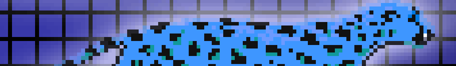


Справочное руководство Free Pascal

Справочное руководство для Free Pascal,
Версия 3.0.0
Версия документа 3.0

Ноябрь 2015 (оригинал)
(перевод на русский язык) Не окончен

Авторы:
Michaël Van Canneyt

Переводчики:
Google переводчик
Белькевич Дмитрий Васильевич
Чигрин Виталий Николаевич (адаптировал и переработал)
Использована более ранняя попытка перевести официальную документацию (к сожалению неизвестны имёна переводчиков)

© GNU

Содержание

Глава Про это руководство ... 7

Обозначения ... 7

Синтаксические диаграммы ... 8

Глава О языке Паскаль ... 9

Глава Глава 1 Элементы языка паскаль ... 10

1.1 Символы ... 10

1.2 Комментарии ... 11

1.3 Зарезервированные слова ... 12

1.3.1 Зарезервированные слова Turbo Pascal ... 12

1.3.2 Зарезервированные слова Free Pascal ... 13

1.3.3 Зарезервированные слова Object Pascal ... 13

1.3.4 Модификаторы ... 14

1.4 Идентификаторы ... 15

1.5 Директивы подсказок ... 16

1.6 Числа ... 17

1.7 Метки ... 18

1.8 Символьные строки ... 19

Глава Глава 2 Константы ... 21

2.1 Обычные константы ... 21

2.2 Типизированные константы ... 22

2.3 Строковые ресурсы ... 23

Глава Глава 3 Типы ... 25

3.1 Базовые типы ... 25

3.1.1 Перечислимые типы ... 26

Целые типы ... 26

Булевы типы ... 28

Типы перечислений ... 29

Типы диапазоны ... 31

3.1.2 Вещественные типы ... 32

3.2 Символьные типы ... 32

3.2.1 Символ (Char или AnsiChar) ... 32

3.2.2 WideChar ... 33

3.2.3 Другие символьные типы ... 33

3.2.4 Однобайтовые строковые тип ... 34

Короткие строки (ShortStrings) ... 34

Строки AnsiString ... 35

Преобразование кодовой страницы ... 38

Необработанная строка байт (Raw ByteString) ... 40

UTF8String ... 40

3.2.5 Многобайтные строковые типы ... 40

Строки Unicode (UnicodeStrings) ... 41

Большие строки (WideStrings) ... 41

3.2.6 Строковые константы (Constant strings) ... 41

3.2.7 PChar - строки завершённые нулём ... 42

3.2.8 Размеры строк ... 43

3.3 Структурированные Типы ... 44

Упакованные структурированные типы ... 44

3.3.1 Массивы ... 46

Статические массивы ... 46
Динамические массивы ... 48
Типы совместимые с Динамическими массивами ... 51
Конструктор Динамического массива ... 52
Упаковка и распаковка массивов ... 53
3.3.2 Записи ... 54
Структура и размер Записи ... 55
Замечания и примеры ... 57
3.3.3 Множества ... 57
3.3.4 Файловый тип ... 58
3.4 Указатели ... 59
3.5 Предварительное описания типа ... 61
3.6 Процедурный тип ... 62
3.7 Тип данных Variant ... 66
3.7.1 Определение ... 66
3.7.2 Вариантные переменные в присвоениях и выражениях ... 68
3.7.3 Варианты и интерфейсы ... 69
3.8 Псевдоним типа ... 69
Глава Глава 4 Переменные ... 72
4.1 Определение ... 72
4.2 Объявление ... 72
4.3 Область видимости(контекст) ... 74
4.4 Инициализированные переменные ... 74
4.5 Инициализация переменных (по умолчанию) ... 75
4.6 Потоконезависимые переменные ... 77
4.7 Свойства (Properties) ... 77
Глава Глава 5 Объекты ... 81
5.1 Объявление ... 81
5.2 Поля ... 83
5.3 Статические поля илм поля классов ... 84
5.4 Конструкторы и деструкторы ... 85
5.5 Методы ... 87
5.5.1 Объявление ... 87
5.5.2 Вызов метода ... 88
Статические методы ... 88
Виртуальные методы ... 89
Абстрактные методы ... 91
Методы класса и статические методы ... 92
5.6 Видимость ... 93
Глава Глава 6 Классы ... 95
6.1 Определения классов ... 95
6.2 Обычные и статические поля ... 99
6.2.1 Обячные поля/переменные ... 100
6.2.2 Переменняе/поля класса ... 101
6.3 Экземпляр класса ... 102
6.4 Уничтожение класса ... 102
6.5 Методы ... 103
6.5.1 Объявление ... 104
6.5.2 Вызов ... 104
6.5.3 Виртуальные методы ... 104
6.5.4 Методы класса ... 106

6.5.5 Конструктор и деструктор класса ... 107
6.5.6 Стасический метод класса ... 108
6.5.7 Методы обработки сообщений ... 110
6.5.8 Использование наследования ... 112
6.6 Свойства ... 114
6.6.1 Определение ... 114
6.6.2 Индексированные свойства ... 117
6.6.3 Массив свойств ... 118
6.6.4 Свойства по умолчанию ... 119
6.6.5 Публикуемые (Published) свойства ... 120
6.6.6 Сохраняемая информация ... 120
6.6.7 Переопределение свойств ... 121
6.7 Свойства класса ... 122
6.8 Вложенные типы, константы и переменные ... 123
Глава Глава 7 Интерфейсы ... 125
7.1 Определение ... 125
7.2 Идентификация интерфейса: GUID ... 126
7.3 Реализация интерфейса ... 127
7.4 Делегация Интерфейса ... 128
7.5 Интерфейсы и COM ... 130
7.6 CORBA и другие интерфейсы ... 130
7.7 Подсчет ссылок ... 131
Глава Глава 8. Дженерики ... 132
8.1 Введение ... 132
8.2 Определение дженерика классов ... 132
8.3 Специализация дженерика класса ... 136
8.4 Ограничения дженериков ... 137
8.5 Совместимость с Delphi ... 139
8.5.1 Элементы синтаксиса ... 139
8.5.2. Ограничения для записей ... 140
8.5.3 Перегрузка типов ... 141
8.5.4 Соглашение о пространствах имен ... 141
8.5.5 Соглашение об области действия ... 141
8.6 Совместимость типов ... 142
8.7 Инициализация по умолчанию ... 145
8.8 Несколько слов об области действия ... 145
8.9 Перегрузка операторов и дженерики ... 148
Глава Глава 9 Расширенные записи ... 150
9.1 Описание ... 150
9.2 Энумераторы расширенной записи ... 152
Глава Глава 10 Хелперы для классов, записей и типов ... 155
10.1 Определение ... 155
10.2 Ограничения для классов хелперов ... 156
10.3 Ограничения на хелперы записей ... 157
10.4 Особенности хелперов простых типов ... 158
10.5 Замечание по видимости и времени жизни хелперов записей и типов ... 160
10.6 Наследование ... 162

10.7 Использование ... 162

Глава Глава 11 Классы Objective-Pascal ... 166

11.1 Введение ... 166
11.2 Объявление классов Objective-Pascal ... 166
11.3 Формальное объявление ... 169
11.4 Распределение и освобождение экземпляров ... 171
11.5 Определения протокола ... 172
11.6 Категории ... 173
11.7 Пространство имён и идентификаторы ... 175
11.8 Селекторы ... 175
11.9 Тип id ... 176
11.10 Перечисления в классах Objective-C ... 176

Глава Глава 12 Выражения ... 178

12.1 Синтаксис выражений ... 179
12.2 Вызов функций ... 180
12.3 Конструкторы множеств ... 183
12.4 Приведение типов значений ... 183
12.5 Приведения типов переменной ... 184
12.6 Приведение невыровненых типов ... 185
12.7 Оператор @ ... 186
12.8 Операторы ... 187
12.8.1 Арифметические операторы ... 187
12.8.2 Поразрядные логические операторы ... 188
12.8.3 Логические операторы (однобитовые) ... 189
12.8.4 Строковый оператор ... 189
12.8.5 Операторы действий над множествами ... 190
12.8.6 Операторы отношения ... 192
12.8.7 Операторы действий над классами ... 193

Глава Глава 13 Операторы ... 196

13.1 Простые операторы ... 196
13.1.1 Оператор присвоения ... 196
13.1.2 Оператор вызова процедуры ... 198
13.1.3 Оператор Goto ... 198
13.2 Структурные операторы ... 199
13.2.1 Составной оператор ... 200
13.2.2 Оператор Case ... 201
13.2.3 Оператор If..then..else ... 202
13.2.4 Оператор For..to/downto..do ... 204
13.2.5 Оператор For..in..do ... 205
13.2.6 Оператор Repeat..until ... 213
13.2.7 Оператор While..do ... 214
13.2.8 Оператор With ... 215
13.2.9 Операторы Исключения ... 217
13.3 Оператор Asm ... 217

Глава Глава 14 Использование функций и процедур ... 218

14.1 Объявление процедуры ... 218
14.2 Объявление функции ... 219
14.3 Результат функции ... 219

14.4 Список параметров ... 220
14.4.1 Параметры-значения ... 220
14.4.2 Параметры-переменные ... 222
14.4.3 Выходные (Out) параметры ... 223
14.4.4 Параметры-константы ... 224
14.4.5 Параметр- открытый массив ... 226
14.4.6 Массив констант ... 228
14.5 Управляение типами со счетчиком ссылок ... 230
14.6 Перегрузка функций ... 235
14.7 Forward объявление подпрограмм ... 235
14.8 Внешние (external) функции ... 237
14.9 Функции на ассемблере ... 238
14.10 Модификаторы ... 238
14.10.1 alias ... 239
14.10.2 cdecl ... 240
14.10.3 export ... 241
14.10.4 inline ... 241
14.10.5 interrupt ... 242
14.10.6 iocheck ... 242
14.10.7 local ... 242
14.10.8 noreturn ... 243
14.10.9 nostackframe ... 243
14.10.10 overload ... 243
14.10.11 pascal ... 245
14.10.12 public ... 245
14.10.13 register ... 246
14.10.14 safecall ... 247
14.10.15 saveregisters ... 247
14.10.16 softfloat ... 247
14.10.17 stdcall ... 247
14.10.18 varargs ... 247
14.11 Неподдерживаемые модификаторы Turbo Pascal ... 248
Глава Глава 15 Перегрузка операторов ... 249
15.1 Введение ... 249
15.2 Объявление оператора ... 249
15.3 Операторы присваивания ... 251
15.4 Арифметические операторы ... 255
15.5 Операторы сравнения ... 257
15.6 Оператор In ... 258
Глава Глава 16 Программы, модули, блоки ... 260
16.1 Программы ... 260
16.2 Модули ... 261
16.3 Namespaces: Уточноние модуля ... 264
16.4 Зависимость модулей ... 267
16.5 Блоки ... 268
16.6 Область действия ... 269
16.6.1 Область действия блока ... 269
16.6.2 Область действия записи ... 270
16.6.3 Область действия класса ... 270
16.6.4 Область действия модуля ... 270

16.7 Libraries (Библиотеки) ... 271
Глава Глава 17 Исключения ... 273
17.1 Оператор Raise ... 273
17.2 Операторы try...except ... 275
17.3 Операторы try...finally ... 277
17.4 Обработка вложенных исключений ... 278
17.5 Классы исключений ... 279
Глава Глава 18 Использование ассемблера ... 281
18.1 Операторы Ассемблера ... 281
18.2 Процедуры и функции Ассемблера ... 281
18.3 Приложение ... 282
Алфавитный указатель ... 283

Про это руководство

Этот документ служит руководством по языку Pascal реализованном компилятором Free Pascal. Оно описывает все конструкции которые поддерживает Free Pascal, а также списки всех поддерживаемых типов данных. Однако оно, не дает подробное объяснение языка Pascal: это не учебник. Цель заключается в создании списка конструкций которые поддерживаются языком Pascal , и показать, где реализация Free Pascal отличается от реализации Turbo Pascal или Delphi.

Компиляторы Turbo Pascal и Delphi ввели различные особенности в язык Pascal. Компилятор Free Pascal эмулирует поведение этих компиляторов в соответствующих режимах: некоторые функции доступны только в случае когда компилятор переключается на соответствующий режим. Когда для определенных функций это необходимо, нужно использовать переключатель командной строки `-M` или директиву `{$MODE}` указанную в исходном тексте. Более подробную информацию о различных режимах можно найти в Справочник пользователя Free Pascal и Справочник программиста Free Pascal.

Предыдущие версии этого документа также содержат справочную документацию к модулю system и модулю objpas. Они были перемещены в справочник RTL.

От переводчика:
Я старался сохранить исходный вариант документации, однако иногда, при переводе, чувствуя что теряется смысл, то пытался объяснить некоторые моменты своими словами. Часть работы сделана не мной, я использовал более раннюю попытку перевода *(к сожалению не знаю имён переводчиков, однако если Вы мне это сообщите, я включу имена в список переводчиков)*. Вся эта информация включена в этот файл. Я старался опираться на документацию под Free Pascal **3.0.0**. Но мог пропустить или не заметить несоответствия. Так что если Вами будут замечены несоответствия или др. замечания просьба присылать их по адресу vchigrin@mail.ru, я постараюсь это исправить. Даже простое "обращение внимание" имеет смысл.

Обозначения

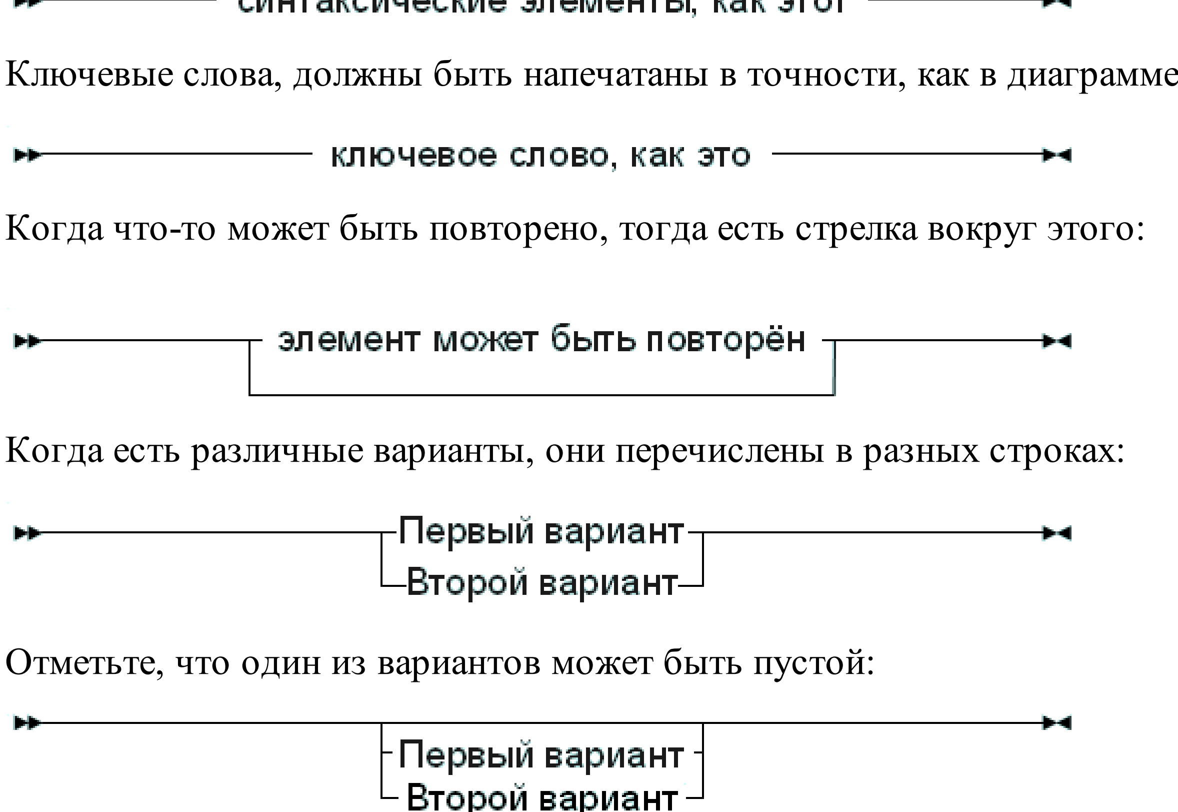
В данном документе мы будем описывать **функции**, **типы** и **переменные** с помощью моноширинного шрифта (`typewriter`).

