобразует запись TTimeStamp в значение TDateTime.

TimeStampToMSecs, функция

```
function TimeStampToMSecs(const TimeStamp: TTimeStamp): Comp;
```

Функция TimeStampToMSecs преобразует запись TimeStamp в количество миллисекунд. Результат имеет тип Comp, а не Int64, поэтому необходимо убедиться, что FPU установлен в режим полной расширенной точности. Подробности приведены в главе 5 в описании типа Comp.

TimeToStr, функция

```
function TimeToStr(Time: TDateTime): string;
```

Функция TimeToStr форматирует компоненту времени в TDateTime в виде строки в соответствии с кратким форматом времени.

TTimeStamp, тип

```
type
  TTimeStamp = record
    Time: Integer;
    Date: Integer;
  end;
```

TTimeStamp содержит по отдельности дату и время. В поле Time содержится количество миллисекунд, прошедших от полуночи, в поле Date – значение, на единицу большее, чем количество дней, прошедших с 1 января 0001 года. Хотя с типом TTimeStamp сложнее работать, чем с TDateTime, он имеет преимущество, т. к. время хранится точно и можно избежать проблем с округлением вещественных значений.

TwoDigitYearCenturyWindow, переменная

```
var TwoDigitYearCenturyWindow: Word = 50;
```

Преобразуя строку в дату, если значение года обозначено двумя цифрами, Delphi использует значение переменной TwoDigitYearCenturyWindow для того, чтобы определить, относится ли год к текущему, предыдущему или следующему веку. Окно (CenturyWindow) – это диапазон в сто лет вокруг текущего года, начиная с текущего года минус TwoDigitYearCenturyWindow. Функция StrToDate выбирает тот год из этого окна, который имеет последние две цифры, совпадающие с двумя цифрами года в исходной строке. Если TwoDigitsYearCenturyWindow равно нулю, год, обозначенный двумя цифрами, всегда будет равен году текущего века. В табл. В-11 приведены некоторые примеры.

Таблица B-11. Пример интерпретации года, состоящего из двух цифр

TwoDigitYearCenturyWindow	Текущий год	Год из двух цифр	Результат функции StrToDate
50	1998	47	2047
50	1998	48	1948
0	1998	97	1997
0	1998	98	1998
50	2001	50	2050
50	2001	51	1951

Локализация

Каждый программист Delphi, распространяющий собственные компоненты, должен разбираться в вопросах локализации, т. к. никогда не известно, кто и где будет работать с компонентом. Чтобы удовлетворить потребности разных пользователей, следует применять `resourcestring` для всех строковых констант, которые могут быть переведены на другой язык. В строках формата следует указывать индексные спецификаторы на случай необходимости перегруппировки аргументов в переведенной версии строки формата.

При форматировании даты, времени и чисел нужно учитывать локальные спецификации для `DecimalSeparator`, `CurrentFormat` и т. д. Пользователь может изменить эти настройки с помощью приложения «Язык и стандарты» из панели управления Windows. Если пользователь хочет разделять компоненты даты вопросительными знаками, ваша программа или компонент должны учитывать эту настройку (которая хранится в переменной `DateSeparator`) и никогда не применять жестко зашитый символ разделителя.

Строки могут содержать символы с пометками или даже быть многобайтовыми строками. Информацию о конкретных функциях форматирования можно найти в их описаниях в других разделах данного приложения. В этом разделе перечислены дополнительные типы, константы и подпрограммы, относящиеся к локализации.

Дополнительная информация о региональных настройках Windows приведена в документации Microsoft Platform SDK, в частности в разделе «National Language Support».

GetFormatSettings, процедура

```
procedure GetFormatSettings;
```

Процедура `GetFormatSettings` загружает локальные настройки Windows для переменных, перечисленных в табл. B-12. Графические

приложения автоматически вызывают эту процедуру, когда изменяются настройки формата (т. е. когда приложение получает сообщение Wm_WinIniChange).

Таблица В-12. Переменные, значения которых устанавливаются процедурой GetFormatSettings

Переменная	Описание
CurrencyDecimals	Количество десятичных знаков для данных типа Currency
CurrencyFormat	Формат положительных значений типа Currency
CurrencyString	Символ, обозначающий данные типа Currency
DateSeparator	Символ для разделения компонентов даты
DecimalSeparator	Символ, отделяющий целую часть от дробной
EraNames	Имена азиатских эр
EraYearOffsets	Годы азиатских эр
ListSeparator	Символ, разделяющий элементы списка
LongDateFormat	Длинный формат даты
LongDayNames	Полные названия дней недели
LongMonthNames	Полные названия месяцев
LongTimeFormat	Длинный формат времени
NegCurrFormat	Формат отрицательных значений типа Currency
ShortDateFormat	Краткий формат даты
ShortDayName	Краткие названия дней недели
ShortMonthNames	Краткие названия месяцев
ShortTimeFormat	Краткий формат времени
ThousandSeparator	Символ для разделения групп тысяч
TimeAMString	Строка, обозначающая время до полудня
TimePMString	Строка, обозначающая время после полудня
TimeSeparator	Символ для разделения компонентов времени

GetLocaleChar, функция

```
function GetLocaleChar(Locale, LocaleType: Integer;
    Default: Char): Char;
```

Функция GetLocaleChar ищет значение LocaleType в региональных настройках Windows с идентификатором Locale и возвращает первый символ результата. Если значение Locale или LocaleType некор-

ректно, возвращается Default. Процедура GetFormatSettings вызывает эту функцию для загрузки значений некоторых локализуемых переменных.

