Borland Pascal with Objects 7.0
Руководство пользователя

(свободный перевод документации фирмы Borland International)

Объектно-ориентированное программирование

Содержание

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ.....3

ОБЪЕКТЫ.....4

НАСЛЕДОВАНИЕ.....5

ОБЪЕКТЫ: НАСЛЕДУЮЩИЕ ЗАПИСИ.....6

Экземпляры объектных типов.....8
Поля объектов.....8
Хорошая и плохая техника программирования.....9

МЕТОДЫ.....10

Совмещенные код и данные.....11
Определение методов.....12
Область действия метода и параметр Self.....13
Поля данных объекта и формальные параметры метода.....14
Объекты, экспортируемые модулями.....15
Секция private.....16
Программирование в "действительном залоге".....16

ИНКАПСУЛЯЦИЯ.....18

Методы: никакого ухудшения.....19
Расширяющиеся объекты.....19
Наследование статических методов.....23

ВИРТУАЛЬНЫЕ МЕТОДЫ И ПОЛИМОРФИЗМ.....26

Раннее связывание против позднего связывания.....27
Совместимость типов объектов.....28

ПОЛИМОРФИЧЕСКИЕ ОБЪЕКТЫ.....30

Виртуальные методы.....31
Проверка диапазонов при вызове виртуальных методов.....34
Расширяемость объекта.....35
Статические методы или виртуальные методы?.....35
Динамические объекты.....36
Размещение и инициализация с помощью процедуры New.....36
Удаление динамических объектов.....37

ДЕСТРУКТОРЫ.....38

Пример размещения динамического объекта.....40
Удаление сложной структуры данных из динамической памяти.....41

ЧТО ЖЕ ДАЛЬШЕ?.....43

ЗАКЛЮЧЕНИЕ.....44

Объектно-ориентированное программирование

Объектно-ориентированное программирование представляет собой метод программирования, который весьма близко напоминает наше поведение. Оно является естественной эволюцией более ранних нововведений в разработке языков программирования. Объектно-ориентированное программирование является более структурным, чем все предыдущие разработки, касающиеся структурного программирования. Оно также является более модульным и более абстрактным, чем предыдущие попытки абстрагирования данных и переноса деталей программирования на внутренний уровень. Объектно-ориентированный язык программирования характеризуется тремя основными свойствами:

1. Инкапсуляция. Комбинирование записей с процедурами и функциями, манипулирующими полями этих записей, формирует новый тип данных - объект.
2. Наследование. Определение объекта и его дальнейшее использование для построения иерархии порожденных объектов с возможностью для каждого порожденного объекта, относящегося к иерархии, доступа к коду и данным всех порождающих объектов.
3. Полиморфизм. Присваивание действию одного имени, которое затем совместно используется вниз и вверх по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, именно ему подходящим.

Языковые расширения Borland Pascal предоставляют вам все средства объектно-ориентированного программирования: большую структурированность и модульность, большую абстрактность и встроенную непосредственно в язык возможность повторного использования. Все эти характеристики соответствуют коду, который является более структурированным, более гибким и более легким для обслуживания.

Объектно-ориентированное программирование порой требует от вас оставить в стороне характерные представления о программировании, которые долгие годы рассматривались, как стандартные. Однако после того, как это сделано, объектно-ориентированное программирование становится простым, наглядным и превосходным средством разрешения многих проблем, которые доставляют неприятности традиционному программному обеспечению.

Дадим хороший совет тому, кто уже имел дело с объектно-ориентированным программированием на других языках. Оставьте в стороне ваши прежние впечатления об объектно-ориентированном программировании и изучайте объектно-ориентированные характеристики Borland Pascal в их собственных терминах. Объектно-ориентированное программирование не является единственным путем, оно представляет собой континуум идей. По своей объект-

ной философии Borland Pascal больше напоминает C++, чем Smalltalk. Smalltalk является интерпретатором, тогда как Borland Pascal с самого начала был чистым компилятором реального кода. Компилятор реального кода выполняет работу иначе (и значительно быстрее), чем интерпретатор. Borland Pascal был сконструирован, как инструмент разработки продуктов, а не как инструмент исследования.

Для тех, кто не имеет об этом ни малейшего понятия, мы не будем подробно объяснять, что такое объектно-ориентированное программирование. В этот вопрос и так уже внесено достаточно путаницы. Поэтому забудьте о том, что люди говорили вам об объектно-ориентированном программировании. Наилучший способ (и, фактически, единственный) изучить что-либо полезное об объектно-ориентированное программировании - это сделать то, что вы уже почти сделали: сесть и попытаться узнать все самостоятельно.

Объекты

Посмотрите вокруг себя... и вы обнаружите яблоко, которое вы купили к завтраку. Допустим, что вы намерены описать яблоко в терминах программирования. Первое, что вы, возможно, попытаетесь сделать, так это рассмотреть его по частям; пусть S представляет площадь кожуры, J представляет содержащийся в яблоке объем жидкого сока, F представляет вес фрукта внутри кожуры, D - число семечек...

Не думайте таким образом. Думайте как живописец. Вы видите яблоко и вы рисуете яблоко. Изображение яблока не есть само яблоко. Оно является символом на плоской поверхности. Но оно не может быть абстрагировано в несколько чисел, каждое из которых расположено отдельно и независимо где-нибудь в сегменте данных. Его компоненты остаются вместе в их существенной взаимосвязи друг с другом.

Объекты моделируют характеристики и поведение элементов мира, в котором мы живем. Они являются окончательной абстракцией данных.

> *Примечание:* Объекты содержат вместе все свои характеристики и особенности поведения.

Яблоко можно разрезать на части, но как только оно будет разрезано, оно больше не будет яблоком. Отношения частей к целому и взаимоотношения между частями становятся понятнее тогда, когда все содержится вместе в одной упаковке. Это называется инкапсуляцией и является очень важным понятием. Немного позже мы к нему вернемся.

Не менее важным является и тот факт, что объекты могут наследовать характеристики и поведение того, что мы называем порождающие, родительские объекты (или предки). Здесь происходит качественный скачок: наследование, возможно, является сегодня единственным самым крупным различием между обычным программированием на Паскале и объектно-ориентированным программированием в Borland Pascal.

Наследование

Целью науки является описание взаимодействий во вселенной. Большая часть работы в науке при продвижении к цели заключается просто в построении генеалогических деревьев. Когда энтомолог возвращается с Амазонки с неизвестным ранее насекомым в банке, то главной его заботой является определение, где это насекомое располагается в гигантской схеме, на которой собраны научные названия всех других насекомых. Аналогичные схемы составлены для растений, рыб, млекопитающих, рептилий, химических элементов, элементарных частиц и космических галактик. Все они выглядят как генеалогические деревья: с единой всеобщей категорией в вершине и все увеличивающимися число категорий, которые лежат ниже этой единственной категории, и разворачиваются веером по мере приближения к границам разнообразия.

Внутри категории "насекомые" имеется два подразделения: насекомые с видимыми крыльями и насекомые со спрятанными крыльями или вообще бескрылые. Среди крылатых имеется большее число категорий; мотыльки, бабочки, мухи и т.д. Каждая категория содержит большое число подкатегорий, а ниже этих подкатегорий может иметься даже еще большее число подкатегорий (см. Рис. 1).

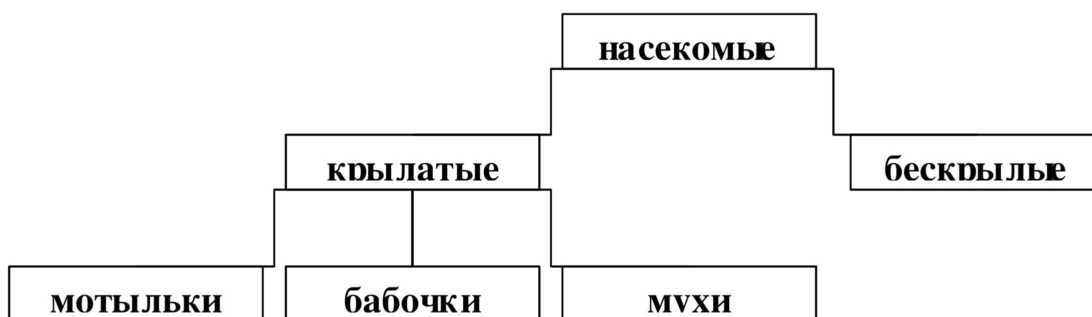

Рис. 1 Таксономическая схема насекомых.

Этот процесс классификации называется таксономией. Это прекрасная начальная метафора для механизма наследования в объектно-ориентированном программировании.

Вопросами, которые задает ученый при попытке классификации некоторого животного или объекта, являются следующие. В чем этот объект похож на другие объекты из общего класса? В чем он отличается от других объектов? Каждый конкретный класс имеет множество свойств поведения и характеристик, определяющих этот класс. Ученый начинает с вершины конкретного генеалогического дерева и проходит по дочерним областям, задавая себе эти вопросы. Наивысший уровень самый общий, а вопросы самые простые, например, крылатое или бескрылое? Каждый последующий уровень является более специфическим, чем предыдущий, и менее общим. В конце концов, ученый добирается до точки подсчета волосков на третьем сегменте задней ноги насекомого - что воистину специфично. (И скорее всего о нем следует сказать, что он не энтомолог.)

Важно помнить то, что если характеристика однажды определена, то все категории, расположенные ниже данного определения, содержат эту характери-

стику. Таким образом, как только вы определи ли насекомое, как члена отряда diptera (мухи), то вам не следует отмечать снова, что у мух имеется одна пара крыльев. Разновидность насекомых, которую мы зовем мухи, наследует эту характеристику от своего отряда.

Как вы поняли, объектно-ориентированное программирование в большой степени является процессом построения генеалогического дерева для структур данных. Одной из важных особенностей, которые объектно-ориентированное программирование добавляет традиционным языкам типа Паскаль, является механизм, с помощью которого типы данных могут наследовать характеристики более простых, более общих типов. Этим механизмом является наследование.

Объекты: наследующие записи

В терминах Паскаля, объект во многом схож с записью, которая является оболочкой для объединения нескольких связанных элементов под одним именем. Предположим, вы хотите разработать программу вывода платежной ведомости, печатающую отчет и показывающую, сколько нужно выплатить каждому служащему за рабочий день. Запись можно организовать следующим образом:

```
TEmployee = record
  Name: string[25];
  Title: string[25];
  Rate: Real;
end;
```

> *Примечание:* По соглашению все типы начинаются с буквы T. Вы также можете следовать этому правилу.

Здесь TEmployee является типом записи, т.е. шаблоном, используемым компилятором для создания переменных типа запись. Переменная типа TEmployee является экземпляром этого типа. Термин "экземпляр" будет вам нередко встречаться в Паскале. Он постоянно применяется теми, кто использует методы объектно-ориентированного программирования, поэтому будет хорошо, если вы начнете мыслить в терминах типов и экземпляров этих типов.

Вы можете оперировать с типом TEmployee двояко. Вы можете рассматривать поля Name, Title и Rate по отдельности, а когда о полях, как о работающих одновременно для описания конкретного рабочего, вы можете рассматривать их совокупность, как TEmployee.