Файлы описываются с помощью этого шрифта: (например filename).

Синтаксические диаграммы

Все элементы языка Pascal объяснены в синтаксических диаграммах. Синтаксические диаграммы как блок-схемы. Чтение синтаксических диаграмм означает, что нужно добраться от левой стороны до правой стороны, следуя за стрелками. Когда правая сторона синтаксической диаграммы достигнута, и она заканчивается единственной стрелкой, это означает, что синтаксическая диаграмма продолжается на следующей строке. Если линия заканчивается **2**-мя стрелками, указывающими друг на друга, тогда диаграмма закончена.

Синтаксические элементы пишутся так



Ключевые слова, должны быть напечатаны в точности, как в диаграмме:

Когда что-то может быть повторено, тогда есть стрелка вокруг этого:

Когда есть различные варианты, они перечислены в разных строках:

Отметьте, что один из вариантов может быть пустой:

Это означает, что как первый так и второй вариант не являются обязательными. Конечно, все эти элементы могут быть объединены и вложены.

О языке Паскаль

Язык Pascal был первоначально разработан Никлаусом Виртом в **1970** году. С того дня он значительно эволюционировал, с большим количеством вкладов различными конструкциями компилятора *(Особенно: Borland)*. Основные элементы были сохранены на протяжении многих лет:

- **Легкий синтаксис**, довольно многословный, но все же легкий для чтения. Идеально подходящий для обучения.
- **Строго типизированный.**
- **Процедурный.**
- **Не чувствительный к регистру.**
- **Позволяет вложенные процедуры.**
- **Встроенные процедуры для обработки ввода/вывода.**

Компиляторы Turbo Pascal и Delphi ввели различные особенности в язык Pascal, наиболее заметными являются более легкая строковая обработка и объектная ориентированность. Компилятор Free Pascal первоначально эмулировал большую часть Turbo Pascal а позже и Delphi. Он эмулирует поведение этих компиляторов в соответствующих режимах: некоторые функции доступны только в том случае если компилятор переключается на соответствующий режим. Когда для определенных функций это необходимо, нужно использовать переключатель командной строки `-M` или директиву `{$MODE}` указанную в исходном тексте. Более подробную информацию о различных режимах можно найти в руководстве пользователя и руководстве программиста.

Глава 1 Элементы языка паскаль

Токены это основа строительных блоков исходного кода: они являются ***'словами'*** языка: символы объединены в лексемы в соответствии с правилами языка программирования. Есть пять классов лексем:

зарезервированные слова — это слова, которые имеют определенный смысл в языке. Они не могут быть изменены или переопределены.

идентификаторы — это имена символов которые определят программист. Они могут быть изменены или использованы повторно. Они подчиняются правилам грамматики языка.

операторы — это обычно символы для математических или других операций: `+`, `-`, `*` и так далее.

разделители — обычно это пробел.

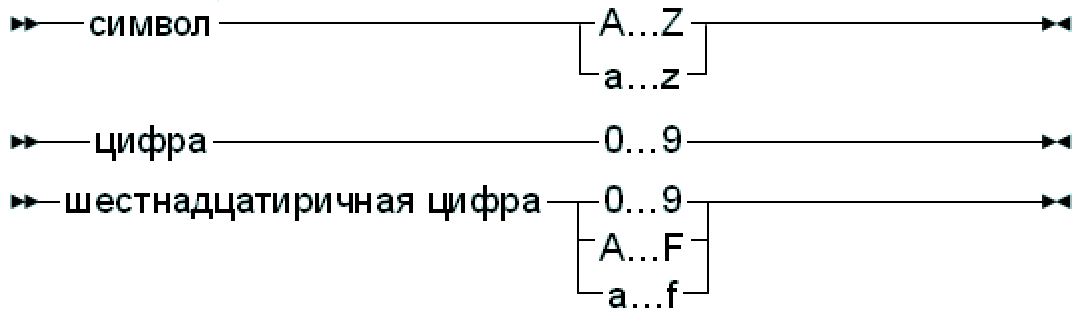
константы — числовые или символьные константы используются, чтобы обозначить фактические значения в исходном коде, такой как **1** *(целочисленная константа)* или **2.3** *(вещественная константа)* или ***Строковая константа*** *(строка: часть текста)*.

В этой главе мы описываем все зарезервированные слова языка Pascal, а также различные способы для обозначения строк, чисел, идентификаторов и т.д.

1.1 Символы

Free Pascal разрешает использовать ***все символы***, ***цифры*** и некоторые ***специальные символы*** в исходном коде.

Распознаваемые символы



Следующие символы имеют особое значение:

```
+  -  *  /  =  <  >  [  ]  .  ,  ()  :  ^  @  –  "  $  #  &
 %
```

и следующие символьные пары также:

```
<<  >>  **  <>  ><  <=  >=  :=  +=  -=  *=  /=  (*  *)  (.
 .)  //
```

В спецификаторе диапазона используется, символьная пара **(.** , она эквивалентна левой квадратной скобке **[**. Аналогично, символьная пара **.)** эквивалентна правой квадратной скобке **]**. При использовании в качестве разделителя комментариев, символьная пара **(*** эквивалентна левой фигурной скобке **{** и символьная пара ***)** эквивалентна правой фигурной скобке **}**. Эти символьные пары сохраняют свое нормальное значение в строковых выражениях.

1.2 Комментарии

Комментарии это фрагменты исходного кода, которые полностью игнорируются компилятором. Они существуют только для выгоды программиста, так он может объяснить определенные фрагменты кода. Для компилятора, это то же самое что их и не было.

Следующая часть кода демонстрирует комментарий:

```
(*  Моя красивая функция возвращает интересный результат  *)
Function Beautiful : Integer;
```

(* и *) использовались как разделители комментариев с самых первых дней языка Pascal. Они были заменены с помощью { и }, как в следующем примере:

```
{ Моя красивая функция возвращает интересный результат  }
Function Beautiful : Integer;
```

Комментарий может занимать несколько строк:

```
{
     Моя прекрасная функция возвращает интересный результат,
но только если аргумент A меньше, чем B.
}
Function Beautiful  (A,B : Integer): Integer;
```

Однострочные комментарии могут быть указаны также с помощью разделителя //:

```
// Моя прекрасная функция возвращает интересный результат
Function  Beautiful : Integer;
```

Комментарий продолжается от символа // до конца строки. Этот вид комментария был представлен Borland в компиляторе Pascal Delphi.

Free Pascal поддерживает использование вложенных комментариев. Следующие конструкции - допустимые комментарии:

```
(*  Это комментарий в старом стиле*)
{    Это комментарий Turbo Pascal }
//  Это комментарий Delphi. Все до конца строки игнорируется.
```

Ниже приводятся допустимые способы вложения комментариев:

```
{  Комментарий  1  (*  Комментарий  2  *)  }
(*  Комментарий  1  {   Комментарий  2  }  *)
{   Комментарий  1  //  Комментарий  2  }
(*  Комментарий  1  //  Комментарий  2  *)
//  Комментарий  1  (*  Комментарий  2  *)
//  Комментарий  1  {   Комментарий  2  }
```

Последние два комментария должны быть на одной строке. Следующие два дадут ошибки:

```
//  Допустимый комментарий { Больше не действительный
комментарий!!
    }
```

и

```
//  Допустимый комментарий  (*  Больше не действительный
комментарий!!
    *)
```

Компилятор будет реагировать с ошибкой ***"недопустимый символ"***, когда он встречает такие конструкции, независимо от переключателя -Mtp.

Замечание:

В режиме совместимости TP и Delphi, вложенные комментарии не допускаются, для максимальной совместимости с существующим кодом для этих компиляторов.

1.3 Зарезервированные слова

Зарезервированные слова являются частью языка Pascal, и как таковые, не могут быть переопределены программистом. Всюду ***на синтаксических диаграммах*** они будут обозначены, **полужирным шрифтом**. Pascal не чувствителен к регистру символов, таким образом компилятор примет любое сочетание верхнего или нижнего регистра для зарезервированных слов.

Мы сделали различия между зарезервированными словами Turbo Pascal и Delphi. В режиме TP, распознаются только зарезервированные слова Turbo Pascal, но зарезервированные слова из Delphi могут быть переопределены. По умолчанию, Free Pascal распознает зарезервированные слова Delphi.

1.3.1 Зарезервированные слова Turbo Pascal

Следующие ключевые слова существуют в режиме совместимости с Turbo Pascal

```
absolute            if                  reintroduce
```

```
and            implementation   repeat
array          in               self
asm            inherited        set
begin          inline           shl
case           interface        shr
const          label            string
constructor    mod              then
destructor     nil              to
div            not              type
do             object           unit
downto         of               until
else           operator         uses
end            or               var
file           packed           while
for            procedure        with
function       program          xor
goto           record
```

1.3.2 Зарезервированные слова Free Pascal

Вдобавок к зарезервированным словам Turbo Pascal Free Pascal также рассматривает следующие слова как зарезервированные слова:

```
dispose        false            true
exit           new
```

1.3.3 Зарезервированные слова Object Pascal

Зарезервированные слова Object Pascal *(используемые в режиме `Delphi` или `Objfpc`)* те же что и в режиме Turbo Pascal, со следующими дополнительными ключевыми словами:

```
as             initialization   property
class          inline           raise
dispinterface  is               resourcestring
except         library          threadvar
exports        on               try
finalization   out
```

finally	packed

1.3.4 Модификаторы

Следующий - список всех ***модификаторов***. Они не являются зарезервированными словами в том смысле, что они могут использоваться в качестве идентификаторов, но в определенных местах, у них есть особое значение для компилятора, то есть, компилятор рассматривает их как часть языка Pascal.

absolute	generic	protected
abstract	helper	public
alias	implements	published
assembler	index	read
bitpacked	interrupt	register
break	iochecks	reintroduce
cdecl	local	result
continue	message	safecall
cppdecl	name	saveregisters
cvar	near	softfloat
default	nodefault	specialize
deprecated	noreturn	static
dynamic	nostackframe	stdcall
enumerator	oldfpccall	stored
experimental	otherwise	strict
export	overload	unaligned
external	override	unimplemented
far	pascal	varargs
far16	platform	virtual
forward	private	write

Замечание:

Предварительно определенные типы, такие как `Byte`, `Boolean` и константы, такие как `maxint` ***не*** являются зарезервированными словами. Они - идентификаторы, объявленные в модуле system. Это означает, что эти типы могут быть переопределены в других модулях. Однако это не поощряется, поскольку это может вызвать путаницу.

Замечание:

Начиная с версии **2.5.1** можно использовать зарезервированные слова в качестве идентификаторов с помощью их экранирования символом &. Это означает, что следующие возможно

```
var
    &var  :  integer;

begin
   &var:= 1;
   Writeln(&var);
end.
```

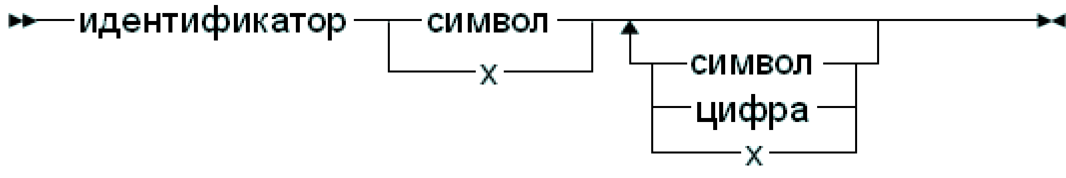
однако, не рекомендуется использовать эту функцию в новом коде, поскольку это делает код менее читабельным. Она предназначена в основном для исправления старого кода, когда список зарезервированных слов изменился и охватывает слово, которое еще не было зарезервировано *(См. также раздел* *1.4 Идентификаторы* [15]*).*

1.4 Идентификаторы

Идентификаторы обозначают определенные программистом ***имена*** для конкретных *констант*, *типов*, *переменных*, *процедур*, *функций*, *модулей* и *программ*. Все определенные программистом имена в исходном коде, исключая зарезервированные слова, используются как идентификаторы.

Идентификаторы могут состоять из значимых символов от **1** до **127** *(букв, цифр и подчеркивания)*, первый из которых должен быть буквой *(**a-z** или **A-Z**)*, или подчеркиванием *(_)*.Следующая диаграмма иллюстрирует синтаксис идентификаторов.

Идентификаторы



Как зарезервированные слова языка Pascal, идентификаторы являются нечувствительными к регистру, то есть, оба идентификатора

```
myprocedure;
```

и

```
MyProcedure;
```

описывает одну и ту же процедуру.

Замечание:

Начиная с версии **2.5.1** можно использовать зарезервированные слова в качестве идентификаторов с помощью их экранирования символом амперсанд *(&)*. Это означает, что следующие возможно

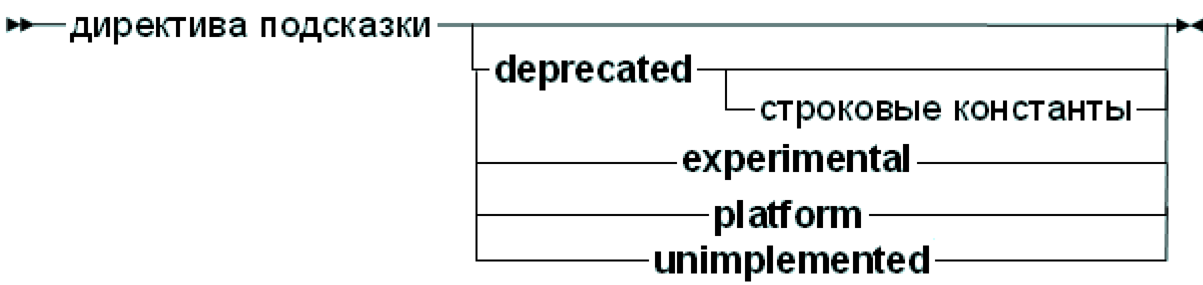
```
program  testdo;
 procedure  &do;
  begin
  end;
begin
    &do;
end.
```

Зарезервированное слово do используется в качестве идентификатора для объявления и для вызова процедуры '**do**'.

1.5 Директивы подсказок.

Большинство идентификаторов (***константы***, ***переменные***, ***функции*** или ***методы***, ***свойства***) могут иметь подсказки добавленные к их определению:

Директива подсказки



Всякий раз, когда с идентификатором, отмеченным директивой подсказки, позже встречается компилятор, он будет выдавать предупреждение, соответствущее указанной подсказке.

deprecated *(устарел)* — использование этого идентификатора осуждается, используйте альтернативу вместо него.

experimental *(экспериментальный)* — использование этого идентификатора экспериментально: он может использоваться, чтобы отметить новые функции, он должен использоваться с осторожностью.

platform *(платформа)* — это - зависимый от платформы идентификатор: он не может быть определен на всех платформах

unimplemented *(невыполненными)* — он должен использоваться только с функциями и процедурами. Он использоваться, чтобы сигнализировать, что определенная функция еще не была реализована.

Пример:

```
Const
    AConst = 12 deprecated;
var
    p : integer platform;
Function Something : Integer; experimental;
begin
    Something:=P+AConst;
end;
begin
    Something;
end.
```

Это привело бы к следующему выводу:

```
testhd.pp(11,15)  Warning: Symbol "p" is not portable
                  (Внимание: Символ "p" не является переносимым)
testhd.pp(11,22)  Warning: Symbol "AConst" is deprecated
                  (Внимание: Символ "AConst" устарел)
testhd.pp(15,3)   Warning: Symbol "Something" is
                  experimental
                  (Внимание: Символ "Something" является
                  экспериментальным)
```

Директивы подсказок могут применяться со всеми видами идентификаторов: **модулями, константами, типами, переменными, функциями, процедурами и методами**.