GetLocaleStr, функция

```
function GetLocaleStr(Locale, LocaleType: Integer;
    const Default: string): string;
```

GetLocaleStr ищет значение LocaleType в региональных настройках Windows с идентификатором Locale и возвращает результат в виде строки. Если значение Locale или LocaleType некорректно, возвращается Default. Процедура GetFormatSettings вызывает эту функцию для загрузки значений некоторых локализуемых переменных.

Languages, функция

```
function Languages: TLanguages;
```

Функция Languages возвращает объект TLanguages, содержащий информацию обо всех регионах, поддерживаемых текущей версией Windows. Дополнительную информацию можно найти в описании типа TLanguages далее в этом разделе.

LeadBytes, переменная

```
var LeadBytes: set of Char = [];
```

Переменная LeadBytes содержит множество символов, которые могут быть первыми байтами многобайтового символа. В региональных настройках для западных стран используется пустое множество, однако для дальневосточных стран следует установить LeadBytes в соответствии с местной кодовой страницей.

ListSeparator, переменная

```
var ListSeparator: Char;
```

ListSeparator – это символ, предназначенный для разделения элементов в списке, выводимом пользователю. Значение по умолчанию берется из региональных настроек Windows.

SysLocale, переменная

```
var SysLocale: TSysLocale;
```

В переменной SysLocale хранится информация о регионе по умолчанию. Подробности рассмотрены ниже в описании типа TSysLocale.

TLanguages, тип

```
type TLanguages = class
 public
    constructor Create;
    property Count: Integer read ...;
    function IndexOf(ID: LCID): Integer;
    property Name[Index: Integer]: string read ...;
```

```
    property NameFromLocaleID[ID: LCID]: string read ...;
    property NameFromLCID[const ID: string]: string read ...;
    property ID[Index: Integer]: string read ...;
    property LocaleID[Index: Integer]: LCID read ...;
    property Ext[Index: Integer]: string read ...;
  end;
```

В TLanguages хранится информация обо всех регионах, поддерживаемых данной локальной версией Windows. Модуль SysUtils создает экземпляр класса TLanguages, так что нет необходимости создавать его дополнительные экземпляры.

В Count содержится количество регионов. IndexOf возвращает индекс идентификатора региона в TLanguages. Другие свойства возвращают информацию о каждом из регионов:

Ext Возвращает краткое имя языка, которое используется Delphi для расширения имени файла DLL ресурсов.

ID Возвращает идентификатор региона, форматированный в виде шестнадцатеричной строки.

LocaleID

Возвращает идентификатор региона.

Name
NameFromLCID
NameFromLocaleID

Возвращает имя региона. Эти свойства отличаются только индексом, по которому выполняется поиск имени. Индекс может быть индексом из TLanguages, идентификатором региона или строкой идентификатора региона.

TSysLocale, mun

```
type
  TSysLocale = packed record
    DefaultLCID: LCID;
    PriLangID: LANGID;
    SubLangID: LANGID;
    FarEast: Boolean;
    MiddleEast: Boolean;
  end;
```

В TSysLocale содержится информация о региональных установках Windows, в частности, идентификатор региона (DefaultLCID), идентификатор первичного языка (PriLangID) и идентификатор вторичного языка (SubLangID). Если региону требуется поддержка многобайтовых символов, FarEast равняется True. Если Windows поддерживает иврит и арабский язык, MiddleEast равняется True.

Исполняемые модули

Все исполняемые модули (программы, библиотеки или пакеты) экспортируют две процедуры: Initialize и Finalize, которые вызывают разделы инициализации и завершения всех модулей, содержащихся в исполняемом модуле. Пакеты специально используют эту возможность для облегчения загрузки DLL пакета (инициализируя при этом свои модули) и выгрузки DLL (после завершения работы модулей). В модуле SysUtils содержится несколько констант, типов и подпрограмм, облегчающих получение информации об исполняемом модуле, а также для загрузки и выгрузки пакетов.

Хотя в именах присутствует слово «Package», большинство подпрограмм, описанных в этом разделе, работают с любыми видами исполняемых модулей.

FinalizePackage, процедура

```
procedure FinalizePackage(Module: HMODULE);
```

Процедура FinalizePackage вызывает секции завершения всех модулей загружаемого исполняемого модуля. Процедура UnloadPackage вызывает FinalizePackage, поэтому у программиста вряд ли возникнет необходимость самому вызвать эту процедуру. Если в исполняемом модуле нет секции завершения, FinalizePackage генерирует исключительную ситуацию EPackageError.

GetPackageDescription, функция

```
function GetPackageDescription(ModuleName: PChar): string;
```

Функция GetPackageDescription возвращает строку описания исполняемого модуля, путь к которому задан параметром ModuleName. Если файл не существует или не может быть загружен, возникает исключительная ситуация EPackageError. Если модуль не имеет описания, возвращается пустая строка. Описание задается директивой компилятора $D или $Description и сохраняется в ресурсе RCDATA с именем DESCRIPTION.

GetPackageInfo, процедура

```
type
  TNameType = (ntContainsUnit, ntRequiresPackage);
  TPackageInfoProc = procedure (const Name: string;
    NameType: TNameType; Flags: Byte; Param: Pointer);

procedure GetPackageInfo(Module: HMODULE; Param: Pointer;
    var Flags: Integer; InfoProc: TPackageInfoProc);
```

Процедура GetPackageInfo вызывает InfoProc для каждого модуля загруженного исполняемого модуля (который может быть пакетом, программой или DLL). Param – это простой указатель, непосредственно передаваемый InfoProc. Процедура устанавливает Flags рав-

ной флагам пакета, описанным в табл. B-13. Заметьте, что pfExeModule не применяется в Delphi 5: его значение должно быть равно $C0000000. Если исполняемый модуль не является модулем Delphi или C++ Builder, GetPackageInfo генерирует исключительную ситуацию.