Предположим, что на вашей фирме работает несколько типов рабочих. Одни из них имеют почасовую оплату, другие - оклад, третьи - тарифную ставку и так далее. Ваша программа должна учитывать все эти типы. Вы можете создать другой тип записи для каждого типа рабочего. Например, для получения данных о том, сколько должен получить рабочий с почасовой оплатой, нужно знать, сколько часов он отработал. Можно построить запись THourly вида:

```
THourly = record
  Name: string[25];
  Title: string[25];
```

```
  Rate: Real;
end;
```

Вы можете также оказаться несколько догадливее и сохранить тип TEmployee путем создания поля типа TEmployee внутри типа THourly:

```
THourly = record
  Worker: TEmployee;
  Time: integer;
end;
```

Такая конструкция работает, и программисты, работающие на Паскале, делают это постоянно. Единственное, чего этот метод не делает, так это то, что он заставляет вас думать о том, с чем вы работаете в вашем программном обеспечении. Вам следует задаться вопросом типа; "Чем почасовик отличается от других рабочих?" Ответ прост: почасовик - это рабочий, которому платится за количество часов работы. Продумайте снова первую часть предложения; почасовик - это рабочий…

Теперь вы поняли!

Запись для работника-почасовика hourly должна иметь все записи, которые имеются в записи employee. Тип THourly является дочерним типом для типа TEmployee. THourly наследует все, что принадлежит TEmployee, и кроме того имеет кое-что новое, что делает THourly уникальным.

Этот процесс, с помощью которого один тип наследует характеристики другого типа, называется наследованием. Наследник называется порожденным (дочерним) типом, а тип, которому наследует дочерний тип, называется порождающим (родительским) типом.

Ранее известные типы записей Паскаля не могут наследовать. Однако Borland Pascal расширяет язык Паскаль для поддержки наследования. Одним из этих расширений является новая категория структуры данных, связанная с записями, но значительно более мощная. Типы данных в этой новой категории определяются с помощью нового зарезервированного слова object. Тип объекта может быть определен как полный, самостоятельный тип в манере описания записей Паскаля, но он может определяться и как потомок существующего типа объекта путем помещения порождающего (родительского) типа в скобки после зарезервированного слова object.

В приводимом здесь примере платежной ведомости два связанных типа объектов могли бы определяться следующим образом:

```
type
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate : Real;
  end;

  THourly = object(TEmployee)
```

```
  Time : Integer;
end;
```

> *Примечание:* Обратите внимание, что здесь использование скобок означает наследование.

Здесь TEmployee является родительским типом, а THourly - дочерним типом. Как вы увидите чуть позднее, этот процесс может продолжаться неопределенно долго. Вы можете определить дочерний тип THourly, дочерний к типу THourly тип и т.д. Большая часть конструирования объектно-ориентированных прикладных программ состоит в построении такой иерархии объектов, являющейся отражением генеалогического дерева объектов в приложениях.

Все возможные типы, наследующие тип TEmployee, называются дочерними типами типа TEmployee, тогда как THourly является непосредственным дочерним типом типа TEmployee. Наоборот, TEmployee является непосредственным родителем типа THourly. Тип объекта (в точности как подкаталог в DOS) может иметь любое число непосредственных дочерних типов, но в то же время только одного непосредственного родителя.

Как показали данные определения, объекты тесно связаны с записями. Новое зарезервированное слово object является наиболее очевидным различием, но как вы увидите позднее, имеется большое число других различий, некоторые из которых довольно тонкие.

Например, поля Name, Title и Rate в типе TEmployee не указаны явно в типе THourly, но в любом случае тип THourly содержит их благодаря свойству наследования. Вы можете говорить о величине Name типа THourly в точности так же, как о величине Name типа TEmployee.

Экземпляры объектных типов

Экземпляры объектных типов описываются в точности так же, как в Паскале описывается любая переменная, либо статическая, либо указатель, ссылающийся на размещенную в динамической памяти переменную:

```
type
  PHourly = ^THourly;
var
  StatHourly: THourly;       { готово }
  DynaHourly: PHourly;       {перед использованием память должна
                               выделяться с помощью New }
```

Поля объектов

Вы можете обратиться к полю объекта в точности так же, как к полю обычной записи, либо с помощью оператора with, либо путем уточнения имени с помощью точки. Например:

```
AnHourly.Rate := 9.45;
```

```
with AnHourly do
begin
  Name := 'Sanderson, Arthur';
  Title := 'Word processor';
end;
```

> *Примечание:* Не забывайте о том, что наследуемые поля объектов не интерпретируются особым образом только потому, что они являются наследуемыми.

Именно сейчас вы должны запомнить (в конце концов это придет само собой), что наследуемые поля являются столь же доступными, как если бы они были объявлены внутри типа объекта. Например, даже если Name, Title и Rate не являются частью описания типа THourly (они наследованы от типа TEmployee), то вы можете ссылаться на них, словно они описаны в THourly:

```
AnHourly.Name := 'Arthur Sanderson';
```

Хорошая и плохая техника программирования

Даже если вы можете обратиться к полям объекта непосредс твенно, это будет не совсем хорошей идеей. Принципы объектно-ориентированного программирования требуют, чтобы поля объектов были исключены из исходного кода, насколько это возможно. Это ограничение поначалу может показаться спорным и жестким, но оно является только частью огромной картины объектно-ориентированное программирования, которую мы нарисуем в этой главе. Со временем вы увидите смысл, скрытый в этом новом определении хорошей практики программирования, хотя имеются некоторые основания приоткрыть его перед тем, как все придет само. А пока же примите на веру: избегайте прямого обращения к полям данных.

> *Примечание:* Borland Pascal позволяет вам делать поля объекта и его методы частными. Подробнее об этом рассказывается ниже.

Итак, как же обращаться к полям объекта? Как читать их и как присваивать им значения?

> *Примечание:* Поля данных объекта - это то, что объект "знает", а методы объекта - это то, что объект "делает".

Ответ заключается в том, что при всякой возможности для доступа к полям данных должны использоваться методы объекта. Метод является процедурой или функцией, описанной внутри объекта и жестко ограниченной этим объектом.

Методы

Методы являются одними из наиболее примечательных атрибутов объектно-ориентированное программирования и требуют некоторой практики перед использованием. Вернемся сначала к исходному вопросу о тщетной попытке структурного программирования, связанной с инициализацией структур данных. Рассмотрим задачу инициализации записи со следующим определением:

```
TEmployee = object
  Name: string[25];
  Title: string[25];
  Rate: Real;
end;
```

Большинство программистов использовали бы оператор with для присвоения полям Name, Title и Rate начальных значений:

```
var
  MyEmployee: Employee;

with MyEmployee do
begin
  Name := 'Sanderson, Arthur';
  Title :=  'Word processor';
  Rate := 9.45;
end;
```

Это тоже неплохо, но здесь мы жестко ограничены одним специфическим экземпляром записи - MyEmployee. Если потребуется инициализировать более одной записи типа Employee, то вам придется использовать большее число операторов with, которые выполняют в точности те же действия. Следующим естественным шагом является создание инициализирующей процедуры, которая обобщает оператор with применительно к любому экземпляру типа TEmployee, пересылаемой в качестве параметра:

```
procedure InitTEmployee(var Worker: TEmployee; AName, ATitle: String;
                                ARate: Real);
Begin
  with Worker do
  begin
    Name := NewName ;
    Title := NewTitle;
    Rate := NewRate;
  end;
end;
```

Это будет работать, все в порядке, однако если вы почувствуете, что при этом тратите больше времени, чем необходимо, то вы почувствуете то же самое,

что чувствовал ранний сторонник объектно-ориентированного программирования.

Это чувство значит, что, ну, скажем, вы разрабатываете процедуру InitTEmployee специально для обслуживания типа TEmployee. Тогда почему вы должны помнить, какой тип записи и какой его экземпляр обрабатывает InitTEmployee? Должен существовать некоторый путь объединения типа записи и обслуживающего кода в одно единое целое.

Такой путь имеется и называется методом. Метод - это процедура или функция, объединенная с данным типом столь тесно, что метод является как бы окруженным невидимым оператором with, что делает экземпляр данного типа доступными изнутри для метода. Определение типа включает заголовок метода. Полное определение метода квалифицируется в имени типа. Тип объекта и метод объекта являются двумя лицами этой новой разновидности структуры, именуемой методом.

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
  procedure Init (NewName, NewTitle: string[25];NewRate: Real);
  end;

procedure TEmployee.Init (NewName, NewTitle: string[25];NewRate: Real);
begin
  Name := NewName ;   { Поле Name объекта TEmployee }
  Title := NewTitle;       { Поле Tutle объекта TEmployee }
  Rate := NewRate;       { Поле Rate объекта TEmployee }
end;
```

Теперь для инициализации экземпляра типа TEmployee вы просто вызываете его метод, словно метод является полем записи, что имеет вполне реальный смысл:

```
var
  AnEmployee: TEmployee;

AnEmployee.Init('Sara Adams, Account manager, 15000'  );
                                        {просто, не так ли?}
```

Совмещенные код и данные

Одним из важнейших принципов объектно-ориентированного программирования является то, что программист во время разработки программы должен думать о коде и о данных совместно. Ни код, ни данные не существуют в вакууме. Данные управляют потоком кода, а код манипулирует образами и значениями данных.

Если ваши код и данные являются разделенными элементами, то всегда существует опасность вызова правильной процедуры с неверными данными или ошибочной процедуры с правильными данными. Забота о совпадении этих элементов возлагается на программиста, и хотя строгая типизация Паскаля здесь помогает, самое лучшее, что он может сделать - это указать на несоответствие.

О том, что действительно существует вместе, Паскаль нигде не сообщает. Если это не отмечено комментарием или не то, о чем вы все время помните, то вы играете с судьбой.

Объект осуществляет синхронизацию кода и данных путем совместного построения их описаний. Реально, чтобы получить значение одного из полей объекта, вы вызываете относящийся к этому объекту метод, который возвращает значение нужного поля. Чтобы присвоить полю значение, вы вызываете метод, который назначает данному полю новое значение.

Однако, Borland Pascal не вынуждает вас делать это. Как всякое структурное программирование, объектно-ориентированное программирование является дисциплиной, которую вы должны навязать себе, используя предоставляемые языком средства. Borland Pascal позволяет вам обращаться к полям объекта непосредственно извне объекта, однако он поощряет вас использовать преимущества объектно-ориентированного программирования и создавать методы для манипулирования полями объекта внутри самого объекта. Borland Pascal позволяет задать принудительную инкапсуляцию с помощью использования описания private в объявлении объекта.

> *Примечание:* Подробнее о принудительной инкапсуляции рассказывается ниже в разделе "Секция private".

Определение методов

Процесс определения методов объектов напоминает модули Borland Pascal. Внутри объекта метод определяется заголовком процедуры или функции, действующей как метод:

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
  procedure Init (AName, ATitle: String; ARate: Real);
  function GetName : String;
  function GetTitle : String;
  function GetRate : Real;
  end;
```

> *Примечание*: Поля данных должны быть описаны перед первым описанием метода.

Как и описания процедур и функций в интерфейсной секции модуля (interface), описание методов внутри объекта говорит, что методы делают, но не говорит, как.