1.6 Числа

Числа по умолчанию записываются в десятичной системе счисления. Вещественные числа (или десятичные) записываются с использованием технической или экспоненциальной нотации *(например **0.314E1**)*.

Для типа целочисленных констант Free Pascal поддерживает четыре формата:

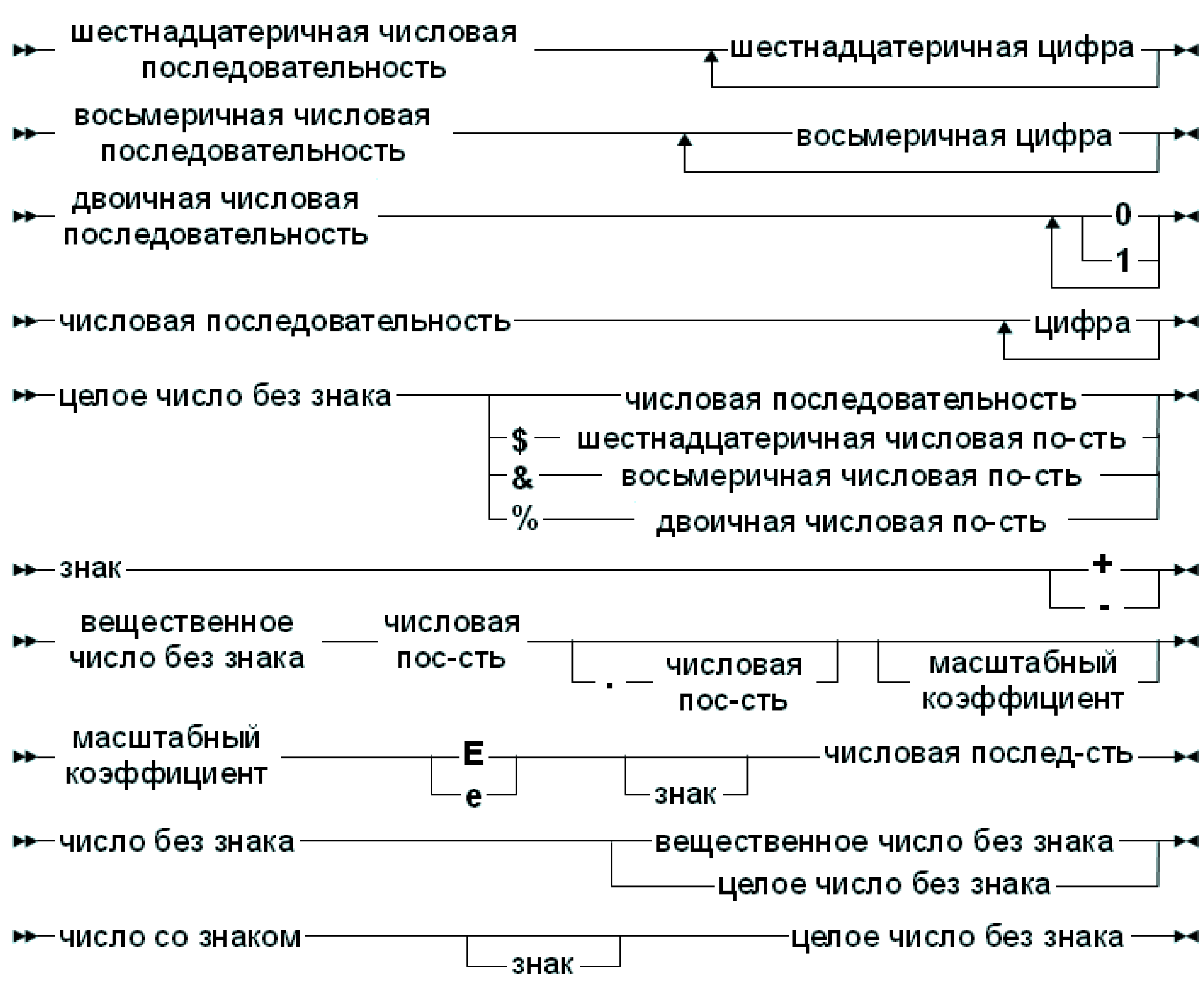
1. Нормальный, ***десятичный формат*** *(по основанию **10**)*. Это - стандартный формат.
2. ***Шестнадцатеричный формат*** *(по основанию **16**)*, таким же образом как и в Turbo Pascal. Чтобы определить константную величину в шестнадцатеричном формате, укажите перед именем знак доллара ($). Таким образом шестнадцатеричное число `$FF` равняется десятичному числу `255`. Отметьте, что регистр является незначащим при использовании шестнадцатеричных констант.
3. С версии **1.0.7** также поддерживается ***Восьмеричный формат*** *(по*

основанию 8). Чтобы определить константу в восьмеричном формате, укажите перед именем знак амперсанда (&). Например 15 определен в восьмеричной нотации как &17.

4. ***Двоичная нотация*** *(по основанию 2)*. Двоичное число может быть определено, указав переде ним знак процента (%). Таким образом, 255 может быть определен в двоичной записи как %11111111.

Следующие диаграммы показывают, синтаксиса для чисел.

Числа



Замечание:

Восьмеричная и ***двоичная*** запись ***не поддерживается*** в режиме эмуляции Delphi или TP.

1.7 Метки

Метка — это имя, места в исходном коде, к которому, можно перейти с другого места с помощью оператора goto. Метка — это стандартный

идентификатор или последовательность цифр.

Метка

метка — числовая последовательность / идентификатор

Замечание:

> Прежде чем использовать метки нужно указать ключи `-Sg` или `-Mtp`. По умолчанию Free Pascal не поддерживает метки *(label)* и оператор перехода *(goto)*. Директива `{$GOTO ON}` также может использоваться, чтобы позволить использование меток и оператора перехода.

Ниже приведены примеры действительных меток:

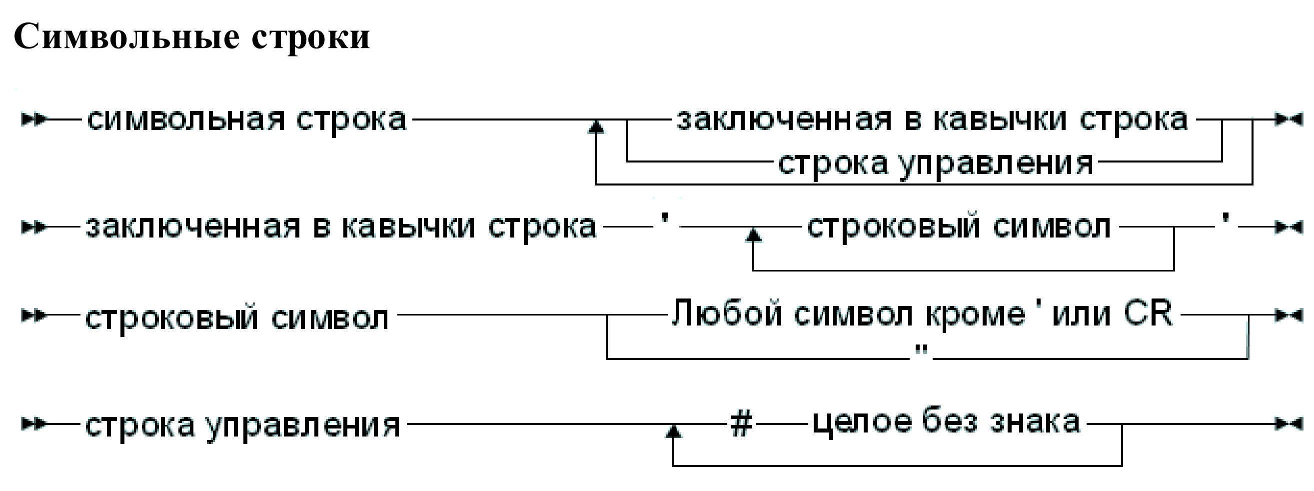
```
Label
  123,
  abc;
```

1.8 Символьные строки

Символьная строка *(или* ***строка****, если коротко)* является последовательностью символов *(измеренных в байтах)*, заключенных в одинарные кавычки, и на одной строке исходного кода программы: никакие литеральные символы возврата каретки или перевода строки не могут присутствовать в строке.

Набор символов ни с чем между кавычками `('')` является ***пустой строкой***.

Символьные строки



Строка состоит из стандартных, **8**-разрядных символов ASCII или символов Unicode (обычно закодированных в UTF-8). Специальная **строка управления**

(control string) может использоваться, чтобы определить символы, которые не могут быть введены с клавиатуры, например такой как #27 для символа **ESC**.

Символ одинарной кавычки может быть встроен в строку, для этого нужно ввести его дважды. Конструкция из языка C экранирования символов в строке (используя наклонную косую черту) не поддерживается в Pascal.

Следующие строковые константы допустимы:

```
'Это строка Pascal'
''
'a'
'Символ табулятора:  '#9'  врезать легко'
```

Недопустимые строки:

```
'строка начинается здесь
 и продолжается здесь'
```

Вышеупомянутая строка должна быть введена как:

```
'строка начинается здесь'#13#10'и продолжается здесь'
```

или

```
'строка начинается здесь'#10'и продолжается здесь'
```

на UNIX подобных системах(включая Mac OS X), и как

```
'строка начинается здесь'#13'и продолжается здесь'
```

на классической Mac-подобной операционной системе.

Возможно использовать другие наборы символов в строках: в этом случае кодовая страница исходного файла должна быть определена с помощью директивы {$CODEPAGE XXX} или с помощью параметра командной строки компилятора -Fc. В этом случае символы в строке будут интерпретироваться как символы из указанной кодовой страницы.

Глава 2 Константы

Так же, как и Turbo Pascal, Free Pascal поддерживает как обычные, так и типизированные константы. Они объявлены в блоке объявления констант в модуле, объявления программы или класса, функции или процедуры *(раздел 16.6 Область действия* [269]*)*.

2.1 Обычные константы

Объявления ***обычных констант*** создаются используя имя идентификатора со следующим за ним знаком "=" , за которым следует необязательное выражение, состоящее из допустимых комбинаций чисел, символов, логических значений или перечисляемых значений по мере их необходимости. Следующая синтаксическая диаграмма показывает, как создавать допустимые объявления обычных констант.

Объявление константы

```
►► объявление ─┬─ идентификатор ─ = ─ выражение ─ директивы ─ ; ─┬─►◄
   константы   │                                  подсказки       │
               └──────────────────────────────────────────────────┘
```

Компилятор должен быть в состоянии вычислить выражение при объявлении константы во время компиляции. Это означает, что большинство функций в библиотеке времени выполнения не может использоваться в объявлениях констант. Тем не менее такие операторы как `+`, `-`, `*`, `/`, `not`, `and`, `or`, `div`, `mod`, `ord`, `chr`, `sizeof`, `pi`, `int`, `trunc`, `round`, `frac`, `odd` могут использоваться. Для получения дополнительной информации по выражениям Глава 12 Выражения [178].

Только константы следующих типов могут быть объявлены:

- Перечисляемые типы.
- Множества.
- Типы указателей *(но единственное позволенное значение `Nil`)*.
- Вещественные типы.
- **Символ** (`Char`).
- **Строка** (`String`).

Все следующие объявления констант допустимые:

```
Const
  e  =  2.7182818;          {  Константа вещественного (Real)
типа.  }
  a  =  2;                  {  Константа порядкового (Integer)
типа.  }
  c  =  '4';                          {  Константа символьного
```

```
типа. }
   s = 'Это строковая константа'; { Константа строкового
типа. }
   sc = chr(32)
   ls = SizeOf(Longint);
   P = Nil;
   Ss = [1,2];
```

Присвоение значения обычной константе не разрешено. Таким образом, учитывая предыдущее объявление, следующее приведет к ошибке компилятора:

```
   s := 'некоторая строка';
```

Для строковых констант тип строки зависит от некоторых установок компилятора. Если требуется определенный тип, должна использоваться типизированная константа, так как объяснено в следующем разделе.

До версии **1.9**, Free Pascal не корректно поддерживает **64**-битные константы. Начиная с версии **1.9**, **64**-разрядные константы могут быть определены.

2.2 Типизированные константы

Иногда необходимо определить ***тип константы***, например для констант сложных структур *(определенных позже в руководстве)*. Их определение довольно просто.

Объявление типизированных констант

В противоположность обычным константам им может быть присвоено значение во время выполнения программы. Это старая концепция из Turbo Pascal, которая была заменена поддержкой инициализированных переменных: Для подробного описания см. 4.4 Инициализированные переменные [74].

Возможностью присвоения значений типизированным константам управляют директива {$J}: она может быть выключена, но включена по умолчанию *(для совместимости с Turbo Pascal)*. ***Инициализированные переменные всегда разрешены***.

Замечание:

нужно подчеркнуть, что типизированные константы автоматически инициализируются при старте программы. Это также верно для ***локальных*** типизированных констант и инициализированных переменных. Локальные типизированные константы также инициализируются при запуске программы. Если их значение было изменено во время предыдущих вызовов функции, они сохранят свое измененное значение, то есть они не инициализируются каждый раз, когда вызывается функция.

2.3 Строковые ресурсы

Resourcestring *(строковой ресурс)* - это специальный вид блока объявления строковых констант. Объявление строкового ресурса очень похоже на объявление строковых констант: строка ресурсов выступает в качестве строки константы, но она может быть локализована посредством ряда специальных подпрограмм из модуля objpas. Блок объявления строкового ресурса разрешен только в режимах Delphi или Objfpc.

Далее идет пример определения строкового ресурса:

```
Resourcestring
   FileMenu  =  '&File...';
   EditMenu  =  '&Edit...';
```

Все строковые константы определенные как строковые ресурсы, размещены в специальных таблицах. Строками в этих таблицах можно управлять во время выполнения программы с помощью специальных подпрограмм из модуля objpas.

Семантически, строки действуют как обычные константы; но присваивать им значения запрещено *(кроме как через специальные подпрограмм из модуля objpas)*. Однако, они могут использоваться в присвоениях или выражениях как обычные строковые константы. Основное использование блока строковых ресурсов должно обеспечить легкое средство интернационализации.

Больше информации про строковые ресурсы вы можете получить в Справочник программиста Free Pascal и в описании модуля objpas.

Замечание:

Отметьте, что строка ресурса, которая объявлена как выражение, не будет изменяться, если части выражения будут изменены:

```
resourcestring
   Part1  =  'First  part  of  a  long  string.';
   Part2  =  'Second  part  of  a  long  string.';
   Sentence  =  Part1+'  '+Part2;
```

Если подпрограмма локализации преобразует `Part1` и `Part2`, то константа `Sentence` не будет переведена автоматически: она имеет отдельную запись в таблицах строк ресурса, и должна для этого быть преобразована отдельно. Вышеупомянутая конструкция просто говорит, что начальное значение `Sentence` равняется `Part1 +'  '+Part2`.

Замечание:

Аналогично, при использовании строк ресурсов в массиве констант, только начальные значения строк ресурсов будут использоваться в массиве: когда отдельные константы будут преобразованы, элементы в массиве сохранят свое исходное значение.

```
resourcestring
   Yes  =  'Yes.';
   No  =  'No.';

Var
   YesNo  :  Array[Boolean]  of  string  =  (No,Yes);
   B  :  Boolean;

begin
   Writeln(YesNo[B]);
end.
```

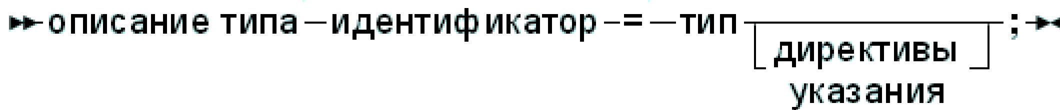
Это напечатает *'Yes.'* или *'No.'* в зависимости от значения `B`, даже если константы `Yes` и `No` были локализованы некоторым механизмом локализации.

Глава 3 Типы

У всех переменных есть ***тип***. Free Pascal поддерживает те же самые основные типы что и Turbo Pascal с некоторыми дополнительными типами из Delphi.