Таблица B-13. Флаги пакетов

Литерал	Значение	Описание
PfNeverBuild	$00000001	Перекомпилируется только явно
PfDesignOnly	$00000002	Пакет только для разработки
pfRunOnly	$00000004	Пакет только времени выполнения
pfIgnoreDupUnits	$00000008	Не проверять на дублирование модулей
pfModuleTypeMask	$C0000000	Маска типа исполняемого модуля
pfExeModule	$00000000	(Литерал не применяется в Delphi 5)
pfPackageModule	$40000000	DLL пакета
pfLibraryModule	$80000000	Другая DLL
	$C0000000	Исполняемая программа
pfProducerMask	$0C000000	Маска компилятора, с пмощью которого выполнялась сборка пакета
pfV3Produced	$00000000	Delphi 3 или C++ Builder 3
pfProducerUndefined	$04000000	Неизвестный компилятор
pfBCB4Produced	$08000000	C++ Builder 4 или более поздняя версия
pfDelphi4Produced	$0C000000	Delphi 4 или более поздняя версия

Аргументы, которые Delphi передает в InitProc, – это имя модуля или пакета, тип (модуль или требуемый пакет), некоторые флаги и параметр Param, переданный ранее процедуре GetPackageInfo. Флаги Flags описывают дополнительную информацию о модуле, как показано в табл. B-14.

Таблица B-14. Флаги модуля

Литерал	Значение	Описание
ufMainUnit	$01	Основной модуль пакета или проекта
ufPackageUnit	$02	Исходный файл пакета (*dpk*-файл)
ufWeakUnit	$04	«Слабо связанный» модуль
ufOrgWeakUnit	$08	«Слабо связанный» модуль и исходном пакете[a]
ufImplicitUnit	$10	Неявно связанный модуль

[a] Как следует из модуля SysUtils, флаг ufOrgWeakUnit устанавливается только для модуля в пакете, в котором он был определен изначально. Практически он устанавливается всегда, когда применяется «слабо связанный» модуль.

InitializePackage, процедура

```
procedure InitializePackage(Module: HMODULE);
```

Процедура InitializePackage вызывает секции инициализации для всех модулей загруженного исполняемого модуля. Она, в свою очередь, автоматически вызывается функцией LoadPackage. Если исполняемый модуль не имеет секции инициализации, генерируется исключительная ситуация.

LoadPackage, функция

```
function LoadPackage(const Name: string): HMODULE;
```

Функция LoadPackage загружает DLL пакета и инициализирует его, запуская секции инициализации всех модулей пакета. Она возвращает дескриптор модуля загруженной DLL. Если Name не соответствует имени пакета, LoadPackage генерирует исключительную ситуацию EPackageError.

Чтобы использовать пакет, обычно требуется организовать регистрацию пакета при его загрузке. IDE Delphi делает это с помощью объявления процедуры Register в каждом модуле. IDE вызывает процедуру Register для каждого модуля после загрузки пакета. Другой подход состоит в создании упреждающего пакета и вызове для секции инициализации каждого модуля процедуры регистрации, определенной в модуле из другого пакета. Приложение также загружает регистрирующий пакет для получения списка зарегистрированных модулей. В примере B-5 показан один из способов создания такого пакета.

Пример B-5. Один из способов динамической загрузки модуля

```
unit Registration;
// Поместите этот модуль в отдельный пакет. Приложение добавляет
// этот пакет в список своих пакетов времени выполнения. Все
// динамически загружаемые пакеты добавляют этот пакет в свои
// списки в секции required.
interface

// Можно и другую интересную информацию, кроме имени
// модуля.
procedure RegisterUnit(const Name: string);
procedure UnregisterUnit(const Name: string);

implementation

uses Windows, SysUtils, Classes;
```

```
var
  UnitList: TStringList;              // список записей о модулях
  CritSect: TRtlCriticalSection;      // защита для многопоточных
                                      // приложений

procedure RegisterUnit(const Name: string);
begin
  EnterCriticalSection(CritSect);
  try
    UnitList.Add(Name);
  finally
    LeaveCriticalSection(CritSect);
  end;
end;

procedure UnregisterUnit(const Name: string);
begin
  EnterCriticalSection(CritSect);
  try
    UnitList.Delete(UnitList.IndexOf(Name));
  finally
    LeaveCriticalSection(CritSect);
  end;
end;

initialization
  UnitList := TStringList.Create;
  InitializeCriticalSection(CritSect);
finalization
  DeleteCriticalSection(CritSect);
  FreeAndNil(UnitList);
end.

// Пример использования модуля Registration.
unit Example;

interface
  ...
implementation
  uses Registration;
  ...
initialization
  RegisterUnit('Example');
finalization
  UnregisterUnit('Example');
end.
```

UnloadPackage, процедура

```
procedure UnloadPackage(Module: HMODULE);
```

Процедура UnloadPackage завершает все модули пакета и затем выгружает DLL. Программист сам отвечает за то, чтобы в приложении не было «висящих» указателей, например, ссылок на классы, события или методы из выгружаемого пакета.

Windows

В этом разделе описаны переменные и функции для работы с Windows API.

GetDiskFreeSpaceEx, переменная

```
var GetDiskFreeSpaceEx: function (Directory: PChar; var FreeAvailable,    To-
talSpace: TLargeInteger; TotalFree: PLargeInteger): Bool stdcall;
```

Первая версия Windows 95 не поддерживает функцию API GetDiskFreeSpaceEx, однако версия OSR2 ее поддерживает, как и Windows 98, и Windows NT. Delphi скрывает эту разницу с помощью переменной GetDiskFreeSpaceEx, указывающей на функцию Windows API, если она существует, и на функцию Delphi для Windows 95a. Обратитесь к описанию функции DiskFree ранее в этом приложении, которая является более простым способом определения свободного места на диске.