Это определяется вне определения объекта, в отдельном описании процедуры или функции. Если метод полностью определяется вне объекта, то имени метода должно предшествовать имя типа объекта, которому принадлежит этот метод, с последующей точкой:

```
procedure TEmployee.Init(AName, ATitle: string;ARate: Real);
begin
  Name := AName;
  Title := ATitle;
  Rate := ARate;
end;

function TEmployee.GetName: String;
begin
  GetName := Name;
end;

function TEmployee.GetTitle: String;
begin
  GetTitle := Title;
end;

function TEmployee.GetRate: Real;
begin
  GetRate := Rate;
end;
```

Метод определения следует методу интуитивного разделения точками для указания поля записи. Кроме наличия определения TEmployee.GetName можно было бы определить процедуру с именем GetName, в имени которой нет предшествующего идентификатора TEmployee. Однако, такая "внешняя" GetName не будет иметь никакой связи с объектом типа TEmployee и будет только запутывать смысл программы.

Область действия метода и параметр Self

Заметьте, что ни в одном из предыдущих примеров конструкция: with объект do... не встречается в явном виде. Поля данных объекта легко доступны с помощью методов объекта. Хотя в исходном коде поля данных объекта и тела методов разделены, на самом деле они совместно используют одну и ту же область действия.

Именно поэтому один из методов TEmployee может содержать оператор GetTitle := Title без какого-либо квалификатора перед Title. И именно поэтому Title принадлежит тому объекту, который вызывает метод. Если объект вызывает метод, то выполняется неявный оператор with myself do method, связывающий объект и его методы в области действия.

Неявный оператор with выполняется путем передачи невидимого параметра методу всякий раз, когда этот метод вызывается. Этот параметр называется Self и в действительности является 32-разрядным указателем на экземпляр объекта, осуществляющего вызов метода. Относящийся к TEmployee метод GetRate приблизительно эквивалентен следующему:

```
function TEmployee.GetRate(var Self: TEmployee): integer;
begin
  GetRate := Self.Rate;
end;
```

> *Примечание:* Синтаксически этот пример не совсем корректен. Он приведен только затем, чтобы дать вам более полное представление о специфической связи между объектом и его методом.

Но важно ли вам знать о существовании параметра Self? Обычно нет. Генерируемый Borland Pascal код выполняет все это автоматически. Однако в некоторых немногочисленных случаях вы можете захотеть проникнуть внутрь метода и использовать параметр Self яво.

> *Примечание:* Явное использование параметра Self допускается, но вы должны избегать ситуаций, в которых это требуется.

Параметр Self является частью физического кадра стека при всех вызовах методов. Методы, используемые как внешние на языке Ассемблера, должны учитывать Self при получении доступа к параметрам метода в стеке.

Поля данных объекта и формальные параметры метода

Выводом из того факта, что методы и их объекты разделяют общую область действия, является то, что формальные параметры метода не могут быть идентичными любому из полей данных объекта. Это является не каким-то новым ограничением, налагаемым объектно-ориентированным программированием, а скорее теми же самыми старыми правилами области действия, которые Паскаль имел всегда. Это то же самое, что и запрет для формальных параметров процедуры быть идентичными локальным переменным этой процедуры:

```
procedure CrunchIt(Crunchee: MyDataRec, Crunchby, ErrorCode: integer);
var
A, B: char;
  ErrorCode: integer;
Begin
…
end;
```

Локальные переменные процедуры и ее формальные параметры совместно используют общую область действия и поэтому не могут быть идентичными. Вы получите сообщение "Error 4: Duplicate identifier" (Ошибка 4; Повторение

идентификатора), если попытаетесь компилировать что-либо подобное, та же ошибка возникает при попытке присвоить формальному параметру метода имени поля объекта, которому данный метод принадлежит.

Обстоятельства несколько отличаются, так как помещение заголовка процедуры внутрь структуры данных является намеком на новшество в Турбо Паскале, но основные принципы области действия Паскаля не изменились.

Объекты, экспортируемые модулями

Имеет смысл определять объекты в модуле посредством описаний типа объекта в интерфейсной части модуля, а тела процедур и мето ды объекта - в секции реализации. Для определения объекта в модуле не требуется никаких специальных соглашений.

> *Примечание:* Экспортируемый - означает "определенный в интерфейсной части модуля".

Модули могут иметь свои собственные приватные (частные) определения типов объектов внутри выполняемой секции, и эти типы подвержены тем же ограничениям, как и всякие другие типы, определенные в секции реализации. Типы объектов, определенные в интерфейсной части модуля, могут иметь дочерние типы объектов, определенные в секции реализации модуля. В том случае, когда модуль B использует модуль A, модуль B также может определять дочерние типы любого типа объекта, экспортируемого модулем A.

Описанные ранее типы объектов и методы можно определить в модуле, как показано в программе WORKERS.PAS на дистрибутивном диске. Чтобы использовать типы объектов и методы, определенные в модуле Workers, вы можете просто использовать этот модуль в своей программе и описать экземпляр типа THourly в секции переменных программы:

```
program HourPrt;

uses WinCrt, Workers;

var
  AnHourly: THourly;
```

Для создания и вывода фамилии рабочего-почасовика, его должности и размера выплаты, представленной переменной AnHourly, вы просто вызываете методы AnHourly, используя следующий синтаксис:

```
AnHourlye.Init('Sara Adams', 'Account manager', 1400);
{записывает в экземпляр Thourly данные для Сары Адамс:
фамилию, должность и размер выплаты. }
AnHourly.Show;
```

> *Примечание:* Объектами могут быть также типизированные константы.

Объекты, будучи очень схожими с записями, могут использоваться внутри оператора with. В этом случае указание имени объекта, являющегося собственником методов, не является необходимым:

```
with AnHourly do
begin
  Init('Sara Adams', 'Account manager', 1400);
  Show;
end;
```

Как и в случаях с записями, объекты могут передаваться в качестве параметра процедуре и (как вы увидите позднее) могут размещаться в динамически распределяемой памяти.

Секция private

В некоторых случаях у вас могут иметься части описаний объектов, которые экспортировать нежелательно. Например, вы можете предусмотреть объекты для других программистов, которые они могут использовать, но не могут непосредственно манипулировать с данными объекта. Чтобы облегчить это, Borland Pascal позволяет задавать внутри объектов приватные (закрытые) поля и методы.

Приватные поля и методы доступны только внутри того модуля, в котором описан объект. В предыдущем примере, если бы тип THourly содержал приватные поля, то доступ к ним можно было бы получить только в модуле THourly. Даже если другие части объекта THourly можно было бы экспортировать, (части, описанные, как приватные, были бы недоступными.

Приватные поля и методы описываются непосредственно после обычных полей и методов, вслед за зарезервированным словом private. Таким образом, полный синтаксис описания объекта будет следующим:

```
type
  NewObject = object(родитель)
    поля;              { общедоступные }
    методы;            { общедоступные }
  private
    поля;              { приватные }
    методы;            { приватные }
  end;
```

Программирование в "действительном залоге"

Большая часть из того, что говорилось об объектах до сих пор, исходит из удобств и перспектив Borland Pascal, поскольку наиболее вероятно, что это именно то, с чего вы начнете. Теперь начнутся изменения, поскольку мы подошли к концепциям объектно-ориентированного программирования с помощью некоторых принципов программирования на стандартном Паскале. Объектно-

ориентированное программирование имеет свое собственное отдельное множество понятий, частично благодаря началам объектно-ориентированного программирования (до некоторой степени ограниченным) в научных кругах, однако также и потому, что эта концепция действительно является радикально отличной от других.

> *Примечание:* Объектно-ориентированные языки однажды метафорично назвали "языками актеров".

Одним, часто забавным, следствием этого явилось то, что объектно-ориентированное программирование фанатично "одушевляет" свои объекты. Отныне данные для вас не емкости, которые вы можете наполнять значениями. С точки зрения нового взгляда на вещи, объ екты выглядят как актеры на подмостках со множеством заученных ролей (методов). Если вы (директор) даете им слово, то актеры начинают декламировать в соответствии со сценарием.

Было бы полезно представить функцию AnHourly.GetPayAmount как, например, дающую распоряжение объекту AnHourly "Вычислить размер вашей ежедневной платы". Центральной концепцией здесь является объект. Этот объект обслуживают как список методов, так и список полей данных, содержащихся в объекте. И ни код, ни данные не являются здесь "директором".

Чтобы быть совсем привлекательным, объект не может быть описан как актер на сцене. Образцу объектно-ориентированного программирования с большим трудом удается моделировать составляющие проблемы как компоненты, а не как логические абстракции. Случайности и закономерности, наполняющие нашу жизнь (от тостеров до телефонных звонков по поводу махровых полотенец) все имеют характеристики (данные) и линии поведения (методы). Характеристики тостера могут включать требуемое напряжение, число гренок, которые он может поджарить одновременно, установку слабого или сильного уровней поджаривания, цвет тостера, его фабричную марку и т.д. Его поведение может включать загрузку кусков хлеба, поджаривание этих кусков и автоматическое выталкивание готовых гренок наружу.

Если мы хотим написать программу имитации кухни, то какой же имеется наилучший способ смоделировать различные приспособления, кроме объектов, с их характеристиками и линиями поведения, закодированными в поля данных и в методах? Фактически, это уже сделано: один из первых объектно-ориентированных языков (Симула-67) был создан как язык для написания таких имитаций.

Есть также причина того, что объектно-ориентированное программирование довольно крепко связано в традиционном смысле с ориентированной на построение графиков средой. Объекты должны быть моделями, и есть ли лучший способ смоделировать объект, чем нарисовать его изображение? Объекты в Borland Pascal должны имитировать компоненты проблему, которую вы пытаетесь разрешить. Примите это во внимание, если в дальнейшем вы намерены эксплуатировать новые объектно-ориентированные расширения Borland Pascal.

Инкапсуляция

Объединение в объекте кода и данных называется инкапсуляцией. Возможно вы сможете предоставить достаточное количество методов, благодаря чему пользователь объекта никогда не будет обращаться к полям объекта непосредственно. Некоторые другие объектно-ориентированные языки, например Smalltalk, требуют обязательной инкапсуляции, однако в Borland Pascal у вас есть выбор, а хорошая практика объектно-ориентированного программирования во многом зависит от вашей добросовестности.

Объекты TEmployee и THourly написаны таким образом, что совершенно исключена необходимость прямого обращения к их внутренним полям данных:

```
type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
  procedure Init (AName, ATitle: string; ARate: Real);
  function GetName : String;
  function GetTitle : String;
  function GetRate : Real;
  function GetPayAmount : Real;
  end;

  THourly = object(TEmployee)
    Time: Integer;
  procedure Init(AName, ATitle: string; ARate:Real, Atime: Integer);
  function GetPayAmount : Real;
  end;
```

Здесь присутствуют только четыре поля данных: Name, Title, Rate и Time. Методы ShowName и ShowTitle выводят фамилию работающего и его должность, соответственно. Метод GetPayAmount использует Rate, а в случае работающего THourly и Time для вычисления суммы выплат работающему. Здесь уже нет необходимости обращаться непосредственно к этим полям данных.

Предположив существование экземпляра AnHourly типа THourly, вы могли бы использовать набор методов для манипулирования полями данных AnHourly, например:

```
with AnHourly do
begin
  Init ('Allison Karlon, Fork lift operator'   12.95, 62
  { Выводит на экран фамилию, должность и сумму выплат}
  Show;
end;
```

Обратите внимание, что доступ к полям объекта осуществляется не иначе, как только с помощью методов этого объекта.