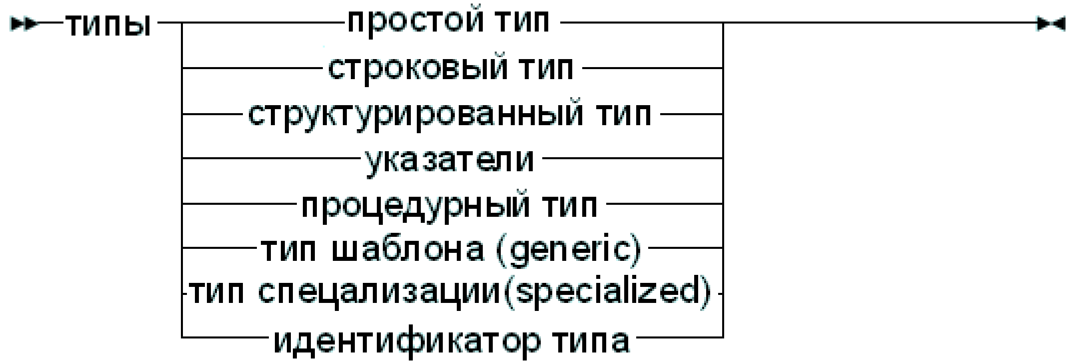
Программист может объявить свои собственные типы, которые в сущности определяют идентификатор, который может использоваться, чтобы обозначить этот пользовательский тип при объявлении переменных далее в исходном коде. Объявление типа происходит в ***блоке объявления типов*** (type) *(секции 16.6 Область действия*[269]*)*, который представляет собой набор деклараций типов, разделенных точкой с запятой:

Определение типа



Есть **8** главных классов типа:

Типы

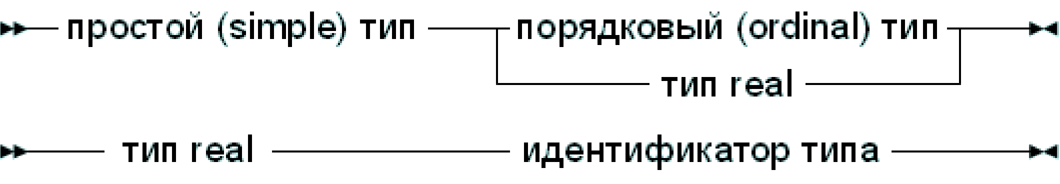


Каждый из этих случаев будут рассмотрен отдельно.

3.1 Базовые типы

Основные или простые типы Free Pascal такие же как и типы Delphi. Мы рассмотрим каждый тип отдельно.

Простой тип



3.1.1 Перечислимые типы

За исключением `Int64`, `QWord` и `Real` *(вещественных)* типов, все *базовые* типы это ***перечислимые типы***. Перечислимые типы имеют следующие характеристики:

1. Перечислимые типы исчисляемы и упорядочены, то есть, в принципе, возможно начать считать их один за другим в указанном порядке. Это свойство позволяет работу функций `Inc`, `Ord`, `Dec` для порядковых типов, которые будут определены.
2. У перечислимых типов есть самое маленькое значение. Попытка применить функцию `Pred` для самого маленького значении сгенерирует ошибку проверки диапазона, если проверка диапазона включена.
3. У перечислимых типов есть самое большое значение. Попытка применить функцию `Succ` для самого большого значении сгенерирует ошибку проверки диапазона, если проверка диапазона включена.

Целые типы

Список предопределенных целочисленных типов представлен в таблице (3.1).

Таблица 3.1: Предопределенные целочисленные типы

Имя
Integer
ShortInt
SmallInt
LongInt
LongWord
Int64
Byte
Word
Cardinal
QWord

Имя
Boolean
ByteBool
WordBool
LongBool
Char

Целочисленные типы, и их диапазоны и размеры, предопределены во Free Pascal, и перечислены в таблице (3.2). Пожалуйста, отметьте, что типы QWord и Int64 не истинно перечисляемые, таким образом, некоторые конструкции языка Pascal не будут работать с этими двумя целочисленными типами.

Таблица 3.2: Предопределенные целочисленные типы

Тип	Диапазон	Размер в байтах
Byte	**0 .. 255**	**1**
ShortInt	**-128 .. 127**	**1**
SmallInt	**-32768 .. 32767**	**2**
Word	**0 .. 65535**	**2**
Integer	или SmallInt или LongInt	**2 или 4**
Cardinal	LongWord	**4**
LongInt	**-2147483648 .. 2147483647**	**4**
LongWord	**0 .. 4294967295**	**4**
Int64	**-9223372036854775808 .. 9223372036854775807**	**8**
QWord	**0 .. 18446744073709551615**	**8**

По умолчанию во Free Pascal тип Integer соответствует типу SmallInt. И типу LongInt в любом из двух режимов Delphi или ObjFPC. Тип Cardinal сейчас всегда соответствует типу LongWord.

Замечание:

Все десятичные константы, которые не входят в диапазон **-2147483648 .. 2147483647** начиная с версии **1.9.0**, автоматически обрабатываются как **64**-разрядные целые константы. Более ранние версии будут конвертировать их в вещественные типизированные константы.

Free Pascal делает автоматическое преобразование типов в выражениях, где используются различные виды целочисленных типов:

1. Каждая платформа имеет ***родной*** размер целого числа, в зависимости от того, платформа **8**-бит, **16**-бит, **32**-бит или **64**-бит. например, на AVR это **8**-разрядная платформа.
2. Каждое целое число меньшее, ***родного*** размера повышается до базовой версии ***родного*** размера. Целые равные ***родному*** размеру сохраняют их разрядность.
3. Результат двоичных арифметических операций *(+, –, * и т.д.)*, определяется следующим образом:
   a. Если по крайней мере один из операндов больше, чем нативный (родной) размер целого, то результатом выбирается наименьший тип, который охватывает диапазон типов обоих операндов. Это означает, что смешивание типа без знака с меньшим или равным по размеру знакового типа, даст тип, который больше, чем оба из них.
   b. Если оба операнда имеют один и тот же тип (размер), то результатом будет такой-же тип. Единственным исключением является вычитание *(–)*: если один из типов без знаковой, операция вычитания даёт знаковый тип FPC *(как и в Delphi, но не в TP7)*.
   c. Смешивние знаковых и беззнаковых типов ***родного*** целого типа - даёт тип, больший по размеру, и со знаком. Это означает, что при смешивании типов `LongInt` и `LongWord` на **32**-разрядных платформах будет производить `Int64`. Аналогично, при смешивании типа `Byte` и `ShortInt` на **8**-разрядных платформах *(AVR)*, результатом будет тип `SmallInt`.

Булевы типы

Free Pascal поддерживает тип Boolean с двумя предопределенными возможными значениями True и False. Только два значения могут быть присвоены логическому типу. Конечно, любое выражение, которое относится к Boolean значению, также может быть присвоено Boolean типу.

Таблица 3.3: Булевы типы

Имя	Размер	Ord(True)
`Boolean`	1	1

ByteBool	**1**	Любое ненулевое значение
WordBool	**2**	Любое ненулевое значение
LongBool	**4**	Любое ненулевое значение

Free Pascal также поддерживает типы ByteBool, WordBool и LongBool. Они имеют тип Byte, Word и Longint, но являются совместимыми с переменными булевого типа: False *(Ложь)* эквивалентна нулю (**0** - нуль), и любое ненулевое значение считается True *(Истиной)* при преобразовании в булево значение. Булево значение True преобразовывается в **–1** в случае, если оно присвоено переменной типа LongBool.

Следующие присвоения корректны, при условии что переменная B типа Boolean *(логического)*:

```
B := True;
B := False;
B := 1<>2; { -  Результат в B  :=  True}
```

Булевы выражения также используются в условиях.

Замечание:

Во Free Pascal, логические выражения по умолчанию всегда вычисляются таким образом, что, когда становится известен результат, остальная часть выражения больше не будет вычисляться: это называется короткой схемой вычисления логического выражения.

В следующем примере функция Func никогда не будут вызвана, что может привести к странным побочным эффектам.

```
...
  B  :=  False;
  A  :=  B  and  Func;
```

Здесь Func является функцией, которая возвращает значение Boolean *(логического)* типа.

Такое поведение контролируется директивой компилятора {$B}.

Типы перечислений

Во Free Pascal поддерживаются ***типы перечислений***. Вдобавок к реализации Turbo Pascal, Free Pascal поддерживает также расширенный C подобный стиль задания перечислений, где определенному элементу списка перечисления присвоено значение.

Перечислимый тип

(см. *Глава 12 Выражения* [178] *для того чтобы увидеть как использовать выражения)* При использовании присвоенных перечисляемых типов, присвоенные элементы должны быть в списке в возрастающем числовом порядке, иначе компилятор будет генерировать ошибку. Выражения, используемые в присвоенных перечисляемых элементах, должны быть известны во время компиляции. Таким образом, следующее объявление является корректным объявлением перечисляемого типа:

```
Type
  Direction = ( North, East, South, West);
```

Си подобный стиль объявления выглядит следующим образом

```
Type
  EnumType = (one, two, three, forty := 40,fortyone);
```

В результате порядковое число `forty` равно **40**, а не **3**, как это бы было без присвоения `': = 40'`. Порядковый номер fortyone равен `41`, а не `4`, как это бы было без присвоения. После присвоения в перечисляемом объявлении компилятор добавляет `1` к присвоенному значению, чтобы присвоить его следующему перечисляемому элементу.

При определении такого перечисляемого типа важно иметь в виду, что перечисляемые элементы должны находится в порядке возрастания. Следующее объявление вызовет ошибку компилятора:

```
Type
  EnumType = (one, two, three, forty := 40, thirty := 30);
```

Необходимо объявлять `сорок` и `тридцать` в правильном порядке. При использовании типов перечисления(имеется ввиду с присвоениями) важно помнить следующие моменты:

1. Функции `Pred` и `Succ` не могут использоваться на этом виде перечисляемых типов. Любая попытка вызвать их приведет к ошибке компилятора.
2. Перечисляемые типы хранятся с использованием типа по умолчанию, независимо от фактического числа значений: компилятор не пытается оптимизировать их для сохранения места. Это поведение может быть изменено с помощью директивы компилятора `{$PACKENUM N}` , которая указывает компилятору минимальное количество байтов, которые будут

использоваться для перечисляемых типов.

```
Type

{$PACKENUM 4}
  LargeEnum = ( BigOne, BigTwo, BigThree );

{$PACKENUM 1}
  SmallEnum = ( one, two, three );

Var S : SmallEnum;
 L : LargeEnum;
begin
  WriteLn ('Small enum : ',SizeOf(S));
  WriteLn ('Large enum : ',SizeOf(L));
end.
```

После запуска мы увидим следующее:

```
Small enum : 1
Large enum : 4
```

Больше информации можно найти в Справочник программиста Free Pascal в разделе директив компилятора.

Типы диапазоны

Тип диапазон является диапазоном значений от порядкового типа *(базовый тип)*. Чтобы определить тип диапазона, необходимо указать его предельные значения: максимальное и минимальное значение типа.

Тип - диапазон

```
►►─тип диапазон──=──константа───...───константа───►◄
```

Некоторые из предопределенных `integer` *(целочисленных)* типов определены как ***типы диапазоны***:

```
Type
  Longint  = $80000000..$7fffffff;
  Integer  = -32768..32767;
  shortint = -128..127;
  byte = 0..255;
  Word = 0..65535;
```

Также могут быть определены типы диапазоны от перечисляемого типа:

```
Type
  Days = (monday,tuesday,wednesday,thursday,friday,saturday,
```

```
  sunday);
    WorkDays = monday .. friday;
    WeekEnd = Saturday .. Sunday;
```

3.1.2 Вещественные типы

Free Pascal использует математический сопроцессор *(или эмуляцию)* для всех вычислений с плавающей точкой. Родной тип `Real` зависим от процессора, но он является либо типом `Single` либо типом `Double`. Поддерживаются только типы плавающей точки стандарта IEEE, и они зависят от целевого процессора и опций эмуляции. Истинно совместимые с Turbo Pascal типы перечислены в таблице (3.4).

Таблица 3.4: Поддерживаемые Вещественные типы

Тип	Диапазон	Значящие разряды	Размер
`Real`	зависит от платформы	???	**4** or **8**
`Single`	**1.5E-45 .. 3.4E38**	**7-8**	**4**
`Double`	**5.0E-324 .. 1.7E308**	**15-16**	**8**
`Extended`	**1.9E-4932 .. 1.1E4932**	**19-20**	**10**
`Comp`	**-2E64+1 .. 2E63-1**	**19-20**	**8**
`Currency`	**-922337203685477.5808 .. 922337203685477.5807**	**19-20**	**8**

Тип `Comp` фактически является, **64**-разрядным целым числом и доступен **не на всех** целевых платформах. Чтобы получить более подробную информацию о поддерживаемых типов для каждой платформы, обратитесь к Справочник программиста Free Pascal.

Тип `Currency` - это вещественный тип с фиксированной точкой со внутренним представлением в виде **64**-битного целого (автоматически нормируемый с множителем **10000**), что минимизирует ошибки округления.

3.2 Символьные типы

3.2.1 Символ (Char или AnsiChar)

Free Pascal поддерживает тип `Char`. `Char` занимает **1** байт, и содержит один ASCII-символ.

Символьную константу можно объявить, заключив символ в одинарные кавычки, следующим образом: `'a'` или `'A'` оба являются символьными константами.

Символ также может быть определен с помощью его символьного значения (обычно ASCII-код), которому предшествует символ (#). Например определение `#65` будет тем же самым что и `'A'`.

Кроме того, символ каре (^) может использоваться в комбинации с буквой, чтобы определить, символ с ASCII кодом меньше чем **27**. Таким образом `^G` равняется `#7` - **G** - седьмая буква в алфавите. Компилятор довольно неаккуратен с символами которые он позволяет указывать после каре, но в общем должен принимать только буквы.

Когда должен быть указан символ одинарной кавычки, он должен быть введен последовательно два раза, таким образом `''''` представляет символ одинарной кавычки.

Чтобы отличить `Char` от `WideChar`, модуль System определяет тип `AnsiChar`, который совпадает с типом `Char`. В будущих версиях FPC, тип `Char` может стать аналогичным типу `WideChar` или `AnsiChar`.

3.2.2 WideChar

Free Pascal поддерживает тип `WideChar`. Тип `WideChar` имеет размер **2** байта, и содержит один UNICODE символ в кодировке UTF-16.

Символ юникода *(unicode)* - это число *(код UTF-16)*, которому предшествует знак (#).

Нормальный символ ANSI *(1 байт)* также может использоваться для `WideChar`, компилятор автоматически преобразует его в **2**-х байтный символ UTF-16.

Ниже приведено определение греческих символов (**Ψ** *(Phi)* и **Ω** *(Omega)*):

```
Const
 C3 : widechar = #$03A8;
 C4 : widechar = #$03A9;
```

Это-же можно сделать путём приведения типа к `WideChar`

```
Const
 C3 : widechar = widechar($03A8);
 C4 : widechar = widechar($03A9);
```

3.2.3 Другие символьные типы

В модуле System (Free Pascal) определены другие типы символов, например

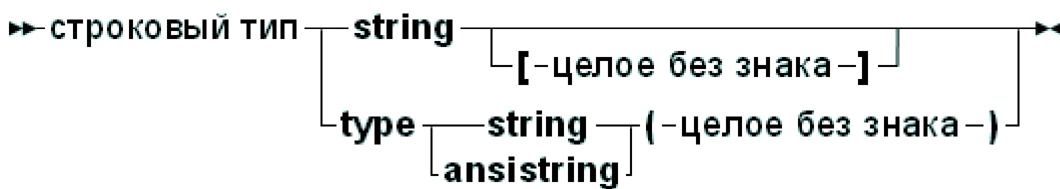
UCS2Char, UCS4Char, UniCodeChar. Но никакой специальной поддержки для этих символьных типов не существует, они были определены только для совместимости с Delphi.