SafeLoadLibrary, функция

```
function SafeLoadLibrary(const Filename: string;
    ErrorMode: UINT = SEM_NoOpenFileErrorBox): HMODULE;
```

Некоторые DLL изменяют состояние FPU, что может повлиять на точность вычисления типов Comp и Currency в Delphi. Функция SafeLoadLibrary полностью аналогична функции LoadLibrary из Windows API, кроме того, что обеспечивает сохранение управляющего состояния FPU. Необязательный второй параметр передается в SetErrorMode, что позволяет контролировать, каким образом ошибки открытия файлов и другие подобные ошибки сообщаются пользователю.

TMultiReadExclusiveWriteSynchronizer, тип

```
type
  TMultiReadExclusiveWriteSynchronizer = class
  public
    constructor Create;
    destructor Destroy; override;
    procedure BeginRead;
    procedure EndRead;
    procedure BeginWrite;
    procedure EndWrite;
  end;
```

Класс TMultiReadExclusiveWriteSynchronizer используется, когда нескольким потокам нужно читать разделяемый ресурс, не изменяя

его. В критических секциях только одному потоку разрешен доступ к разделяемому ресурсу, даже если поток не изменяет его. Кроме того, TMultiReadExclusiveWriteSynchronizer улучшает производительность, разрешая доступ на чтение нескольким потокам одновременно.

Каждый читающий поток вызывает BeginRead для получения доступа на чтение и EndRead, когда чтение завершается. Если потоку требуется модифицировать ресурс, он вызывает BeginWrite, который ожидает, пока все потоки-читатели не вызовут EndRead, и затем получает монопольный доступ на запись. EndWrite вызывается для снятия блокировки. Если поток заблокировал синхронизатор, чтобы осуществить запись, все вызовы BeginRead в других потоках блокируются до завершения записи потоком.

Поток должен уступить доступ на чтение перед попыткой получения доступа на запись, иначе может возникнуть взаимная блокировка, при которой два потока «зависают» на вызовах метода BeginWrite, когда один поток ожидает вызова другим потоком метода EndRead.

Win32BuildNumber, переменная

```
var Win32BuildNumber: Integer;
```

В Win32BuildNumber содержится номер выпуска (build number) операционной системы. В Windows 95 и 98 младшее слово – это номер выпуска, в старшем хранятся старший и младший номера версии (те же значения, что и Win32MajorVersion и Win32MinorVersion).

Win32CSDVersion, переменная

```
var Win32CSDVersion: string;
```

В Win32CSDVersion содержится дополнительная информация об операционной системе, например, номер сервисного пакета NT. Фактический текст зависит от операционной системы.

Win32MajorVersion, переменная

```
var Win32MajorVersion: Integer;
```

В Win32MajorVersion содержится старшая часть номера версии операционной системы, например, 4 для Windows NT 4.0 или Windows 95.

Win32MinorVersion, переменная

```
var Win32MinorVersion: Integer;
```

В Win32MinorVersion содержится младшая часть номера версии операционной системы, например, 0 для Windows NT 4.0 или Windows 95 либо 10 для Windows 98.

Win32Platform, переменная

```
var Win32Platform: Integer;
```

В Win32Platform содержится идентификатор платформы, который может быть равен одному из значений из табл. В-15 (литералы объявлены в модуле Windows).

Таблица В-15. Значения Win32Platform

Литерал	Описание
Ver_Platform_Win32s	Win32s для Windows 3.1
Ver_Platform_Win32_NT	Windows NT или Windows 2000
Ver_Platform_Win32_Windows	Windows 95 или Windows 98

Для идентификации операционной системы Windows можно использовать следующий код:

```
function GetOS: string;
begin
  case Win32Platform of
  Ver_Platform_Win32_NT:
    Result := Format('Windows NT %d.%d (Build %d %s)',
      [Win32MajorVersion, Win32MinorVersion,
       Win32BuildNumber, Win32CSDVersion]);
  Ver_Platform_Win32_Windows:
    if Win32MinorVersion = 0 then
      Result := Format('Windows 95 %s (%d.%2.2d.%d)',
        [Win32CSDVersion, Win32MajorVersion, Win32MinorVersion,
         Win32BuildNumber and $FFFF])
    else
      Result := Format('Windows 98 %s (%d.%2.2d.%d)',
        [Win32CSDVersion, Win32MajorVersion, Win32MinorVersion,
        Win32BuildNumber and $FFFF]);
   Ver_Platform_Win32s:
     Result := 'Win32s';
   else
     Result := Format('Windows %d.%2.2d.%d %s',
       [Win32MajorVersion, Win32MinorVersion,
        Win32BuildNumber, Win32CSDVersion])
   end;
end;
```

Прочее

В этом разделе перечислено все, что не подходит под другие категории.

AllocMem, функция

```
function AllocMem(Size: Cardinal): Pointer;
```

Функция AllocMem полностью аналогична GetMem, за исключением того, что она инициализирует выделенную память нулями. Если AllocMem применяется для выделения памяти записи, содержащей длинные строки, интерфейсы или Variant, Initialize вызывать не нужно. Память освобождается процедурой FreeMem.

CompareMem, функция

```
function CompareMem(P1, P2: Pointer; Length: Integer): Boolean;
```

Функция CompareMem сравнивает Length байтов, указанных P1 и P2, на равенство. Функция возвращает True, если две области памяти содержат идентичное содержимое, и False, если хотя бы один байт отличается.