Методы: никакого ухудшения

Добавление этих методов незначительно увеличивает объем исходного кода, однако развитый компоновщик Borland Pascal выбрасывает код любого метода, который ни разу не вызывается в программе. Поэтому вам не следует отступать при предоставлении объекту того или иного метода, который имеет одинаковые шансы быть как использованным, так и неиспользованным в каждой программе, в которой задействован данный тип объекта. Неиспользуемые методы ничего не будут стоить вам как в части качества выполнения программы, так и в части ее размера, - если они не используются в программе, то они попросту отсутствуют в ней.

Замечание по поводу абстрактности данных: Имеется громадное преимущество в возможности полностью отсоединить THourly от глобальных ссылок. Если ничто вне объекта не "знает" о представлении его внутренних данных, то программист, контролирующий объект, может изменять детали внутреннего представления данных до тех пор, пока не изменится заголовок метода.

Внутри самого объекта данные могут быть представлены в виде массива, однако позднее (возможно, что сфера действия прикладной программы расширяется и объем ее данных растет) в качестве более эффективного представления данных может быть признано двоичное дерево. Если объект полностью инкапсулирован, изменение представления данных с массива на двоичное дерево вообще не изменит использование объекта. Интерфейс с объектом останется полностью тем же, позволяя программисту изящно приспосабливать эксплуатационные качества объекта без изменения кода, использующего объект.

Расширяющиеся объекты

Люди, которые впервые сталкиваются с Паскалем, зачастую считают само собой разумеющейся гибкость стандартной процедуры Writeln, которая позволяет единственной процедуре обрабатывать параметры многих различных типов:

```
Writeln(CharVar);    { Вывести значение символьного типа }
Writeln(IntegerVar); { Вывести целое значение }
Writeln(RealVar);    { Вывести значение с плавающей точкой }
```

К сожалению, стандартный Паскаль не предоставляет лично вам никаких возможностей для создания столь же гибких процедур.

Объектно-ориентированное программирование решает эту проблему с помощью наследования: если определен порожденный тип, то методы порождающего типа наследуются, однако при желании они могут переопределяться. Для переопределения наследуемого метода попросту опишите новый метод с тем же именем, что и наследуемый метод, но с другим телом и (при необходимости) с другим множеством параметров.

Простой пример прояснит как процесс так и его смысл. Давайте определим дочерний по отношению к TEmployee тип, представляющий работника, которому платится часовая ставка:

```
const
PayPeriods = 26;              { периоды выплат }
OvertimeThreshold = 80;   { на период выплаты }
OvertimeFactor = 1.5;       { почасовой коэффициент }

type
  THourly = object(TEmployee)
    Time: Integer;
  procedure Init(AName, ATitle: string; ARate:Real, Atime: Integer);
  function GetPayAmount : Real;
  end;

procedure THourly.Init(AName, ATitle: string;ARate: Real, Atime: Integer);
begin
  TEmployee.Init(AName, ATitle, ARate);
  Time := ATime;
end;

function THourly.GetPayAmount: Real;
var
  Overtime: Integer;
Begin
  Overtime := Time - OvertimeThreshold;
  if Overtime > 0
  then
    GetPayAmount := RoundPay(OvertimeThreshold * Rate +
                              Rate OverTime * OvertimeFactor * Rate)
  else
    GetPayAmount := RoundPay(Time * Rate)
end;
```

Человек, которому платится часовая ставка, является работающим: он обладает всем тем, что мы используем для определения объекта TEmployee (фамилией, должностью, ставкой), и лишь количество получаемых почасовиком денег зависит от того, сколько часов он отработал за период, подлежащий оплате. Таким образом, для THourly требуется еще и поле времени, Time.

Так как THourly определяет новое поле, Time, его инициализация требует нового метода Init, который инициализирует и время, и наследованные поля. Вместо того, чтобы непосредственно присвоить значения наследованным полям, таким как Name, Title и Rate, почему бы не использовать вновь метод инициализации объекта TEmployee (иллюстрируемый первым оператором THourly.Init), где Ancestor есть идентификатор типа родового типа объекта, а Method есть идентификатор метода данного типа.

Заметьте, что вызов метода, который вы переопределяете, не является единственно хорошим стилем. В общем случае возможно, что TEmployee.Init выполняет важную, однако скрытую инициализацию. Вызывая переопределяемый метод, вы должны быть уверены в том, что порожденный тип объекта включает

функциональность родителя. Кроме того, любое изменение в родительском методе автоматически оказы вает влияние на все порожденные.

После вызова TEmployee.Init, THourly.Init может затем выполнить свою собственную инициализацию, которая в этом случае состоит только в присвоении значения, переданного в ATime.

Другим примером переопределяемого метода является функция THourly.GetPayAmount, вычисляющая сумму выплат работающему на почасовой ставке. В действительности, каждый тип объекта TEmployee имеет свой метод GetPayAmount, так как тип работающего зависит от того, как производится расчет. Метод THourly.GetPayAmount должен учитывать, сколько часов работал сотрудник, были ли сверхурочные работы, каков коэффициент увеличения за сверхурочные работы и так далее. Метод TSalaried.GetPayAmount должен лишь делить ставку работающего на число выплат в каждом году.

```
unit Workers;

interface

const
  PayPeriods = 26;        {в год}
  OvertimeThreshold = 80;   {за каждый период оплаты}
  OvertimeFactor =1.5;        {увеличение против обычной оплаты}

type
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
  procedure Init (AName, ATitle: string; ARate: Real);
  function GetName : String;
  function GetTitle : String;
  function GetRate : Real;
  function GetPayAmount : Real;
  end;

  THourly = object(TEmployee)
    Time: Integer;
  procedure Init(AName, ATitle: string; ARate:Real, Atime: Integer);
  function GetPayAmount : Real;
  function GetTime : Real;
  end;

  TSalaried = object(TEmployee)
  function GetPayAmount : Real;
  end;

  TCommissioned = object(TSalaried)
    Commission : Real;
```

```
    SalesAmount : Real;
  constructor Init (AName, ATitle: String;
  ARate, ACommission, ASalesAmount: Real);
  function GetPayAmount : Real;
  end;

implementation

function RoundPay(Wages: Real) : Real;
{ округляем сумму выплат, чтобы игнорировать суммы меньше пенни }
begin
  RoundPay := Trunc(Wages * 100) / 100;
```

TEmployee является вершиной нашей иерархии объектов и содержит первый метод GetPayAmount.

```
function TEmployee.GetPayAmount : Real;
begin
  RunError(211);   { дать ошибку этапа выполнения }
end;
```

Может вызвать удивление тот факт, что метод дает ошибку времени выполнения. Если вызывается TEmployee.GetPayAmount, то в программе возникает ошибка. Почему? Потому что TEmployee является вершиной нашей иерархии объектов и не определяет реального рабочего; следовательно, ни один из методов TEmployee не вызывается определенным образом, хотя они и могут быть наследованными. Все наши работники являются либо почасовиками, либо имеют оклады, либо работают на сдельщине. Ошибка на этапе выполнения прекращает выполнение программы и выводит 211, что соответствует сообщению об ошибке, связанной с вызовом абстрактного метода (если ваша программа по ошибке вызывает TEmployee.GetPayAmount).

Ниже приводится метод THourly.GetPayAmount, в котором учитываются такие вещи как сверхурочная оплата, число отработанных часов и так далее.

```
function THourly.GetPayAMount : Real;
var
  OverTime: Integer;
begin
  Overtime := Time - OvertimeThreshold;
  if Overtime > 0
  then
    GetPayAmount := RoundPay(OvertimeThreshold * Rate +
                            Rate OverTime * OvertimeFactor * Rate)
  else
    GetPayAmount := RoundPay(Time * Rate)
end;
```

Метод TSalaried.GetPayAmount намного проще; в нем ставка делится на число выплат:

```
function TSalaried.GetPayAmount : Real;
begin
  GetPayAmount := RoundPay(Rate / PayPeriods);
end;
```

Если взглянуть на метод TCommissioned.GetPayAmount, то будет видно, что он вызывает TSalaried.GetPayAmount, вычисляет комиссинные и прибавляет их к величине, возвращаемой методом TSalaried.GetPayAmount.

```
function TCommissioned.GetPayAmount : Real;
begin
  GetPayAmount := RoundPay(TSalaried.GetPayAmount +
                              Commission * SalesAmount);
end;
```

> *Важн ое замечание:* Хотя методы могут быть переопределены, поля данных переопределяться не могут. После того, как вы опреде лили поле данных в иерархии объекта, никакой дочерний тип не может определить поле данных в точности с таким же именем.

Наследование статических методов

Все показанные до сих пор методы, связанные с типами объектов TEmployee, THourly, TSalaried и TCommissioned, являются статическими методами. Однако, со статическими методами связана проблема наследования.

Для того, чтобы разобраться с этой проблемой, отложим в сторону наш пример с платежной ведомостью и рассмотрим другой упрощенный и нереалистичный, но показательный пример. Вернемся к крылатым насекомым. Предположим, что нужно создать программу, которая будет рисовать на экране различные типы летающих насекомых.Предположим, вы решили, что на вершине иерархии будет находиться объект Winged. Пусть вы планируете, что новые типы объектов летающих насекомых как будут строиться как потомки Winged. Например, вы можете создать тип объекта Bee, который отличается от родственных крылатых насекомых тем, что имеет жало и полосы. Конечно, у пчелы есть другие отличающие ее характеристики, но в нашем примере это может выглядеть следующим образом:

```
type
  TWinged = object(Insect)
  procedure Init(AX, AY: Integer) { инициализирует экземпляр }
  procedure Show;   { отображает крылатое насекомое на экране }
  procedure Hide;   { стирает крылатое насекомое экрана }
  procedure MoveTo(NewX, NewY : Integer);{перемещает насекомое }
  end;
```

```
type
  TBee = object(Winged)
  procedure Init(AX, AY: Integer) { инициализирует экземпляр Bee }
  procedure Show;          { отображает пчелу на экране }
  procedure Hide;          { стирает пчелу с экрана }
  procedure MoveTo(NewX, NewY : Integer);    { перемещает пчелу }
  end;
```

И TWinged, и TBee имеют по четыре метода. TWinged.Init и TBee.Init инициализируют экземпляр соответствующих объектов. Метод TWinged.Show знает, как рисовать крылатое насекомое на экране, а метод TBee.Show - как рисовать пчелу (крылатое насекомое с полосками на теле и с жалом). Метод TWinged.Hide знает, как стирать крылатое насекомое с экрана, а метод TBee.Hide - как стирать пчелу. Два метода Show отличаются друг от друга, равно как и два метода Hide.

Однако, методы TWinged.MoveTo и TBee.MoveTo полностью одинаковы. В нашем примере X и Y определяют положение на экране.

```
procedure TWinged.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;    {новая координата X на экране}
  Y := NewY;    {новая координата Y на экране}
  Show;
end;

procedure TBee.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;    {новая координата X на экране}
  Y := NewY;    {новая координата Y на экране}
  Show;
end;
```

Не изменилось ничего, кроме копирования программы и постановки квалификатора TBee перед идентификатором MoveTo. Так как методы одинаковы, зачем нужно помещать MoveTo в TBee? Ведь Bee автоматически наследует MoveTo от TWinged. Поэтому не нужно переопределять метод MoveTo из TWinged, но это именно то место, где возникает проблема в случае статических методов.