3.2.4 Однобайтовые строковые тип

Free Pascal поддерживает строковый тип (String) , как это определено в Turbo Pascal: последовательность символов с необязательным указанием размера. Он также поддерживает AnsiStrings *(строки неограниченной длинны)* и информация о кодовой странице *(начиная с версии **3.0** Free Pascal)* как и в Delphi.

Для объявления строковой переменной, используйте следующую спецификацию типа:

Строковые типы



Если есть спецификатор размера *(в квадратных скобкках)*, то он указывает **максимальный** размер, **максимальный** размер строки по умолчанию *(если скобок нет)* – составляет **255** *(символов)*. Если есть спецификатор кодовой страницы, *(с помощью круглых скобок)* он указывает на тип AnsiString с сопутствующей информацией о кодовой странице.

Смысл объявления строки без указания размера и кодовой страницы интерпретируется в зависимости от директивы {$H}.

```
var
  A : String;
```

Если не указан размер или кодовая страница, так можно объявить AnsiString или ShortString *(короткую строку)*.

Вне зависимости от фактического типа строк, AnsiString и ShortString могут быть взаимозаменяемы. Компилятор всегда заботится о необходимых преобразованиях типов. Но результатом выражения, которое содержит AnsiStrings и ShortString, всегда будет AnsiString.

Короткие строки (ShortStrings)

Строка будет объявлена как короткая (ShortString) в следующих случаях:

1. Если директива {$H-} строка всегда объявляется как короткая (ShortString).
2. Если директива {$H+} и указана максимальная длина (размер) строки, строка будет объявлена как короткая (ShortString).

Короткие строки всегда используют кодовую страницу системы *(текущую, по умолчанию)*. Тип ShortString определен как строка размером **255** символов:

```
ShortString = String[255];
```

Если размер строки не указан, берется в качестве значения по умолчанию **255**. Фактическая длина строки может быть получена с помощью функции Length. Например:

```
{$H-}
 Type
   NameString = String[10];
   StreetString = String;
```

NameString может содержать максимум **10** символов. В то время как StreetString может содержать до **255** символов.

Замечание:

Короткие строки имеют максимальную длину **255** символов: при определении максимальной длины, она не может превышать **255**. Если будет предпринята попытка установить длину больше **255**, компилятор выдаст сообщение об ошибке:

```
Error: string length must be a value from 1 to 255
Ошибка: длина строки должна быть в диапазоне от 1 до 255
```

Для коротких строк, длина хранится в символе с индексом **0**. Поскольку старый код Turbo Pascal зависит от этого, он реализован так же и во Free Pascal.

Несмотря на это, чтобы писать переносимый код, лучше устанавливать длину ShortString с помощью вызова SetLength, а для ее получения использовать вызов Length. Эти функции будут работать всегда, независимо от внутреннего представления ShortStrings или других строк: это позволяет легкое переключаться между различными строковыми типами.

Строки AnsiString

AnsiStrings это строки, которые не имеют ограничение по длине *(Начиная с версии 3.0 Free Pascal)*. Для их ведется подсчет количества ссылок на них, и гарантируется что они будут завершены нулем.

Внутренне, AnsiString интерпретируется как указатель: фактически содержимое строки находится в "куче", в ней выделяется столько памяти, сколько необходимо для хранения содержимого строки.

Если не определена кодовая страница, то берётся кодовая страница системы по умолчанию. Эта кодовая страница определяется константой `DefaultSystemCodePage` в модуле System.

Это все обрабатывается прозрачно для программиста, то есть ими можно управлять как обычными короткими строками. AnsiStrings может быть определен, используя предопределенный тип `AnsiString`.или используя ключевое слово `String` в режиме `{$H+}`.

Замечание:

> Завершающий нуль не означает, что нулевые символы (`char(0)` или `#0`) не могут использоваться внутри строки: завершение нулём ***не используется*** внутри, но поддерживается для удобства использования с внешними подпрограммами, которые ожидают завершенную нулем строку *(как делают большинство подпрограмм C)*.

Если включена директива переключатель `{$H}`, то определение строки, использует ключевое слово `String` не содержащее спецификатора длины, будет расцениваться как `AnsiString`. Если будет присутствовать спецификатор длины, то будет использоваться тип `ShortString`, независимо от наличия директивы `{$H}`.

Если строка пустая (`''`), то внутреннее представление строки — указатель, равный `Nil`. Если строка не пуста, то указатель указывает на структуру в динамической памяти.

Внутреннее представление в виде указателя, и автоматическое завершение нулевым символом позволяет преобразовать тип `AnsiString` к `PChar`. Когда строка пуста (в таком случае указатель равен `Nil`) компилятор удостоверяется, что преобразованный тип `PChar` указывает на нулевой байт.

Присвоение одного `AnsiString` другому не приводит к фактическому перемещению строки. Оператор

```
S2:=S1;
```

приводит к тому что счетчик ссылок указывающих на `S2` уменьшается на **1**, а указывающих на `S1` увеличивается на **1**, и наконец `S1` (как указатель) копируется в `S2`. Это - существенное ускорение работы кода.

Если счетчик ссылок строки достигает нуля, то память, занятая строкой, освобождается автоматически, и указатель устанавливается в `Nil`, таким образом, не возникают утечки памяти.

При объявлении переменной типа `AnsiString`, компилятор Free Pascal первоначально выделяет память только для указателя, не больше. Этот указатель гарантировано будет равен `Nil`, означая, что строка первоначально пуста. Это истинно для локальных и глобальных переменных типа `AnsiStrings` или переменных которые являются частью структур (**массивов**, **записей** или **объектов**).

Это вносит накладные расходы. Например, объявление,

```
Var
  A : Array[1..100000] of string;
```

скопирует значение Nil 100 000 раз в A. Когда A покидает область видимости, счетчик ссылок каждой из 100000 строк будет уменьшен на единицу для каждой строки в отдельности. Все это происходит невидимо для программиста, но при рассмотрении проблем с производительностью, это важно.

Память для содержимого строки будет выделена только тогда, когда строке будет присвоено значение. Если строка выходит из области видимости, то ее счетчик ссылок автоматически уменьшается на 1. Если счетчик ссылок достигает нуля, память, выделенная для строки, освобождается.

Если символу строки, у которой счетчик ссылок больше чем 1, присвоено значение, как в следующих операторах:

```
S:=T;  { количество ссылок на S и T в настоящее время равно
2}
S[I]:='@';
```

то перед присвоением создается копия строки. Этот подход известен как *копирование при записи*. Существует возможность получить строку с количеством ссылок гарантировано равным 1 используя функцию UniqueString.

```
S:=T;
R:=T; // Счетчик ссылок T не меньше 3
UniqueString(T);
// Счетчик ссылок T гарантировано равен 1
```

Рекомендуется делать это, при преобразовании типа AnsiString к типу PChar и при передаче его в подпрограмму C, которая изменяет переданную строку.

Чтобы получить длину строки AnsiString нужно использовать функцию Length: длина строки не находится в нулевом символе. Конструкция

```
L:=ord(S[0]);
```

была допустима в Turbo Pascal для ShortStrings, но она не корректна для AnsiStrings. Компилятор выдаст предупреждение, если встретятся с такой конструкцией.

Чтобы установить длину строки AnsiString, нужно использовать функцию SetLength. Константы AnsiStrings имеют счетчик ссылок равный -1 и обрабатываются особенно. То же замечание нужно сделать и относительно функции Length: конструкция

```
L:=12;
S[0]:=Char(L);
```

была допустима в Turbo Pascal для ShortStrings, но она не корректна для AnsiStrings. Компилятор выдаст предупреждение, если встретятся с такой конструкцией.

AnsiStrings преобразуются компилятором в ShortStrings если нужно, это означает, что AnsiStrings и ShortStrings могут быть "смешаны" *(в коде)*.

AnsiStrings может быть преобразован к типу PChar или типам Pointer:

```
Var P : Pointer;
    PC : PChar;
    S : AnsiString;
begin
   S :='Это AnsiString';
   PC:=Pchar(S);
   P :=Pointer(S);
```

Между двумя преобразованиями типов есть различие. Когда пустая AnsiString строка будет преобразована к типу указатель, указатель будет равен Nil. Если пустая AnsiString строка будет преобразована к типу PChar, то результатом будет указатель на нулевой байт (пустую PChar строку) .

Результатом такого преобразования следует пользоваться с осторожностью. Лучше использовать результат такого приведения только для чтения, т.е. он подходит только как параметр для передачи в процедуру, которая нуждается в аргументе-константе типа PChar.

Поэтому ***не*** желательно делать приведения типов в следующих случаях:

1. **выражениях**.
2. **строках**, у которых счетчик ссылок, больше чем **1**. В этом случае Вы должны вызвать Uniquestring, чтобы гарантировать, что у строки счетчик ссылок равен **1**.

Преобразование кодовой страницы

Так как строки имеют информацию о кодовой странице, связанную с ними, важно знать, в какой кодовой странице используется данная строка:

1. Короткие строки (ShortString) всегда используют системную кодовую страницу *(по умолчанию)*.
2. AnsiStrings использует системную кодовую страницу *(по умолчанию)*.
3. **Одиночные байтовые строки** объявляются с текущей кодовой страницей.
4. Тип RawByteString не имеет информации о кодовой странице, связанной с ним.

Компилятор преобразует кодовую страницу нужным образом: При присвоении строки, кодовая страница исходной строки всегда будет преобразовываться в

кодовую страницу результирующей строки.

Это означает, что в следующем коде:

```
Type
  TString1 = Type String(1252);
  TString2 = Type String(1251);
Var
  A : TString1;
  B : TString2;
begin
  A:='123'+'345'+intToStr(123);
  B:=A;
  Writeln('B : ',StringRefCount(B));
  Writeln('A : ',StringRefCount(A));
end.
```

Компилятор преобразует содержимое строки B в соответствии с кодовой страницей строки A. Обратите внимание, что если происходит преобразование кодовой страницы, то ***не используется*** механизм счетчика ссылок: для новой строки будет выделена память.

Автоматизированное преобразование кодовых страниц может серьезно замедлить код, поэтому необходимо соблюдать осторожность, чтобы преобразования кодовых страниц было сведено к минимуму.

Кодовую страницу строки можно задать явно с помощью процедуру SetCodePage модуля System. Вызов этой процедуры будет преобразовывать значение строки к запрашиваемой кодовой страницы.

Примечание:

Преобразование кодовой страницы может привести к потере данных: если в результирующей кодовой странице нет нужного символа *(символ не определён)*.

Примечание:

Поддержка кодовой страницы требует некоторые вспомогательные процедуры, они реализованы в менеджере UnicodeString. В Windows для этого используются системные процедуры. В Unices, используется модуль cwString для установления связи с библиотекой C и использует её поддержку и её функции преобразования. В качестве альтернативы, можно использовать модуль fpWideString который содержит менеджер UnicodeString реализованный в Object Pascal.

Необработанная строка байт (RawByteString)

Предопределенный тип RawByteString является строка типа AnsiString, без информации о кодовой странице (CP _NONE):

```
Type
  RawByteString = type AnsiString(CP_NONE);
```

Он обрабатывается так что, если процедура преобразования встречает CP_NONE в целевой строке - преобразование кодовой страницы не выполняется, сохраняется кодовая страница исходной строки.

По этой причине большинство подпрограммы в модулях SysUtils и System используют тип строка байт (RawByteString).

UTF8String

Однобайтовые строки могут хранить ***только*** набор символов, доступных в этой кодовой страницы. Символы, которые ***нет*** в кодовой странице, не могут быть в этой строке. Unicode кодировка UTF-8 является кодировкой, которая может быть использована с однобайтными строками: Таблица ASCII-символов *(128 разрядная таблица)* в этой кодировке строго CP_ACP. Этот факт используется для пере определения одного байта строки, содержащей все символы:

```
Type
  UTF8String = type AnsiString(CP_UTF8);
```

Строка типа UTF8String может использоваться для представления всех символов Unicode. Это возможно! Так как для символа юникода *(unicode)* может потребоваться несколько байтов, которые будут представлены в кодировке UTF-8, однако есть ***два замечания***, которые надо учитывать при использовании UTF8String:

1. Осторожно нужно пользоваться ***индексом символа*** - т.к размер символа *(в общем случае)* неравен **1** байт, то ***индекс символа*** и ***номер байта*** - не одно и то-же: то есть выражение S[i] не всегда указывает на допустимым символом в строке S *(типа UTF8String)*.
2. Количество байт в строке ***не равно*** числу символов в строке. Стандартная функция length ***не может*** быть использована ***для получения длины символа***, она всегда будет возвращать длину **1** байт.

Для всех *(остальных)* кодовых страниц, количество однобайтных символов в строке равна длине байт строки.

3.2.5 Многобайтные строковые типы

Для многобайтными типов строк, символ имеет базовый размер по минимум **2** байта. Это означает, что он может быть использован для хранения символа юникода в кодировке UTF16 или в UCS2.

Строки Unicode (UnicodeStrings)

UnicodeStrings *(используется для представления строк символов Unicode)* реализованы во многом так же, как AnsiStrings: подсчет ссылок, нулевой символ в конце массива, только они реализованы в виде массивов символов WideChars вместо обычных Chars. WideChar является двухбайтовым символом (элементом DBCS: набор двухбайтовых символов). Для UnicodeStrings (WideStrings) в основном применяются те же правила что и для AnsiStrings. Компилятор прозрачно преобразует UnicodeStrings (WideStrings) к AnsiStrings и наоборот.

Аналогично преобразованию типа AnsiString к типу PChar завершенному нулем массиву символов, UnicodeString может быть преобразован в PUnicodeChar, завершенный нулем массив символов. Отметьте, что массив PUnicodeChar завершен **2**-мя нулевыми байтами вместо **1**, таким образом, преобразование типа к PChar не является автоматическим.

Компилятор не предоставляет поддержки для любого преобразования из Unicode в AnsiStrings или наоборот. Модуль system содержит менеджер обработки UnicodeString, который может быть инициализирован с некоторым специфичным для ОС подпрограммами обработки строк unicode. Для получения дополнительной информации см. руководство модуля system.

Строка unicode может быть построена таким же образом, как WideChar:

```
Const
 ws2: unicodestring = 'Ψ Ω: '#$03A8' '#$03A9;
```

Большие строки (WideStrings)

WideStrings (используется для представления строк символов Unicode в приложениях COM) реализованы во многом так же, как UnicodeStrings. В отличие от последних, для них ***не*** подсчитываются ссылки, и на Windows они ассоциированы со специальными Windows -функциями, которые позволяют им быть использованными для автоматизации OLE. Это означает, что они реализованы в виде заканчивающихся нулем массивов символов WideChars, вместо обычных символов Char. WideChar является двухбайтовым символом (элемент DBCS: набор двухбайтовых символов). В основном для WideStrings применяются те же правила что и для AnsiStrings. Как и с UnicodeStrings, компилятор прозрачно преобразует WideStrings к AnsiStrings и наоборот.

Для приведения типов и преобразования, применяются те же правила, что и для типа UnicodeString.

3.2.6 Строковые константы (Constant strings)

Чтобы определить строковую константу, ее нужно заключить в одинарные

кавычки, так же, как тип Char, только здесь допускается более одного символа. Учитывая, что S имеет тип String, следующие присвоения допустимы:

```
S := 'Это строка.';
S := 'Один'+', Два'+', Три';
S := 'This isn''t difficult !';//Так определяется апостроф
S := 'Это дополнительный символ: ' #145 ' !';
```

Как может быть замечено, символ одинарной кавычки определен **2**-мя символами одинарной кавычки следующих друг за другом. Непечатные символы могут быть определены с помощью их символьных значений (обычно это ASCII код). Пример также показывает, что могут быть добавлены две строки. Результирующая строка - это конкатенация первой и второй строки, без пробелов между ними. ***Строки не могут быть вычтены*** *(знак "–")*.

Строка сохраняется как AnsiString или ShortString, в зависимости от установки директивы-переключателя {$H}.