FreeAndNil, процедура

```
procedure FreeAndNil(var Obj);
```

Когда ссылка на объект хранится в переменной или в поле, передавайте ссылку на объект процедуре FreeAndNil вместо вызова метода Free. FreeAndNil вызывает Free и присваивает nil переменной или полю.

Процедура FreeAndNil особенно полезна в многопоточных и реентерабельных приложениях, т. к. гарантирует, что переменная установлена в nil до освобождения объекта. Другими словами, в переменной никогда не хранится неверная ссылка.

Int64Rec, тип

```
type
  Int64Rec = packed record
    Lo, Hi: LongWord;
  end;
```

Int64Rec облегчает доступ к старшему и младшему длинным словам Int64 или другого 64-битового типа (такого как Double).

LoadStr, функция

```
function LoadStr(Ident: Integer): string;
```

Функция LoadStr загружает строковый ресурс (или пустую строку, если ресурс с идентификатором Ident не существует). Функция существует только для обратной совместимости. В новом коде следует использовать объявления resourcestring.

LongRec, тип

```
type
  LongRec = packed record
    Lo, Hi: Word;
  end;
```

LongRec облегчает доступ к старшему и младшему словам LongWord или другого 32-битового типа (такого как Single).

PByteArray, тип

```
type PByteArray = ^TByteArray;
type TByteArray = array[0..32767] of Byte;
```

Чаще всего PByteArray применяется в преобразованиях типа, что позволяет рассматривать область памяти как массив байт.

PWordArray, тип

```
type PWordArray = ^TWordArray;
type TWordArray = array[0..16383] of Word;
```

Чаще всего PWordArray используется в преобразованиях типа, что позволяет рассматривать область памяти как массив слов.

Supports, функция

```
function Supports(const Instance: IUnknown; const Intf: TGUID;
    out Inst): Boolean; overload;
function Supports(Instance: TObject; const Intf: TGUID;
    out Inst): Boolean; overload;
```

Функция Supports – это еще один способ преобразования ссылки на объект или интерфейс в другой GUID или тип интерфейса. Во многих случаях Supports оказывается более удобной в использовании, чем QueryInterface. Если Instance не равен nil, и если он поддерживает интерфейс Intf, функция присваивает переменной Inst требуемый интерфейс и возвращает True. В противном случае Inst присваивается nil, а функция возвращает False.

TIntegerSet, тип

```
type TIntegerSet = set of 0..SizeOf(Integer) * 8 - 1;
```

С множествами обычно проще работать, чем с битовыми масками. Тип TIntegerSet – это множество целых от 0 до 31. Таким образом, для того чтобы проверить, установлены ли два старших бита в целом значении, попробуйте использовать следующий способ:

```
if ([30,31] * TIntegerSet(IntValue)) = [30,31] then
  TwoMostSignificantBitsAreSet(IntValue);

// То же самое с применением битовых масок читается хуже:
if (IntValue and $C0000000) = $C0000000 then
  TwoMostSignificantBitsAreSet(IntValue);
```

TMethod, тип

```
type
  TMethod = record
    Code, Data: Pointer;
  end;
```

Все ссылки на методы хранятся в формате, описываемым типом TMethod. Можно преобразовать метод к типу TMethod и обратно, так же как и свойство-событие.

TProcedure, тип

```
type TProcedure = procedure;
```

TProcedure – это простой процедурный тип для процедуры без параметров.

TSysCharSet, тип

```
type TSysCharSet = set of Char;
```

TSysCharSet – это множество всех символов ANSI.

TWordRec, тип

```
type
  WordRec = packed record
    Lo, Hi: Byte;
  end;
```

WordRec облегчает доступ к старшему и младшему байту Word или другого 16-битового типа (такого, как SmallInt).

Алфавитный указатель

Symbols

$A 503
$Align 503
$AppType 15, 504
$Assertions 504
$B 505
$BootEval 506
$C 507
$D 507
$DenyPackageUnit 509
$Description 509
$DebugInfo 507
$Define 507
$DefinitionInfo 508
$DesignOnly 510
$E 510
$Else 511
$EndIf 511
$ExtendedSyntax 512
$ExternalSum 512
$G 513
$H 513
$Hints 513
$HppEmit 514
$I 515
$IfNDef 516
$IfOpt 517
$IfDef 515
$ImageSize 518
$ImplicitBuild 518
$ImportedData 519
$Include 519
$IOChecks 520
$J 35, 521
$L 521
$Link 521
$LongStrings 522
$LocalSymbols 522
$M 523
$MaxStackSize 523
$MinStackSize 525
$MinEnumSize 524
$NoDefine 525
$NoInclude 526
$O 526
$ObjExportAll 526
$OpenStrings 527
$Optimization 527
$OverflowChecks 529
$P 530
$Q 530
$R 530
$RangeChecks 530
$RealCompatibility 531
$ReferenceInfo 532
$Resource 532
$RunOnly 533
$SafeDivide 534
$StackFrames 534
$T 31, 535
$TypeAddress 31
$TypedAddress 535
$TypeInfo 536
$U 536
$Undef 537
$V 537
$VarStringChecks 537
$W 538
$Warnings 538
$WeakPackageUnit 539
$WriteableConst 35, 540
$X 541
$Z 541
_AddRef 73
_Release 73
реализация 12
элементарная операция 124

A

Деструктор 61
Вариантные типы 30
Базовый класс 52
Вещественные типы 20
abstract 56
AbstractErrorProc 39
Взаимная блокировка 127