Термин "статический" был выбран для описания методов, не являющихся виртуальными - термин, который мы введем далее. Фактически, виртуальные методы являются решением этой проблемы, но прежде чем понять решение, вам следует разобраться в самой проблеме.

Признаки проблемы состоят в следующем: пока копия метода MoveTo не будет помещена в область действия TBee для подавления метода MoveTo объек-

та TWinged, метод не будет работать правильно, если он будет вызываться из объекта типа TBee. Если TBee запускает метод MoveTo объекта TWinged, так то, что движется по экрану, является крылатым насекомым, а не пчелой. Только когда TBee вызывает копию метода MoveTo, определенного в его собственной области действия, на экране с помощью вызовов Show и Hide будут рисоваться и стираться пчелы.

Почему это так? Это объясняется способом, которым компилятор разрешает вызовы методов. Когда компилируются методы Bee, то сначала встречаются TWinged.Show и TWinged.Hide и их код компилируется в сегмент кода. Немного позднее в файле встречается метод Winged.MoveTo, который вызывает TWinged.Show и TWinged.Hide. Как и при вызове любой процедуры, компилятор замещает ссылки на TWinged.Show и TWinged.Hide в исходном коде на их адреса, сгенерированные в сегменте кода. Таким образом, когда вызывается код TWinged.MoveTo, он, в свою очередь, вызывает TWinged.Show и TWinged.Hide со всеми вытекающими последствиями.

До сих пор это был типичный для Borland Pascal сценарий и он был бы справедлив (за исключением номенклатуры), начиная с версии 1.0 Turbo Pascal 1983 года. Однако, дело меняется, когда вы включаете в этот сценарий принцип наследования. Когда TBee наследует метод от TWinged, он (TBee) использует метод в точности так, как тот был откомпилирован.

Снова посмотрите, что должен наследовать TBee, если он наследует TWinged.MoveTo:

```
procedure TWinged.MoveTo(NewX, NewY: integer);
begin
  Hide;              { Вызов Winged.Hide }
  X := NewX;
  Y := NewY;
  Show               { Вызов Winged.Show }
end;
```

Комментарии здесь приведены для того, чтобы подчеркнуть тот факт, что если Bee вызывает метод TWinged.MoveTo, то он также вызывает TWinged.Show и TWinged.Hide, а не TBee.Show и TBee.Hide. Поскольку TWinged.MoveTo вызывает методы TWinged.Show и TWinged.Hide, TWinged.MoveTo нельзя наследовать. Вместо этого, он должен быть переопределен своей второй копией, которая вызывает копии Show и Hide, определенные внутри области действия второй копии, то есть, TBee.Show и TBee.Hide.

При разрешении вызовов методов, логика компилятора работает так: при вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта. Тип TBee определяет методы с именами Init, Hide, Show и MoveTo. Если метод TBee должен был вызвать один из этих четырех методов, то компилятор заменил бы вызов на адрес одного из собственных методов Bee.

Если в типе объекта не определен метод с таким именем, то компилятор поднимается выше к непосредственному родительскому типу в поисках метода с указанным именем. Если метод с таким именем найден, то адрес родительского метода замещает имя в исходном коде дочернего метода. Если метод с таким

именем не найден, то компилятор продолжает продвигаться вверх по родительским объектам в поисках метода. Если компилятор наталкивается на самый первый(высший) тип объекта, то он выдает сообщение об ошибке, указывающее, что ни одного такого метода не определено.

Однако, если статический наследуемый метод найден и используется, то вы должны помнить, что вызываемый метод является в точности таким, как он определен и компилирован для родительского типа. Если родительский метод вызывает другие методы, то вызываемые методы будут также родительскими методами, даже если дочерний объект содержит методы, которые переопределяют родительские.

Виртуальные методы и полиморфизм

Обсуждаемые до сих пор методы являются статическими. Они являются статическими в том же смысле, в каком статической является статическая переменная: компилятор размещает ее и разрешает все ссылки на нее во время компиляции. Как вы видели, объекты и статические методы могут быть мощным инструментом для составления сложных программ.

Однако иногда это не лучший способ для управления методами.

Проблемы, аналогичные описанной в предыдущем разделе, возникают из-за разрешения ссылок на метод во время компиляции. Выход заключается в том, что метод должен быть динамическим, а ссылки на него должны разрешаться во время выполнения. Чтобы это стало возможным, нужно иметь некоторые специальные механизмы, однако Borland Pascal предоставляет эти механизмы за счет поддержки им виртуальных методов.

> *Важное замечание:* Виртуальные методы предоставляют максимально мощный инструмент для обобщения, именуемого полиморфизмом. Полиморфизм в переводе с греческого означает "многообразие" и является способом присвоения действию имени, которое разделяется вверх и вниз объектами иерархии, причем каждый объект иерархии использует это действие соответствующим ему образом.

Уже описанная простая иерархия крылатых насекомых являет собой хороший пример полиморфизма в действии, предоставляемого посредством виртуальных методов.

Каждый тип объекта в нашей иерархии представляет отдельный тип фигуры на экране: крылатое насекомое или пчелу. Определенно, имеет смысл сказать, что вы можете показать на экране точку или окружность. Позднее, если вам понадобится определить объекты для представления на экране других типов крылатых насекомых, таких как мотыльки, стрекозы, бабочки и т.д., вы могли бы написать метод для каждого из них, который будет выводить объект на экран. В новых терминах объектно-ориентированного программирования вы могли бы сказать, что все эти типы крылатых насекомых имеют способность показать самих себя на экране. Это большая часть из того, что является для них общим.

Что является особым для каждого типа объекта, так это способ, которым он должен показать самого себя на экране. Например, у пчелы на экране должны

рисоваться черные полоски на туловище. Можно показать на экране любой тип крылатых насекомых, но механизм рисования каждого является сугубо специфическим. Одно слово "нарисовать" используется для рисования (буквально) многих крылатых насекомых. Аналогично, если вернуться к нашему примеру платежной ведомости, то слово "GetPayAmount" вычисляет размер выплат для нескольких категорий работающих.

Это были примеры полиморфизма, а виртуальными методами является то, что реализует его в Borland Pascal.

Раннее связывание против позднего связывания

Различие между вызовом статического метода и динамического метода является различием между решением сделать немедленно и решением отложить. Когда вы кодируете вызов статического метода, вы по существу говорите компилятору; "Ты знаешь, чего я хочу. Пойди и вызови это." С другой стороны, применение вызова виртуального метода, подобно разговору с компилятором; "Ты не знаешь пока, чего я хочу. Когда придет время, задай вопрос о конкретном экземпляре."

Подумайте об этой метафоре в терминах проблемы MoveTo, упомянутой в предыдущем разделе. Вызов TBee.MoveTo может привести только к одному - выполнению MoveTo, ближайшей в объектной иерархии. В этом случае TBee.MoveTo по-прежнему будет вызывать определение MoveTo для TWinged, так как TWinged является ближайшим к TBee типом вверх по иерархии. Если предположить, что не определен никакой дочерний тип, который определяет собственный метод MoveTo, переопределяющий MoveTo типа TWinged, то любой порожденный по отношению к TWinged тип будет по-прежнему вызывать тот же самый экземпляр метода MoveTo. Решение может быть принято во время компиляции и это все, что должно быть сделано.

Однако совсем другое дело, когда метод MoveTo вызывает Show. Каждый тип фигуры имеет свой собственный экземпляр Show, поэтому то, какой экземпляр Show вызывается методом MoveTo, полностью зависит от того, какая реализация объекта вызывает MoveTo. Именно поэтому решение о вызове метода Show внутри экземпляра MoveTo должно быть отложено: при компиляции кода MoveTo не может принято никакого решения относительно того, какой метод Show должен быть вызван. Эта информация недоступна во время компиляции, поэтому решение должно быть отложено до тех пор, пока программа не начнет выполняться, и пока нельзя будет запросить экземпляр объекта, вызывающий MoveTo.

Процесс, с помощью которого вызовы статических методов однозначно разрешаются компилятором во время компиляции в один метод, называется ранним связыванием. При раннем связывании вызывающий и вызываемый методы связываются при первой же возможности, т.е. во время компиляции. При позднем связывании вызывающий и вызываемый методы не могут быть связаны во время компиляции, поэтому включается механизм, позволяющий осуществить связывание несколько позднее, когда вызов действительно произойдет.

Сущность механизма интересна и тонка, и немного позднее вы увидите, как он работает.

Совместимость типов объектов

Наследование до некоторой степени изменяет правила совместимости типов в Borland Pascal. Помимо всего прочего, порожденный тип наследует совместимость типов всех своих порождающих типов. Эта расширенная совместимость типов принимает три формы:

- между реализациями объектов;
- между указателями на реализации объектов;
- между формальными и фактическими параметрами.

Однако очень важно помнить, что во всех трех формах совместимость типов расширяется только от потомка к родителю. Другими словами, дочерние типы могут свободно использоваться вместо родительских, но не наоборот.

В модуле WORKERS.TPU TSalaried является потомком TEmployee, а TCommissioned - потомком TSalaried. Помня об этом, рассмотрим следующие описания:

```
type
  PEmployee = ^TEmployee;
  PSalaried = ^TSalfried;
  PCommissioned = ^TCommissioned;
var
  AnEmployee: TEmployee;
  ASalaried: TSalaried;
  PCommissioned: TCommissioned;
  TEmployeePtr: PEmployee;
  TSalariedPtr: PSalaried;
  TCommissionedPtr: PCommissioned;
```

При данных описаниях справедливы следующие операторы присваивания:

```
AnEmployee :=ASalaried;
ASalaried := ACommissioned;
TCommissionedPtr := ACommissioned;
```

> *Примечание:* Порождающему объекту можно присвоить экземпляр любого из его порожденных типов. Обратные присваивания недопустимы.

Эта концепция является новой для Паскаля, и в начале, возможно, вам будет трудновато запомнить, в каком порядке следует совместимость типов. Думайте следующим образом: источник должен быть в состоянии полностью заполнить приемник. Порожденные типы содержат все, что содержат их порождающие типы благодаря свойству наследования. Поэтому порожденный тип имеет либо в точности такой же размер, либо (что чаще всего и бывает) он больше своего родителя, но никогда не бывает меньше. Присвоение порождающего (родительского) объекта порожденному (дочернему) могло бы оставить некоторые поля порожденного объекта неопределенными, что опасно и поэтому недопустимо.

В операторах присваивания из источника в приемник будут копироваться только поля, являющиеся общими для обоих типов. В операторе присваивания:

```
AnEmployee := ACommissioned;
```

Только поля Name, Title и Rate из ACommissioned будут скопированы в AnEmployee, т.к. только эти поля являются общими для TCommissioned и TEmployee. Совместимость типов работает также между указателями на типы объектов и подчиняется тем же общим правилам, что и для реализаций объектов. Указатель на потомка может быть присвоен указателю на родителя. Если дать предыдущие определения, то следующие присваивания указателей будут допустимыми:

```
TSalariedPtr:= TCommissionedPtr;
TEmployeePtr:= TSalariedPtr;
TEmployeePtr:= PCommissionedPtr;
```

Помните, что обратные присваивания недопустимы.