3.2.7 PChar - строки завершённые нулём

Free Pascal поддерживает реализацию типа PChar как в Delphi. PChar определен как указатель на тип Char, но позволяет дополнительные операции. Тип PChar лучше всего воспринимать как Pascal-эквивалент для завершенных нулем строк из языка Си, то есть переменная типа Pchar это указатель, который указывает на массив символов завершенных нулем (#0). Free Pascal поддерживает инициализацию типизированных констант PChar, или прямого присвоения. Следующие части кода эквивалентны:

```
program one;
var P : PChar;
begin
  P := 'Это заканчивающаяся нулём строка.';
  WriteLn (P);
end.
```

Приведет к тому же, что и следующий код

```
program two;
const P : PChar = 'Это заканчивающаяся нулём строка.';
begin
  WriteLn (P);
end.
```

Эти примеры также показывают, что возможно записать содержание строки в файл типа Text. Модуль strings содержит процедуры и функции, для манипуляции типом PChar как в стандартной библиотеке C. Так как PChar эквивалентен указателю на переменную типа Char, также возможно сделать следующее:

```
Program three;
Var S : String[30];
    P : PChar;
begin
  S := 'Это заканчивающаяся нулём строка.'#0;
  P := @S[1];
  WriteLn (P);
end.
```

Этот код приведет к тому же результату что и предыдущие два примера. *Завершенные нулем* строки не могут быть объединены как нормальные Pascal строки. Если нужно объединить две PChar строки; должны использоваться ответствующие функции из модуля strings.

Однако, возможно использовать адресную арифметику с указателями. Операторы + и − могут использоваться для выполнения операций над указателями PChar. В таблице (3.5), **P** и **Q** имеют тип PChar, а **I** имеет тип LongInt.

Таблица 3.5: Адресная арифметика с указателями PChar

Операция	Результат
P + I	Добавляет I по адресу, на который указывает P.
I + P	Добавляет I по адресу, на который указывает P.
P - I	Вычитает I из адреса, на который указывает P
P - Q	Возвращает как целое, расстояние между **2**-мя адресами (или количество символов между P и Q)

3.2.8 Размеры строк

Память, занятая строкой зависит от типа строки. Некоторые типы строк выделяют память в "куче", другие в стеке. Таблица 3.6 показывает как выделяется память для различных типов строк. В таблице используются следующие сокращения:

1. L - Фактическая длина строки (в байтах).
2. HS - Старше Free Pascal 2.7.1 - **16** байт, младше - зависит от версии Free Pascal.
3. WHS - Размер составляет **4** байта для всех версий Free Pascal.
4. UHS - Размер составляет **8** байт для всех версий Free Pascal.

Таблица 3.6: Размеры типов строка в памяти

Тип строки	Размер в стеке	Размер в "куче"
ShortString	Указанная длина + 2	**0**
AnsiString	Размер указателя *(Pointer)*	L + **1** + HS
WideString	Размер указателя *(Pointer)*	**2***(L + **1**) + WHS
UnicodeString	Размер указателя *(Pointer)*	**2***(L + **1**) + UHS
Pchar	Размер указателя *(Pointer)*	L+**1**

3.3 Структурированные Типы

Структурированный тип это тип, который может хранить несколько значений в одной переменной. Структурированные типы могут быть неограниченно вложены друг в друга.

Структурный тип

В отличие от Delphi, Free Pascal ключевое слово `Packed` ***не поддерживает*** для структурированных типов. В следующих разделах описывается каждый из возможных структурированных типов. Будут указаны, типы которые поддерживают ключевое слово `packed`.

Упакованные структурированные типы

После объявления структурированного типа, не нужно делать никаких предположений о внутреннем размещении элементов в типе. Компилятор разместит элементы структуры в памяти, так как по его мнению, будет самым подходящим. Таким образом, порядок элементов будет сохранен, но расположение элементов не гарантируются, и частично управляется директивой `$PACKRECORDS` *(эта директива объяснена в Справочник*

программиста Free Pascal).

Тем не менее, Free Pascal позволяет контролировать размещение с помощью ключевых слов `Packed` и `Bitpacked`. Смысл этих ключевых слов зависит от контекста:

Bitpacked

В этом случае компилятор попытается выровнять перечислимые типы на разрядных границах, как объяснено ниже.

Packed

Значение ключевого слова `Packed` зависит от ситуации:

1. В режиме `MACPAS` оно эквивалентно ключевому слову `Bitpacked`.
2. В других режимах, с ***включенной*** директивой `$BITPACKING`, оно также эквивалентно ключевому слову `Bitpacked`.
3. В других режимах, с ***выключенной*** `$BITPACKING`, оно означает нормальную упаковку на границах байтов.

Упаковка на границах байта означает, что каждый новый элемент структурированного типа начинается на границе байта.

Механизм байт-упаковки прост: компилятор выравнивает каждый элемент структуры на первой доступной границе байта, даже если размер предыдущего элемента *(небольшие перечислимые типы, типы под диапазоны)* является меньше чем байт.

При использовании механизма битовой упаковки компилятор для каждого перечислимого типа вычисляет, количество битов необходимых для его размещения. Следующий указанный тип будет размещен начиная со следующего свободного бита.

Не перечислимые типы - которые включают, но не ограничены - множествами, вещественными числами, строками, *(побитно упакованными)* записями, *(побитно упакованными)* **массивами**, **указателями**, **классами**, **объектами**, и **процедурными переменными**, выравниваются на первой доступной границе байта.

Отметьте, что внутреннее представление побитовой упаковки непрозрачно: в будущем оно может измениться в любое время. Более того: внутренняя упаковка зависит от порядка байтов на платформе, для которой производилась компиляция, и никакое преобразование между платформами не возможно. Это делает побитно упакованные структуры неподходящими для сохранения на диск или передачу через сеть. Однако используется тот же формат что,и в Компиляторе Pascal GNU, и команда Free Pascal стремится сохранять эту совместимость в будущем.

Есть еще некоторые ограничения на элементы побитно упакованных структур:

- Адрес не может быть получен, если размер в битах не кратен числу **8**, и элемент, не выровнен на границе байта.
- Элемент побитно упакованной структуры не может использоваться в

качестве var-параметра, если его размер в битах не кратен числу **8**, и элемент, не выровнен на границе байта.

Для определения размера элемента в побитно упакованной структуре, используется функция `BitSizeOf`. Она возвращает размер элемента - в битах. Для других типов или элементов структур, которые не упакованы побитно, она просто возвратит размер в байтах, умноженных на **8**, то есть, возвращаемое значение равное `8*SizeOf`.

Размер побитно упакованных записей и массивов ограничен:

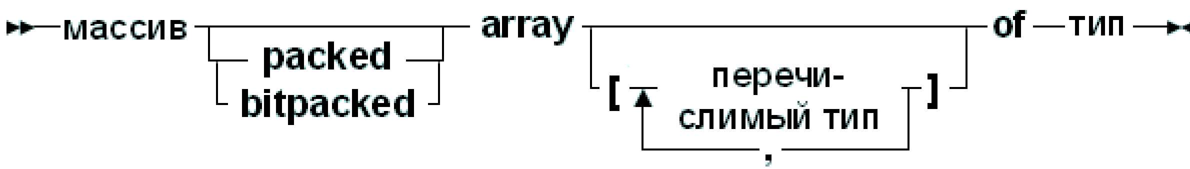
- На **32-х** битных системах максимальный размер составляет $\mathbf{2^{29}}$ байтов (512 MB).
- На **64-х** битных системах максимальный размер составляет $\mathbf{2^{61}}$ байтов.

Причина такого ограничения состоит в том, что смещение элемента должно быть вычислено с помощью максимально большого целочисленного типа в данной системе.

3.3.1 Массивы

Free Pascal поддерживает массивы, как в Turbo Pascal. Также поддерживаются многомерные и (побитно) упакованные *[(bit)packed]* массивы, а также динамические массивы Delphi:

Тип массив



Статические массивы

Когда диапазон массива включен в определение массива, он называется ***статическим массивом***. Попытка доступа к элементу с индексом, который находится вне указанного диапазона будет генерировать ошибку времени выполнения (если включена проверка диапазона). Ниже приведен пример правильного объявления массива:

```
Type
  RealArray = Array [1..100] of Real;
```

Индексы допустимые для доступа к элементу массива находятся в границе от **1** до **100**, включительно с **1** и **100**. Как и в Turbo Pascal, если тип массива является сам по себе массивом, можно объединить два массива в один многомерный массив. Следующее объявление:

```
Type
  APoints = array[1..100] of Array[1..3] of Real;
```

эквивалентно объявлению:

```
Type
  APoints = array[1..100,1..3] of Real;
```

Функции `High` и `Low` возвращают самое высокое и низкое разрешённое значения индекса массива. В приведенном выше случае, это будет **100** и **1**. Вы должны по возможности использовать их, так как они улучшают управляемость вашего кода. Использование обеих функций является столь же эффективно, как использование констант, так как они вычисляются во время компиляции.

Когда переменная типа статический массив присвоена другой переменной, копируется содержание целого массива. Это также истинно для многомерных массивов:

```
program testarray1;

Type
  TA = Array[0..9,0..9] of Integer;

var
  A,B : TA;
  I,J : Integer;
begin
  For I:=0 to 9 do
    For J:=0 to 9 do A[I,J]:=I*J;
  For I:=0 to 9 do
    begin
      For J:=0 to 9 do Write(A[I,J]:2,' ');
      Writeln;
    end;
  B:=A;
  Writeln;
  For I:=0 to 9 do
    For J:=0 to 9 do A[9-I,9-J]:=I*J;
  For I:=0 to 9 do
    begin
      For J:=0 to 9 do Write(B[I,J]:2,' ');
      Writeln;
    end;
end.
```

Выводом этой программы будут **2** идентичных матрицы.

Динамические массивы

Начиная с версии **1.1**, Free Pascal также поддерживает, динамические массивы: В этом случае диапазон массива не указывается, как в следующем примере:

```
Type
  TByteArray = Array of Byte;
```

При объявлении переменной типа динамический массив начальная длина массива равна нулю. Фактическая длина массива должна быть установлена стандартной функцией SetLength, которая выделит память необходимую для размещения элементов массива в куче. Следующий пример установит длину массива равную **1000**:

```
Var
  A : TByteArray;

begin
  SetLength(A,1000);
```

После вызова SetLength, допустимыми индексами массива будут числа от **0** до **999**: индексация массива всегда начинается с нуля.

Обратите внимание, что длина массива задается в элементах, а не в байтах выделяемой памяти *(хотя они могут быть и одинаковыми)*. Объем выделенной памяти равняется размеру массива умноженном на размер одного элемента в массиве. Память будет освобождена при выходе из текущей процедуры или функции.

Также есть возможность изменить размер массива: в этом случае будет сохранено настолько много элементов сколько поместится в новом размере массива. Размер массива может быть установлен в ноль, это эффективный способ сбросить переменную.

Попытка получить доступ к элементу массива с индексом, выходящим за границы объявленного диапазона, всегда генерирует ошибку периода выполнения.

Для динамических массивов ведется подсчет ссылок: присвоение одной переменной типа динамический массив, другой переменной, позволит обеим переменным указывать на тот же самый массив. В отличие от AnsiStrings, присвоение значения одному элементу массива будет отражено и в другом: не делается никакой копии массива при записи. Рассмотрите следующий пример:

```
Var
  A,B : TByteArray;

begin
  SetLength(A,10);
  A[0]:=33;
  B:=A;
```

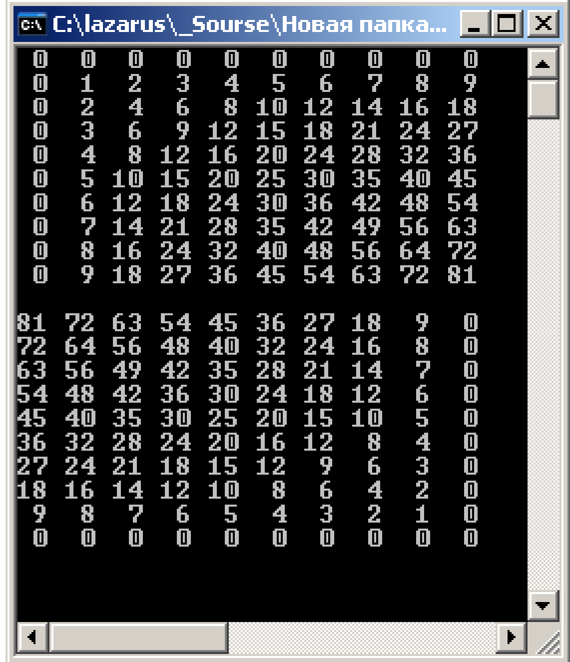
```
  A[0]:=31;
```

После второго присвоения первый элемент в B будет также содержать **31**.

Это также можно заметить по выводу следующего примера:

```
program testarray1;

Type
  TA = Array of array of Integer;

var
  A,B : TA;
  I,J : Integer;

begin
  Setlength(A,10,10);
  For I:=0 to 9 do For J:=0 to 9 do A[I,J]:=I*J;
  For I:=0 to 9 do
   begin
     For J:=0 to 9 do Write(A[I,J]:2,' ');
     Writeln;
   end;
  B:=A;
  Writeln;
  For I:=0 to 9 do For J:=0 to 9 do  A[9-I,9-J]:=I*J;
  For I:=0 to 9 do
   begin
     For J:=0 to 9 do Write(B[I,J]:2,' ');
     Writeln;
   end;
end.
```

Выводом этой программы будет матрица чисел, а затем будет выведена такая же матрица, но зеркально.



Как отмечалось ранее, для динамических массивов ведется подсчет ссылок: если в одном из предыдущих примеров A выйдет за границы области а B нет, то массив ещё не будет освобожден: счетчик ссылок A (и B) уменьшится на **1**. Как только счетчик ссылок достигнет нуля, память, выделенная для содержания массива, освобождается.

Вызов SetLength установит счетчик ссылок на указанный массив в 1, если он больше, то после вызова SetLength переменные динамического массива больше не будут указывали на одну и ту же память.

```
program testunique;

Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;
begin
  Setlength(A,10);
  For I:=0 to 9 do A[I]:=I;
  B:=A;
  SetLength(B,6);
  A[0]:=123;
  For I:=0 to 5 do Writeln(B[I]);
end.
```

Также возможно скопировать и/или изменить размер массива с помощью стандартной функции Copy, которая действует как функция копирования для строк:

```
program testarray3;
```

```
Type
  TA = array of Integer;

var
  A,B : TA;
  I : Integer;
begin
  Setlength(A,10);
  For I:=0 to 9 do A[I]:=I;
  B:=Copy(A,3,6);
  For I:=0 to 5 do Writeln(B[I]);
end.
```

Функция `Copy` скопирует **6** элементов массива в новый массив. Начиная с элемента по индексу **3** *(то есть четвертого элемента)* массива.

Функция `Length` возвратит число элементов в массиве. Функция `Low` для динамического массива будет всегда возвращать **0**, а функция `High` возвратит значение `Length-1`, то есть, значение самого большого позволенного индекса массива.