взаимное исключение 138
AdjustLineBreaks 580
Библиотека 16
Библиотека типов COM 31
AfterConstruction 437
AllocMem 626
AnsiCompareFileName 580
AnsiCompareStr 581
AnsiCompareText 581
AnsiExtractQuotedStr 585
AnsiLastChar 581
AnsiLowerCase 581
AnsiLowerCaseFileName 581
AnsiPos 582
AnsiQuotedStr 582
AnsiSameStr 582
AnsiSameText 582
AnsiStrComp 586
AnsiStrIComp 586
AnsiStrLastChar 586
AnsiStrLComp 586
AnsiStrLIComp 586
AnsiStrLower 586
AnsiStrPos 587
AnsiStrRScan 587
AnsiStrScan 587
AnsiStrUpper 587
AnsiUpperCase 582
AnsiUpperCaseFileName 582
AppendStr 590
Append 41
array of const 25
as 17
AssignStr 590
AssertErrorProc 39
Assign 41
AssignFile 41
Automated 114
вытеснение 122
automated 64

B

BeforeDestruction 437
BeginThread 136
BlockRead 41
BlockWrite 41
Boolean 29
Byte 20
ByteToCharIndex 583
ByteToCharLen 583
ByteType 583
ByteBool 30

C

завершение 112
Cardinal 20
завершение 13
ChangeFileExt 577
CharToByteIndex 583
CharToByteLen 583
class 46
ClassInfo 435
ClassNameIs 435
ClassParent 435
ClassType 437
CleanupInstance 437
Close 41
CloseFile 41
CompareMem 627
CompareStr 590
CompareText 590
ComponentSate 100
constructor 58
console application 15
Create 435
CreateDir 575
Create 51
CurrencyDecimals 595
CurrencyFormat 595
CurrencyString 595
CurrToStr 596
CurrToStrF 596
Currency 21

D

Date 606
DateDelta 606
DateSeparator 606
DateTimeToFileDate 607
DateTimeToStr 607
DateTimeToString 607
DateTimeToSystemTime 609
DateTimeToTimeStamp 609
DateToStr 609
DayOfWeek 610
DecimalSeparator 596
DecodeDate 610

DecodeTime 610
default 100
DefaultHandler 437
Delphi для Linux 4
Destroy 435
destructor 61
deadlock 127
default 68
dfm 11
DiskFree 576
DiskSize 576
Dispatch 438
DisposeStr 590
Dispose 91
DLL 16
    базовый адрес 17
    динамическая память 17
    объекты 17
DMT 56
Double 20
dynamic 56

E

Критическая секция 124
Класс 46, 50
    абстрактный 57
    ссылка 47
Класс-предок 52
Инициализация 112
Инициализация 13
компонент 99
Конструктор 58
конструктор 51
Константа
    типизированная 34
Конвейер 138
Компоненты 83
консольное приложение 15
информация о типе времени выполнения 92
Логические типы 29
Локальные данные потока 136
Интерфейс 72
интерфейс 12
EmptyStr 591
EncodeDate 610
EncodeTime 610
EndThread 136
Исключительная ситуация 126
исключительная ситуация 36
Eof 41
EraNames 610
EraYearOffsets 610
Erase 42
ErrorProc 38, 39
ErrorAddr 39
event 137
ExcludeTrailingBackSlash 577
Exp 239
ExpandFileName 577
ExpandUNCFileName 577
Export 239
Exports 239
Extended 241
External 244
ExtractFileDir 577
ExtractFileDrive 577
ExtractFileExt 577
ExtractFileName 578
ExtractFilePath 578
ExtractRelativePath 578
ExtractShortPathName 578
ExceptClsProc 39
ExceptProc 37, 40
Exception 36, 126
ExceptionClass 39
ExceptObjProc 39
exports 16
Extended 20

F

Far 245
FieldAddress 438
File 245
FileClose 573
FileDateToDateTime 611
FileMode 247
FilePos 248
FileSearch 573
FileSetAttr 573
FileSize 248
FillChar 249
Finalization 250
finalization 112
Finalize 252
FinalizePackage 620
Finally 253
FindClassHInstance 253

FindCmdLineSwitch 584
FindFirst 574
FindHinstance 254
FindNext 575
FileSize 42
finalization 13
FloatToDecimal 596
FloatToStr 597
FloatToStrF 597
FloatToText 598
FloatToTextFmt 598
FmtLoadStr 600
FmtStr 600
form 11
Format 600
FormatBuf 602
FormatCurr 602
FormatDateTime 611
FormatFloat 602
Forward 256
Frac 258
Free 61, 438
FreeAndNil 52, 627
FreeInstance 438
FreeInstance 61
FreeMem 258
FreeMemory 259
Function 259
Futures 148

G

GetCurrentDir 575
GetDir 261
GetDiskFreeSpaceEx 624
GetEnumName 105
GetEnumProp 105
GetEnumValue 105
GetFloatProp 105
GetFormatSettings 616
GetHeapStatus 262
GetInt64Prop 105
GetInterface 438
GetInterfaceEntry 435
GetInterfaceTable 435
GetLocaleChar 617
GetLocaleStr 618
GetMem 263
GetMemory 264
GetMemoryManager 264
GetMethodProp 106
GetObjectProp 106
GetObjectPropClass 106
GetOrdProp 106
GetPackageDescription 620
GetPackageInfo 620
GetPackageInfoTable 265
GetPropInfo 106
GetPropInfos 107
GetPropList 104, 107
GetPropValue 107
GetSetProp 107
GetStrProp 107
GetTypeData 101, 108
GetVariantProp 108
Goto 265
GUID 76

H

Halt 267
HeapAllocFlags 267
HexDisplayPrefix 602
Hi 268
High 24, 268
HInstance 269
HPrevInst 270
HResult 271