Формальный параметр (либо значение, либо параметр-переменная) данного объектного типа может принимать в качестве фактического параметра объект своего же типа или объекты всех дочерних типов. Если определить заголовок процедуры следующим образом:

```
procedure CalcFedTax(Victim: TSalaried);
```

то допустимыми типами фактических параметров могут быть Tsalaried или TCommissioned, но не тип TEmployee. Victim также может быть параметром-переменной. При этом выполняются те же правила совместимости.

> *Замечание:* Имейте в виду, что между параметрами-значениями и параметрами-переменными есть коренное отличие. Параметр-значение является указателем на действительный, посылаемый в качестве параметра объект, тогда как параметр-переменная является только копией фактического параметра. Более того, эта копия включает только те поля, которые входят в тип формального параметра-значения. Это означает, что фактический параметр буквально преобразуется к типу формального параметра. Параметр-переменная больше напоминает приведение к образцу, в том смысле, что фактический параметр остается неизменным.

Аналогично, если формальный параметр является указателем на тип объекта, фактический параметр может быть указателем на этот тип объекта или на любой дочерний тип. Пусть дан заголовок процедуры:

```
procedure Worker.Add (AWorker: PSalaried);
```

тогда допустимыми типами фактических параметров могут быть PSalaried или PCommissioned, но не тип PEmployee.

Полиморфические объекты

При чтении предыдущего раздела вы, возможно, задали себе вопрос: "Если любой порожденный от типа параметра тип может передаваться в качестве параметра, то как же пользователь параметра узнает, какой тип объекта он получил?" Фактически, пользователь явно этого и не знает. Точный тип фактического параметра не известен во время компиляции. Фактический параметр может быть объектом любого дочернего от параметра-переменной типа, и именно поэтому он называется полиморфическим объектом.

Теперь, чем же именно хорош полиморфический объект? Прежде всего полиморфические объекты позволяют обрабатывать объекты, чей тип неизвестен на момент компиляции. Это общее замечание настолько ново для образа мышления Паскаля, что пример для вас не появится незамедлительно. (Со временем вы будете удивлены, насколько естественно это выглядит. То есть, когда вы действительно станете объектно-ориентированным программистом.)

Предположим, что вы написали инструментальное средство для вычерчивания графиков, поддерживающее многочисленные типы фигур: точки, окружности, квадраты, прямоугольники, кривые и т.д. В качестве части этого инструментального средства вы хотите написать программу, которая будет перемещать графическую фигуру по экрану с помощью устройства типа "мышь".

При старом способе необходимо было написать отдельную процедуру перемещения для каждого типа графической фигуры, поддерживаемой инструментальным средством. Вы должны были бы написать DragButterfly, DragBee, DragMoth и т.д. Несмотря на то, что строгая типизация (проверка типов) Паскаля позволяла это (и не забывайте, что всегда существуют способы обойти строгую типизацию), различия между типами графических фигур едва ли позволили бы написать действительно общую программу перемещения.

В конце концов, пчела имеет полоски и жало, бабочка имеет большие цветные крылья, а стрекоза имеет переливчатые цвета, хвост, да что говорить...

С этой точки зрения, "сообразительные" программисты, работающие на Турбо Паскале, сделают шаг вперед и скажут: "Поступайте так: передайте запись о крылатом насекомом процедуре DragIt в качестве ссылки указателя общего вида. В процедуре DragIt проверяйте свободное поле по фиксированному смещению внутри записи о крылатом насекомом для определения, какого вида это насекомое, а затем сделайте переход с помощью оператора case:

```
case FigureIDTag of
  Bee       : DragBee;
  Butterfly : DragButterfly;
  Dragonfly : DragDragonfly;
  Mocquito  : DragMocquito;
```

Ну, размещение семнадцати маленьких чемоданчиков внутри одного большого является незначительным шагом вперед, но в чем же заключается проблема, ожидающая нас на этом пути?

Что случится, если пользователь инструментального средства определит несколько новых типов крылатых насекомых?

В самом деле, что? Что если пользователь захочет работать со среднеазиатскими фруктовыми мухами? В вашей программе нет типа Fruitfly, поэтому DragIt не содержит метки Fruitfly в операторе case и, следовательно, отвергнет перемещение нового рисунка Fruitfly. Будучи представленным в процедуре DragIt, Fruitfly будет выпадать из оператора case в ветвь else этого оператора как "нераспознанное насекомое".

Откровенно говоря, создание для продажи инструментального средства без исходного кода страдает этой проблемой. Инструментальное средство может работать только с типами данных, которые "известны" ему, т.е. которые определены разработчиком инструментального средства. Пользователь инструментального средства оказывается бессильным перед расширением его функций в направлении, непредвиденном разработчиком. То, что пользователь купил, то он и получил. И точка.

Выходом из проблемы является использование правил расширенной совместимости типов Borland Pascal для объектов и разработка прикладных программ с использованием полиморфических методов. Если процедура DragIt инструментального средства установлена так, что может работать с полиморфическими объектами, то она будет работать с любыми объектами, определенными в инструментальном средстве, и с любыми дочерними объектами, которые вы определите сами. Если типы объектов инструментального средства используют виртуальные методы, то объекты и программы инструментального средства могут работать со сделанными вами графическими фигурами в собственных терминах самих фигур. Определенный вами сегодня виртуальный метод может вызываться файлом модуля (.TPU, .TPW или .TPP) инструментального средства, который был написан и оттранслирован год назад. Объектно-ориентированное программирование дает такую возможность, а виртуальные методы являются ключом к ней.

Понимание того, как виртуальные методы делают возможными такие вызовы полиморфических методов требует пояснения описания и использования виртуальных методов.

Виртуальные методы

Метод становится виртуальным, если за его объявлением в типе объекта стоит новое зарезервированное слово virtual. Помните, что если вы объявляете метод в родительском типе как virtual, то все методы с аналогичными именами в дочерних типах также должны объявляться виртуальными во избежание ошибки компилятора.

Ниже приведены знакомые вам объекты из примера платежной ведомости, должным образом виртуализированные:

```
type
```

```
  PEmployee = ^TEmployee;
  TEmployee = object
    Name, Title: string[25];
    Rate: Real;
  constructor Init (AName, ATitle: String; ARate: Real);
  function GetPayAmount : Real; virtual;
  function GetName : String;
  function GetTitle : String;
  function GetRate : Real;
  procedure Show; virtual;
  end;

  PHourly = ^THourly;
  THourly = object(TEmployee);
    Time: Integer;
  constructor Init (AName, ATitle: String; ARate: Real;Time: Integer);
  function GetPayAmount : Real; virtual;
  function GetTime : Integer;
  end;

  PSalaried = ^TSalaried;
  TSalaried = object(TEmployee);
  function GetPayAmount : Real; virtual;
  end;

  PCommissioned = ^TCommissioned;
  TCommissioned = object(Salaried);
    Commission : Real;
    SalesAmount : Real;
  constructor Init(AName,ATitle:String;
                      ARate,ACommission,ASalesAmount:Real);
  function GetPayAmount : Real; virtual;
  end;
```

А ниже приводится пример для насекомых, дополненный виртуальными методами.

```
type
  TWinged = object(Insect)
  constructor Init (AX, AY : Integer)
  procedure Show; virtual;
  procedure Hide; virtual;
  end;
type
  TBee = object(TWinged)
  constructor Init (AX, AY : Integer)
  procedure Show; virtual;
```

```
    procedure Hide; virtual;
    end;
```

Прежде всего обратите внимание, что метод MoveTo, показанный для типа TBee, теперь удален из его определения. Теперь типу TBee больше нет нужды переопределять метод MoveTo типа TWinged с помощью немодифицируемой копии, компилируемой в его собственной области действия. Вместо этого, MoveTo теперь может наследоваться от TWinged со всеми вложенными в MoveTo вызовами, которые, однако, будут вызывать методы из TBee, а не из TWinged, как это происходило в полностью статической иерархии объектов.

Отметьте также новое зарезервированное слово constructor (конструктор), заменившее зарезервированное слово procedure для TWinged.Init и TBee.Init. Конструктор является специальным типом процедуры, которая выполняет некоторую установочную работу для механизма виртуальных методов. Более того, конструктор должен вызываться перед вызовом любого виртуального метода. Вызов виртуального метода без предварительного вызова конструктора может привести к блокированию системы, а у компилятора нет способа проверить порядок вызова методов.

Каждый тип объекта, имеющий виртуальные методы, обязан иметь конструктор.

> *Предупреждение:* Конструктор должен вызываться перед вызовом любого другого виртуального метода. Вызов виртуального метода без предыдущего обращения к конструктору может вызвать блокировку системы, и компилятор не сможет проверить порядок, в котором вызываются методы.

> *Примечание:* Для конструкторов объекта мы предлагает использовать идентификатор Init.

Каждый отдельный экземпляр объекта должен инициализироваться отдельным вызовом конструктора. Недостаточно инициализировать один экземпляр объекта и затем присваивать этот экземпляр другим. Другие экземпляры, даже если они могут содержать правильные данные, не будут инициализированы оператором присваивания и заблокируют систему при любых вызовах их виртуальных методов. Например:

```
var
  FBee, GBee: Bee;    { создать два экземпляра Bee }
begin
  FBee.Init(5, 9)     { вызов конструктора для FBee }
  GBee := FBee;       { Gbee недопустим! }
end;
```

Что же именно создает конструктор? Каждый тип объекта содержит нечто, называемое таблицей виртуального метода (ТВМ) в сегменте данных. ТВМ содержит размер типа объекта и для каждого виртуального метода указатель на

код, выполняющий данный метод. Конструктор устанавливает связь между вызывающей его реализацией объекта и ТВМ типа объекта.

Важно помнить, что имеется только одна ТВМ для каждого типа объекта. Отдельные экземпляры типа объекта (т.е. переменные этого типа) содержат только соединение с ТВМ, но не саму ТВМ. Конструктор устанавливает значение этого соединения в ТВМ. Именно благодаря этому вы нигде не можете запустить выполнение перед вызовом конструктора.

Проверка диапазонов при вызове виртуальных методов

В процессе разработки программы вам, возможно, захочется повысить меры безопасности, которая снижается из-за вызовов виртуальных методов Borland Pascal. Если директива $R находится во включенном состоянии, {$R+}, то все вызовы виртуальных методов будут проверяться на состояние инициализации для выполняющих вызовы реализаций. Если выполняющая вызов реализация еще не была инициализирована конструктором, то произойдет ошибка проверки диапазона исполняющей системы.

Примечание: Состоянием по умолчанию является {$R-}.

После того, как вы хорошенько перетрясли программу и удостоверились, что отсутствуют вызовы методов из неинициализированных реализаций, вы можете до некоторой степени ускорить выполнение программы путем переключения директивы $R в пассивное состояние. После этого проверка вызовов методов из неинициализированных реализаций осуществляться не будет, что оставляет вероятность блокировки системы, если будет выявлена такая ошибка.

Примечание: Виртуальный однажды - виртуальный всегда

Вы уже вероятно обратили внимание, что как TWinged, так и TBee содержат методы, называемые Show и Hide. Все заголовки методов для Show и Hide объявлены виртуальными и снабжены зарезервированным словом virtual. Как только родительский тип объекта объявляет метод виртуальным, все его потомки также должны объявить этот метод виртуальным. Другими словами, статический метод никогда не сможет переопределить виртуальный метод. Если вы попытаетесь сделать это, то компилятор выдаст сообщение об ошибке.