Типы совместимые с Динамическими массивами

Object Pascal является строго типизированным языком. Два формально разных типа иногда рассматриваются как совместимые (то есть значение одного типа может быть присвоено переменной другого типа) это возможно при определенных обстоятельствах. Динамические массивы совместимы по присваиванию, если они используют один и тот же тип элемента *(массива)*. Это означает, что следующий код будет скомпилирован:

```
{$mode objfpc}
Type
  TA = Array of Integer;
  TB = Array of Integer;

Var
  A : TA;
  B : TB;
begin
  SetLength(A,1);
  A[0]:=1;
  B:=A;
end.
```

Но следующий код ***не будет***, не смотря на то, что типы `Integer` и `Word` совместимы присваивания:

```
{$mode objfpc}
Type
  TA = Array of Word;
  TB = Array of Integer;

Var
  A : TA;
  B : TB;
begin
  SetLength(A,1);
  A[0]:=1;
  B:=A;
end.
```

Конструктор Динамического массива

В версии **3.0** Free Pascal, тип динамические массивы имеют ***конструктор***. Это обеспечивает компилятор из внутреннего устройсва динамических массивов. До версии **2.6.4**, был единственный способ инициализировать динамические массивы следующим образом:

```
Type
  TIntegerArray = Array of Integer;

var
  A : TIntegerArray;
begin
  SetLength(A,3);
  A[0]:=1;
  A[1]:=2;
  A[3]:=3;
  Writeln(Length(A));
end.
```

В версии **3.0** Free Pascal, динамический массив можно инициализировать с помощью конструктороподобного синтаксиса. Вызывается конструктор `Create`, и принимает в качестве параметров значения, тип которых соответствует типу элементов массива, а количество - будет определять размерность массива. Это означает, что приведённая выше инициализация может быть сделана, так:

```
Type
  TIntegerArray = Array of Integer;

var
  A : TIntegerArray;
begin
```

```
    A:=TIntegerArray.Create(1,2,3);
    Writeln(Length(A));
  end.
```

Обратите внимание, что это не будет работать для динамических массивов, для которых не был создан свой тип. То есть, следующий код работать не будет:

```
  var
    A : Array of Integer;
  begin
    A:=Array of Integer.Create(1,2,3);
    Writeln(Length(A));
  end.
```

Этот подход также работает рекурсивно, для многомерных массивов:

```
  Type
    TIntegerArray = Array of Integer;
    TIntegerArrayArray = Array of TIntegerArray;

  var
    A : TIntegerArrayArray;
  begin
    A:=TIntegerArrayArray.Create(TIntegerArray.Create(1,2,3),
                                 TIntegerArray.Create(4,5,6),
                                 TIntegerArray.Create(7,8,9));
    Writeln('Length ',length(A));
  end.
```

Однако, поскольку он является конструктором *(его код выполняется во время выполнения)* не представляется возможным использовать его в синтаксисе инициализации переменной. То есть, следующий код работать не будет:

```
  Type
    TIntegerArray = Array of Integer;

  var
    A : TIntegerArray = TIntegerArray.Create(1,2,3);
  begin
    Writeln('Length ',length(A));
  end.
```

Упаковка и распаковка массивов

Массивы могут быть ***упакованы*** *(или побитно упакованы (*`bitpacked`*))* и ***распакованы***. Два типа массива, у которых одинаковый диапазон допустимых индексов и тип элементов, но которые по-разному упакованы, ***не являются*** совместимым для присвоений.

Однако, возможно преобразовать нормальный массив в побитно упакованный

массив процедурой pack *(упаковки)*. Обратная операция также возможна; побитно упакованный массив может быть преобразован в обычно упакованный массив, используя процедуру unpack *(распаковки)*, как в следующем примере:

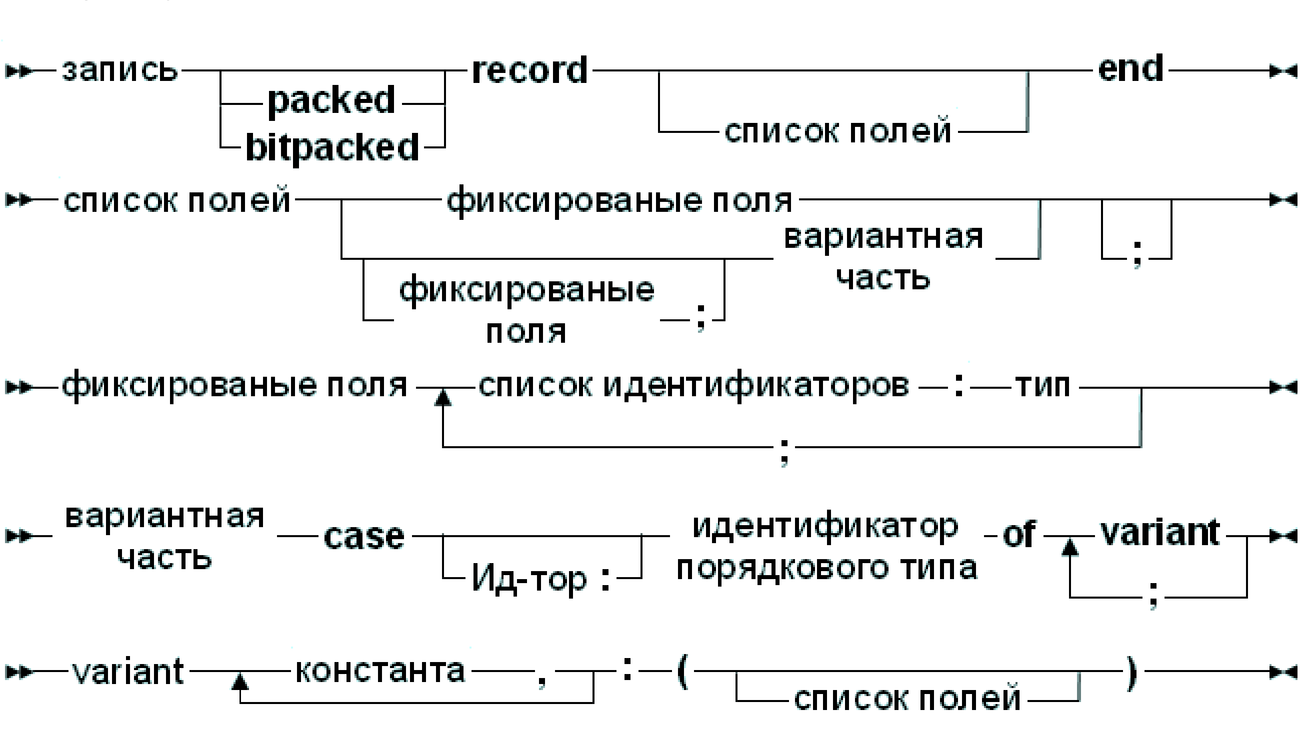
```
Var
  foo : array [ 'a'..'f' ] of Boolean
    = ( false, false, true, false, false, false );
  bar : packed array [ 42..47 ] of Boolean;
  baz : array [ '0'..'5' ] of Boolean;
begin
  pack(foo,'a',bar);
  unpack(bar,baz,'0');
end.
```

Больше информации о подпрограммах pack *(упаковки)* и unpack *(распаковки)*, можно найти в описании модуля system.

3.3.2 Записи

Free Pascal поддерживает фиксированные ***записи***, а также записи с вариантными полями. Синтаксическая диаграмма для записей

Тип запись



Так, действительными являются следующие объявления типов записей:

```
Type
   Point = Record
      X,Y,Z : Real;
```

```
end;

RPoint = Record
   Case Boolean of
      False : (X,Y,Z : Real);
      True : (R,theta,phi : Real);
end;

BetterRPoint = Record
   Case UsePolar : Boolean of
      False : (X,Y,Z : Real);
      True : (R,theta,phi : Real);
end;
```

Вариантная часть должна быть последней в записи. Опциональный идентификатор в операторе `case` служит для того, чтобы получить доступ к значению поля, которое иначе было бы невидимо для программиста. Оно может использоваться, чтобы видеть, какая вариантная часть является активной в определенное время. *(Это нужно для поддержания этого поля)* В действительности оно представляет новое поле в записи.

Замечание:

Возможно вложить различные части, как в записи:

```
Type
   MyRec = Record
           X : Longint;
           Case byte of
             2 : (Y : Longint;
                  case byte of
                  3 : (Z : Longint);
                  );
           end;
```

Структура и размер Записи

На **структуру** и **размер** записи влияют пять аспектов:

- Размер его полей.
- Запрос на выравнивание полей, которое зависит от платформы. Обратите внимание, что запрос на выравнивание полей внутри записи могут отличаться для разных (на разных платформах) переменных этого типа. Кроме того, на расположение поля внутри записи могут также влиять требования к выравниванию записи.
- Текущая активная установка `{$ALIGN N}` или `{$PACKRECORDS N}` *(эти параметры переопределяют друг друга, поэтому действует последний; обратите внимание , что эти директивы не должны принимать одни и*

те же аргументы, обратитесь к Справочник программиста Free Pascal для получения дополнительной информации).

- Текущая установка параметра {$CODEALIGN RECORDMIN = X}.
- Текущая установка параметра {$CODEALIGN RECORDMAX = X}.

Расположение и размер вариантных частей записей определяется путем замены их с поля, тип которого является запись с первым элементом поле типа поля тега , если идентификатор был объявлен для этого поля тега, за которым следуют элементы самой большой вариант.

Расположение и размер вариантных частей записей определяется путем наложения *(в памяти)* всех вариантных полей одного ранга, ***таким образом поля что поля одного типа могут быть преобразованы в поля другого***, таким образом размер всей записи определяется самой большой вариантной частью.

Смещение каждого следующего *(F2)* поля, это сумма смещения и размера предыдущего поля *(F1)*, оно округляется до степени двойки *(F2)*. Требуемое выравнивание рассчитывается следующим образом:

- Выравнивание будет по умолчанию для полей данного типа, оно может корректироваться если, нужно для данного типа в записи.
- Если требуемое выравнивание хуже, чем активная в настоящее время установка {$CODEALIGN RECORDMIN = X}, то применяется значение X.
- Если в данный момент активна установка {$ALIGN N} или {$PACKRECORDS N}
  - o числовое значение: если требуемое выравнивание больше, чем N, оно изменяется на N. Т.е. если N равно **1**, то все поля будут размещены друг за другом.
  - o RESET или DEFAULT: результат выравнивания зависит от целевой платформы.
  - o C :выравнивание регулируется правилами, указанными в официальном ABI *(Application Binary Interface) (Двоичный (бинарный) Интерфейс Приложений)* для текущей платформы.
  - o POWER/POWERPC, MAC68K: значение выравнивания определяется официальными правилами ABI для соответственно платформ Macintosh PowerPC *(классического)* или Macintosh 680x0.

Размер записи равен сумме размеров всех полей записи, при этом смещения каждого поля это размер предыдущего округлённый до кратного требуемому выравниванию для записей. Выравнивание записей определяется следующим образом:

- Общее выравнивание выбирается исходя из выравнивания каждого поля в записи, как поля с наибольшим выравниванием.
- Если текущая установка {$ALIGN N} или {$PACKRECORDS N} отличается от выравнивания в C и выравнивание больше, чем активная {$CODEALIGN RECORDMAX=X}, то выравнивание изменяется на X.
- Если текущая установка {$ALIGN N} или {$PACKRECORDS N} равен C, то нужное выравнивание определяется официальными правилами ABI.

Замечания и примеры

Free Pascal также поддерживает ***'упакованную запись'*** *(packed record)*, которая представляет собой запись, где все элементы выровнены по границе байта. В результате, два следующих объявления эквивалентны:

```
{$PackRecords 1}
Trec2 = Record
  A : Byte;
  B : Word;
  end;
{$PackRecords default}
```

и

```
Trec2 = Packed Record
  A : Byte;
  B : Word;
  end;
```

Обратите внимание на `{$PackRecords Default}` после первого объявления, **чтобы восстановить настройки по умолчанию**!

Учитывая зависимость от характера платформы, как записи расположены в памяти, единственный способ обеспечить совместимое расположение на разных платформах можно с помощью `{$PACKRECORDS 1}` *(учитывая , что все поля объявляются с помощью записи имеют те же значения на этой же платформе)*.

В частности, если должен быть прочитан ***типизированный файл*** с записями, созданный программой на Turbo Pascal, то прочитать файл правильно ***не всегда*** будет возможно. Причина заключается в том, что установка `{$PACKRECORDS N}` по умолчанию Free Pascal не обязательно совместим с Turbo Pascal. Оно может быть изменено в зависимости от установки `{$PACKRECORDS 1}` или `{$PACKRECORDS 2}`, используемой в программе на Turbo Pascal , которая создала файл *(хотя она может закончится неудачно и с `{$PACKRECORDS 2 }` из - за различных требований выравнивания типа между **16** битной MSDOS и текущей платформой)*.

То же замечание относится и к Delphi: обмен данными будет возможен, только если ***и источник, и приёмник*** используют упакованную запись, или если оба находятся на одной платформе, и используют те же настройки `{$PACKRECORDS X}`.

3.3.3 Множества

Free Pascal поддерживает ***тип множество*** как в Turbo Pascal. Прототип объявления множеств следующий:

Тип множество

▸▸— тип множество — set — of — перечисляемый тип —▸◂

Каждый из элементов ***множества*** должен иметь тип `TargetType.TargetType` может быть любым перечислимым типом из диапазона **0..255**. Множество может содержать ***не более*** **255** элементов. Далее следует допустимое объявление множества:

```
Type
  Junk = Set of Char;

  Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  WorkDays : Set of days;
```

Учитывая эти объявления, следующее присвоение является допустимым:

```
WorkDays := [Mon, Tue, Wed, Thu, Fri];
```

Компилятор хранит маленькие множества *(меньше чем 32 элемента)* в `Longint`, если диапазон типа позволяет это. Это делает более быстрой обработку и уменьшает размер программы. В противном случае, множества хранятся в **32**-х байтах.

Над множествами определено несколько операций: объединение или пересечение, добавление или удаление элементов, сравнение. Они документированы в разделе 12.8.5 Операторы действий над множествами [190].

3.3.4 Файловый тип

Файловые типы - типы, которые хранят последовательность некоторого базового типа, который может быть любым типом кроме другого файлового типа. Он может содержать (в принципе) бесконечное число элементов. Файловый тип используются обычно, чтобы хранить данные на диске. Однако, ничто не мешает программисту написать файловый драйвер, который хранит данные ,например в памяти.

Вот описание типа для файлового типа:

Файловый тип

Если не указан идентификатор типа, то файл будет не типизированным файлом; его можно рассматривать как эквивалентный файл байтов. Не типизированные

файлы требуют, чтобы для их обработки использовались специальные команды *(см. Blockread, Blockwrite)*. Следующее объявление объявляет файл записей:

```
Type
  Point = Record
    X,Y,Z : real;
    end;
  PointFile = File of Point;
```

Внутренне, файлы представлены в виде записи FileRec, которая объявлена в модуле Dos или SysUtils.

Специальный тип файла - Текстовый файл (Text), представленный записью TextRec. Файлы текстового *(Text)* типа используют специальные подпрограммы ввода – вывода. Файловые типы Input, Output и StdErr определены в модуле system: они все ***текстового типа*** (Text), и открываются в коде инициализации модуля system.

3.4 Указатели

Free Pascal поддерживает использование указателей. Переменная типа указатель содержит адрес в памяти, где могут храниться данные другой переменной. Тип указателя может быть определен следующим образом:

Тип указатель

```
►►─тип указатель──=──^──идентификатор типа──►◄
```

Как может быть замечено по этой схеме, указатели типизированы, это означает, что они указывают на определенный вид данных. Тип этих данных должен быть известен во время компиляции.

Разыменованный указатель *(обозначается добавлением знака* ^ *после имени переменной)* ведет себя как переменная. Эта переменная имеет тип объявленный при объявлении указателя, и переменная сохранена *(размещена)* по адресу, на который указывает переменная указателя. Рассмотрите следующий пример:

```
Program pointers;
type
 Buffer = String[255];
 BufPtr = ^Buffer;
Var B  : Buffer;
   BP : BufPtr;
   PP : Pointer;
```

```
  и т.д.
```

В этом примере BP - ***указатель на тип*** Buffer; в то время как B - переменная типа Buffer. Переменная B занимает **256** байтов памяти, а BP только **4** (или **8**) байтов: память достаточную, чтобы сохранить адрес.

выражение

```
BP^
```

известно как разыменование BP. Результат имеет тип Buffer, таким образом,

```
BP^[23]
```

обозначает *(указывает на)* **23**-ий символ в строке, на которую указывает BP.

Замечание:

> Free Pascal обрабатывает указатели, так же как и C. Это означает, что указатель на некоторый тип может рассматриваться как массив этого типа.