I

преобразование в последовательную форму 130
Программа 14
Процесс 137
процедура 42
Наследование 52
Массив 23
    Динамический 23
    Открытые вариантного типа 25
    Открытый 24
Метод
    динамический 111
Объект 46, 51
Объект-свойство 70
Объявления типов 34
Перегрузка 43
Переменная
    потоковая 35
Параметр

по умолчанию 43
Пакет 11, 18
для разработки 18
для выполнения 18
создание 19
Менеджер памяти 83
метакласс 47, 51
Метод 54
абстрактный 56
виртуальный 55
динамический 56
перегружаемый 57
статический 55
Метод класса 55
методами 46
IDispatch 271
If 272
Многопоточное программирование 122
Модуль 10
Поле 46, 54
Поток 121
Implementation 273
Implements 273
implements 73
implementation 12
In 275
Inc 276
Include 277
IncludeTrailingBackSlash 580
IncMonth 611
Index 277
index 101
Inherited 278
InheritsFrom 436
Initialization 280
initialization 112
Initialize 281
InitializePackage 622
InitInstance 436
InitProc 282
initialization 13
Insert 284
InstanceSize 436
Int 285
Int64 20, 285
Int64Rec 627
Integer 286
Interface 287
IntToHex 603
IntToStr 603
Interface 72
interface 12
Integer 20
IOResult 289
Is 291
is 17, 53
IsConsole 291
IsDelimiter 584
IsLeapYear 611
IsLibrary 292
IsMemoryManagerSet 293
IsMultiThread 293
IsPathDelimiter 584
IsPublishedProp 108
IsStoredProp 108
IsValidIdent 584
IsMultThread 136
IsMultiThread 122
Мьютекс 138, 139
IUnknown 73, 294

J

JITEnable 295

L

Label 295
Languages 618
LastDelimiter 584
LeadBytes 618
Length 296
LibModuleList 297
Library 297
library 16
ListSeparator 618
Linux 4
Ln 299
Lo 299
LoadPackage 622
LoadResouceModule 300
LoadResString 300
LoadStr 627
long string 27
LongBool 301
LongBool 30
LongDateFormat 611
LongDayNames 611
LongInt 301

LongMonthNames 611
LongRec 627
LongTimeFormat 612
LongWord 20, 302
LongInt 20
Low 24, 302
LowerCase 591

M

MainInstance 303
MainThreadID 304
MaxInt 304
MaxLongInt 304
Message 305
message 81, 82, 137
MethodAddress 436
MethodName 437
memory-mapped file 138
MkDir 306
Mod 306
ModuleIsCpp 307
ModuleIsLib 307
ModuleIsPackage 308
ModuleUnloadList 308
MonthDays 612
Move 309
MSecsPerDay 612
MSecsToTimeStamp 612
mutex 138, 139

N

Счетчик ссылок 73
Счетчики ссылок 78
Свойство 65
    индексированное 67
    массив 67
свойство 46, 98, 103
Name 309
NaN 21
Семафор 138
Near 310
NegCurrFormat 603
Синхронизация потоков 123
Символьная константа 28
New 91, 310
NewInstance 437
NewStr 591
Nil 311
Событие 137
Сообщение 81, 137
Ссылка
    на объект 47
Ссылка на класс 50
Строка 27
    длинная 27
    Короткая 27
    Многобайтовая 29
    с завершающим нулем 27
    Широкая 27
nodedefault 100
Now 612
NullStr 591

O

Уровни доступа 63
Таблица виртуальных методов 92
Таблица динамических методов 111
таблица виртуальных методов 50, 53
таблица динамических методов класса 56
Файл в памяти 138
Целые типы 20
Object 315
object 90
Object Pascal 10
Odd 317
Указатель 31
    процедурный 33
    на функции и методы 33
Типы данных 19
Of 317
Форма 11
OleAuto 114
OleStrToString 317
OleStrToStrVar 318
OleVariant 318
OleVariant 30
On 319
on 36
Функция 42
OpenString 320
Or 321
Ord 322
Фьючерс 148
Out 322
Output 323
Overload 324

overload 43, 57
Override 325
override 56

P

Package 325
    design-time 18
package 11
PackageUnitEntry 327
PackageInfo 326
PackageInfoTable 326
Packed 327
PAnsiChar 329
PAnsiString 329
Pascal 331
ParamStr 330
ParamCount 330
PByteArray 628
PChar 28, 332
PCurrency 332
PDateTime 332
PExtended 333
PGUID 333
PInt64 334
PInterfaceEntry 334
PInterfaceTable 334
pipe 138
PLibModule 334
PMemoryManager 335
PModuleUnloadRec 335
Pos 337
Pointer 335
POleVariant 336, 338
Pred 338
Private 339
private 63
Program 342
program 14
PropIsType 108
PropType 108
Procedure 339
Property 343
property 65
Protected 345
protected 64
PString 346
Ptr 346
PTypeData 101
PTypeInfo 101
Published 94, 98, 103
Public 347
public 64
Published 348
published 54, 64
PUniEntryTable 349
PVarData 349
PVarRec 350
PVarArray 349
PVariant 350
PWideChar 350
PWideString 351
PWordArray 628

Q

QuotedStr 591
QueryInterface 73, 74

R

RandSeed 356
Random 354
Randomize 355
Raise 351
raise 39
RaiseList 353
RemoveDir 576
RenameFile 575
ReplaceTime 612
ReplaceDate 612
Requires 370
Rewrite 42, 375
Read 42, 356, 357
ReadLn 42, 359
Readonly 360
Real 20, 360
Real48 361
ReallocMem 361
ReallocMemory 363
Record 363
RefAnyProc 365
Register, директива 366
Register, процедура 366
RegisterModule 367
Reintroduce 368
RemoveModuleUnloadProc 368
Rename 42, 369
Repeat 369
Reset 42, 371