Также следует помнить, что после того, как метод стал виртуальным, его заголовок не может изменяться в объектах более низкого уровня иерархии. Вы можете представлять себе каждое определение виртуального метода как ворота для всех их. Исходя из этих соображений, заголовки всех реализаций одного и того же виртуального метода должны быть идентичными, включая число параметров и их типы. Это не относится к статическим методам: статический метод, переопределяющий другой, может иметь отличное число параметров и типы этих параметров, в зависимости от необходимости.

Это целый новый мир.

Расширяемость объекта

Важным замечанием, касающимся модулей типа WORKERS.PAS, является то, что типы объектов и методы, определенные в модуле, могут поставляться пользователю в форме .TPU, .TPW или .TPP т.е. в форме, способной к непосредственной компоновке, без исходного кода. (Нужно просмотреть только листинг интерфейсной части модуля.). Используя полиморфические объекты и виртуальные методы, пользователь файла .TPU, .TPW или .TPP сможет добавлять характеристики для приспособления модуля к своим нуждам.

Новое понятие о добавлении функциональных характеристик в программу без предоставления ее исходного кода называется способностью к расширению. Способность к расширению является естественным следствием наследования: вы наследуете все, чем обладают порождающие типы, а затем добавляете новые нужные вам возможности. Позднее связывание позволяет, чтобы новое связывалось со старым во время выполнения программы, благодаря чему расширение существующего кода выглядит "бесшовным" и стоит вам в части выполнения не более, чем быстрое путешествие по таблице виртуального метода.

Статические методы или виртуальные методы?

В общем случае, вам следует делать методы виртуальным. Используйте статические методы только в том случае, если вы хотите получить оптимальную эффективность скорости выполнения и использования памяти. Однако в этом случае, как вы видели, вы теряете возможности расширения.

Предположим, что вы описываете объект с именем Ancestor и внутри этого объекта вы описываете метод с именем Action. Как вы определяете, каким должен быть метод, виртуальным или статическим? Здесь приводится правило большого пальца: сделайте метод Action виртуальным, если имеется вероятность, что будущие наследники объекта Ancestor будут переопределять Action, а вы хотите, чтобы будущий код был доступен Ancestor.

С другой стороны, помните, что если у объекта имеются любые виртуальные методы, то для этого объекта в сегменте данных будет создана таблица виртуальных методов (ТВМ) и каждый экземпляр этого объекта будет иметь связь с ТВМ. Каждый вызов виртуального метода должен проходить через ТВМ, тогда как статические методы вызываются непосредственно. Хотя просмотр ТВМ является весьма эффективным, вызов статического метода все равно остается немного более быстрым, чем вызов виртуального. И если в вашем объекте нет виртуальных методов, то и ТВМ отсутствует в сегменте данных и (что более важно) в каждом экземпляре объекта отсутствуют связи с ТВМ.

Дополнительная скорость и эффективное использование памяти для статических методов должно уравновешиваться гибкостью, которую допускают виртуальные методы: вы можете расширить имеющийся код спустя много времени после его компиляции. Помните, что пользователь вашего типа объекта может рассматривать пути его использования, которые вам и не снились, что, в конечном счете, имеет основное значение.

Динамические объекты

Все приведенные до сих пор объекты имели статические реализации типов объектов, которым в объявлении var присваивались имена и которые размещались в сегменте данных или в стеке.

```
var
  ASalaried: TSalaried;
```

> *Примечание:* Использование здесь слова "статический" не имеет отношения к статическим методам.

Объекты могут размещаться в динамической памяти и ими можно манипулировать с помощью указателей, как и с тесно связанными с ними типами записей, что всегда имело место в Паскале. Турбо Паскаль включает несколько мощных расширений для выполнения динамического размещения и удаления объектов более легкими и более эффективными способами.

Объекты могут размещаться, как области памяти, на которые ссылается указатель, с помощью процедуры New:

```
var
  CurrentPay: Real;
  P: ^TSalaried;

New(P);
```

Как и для типов записей, процедура New выделяет в динамической памяти пространство, достаточное для размещения реализации указателя базового типа и возвращает адрес этого пространства в указателе.

Если динамический объект содержит виртуальные методы, то он должен инициализироваться с помощью вызова конструктора перед тем, как будет вызван любой из его методов:

```
P^.Init('Sara Adams', 'Account manager', 2400);
```

Затем вызовы методов могут происходить в обычном порядке, с использованием имени указателя и ссылочного символа вместо имени реализации, которое использовалось бы при обращении к статически размещенному объекту:

```
CurrentPay := P^.GetPayAmount;
```

Размещение и инициализация с помощью процедуры New

Borland Pascal расширяет синтаксис процедуры New, что является более компактным и более удобным средством выделения пространства для объекта в динамически распределяемой области памяти и инициализации объекта с помощью только одной операции. Теперь процедура New может вызываться с двумя

параметрами: имя указателя используется в качестве первого параметра, а вызов конструктора - в качестве второго параметра:

```
New(P, Init('Sara Adams', 'Account manager', 2400));
```

Если для процедуры New используется расширенный синтаксис, то конструктор Init действительно выполняет динамическое размещение, используя специальный код входа, сгенерированного как часть компиляции конструктора. Имя реализации не может предшествовать Init, т.к. в то время, когда процедура New вызвана, реализация, инициализируемая с помощью Init, еще не существует. Компилятор идентифицирует правильный вызываемый метод Init посредством типа указателя, пересылаемого в качестве первого параметра.

Процедура New также была расширена для возможности использо вания ее как функции, которая возвращает значение указателя. Посылаемый New параметр является типом указателя на объект, а не самой переменной-указателем:

```
type
  PSalaried = ^TSalaried;

var
  P: PSalaried;

P := New(PSalaried);
```

Обратите внимание, что в данной версии функциональная форма расширения процедуры New применима ко всем типам данных, а не только к типам объектов.

Функциональная форма New, как и процедурная форма, также может воспринимать конструктор объектного типа в качестве второго параметра:

```
P := New(PSalaried, Init('Sara Adams', 'Account manager', 2400));
```

В Borland Pascal осуществлено также параллельное расширение процедуры Dispose, это подробно обсуждается в следующем разделе.

> *Примечание:* Новая стандартная процедура Fail поможет вам в конструкторах выполнить восстановление при ошибке.

Удаление динамических объектов

Также, как и обычные записи Паскаля, размещаемые в динамически распределяемой области памяти, объекты могут удаляться процедурой Dispose, если они больше не нужны:

```
Dispose (P);
```

Однако, при избавлении от ненужного объекта может понадобиться нечто большее, чем простое освобождение занимаемой им динамической памяти. Объект может содержать указатели на динамические структуры или объекты, которые нужно освободить или очистить в определенном порядке, особенно если вы оперируете сложной динамической структурой данных. Что бы ни нужно было сделать для очистки динамического объекта в каком-либо порядке, это все должно быть объединено в один метод таким образом, чтобы объект мог быть уничтожен с помощью одного вызова метода:

```
MyComplexObject.Done;
```

Метод Done должен инкапсулировать все детали очистки своего объекта, а также всех структур данных и вложенных объектов.

> *Примечание:* Мы советуем использовать для удаления методов, работающих с объектами, которые более не нужны, использовать идентификатор Done.

Допустимо и часто бывает полезно определять несколько методов очистки для данного типа объекта. В зависимости от того, как они размещены или используются, или в зависимости от состояния и режима объекта на момент очистки, сложные объекты могут потребовать очистки несколькими различными путями.

Деструкторы

Borland Pascal предоставляет специальный тип метода, называемый "сборщиком мусора" или деструктором, для очистки и удаления динамически размещенного объекта. Деструктор объединяет шаг удаления объекта с какими-либо другими действиями или задачами, необходимыми для данного типа объекта. Для единственного типа объекта можно определить несколько деструкторов.

Деструктор определяется совместно со всеми другими методами объекта в определении типа объекта:

```
type
  TEmployee = object
    Name: string[25];
    Title: string[25];
    Rate: Real;
  constructor Init(AName, ATitle: String; ARate: Real);
  destructor Done; virtual;
  function GetName: String;
  function GetTitle: String;
  function GetRate: Rate; virtual;
  function GetPayAmount: Real; virtual;
end;
```

Деструкторы можно наследовать, и они могут быть либо статическими, либо виртуальными. Поскольку различные программы завершения обычно требуют различные типы объектов, мы рекомендуем, чтобы деструкторы всегда были виртуальными, благодаря чему для каждого типа объекта будет выполнен правильный деструктор.

Запомните, что зарезервированное слово destructor не требуется указывать для каждого метода очистки, даже если определение типа объекта содержит виртуальные методы. Деструкторы в действительности работают только с динамически размещенными объектами. При очистке динамически размещенного объекта, деструктор осуществляет специальные функции: он гарантирует, что в динамически распределяемой области памяти всегда будет освобождаться правильное число байтов. Не может быть никаких опасений по поводу использования деструктора применительно к статически размещенным объектам; фактически, не передавая типа объекта деструктору, вы лишаете объект данного типа полных преимуществ управления динамической памятью в Borland Pascal.

Деструкторы в действительности становятся самими собою тогда, когда должны очищаться полиморфические объекты и когда должна освобождаться занимаемая ими память. Полиморфические объекты - это те объекты, которые были присвоены родительскому типу благодаря правилам совместимости расширенных типов Borland Pascal. Экземпляр объекта типа THourly, присвоенный переменной типа TEmployee, является примером полиморфического объекта. Эти правила также могут быть применены к объектам; указатель на THourly может свободно быть присвоен указателю на TEmployee, а указуемый этим указателем объект опять же будет полиморфическим объектом.

Термин "полиморфический" является подходящим, так как код, обрабатывающий объект, не знает точно во время компиляции, какой тип объекта ему придется, в конце концов, обработать. Единственное, что он знает, это то, что этот объект принадлежит иерархии объектов, являющихся потомками указанного типа объекта.

Очевидно, что размеры типов объектов отличаются. Поэтому, когда наступает время очистки размещенного в динамической памяти полиморфического объекта, то как же Dispose узнает, сколько байт динамического пространства нужно освобождать? Во время компиляции из полиморфического объекта нельзя извлечь никакой информации относительно размера объекта.

Деструктор разрешает эту головоломку путем обращения к тому месту, где эта информация записана: в ТВМ переменных реализаций. В каждой ТВМ типа объекта содержится размер в байтах данного типа объекта. Таблица виртуальных методов любого объекта доступна посредством скрытого параметра Self, посылаемого методу при вызове метода. Деструктор является всего лишь разновидностью метода и поэтому, когда объект вызывает его, деструктор получает копию Self через стек. Таким образом, если объект является полиморфическим во время компиляции, он никогда не будет полиморфическим во время выполнения благодаря позднему связыванию.

Для выполнения этого освобождения памяти при позднем связывании деструктор нужно вызывать, как часть расширенного синтаксиса процедуры Dispose:

```
Dispose(P, Done);
```

Примечание: Вызов деструктора вне процедуры Dispose вообще не выполняет никакого освобождения памяти.