Resident 372
Resourcestring 372
Result 44, 374
RmDir 376
Round 376
RTTI 34, 92
Runtime Type Information 92
RunError 378

S

SafeCallException 438
SafeLoadLibrary 624
SameText 591
SafeCall 379
SafeCallErrorProc 40, 380
SecsPerDay 612
Self 54, 383
Set 384
Set8087CW 386
SetCurrentDir 576
SetEnumProp 109
SetFloatProp 109
SetInt64Prop 109
SetMethodProp 109
SetObjectProp 109
SetOrdProp 110
SetPropValue 110
SetSetProp 110
SetString 391
SetStrProp 110
SetVariantProp 110
SetLength 388
SetMemoryManager 85, 389
SetRaiseList 391
SetTextBuf 392
Seek 42, 381
SeekEof 382
SeekEoln 382
semaphore 138
serialize 130
SetInOutRes 387
ShareMem 17, 88
Shl 393
short string 27
ShortDateFormat 613
ShortDayNames 613
ShortMonthNames 613
ShortString 394
ShortTimeFormat 613
ShortInt 20, 394
Shr 395
Sin 396
Single 20, 397
SizeOf 399
Slice 400
SmallInt 20, 400
Sqr 401
Sqrt 401
StdCall 402
stored 98, 108
Stored 403
stored 68
Str 404
StrAlloc 587
StrBufSize 587
StrByteType 587
StrCat 588
StrComp 591
StrCopy 588
StrDispose 588
StrECopy 588
StrEnd 588
StrFmt 604
StrIComp 592
String 406
StringOfChar 406
StringReplace 584
StringToOleStr 406
StringToWideChar 407
StrLCat 588
StrLComp 592
StrLCopy 589
StrLen 589
StrLFmt 604
StrLIComp 592
StrLower 592
StrMove 589
StrNew 589
StrPas 592
StrPCopy 589
StrPLCopy 589
StrPos 592
StrRScan 593
StrScan 593
StrToCurr 604
StrToDate 613
StrToDateTime 613
StrToFloat 604

StrToInt 605
StrToInt64 604
StrToInt64Def 605
StrToIntDef 605
StrToTime 614
StrUpper 593
Succ 408
Supports 628
Swap 408
SysFreeMem 410
SysGetMem 412
SysInit 14
SysLocale 618
SysReallocMem 413
System 14
SystemTimeToDateTime 614
SysUtils 39

T

TClass 414
TComponent 99
TComponent 83
TDateTime 415
TEnumModuleFuncLW 417
Test8086 417
Test8087 417
TestFDIV 418
Text 418
TextFile 419
TextToFloat 605
TGUID 423
THeapStatus 424
Then 426
ThousandSeparator 606
thread local storage 426
ThreadVar 426
thread 121
thread local storage 35, 136
Time 614
TimeAMString 614
TimePMString 614
TimeSeparator 614
TimeStampToDateTime 614
TimeStampToMSecs 615
TimeToStr 615
TIntegerSet 628
TInterfacedClass 428
TInterfacedObject 428
TInterfaceEntry 429
TInterfaceTable 430
TInterfacedObject 73
TLanguages 618
TLibModule 430
TLS 35, 426
TlsIndex 431
TlsLast 432
TMemoryManager 432
TMethod 628
TModuleUnloadProcLW 432
TModuleUnloadRec 433
TMultiReadExclusiveWriteSynchronizer 624
TMultiReadExclusiveWriteSynchronizer 126
To 433
TObject 434
TProcedure 629
TResStringRec 439
Trim 593
TrimLeft 594
TrimRight 594
Trunc 439
Truncate 440
Try 440
try-except 36
try-finally 36
TSysCharSet 629
TSysLocale 619
TThreadFunc 442
TThread 130
TTimeStamp 615
TTypeInfo 101
TVarArray 443
TVarArrayBound 444
TVarData 444
TVarRec 446
TVarData 30
TVarRec 25
TwoDigitYearCenturyWindow 615
TWordRec 629
Type 447
type 34
type library 31
TypeInfo 101, 449
TypInfo 101

U

Unassigned 450

UniqueString 450
Unit 10, 451
UnitEntryTable 452
Unicode 29
UnloadPackage 623
UnregisterModule 452
Until 452
UpCase 453
UpperCase 595
use 12
Uses 454

V

Val 455
Var 456
VarArrayCreate 457
VarArrayDimCount 458
VarArrayHighBound 458
VarArrayLock 459
VarArrayLowBound 460
VarArrayOf 461
VarArrayRedim 461
VarArrayRef 462
VarArrayUnlock 463
VarAsType 464
VarCast 464
VarClear 465
VarCopy 465
VarDispProc 466
VarFromDateTime 467
Variant 467
VarIsArray 469
VarIsEmpty 470
VarIsNull 470
VarToDateTime 471
VarToStr 471
VarType 472
Virtual 473
virtual 56
VMT 50, 53, 92

W

While 474
WideChar 474
WideCharLenToString 475
WideCharLenToStrVar 475
WideCharToString 475
WideCharToStrVar 476
WideString 476
Win32BuildNumber 625
Win32CSDVersion 625
Win32MajorVersion 625
Win32MinorVersion 625
Win32Platform 626
With 477
wide string 27
Word 20, 479
WordBool 479
WordBool 30
WrapText 585
Write, директива 479
Write, процедура 481
WriteLn 482
WriteOnly 483
Write 42
WriteLn 42

X

Xor 483

Z

zero-terminated string 27

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-019-7, название «Delphi. Справочник» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.