Здесь происходит на самом деле то, что сборщик мусора объекта, на который указывает P, выполняется как обычный метод. Однако, как только последнее действие выполнено, деструктор ищет размер реализации своего типа в ТВМ и пересылает размер процедуре Dispose. Процедура Dispose завершает процесс путем удаления правильного числа байт пространства динамической памяти, которое (пространство) до этого относилось к P^. Число освобождаемых байт будет правильным независимо от того, указывал ли P на экземпляр типа TSalaried, или он указывал на один из дочерних типов типа TSalaried, например, на TCommissioned.

Заметьте, что сам по себе метод деструктора может быть пуст и может выполнять только эту функцию:

```
destructor AnObject.Done;
begin
end;
```

То, что делается полезного в этом деструкторе, не является достоянием его тела, однако при этом компилятором генерируется код эпилога в ответ на зарезервированное слово destructor. Это напоминает модуль, который ничего не экспортирует, но который осуществляет некоторые невидимые действия за счет выполнения своей секции инициализации перед стартом программы. Все действия происходят "за кулисами".

Пример размещения динамического объекта

Последний пример программы даст вам возможность приобрести некоторые навыки в использовании размещенных в динамической памяти объектов, включая использование для удаления объекта деструктора. Программа показывает, как в динамической памяти может быть создан связанный список рабочих объектов и как он за ненадобностью может быть очищен при помощи деструктора.

Построение связанного списка объектов требует, чтобы каждый объект содержал указатель на следующий объект списка. Тип TEmployee не содержит таких указателей. Простым выходом из этой ситуации было бы добавление указателя в TEmployee , благодаря чему можно быть уверенным, что все потомки TEmployee наследуют такой указатель. Однако, добавление чего-либо в TEmployee требует от вас наличия исходного кода, а как говорилось ранее, одним из преимуществ объектно-ориентированного программирования является возможность расширения объектов без необходимости их перекомпиляции.

Решение, которое не требует никаких изменений TEmployee, создает новый тип объекта, не являющийся потомком TEmployee. Тип StaffList представ-

ляет собой очень простой объект, целью которого является создание заголовков для объектов типа TEmployee. Так как TEmployee не содержит никаких указателей на следующий объект в списке, то простой тип записи TNode осуществляет этот сервис. TNode даже проще, чем StaffList в том, что TNode не является объектом, не содержит ни одного метода и не имеет никаких данных, за исключением указателя на тип TEmployee и указателя на следующий узел списка.

TStaffList содержит метод, который позволяет ему добавлять нового рабочего в связанный список записей TNode путем внесение нового экземпляра TNode непосредственно после самого себя в качестве указуемого с помощью указателя поля TNodes. Метод Add принимает указатель на объект типа TEmployee, но не сам объект. Из-за расширенной совместимости типов Турбо Паскаля указатели на любого потомка типа TEmployee также должны передаваться в TList.Add в параметре Item.

Программа WorkList описывает статическую переменную Staff типа TStaffList и строит связанный список из пяти узлов. Каждый узел указывает на отдельный рабочий объект, который является либо TEmployee, либо одним из его потомков. Перед созданием каждого динамического объекта и после того, как объект создан, возвращает число байт свободной динамической памяти. Наконец, полная структура, включающая пять записей TNode и пять объектов типа TEmployee, очищается и удаляется из динамической памяти с помощью одного вызова деструктора статического объекта Staff типа TStaffList.

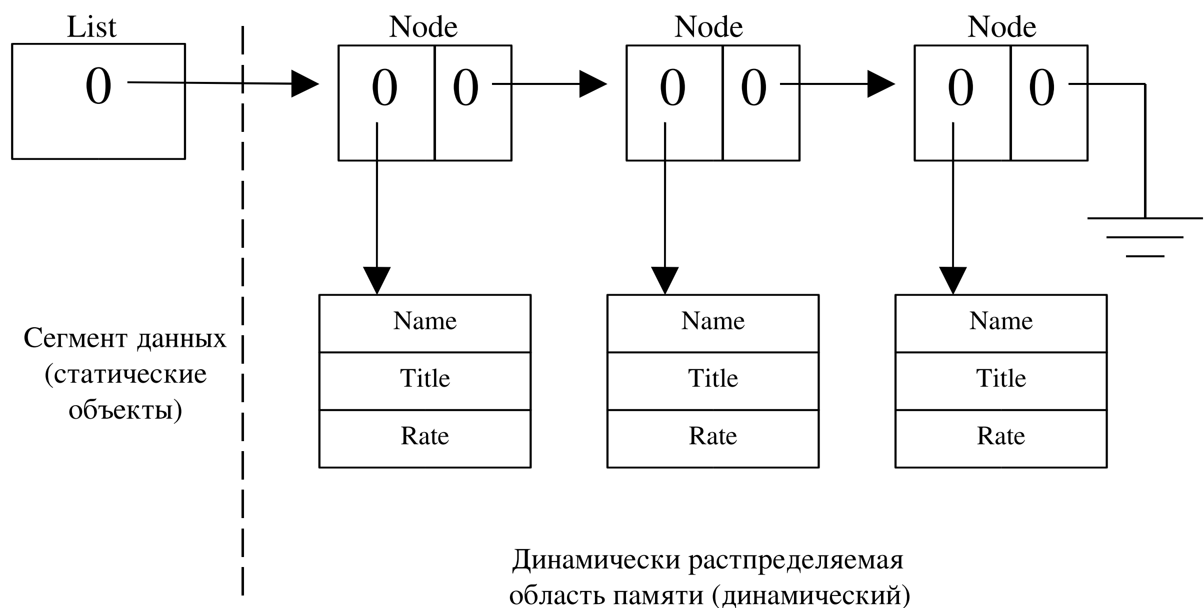

Рис. 2 Схема структур данных программы ListDemo.

Удаление сложной структуры данных из динамической памяти

Деструктор Staff.Done стоит того, чтобы рассмотреть его внимательно. Уничтожение объекта TStaffList включает удаление трех различных типов структур: полиморфических объектов рабочих структур в списке, записей TNode, поддерживающих список, и (если он размещен в динамической памяти)

объект TList, который озаглавливает список. Весь процесс запускается путем единственного вызова деструктора объекта TStaffList:

```
Staff.Done;
```

Код деструктора заслуживает более подробного изучения:

```
destructor StaffList.Done;
var
  N: TNodePtr;
begin
  while TNodes <> nil do
  begin
    N := TNodes;
    Dispose(N^.Item, Done);
    TNodes:= N^.Next;
    Dispose (N);
  end;
end;
```

Список очищается начиная с "головы" списка с помощью алгоритма "из руки в руку", который до некоторой степени напоминает дерганье за веревку воздушного змея: два указателя (указатель TNodes внутри Staff и рабочий указатель N) изменяют свои ссылки в списке, тогда как первый элемент списка удаляется. Вызов процедуры Dispose освобождает память, занимаемую первым объектом TEmployee в списке (Item^), затем TNodes продвигается на следующую запись списка с помощью оператора TNodes := N^.Next, сама запись TNode удаляется, и процесс продолжается до полного очищения списка.

Важным моментом в деструкторе Done является способ, которым удаляются из списка объекты TEmployee:

```
Dispose(N.Item, Done);
```

Здесь N.Item является первым объектом TEmployee в списке, а вызываемый метод Done является деструктором этого объекта. Запомните, что действительный тип N^.Item^ не обязательно является типом TEmployee, однако он может быть любым дочерним типом типа TEmployee. Очищаемый объект является полиморфическим и поэтому нельзя сделать никаких предположений относительно его действительного размера или точного его типа на этапе компиляции. В приведенном выше вызове Dispose, как только Done выполнит все содержащиеся в нем операторы, "невидимый" код эпилога ищет размер реализации очищаемого объекта в ТВМ этого объекта. Метод Done передает размер процедуре Dispose, которая затем освобождает точное количество динамической памяти, в действительности занимаемой полиморфическим объектом.

Помните, что если должно освобождаться правильное количество динамической памяти, то полиморфический объект должен очищаться только посредством вызова передаваемого Dispose деструктора. В примере программы

Staff объявляется как статическая переменная в сегменте данных. Staff мог бы столь же легко разместиться в динамической памяти и "прикрепиться к реальному миру" посредством указателя типа ListPtr. Если заголовок списка также является динамическим объектом, то удаление структуры можно осуществить путем вызова деструктора, выполняющегося внутри Dispose:

```
var
  Staff: TStaffListPtr;
Begin
  Dispose(Staff, Done);
```

Здесь процедура Dispose вызывает метод деструктора Done для очистки структуры в динамической памяти. Затем, когда Done завершается, Dispose освобождает память, на которую указывает Staff, удаляя, как правило, из динамической памяти также и заголовок списка.

Программа WORKLIST.PAS (находящаяся на вашем диске) использует тот же модуль WORKERS.PAS, что и раньше Она создает объект List, являющийся оглавлением связанного списка из пяти полиморфических объектов, совместимых с TEmployee, а затем удаляет всю динамическую структуру данных с помощью единственного вызова деструктора Staff.Done.

Что же дальше?

Как и во всяком другом аспекте машинного программирования, вы не преуспеете в объектно-ориентированном программировании, если будете только читать о нем, но вы добьетесь результата, если начнете программировать. Большинство людей, при первом столкновении с объектно-ориентированном программированием, начинают бормотать с придыханием; "Я не могу постичь этого". "Ага!" приходит позднее, ночью, когда целостная концепция является к нам в одно прекрасное мгновение, и мы, побросав свои никчемные дела, используем это мгновение для обращения к богу. Как лицо женщины, возникающее из чернильных пятен Роша, то, что до этого было смутным, становится очевидным и затем легким.

Самое лучшее, что вы можете сделать в качестве первого шага в объектно-ориентированном программировании, так это взять модуль WORKERS.PAS (он находится на вашем диске) и расширить его. Как только вы воскликните "Ага!", начинайте строить ориентированные на объекты концепции в вашей повседневной программистской жизни. Возьмите несколько имеющихся утилит, которые вы используете каждый день, и переосмыслите их в ориентированных на объекты терминах. Посмотрите критически на "овощное рагу" вашей библиотеки процедур и попытайтесь найти в них объекты, затем перепишите процедуры в объектной форме. Вы убедитесь, что библиотеки объектов станет намного легче использовать в будущих проектах. Даже самые незначительные ваши начальные инвестиции в программные усилия станут навсегда излишними. У вас едва ли возникнет необходимость переписывать объект с самого начала. Если он работает как надо, то используйте его. Если объекту чего-либо не хватает, то

расширьте его. Но если он работает хорошо, то нет смысла выбрасывать из него что-либо.

Заключение

Объектно-ориентированное программирование является прямым следствием усложнения современных приложений, усложнения, которое часто заставляет многих программистов в отчаянии вскидывать вверх руки. Наследование и инкапсуляция являются максимально эффективными средствами для управления сложностью. (Существует разница между десятью тысячами насекомых, классифицированных по таксономической схеме, и десятью тысячами насекомых, жужжащих возле ваших ушей.) Представляя собой значительно большее, чем просто структурное программирование, объектно-ориентированное программирование вносит рациональный порядок в структуру программного обеспечения ЭВМ, что, как и таксономическая схема, устанавливает порядок, не устанавливая пределов.

Добавьте сюда перспективы возможности расширения и повторного использования существующего кода и все это начнет звучать настолько хорошо, что будет походить на правду. Вы думаете, что это невозможно?

Но это же Borland Pascal! Слово "невозможно" в нем не определено.