С этой точки зрения, указатель указывает на нулевой элемент этого массива. Таким образом, следующее объявление указателя

```
Var p : ^Longint;
```

может считаться эквивалентным следующему объявлению массива:

```
Var p : array[0..Infinity] of Longint;
```

Разница в том, что предыдущее объявление выделяет память для указателя *(но не для массива)*, а второе объявление выделяет память для всего массива. Если используется предыдущее объявление, память должна быть выделена вручную, с помощью функции GetMem. Обращение к p^, то же самое что и к p[0]. Возможно, следующая программа более ясно иллюстрирует это:

```
program PointerArray;
var i : Longint;
   p : ^Longint;
   pp : array[0..100] of Longint;
begin
 for i := 0 to 100 do pp[i] := i; {Заполняем массив}
 p := @pp[0];     { Записываем в p указатель на pp }
 for i := 0 to 100 do
   if p[i]<>pp[i] then
 WriteLn ('Ох, проблема!')
end.
```

Free Pascal поддерживает адресную арифметику с указателями, так как это делает C. Это означает что, если P — типизированный указатель, инструкции

```
Inc(P);
Dec(P);
```

Увеличат, и соответственно уменьшат адрес указателя на размер типа, на

который ссылается указатель P. Например

```
Var P : ^Longint;
...
Inc (p);
```

увеличит P на **4**, потому что размер Longint **4** *(байта)*. Если указатель не типизирован, предполагается размер в **1** байт *(то есть как будто бы указатель был указателем на байт: ^Byte)*

Нормальные арифметические операторы также могут использоваться с указателями, то есть, следующее операции допустимы:

```
var  p1,p2 : ^Longint;
L : Longint;
begin
  P1 := @P2;
  P2 := @L;
  L := P1-P2;
  P1 := P1-4;
  P2 := P2+4;
end.
```

Здесь, значение, которое добавляется или вычитается, умножается *на* размер типа на который указывает указатель. В предыдущем примере P1 будет уменьшен на **16**, а P2 будет увеличен на **16** байт.

3.5 Предварительное описания типа

Программы иногда должны поддерживать связанный список записей. Каждая запись содержит указатель на следующую запись *(и возможно на предыдущую запись)*. Для безопасности типов, лучше определять этот указатель как типизированный указатель, таким образом, память под следующую запись может быть выделена в куче, используя вызов New. Для того, чтобы сделать это, запись должна быть определена примерно так:

```
Type
  TListItem = Record
    Data : Integer;
    Next : ^TListItem;
  end;
```

При попытке скомпилировать это, компилятор, при нахождении объявления переменной Next, будет жаловаться, что тип TListItem еще не определен: Это корректно, не смотря на то, что объявление типа все еще анализируется.

Чтобы иметь возможность использовать элемент Next как типизированный указатель, нужно использовать 3.5 Предварительное описания типа [61]:

```
Type
  PListItem = ^TListItem;
```

```
TListItem = Record
  Data : Integer;
  Next : PTListItem;
end;
```

Когда компилятор встречается с объявлением типизированного указателя, где тип, на который ссылаются, еще не известен, он откладывает связывание ссылки на потом. Объявление указателя это 3.5 Предварительное описания типа 61.

Тип, на который ссылаются, должен быть объявлен позже в том же самом блоке `Type`. Блок не может прерываться *(не должно быть других блоков между определением типизированного указателя и типом, на который ссылаются)*. Даже слово `Type` не может использоваться повторно: т.к оно начнет новый блок описания типа (`Type`), заставляя компилятор связать все ожидающие связывания объявления в текущем блоке.

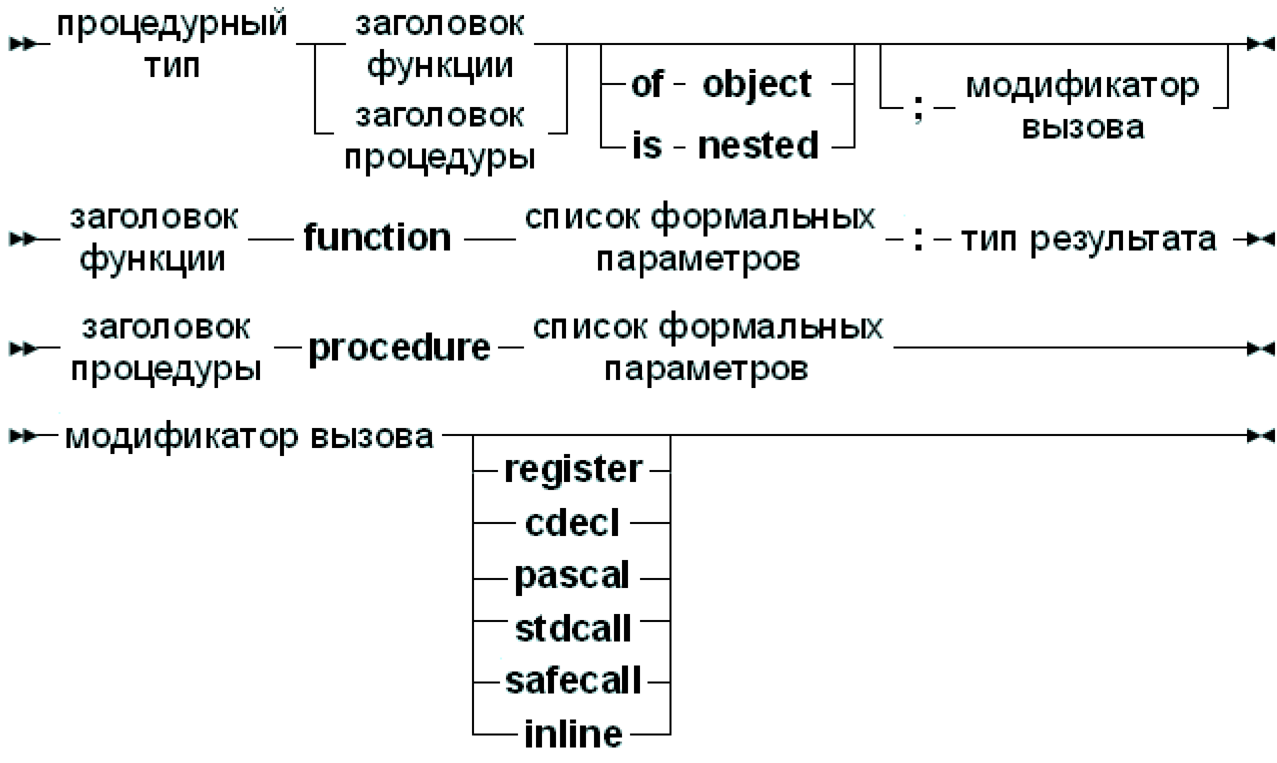
В большинстве случаев, объявление ссылочного типа следует сразу после определения типа указателя, как показано в приведенном выше листинге. Предварительно описанный тип может быть использован в любом объявлении типа после его описания.

Отметим, что предварительное описание типа возможно только с типами указателей и классов, но не с другими типами.

3.6 Процедурный тип

Free Pascal имеет поддержку ***процедурных типов***, она мало чем отличается от ее реализации в Turbo Pascal или Delphi. Объявление типа остается тем же самым, как может быть замечено по следующей синтаксической схеме:

Процедурный тип



Для описания списков формальных параметров см. главу Глава 14 Использование функций и процедур [218]. Два следующих примера - допустимые описания типа:

```
Type
  TOneArg = Procedure (Var X : integer);
  TNoArg = Function : Real;
var
    proc : TOneArg;
    func : TNoArg;
```

Можно присвоить следующие значения переменной процедурного типа:

1. `Nil`, как для указателей обычной процедуры так и указателей метода.
2. Ссылку на переменную процедурного типа, то есть другой переменной того же типа.
3. Глобальный адрес процедуры или функции, с соответствующим заголовком и соглашением о вызове функции или процедуры.
4. Адрес метода.

Учитывая эти объявления, следующие присвоения допустимы:

```
Procedure printit (Var X : Integer);
begin
  WriteLn (x);
end;
...
Proc := @printit;
```

```
Func := @Pi;
```

Из этого примера, ясна разница с Turbo Pascal: в Turbo Pascal нет необходимости использовать оператор адреса (@) при присвоении значения переменной процедурного типа, тогда как в Free Pascal это необходимо. В случае использования директив-переключателей -MDelphi или -MTP, оператор адреса(@) может быть опущен.

Замечание:

модификаторы, касающиеся соглашения о вызове, должны быть такими же как и в объявлении процедурного типа; то есть следующий код вызвал бы ошибку:

```
Type TOneArgCcall = Procedure (Var X : integer);cdecl;
var proc : TOneArgCcall;
Procedure printit (Var X : Integer);
begin
  WriteLn (x);
end;
begin
Proc := @printit;
end.
```

Поскольку тип TOneArgCcall это процедура, которая использует соглашение о вызове Cdecl.

В случае если, добавляется модификатор is nested *(вложен)*, то процедурная переменная может использоваться вложенными процедурами. Это требует, чтобы код компилировался в режиме MACPAS или ISO, или что был активирован переключатель режима {$modeswitch nestedprocvars} :

```
{$modeswitch nestedprocvars}
program tmaclocalprocparam3;

type
  tnestedprocvar = procedure is nested;

var
  tempp: tnestedprocvar;

procedure p1( pp: tnestedprocvar);
begin
  tempp:=pp;
  tempp
end;

procedure p2( pp: tnestedprocvar);
var
```

```
    localpp: tnestedprocvar;
  begin
    localpp:=pp;
    p1( localpp)
  end;

  procedure n;
  begin
    writeln( 'Вызов через n');
  end;

  procedure q;
  var qi: longint;

    procedure r;
    begin
      if qi = 1 then
        writeln( 'Успех для r')
      else
        begin
        writeln( 'сбой');
        halt( 1)
      end
    end;

  begin
    qi:= 1;
    p1( @r);
    p2( @r);
    p1( @n);
    p2( @n);
  end;

  begin
    q;
  end.
```

В случае, если кто-то хочет назначить метод класса переменной процедурного типа, процедурный тип должен быть объявлен с модификатором `of object`.

В следующем примере представлены допустимые объявления двух процедурных переменных для метода *(также известные как обработчиков событий, потому что их использования в дизайне GUI)*:

```
  Type
     TOneArg = Procedure(Var X : integer) of object;
     TNoArg  = Function : Real of object;
```

```
var
  oproc : TOneArg;
  ofunc : TNoArg;
```

Объявление этих функций показывают как это правильно сделать. При их вызове, `Self` будет указывать на экземпляр объекта, который был использован для объявления процедуры метода.

Следующие методы объекта могут быть присвоены переменным `oproc` и `ofunc`:

```
Type
  TMyObject = Class(TObject)
    Procedure DoX (Var X : integer);
    Function  DoY: Real;
  end;

Var
  M : TMyObject;

begin
  oproc:=@M.DoX;
  ofunc:=@M.DOY;
end;
```

При вызове `oproc` и `ofunc`, `Self` будет равна `M`.

Этот механизм иногда называют `Delegation` *(делегация)*.

3.7 Тип данных Variant

3.7.1 Определение

Начиная с версии **1.1**, в FPC введена поддержка типа `variants`. Для получения максимальной поддержки `variant` нужно добавить модуль variants в секцию `uses` каждого модуля который его использует: Модуль variants содержит поддержку просмотра и преобразования типа `variant`, кроме поддержки, предлагаемой по умолчанию модулями System или ObjPas.

Тип значения, хранящегося в переменной `variant` определяется во время выполнения: он зависит от того, что было ей присвоено. Почти любой простой тип может быть присвоен вариантной переменной: ***порядковые типы***, ***строковые типы***, тип `Int64`.

Структурированные типы, такие как ***множества***, ***записи***, ***массивы***, ***файлы***, ***объекты*** и ***классы*** ***не совместимы*** с типом `variant`, в отличии от указателей.

Интерфейсы и COM или CORBA объекты могут быть присвоены вариантной переменной *(в основном потому, что они просто являются указателями)*.

Это означает, что допустимы следующие присвоения:

```
Type
  TMyEnum = (One,Two,Three);

Var
  V : Variant;
  I : Integer;
  B : Byte;
  W : Word;
  Q : Int64;
  E : Extended;
  D : Double;
  En : TMyEnum;
  AStr : AnsiString;
  WS : WideString;
begin
  V:=I;
  V:=B;
  V:=W;
  V:=Q;
  V:=E;
  V:=En;
  V:=D:
  V:=AStr;
  V:=WS;
end;
```

Обратные присвоения, конечно, также верны.

Вариантная переменная может содержать массив значений: Все элементы массива имеют одинаковый тип *(но они могут быть типа* `'variant'`). Для вариантной переменной, которая содержит массив, вариант может быть проиндексирован:

```
Program testv;

uses variants;

Var
  A : Variant;
  I : integer;
begin
  A:=VarArrayCreate([1,10],varInteger);
```

```
    For I:=1 to 10 do A[I]:=I;
  end.
```

Для объяснения VarArrayCreate см. описание модуля variants.

Обратите внимание, что, когда массив содержит строку, она не считается "***массивом символов***" и потому вариантная запись ***не может*** быть проиндексирована для получения символа в определенной позиции в строке.

3.7.2 Вариантные переменные в присвоениях и выражениях

Как видно из определения выше, большинство простых типов могут быть присвоены вариантной переменной. Точно так же вариантная переменная может быть присвоена простому типу: Если возможно, значение вариантной переменной будет преобразовано к типу который присваивается. Это может привести к ошибке: Присвоение вариантной переменной, содержащей строку, целочисленной переменной приведет к ошибке, если строка не содержит допустимое целое число. В следующем примере, первое присвоение будет работать а второе завершится ошибкой:

```
program testv3;

uses Variants;

Var
  V : Variant;
  I : Integer;
begin
  V:='100';
  I:=V;
  Writeln('I : ',I);
  V:='Что-то другое';
  I:=V;
  Writeln('I : ',I);
end.
```

Первое присвоение будет работать, но второе нет, поскольку '***Что-то другое***' не может быть преобразовано в допустимое целочисленное значение. В результате будет сгенерировано исключение EConvertError.

Результатом выражения, включающего тип variant, снова будет тип variant, но он может быть присвоен переменной другого типа - если результат может быть преобразован в переменную этого типа.

Обратите внимание, что выражения, включающие варианты требуют больше времени для вычислений, и поэтому его следует использовать с осторожностью. Если нужно сделать много расчетов, то лучше избежать использования типа variant.

При рассмотрении неявных преобразований типов *(например,* `byte` *к* `integer`*,* `integer` *к* `double`*,* `char` *к* `string`*)* компилятор ***проигнорирует*** варианты пока явно не встретит их в выражении.

3.7.3 Варианты и интерфейсы

Замечание:

> Поддержка интерфейс диспетчиризации *(dispatch)* для типов `variant` в настоящее время нарушена в компиляторе.

Переменные типа `variant` *(variants)* могут содержать ссылку на интерфейс - обычный интерфейс *(наследовано от* `IInterface`*)* или интерфейс диспетчеризации *(наследовано от* `IDispatch`*)*. Переменные типа `variant` содержат ссылку на интерфейс диспетчеризации могут быть использованы для управления объектом: компилятор использует позднее связывание для вызова функции интерфейса: ***не выполняется*** никакой проверки на допустимость имен функций, параметров или аргументов этих функций. Тип результата тоже ***не проверяется***. Компилятор просто вставит код, для чтобы сделать вызова и получения результата.

Это значит, что вы можете сделать в Windows следующее:

```
Var
   W : Variant;
   V : String;

begin
   W:=CreateOleObject('Word.Application');
   V:=W.Application.Version;
   Writeln('Установленная версия MS Word: ',V);
end;
```

Стока

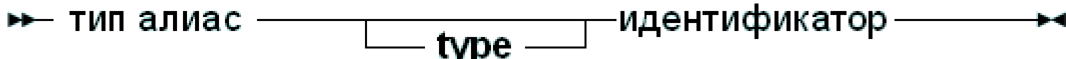
```
V:=W.Application.Version;
```

выполняется путем вставки необходимого кода вызова функции интерфейса диспетчеризации, хранящийся в перемнной `W` типа `variant`, чтобы выполнить вызов, и получить информацию о результате.

3.8 Псевдоним типа

Псевдоним типа это способ дать типу другое имя, но может быть использован и для создания новых типов. Какой из способов использовать, зависит от ваших потребностей

Псевдоним типа



В первом случае мы лишь даём типу другое имя:

```
Type
 MyInteger = Integer;
```

Мы создаём новое имя для типа `Integer`, но не создаём новый тип. То есть, две переменные:

```
Var
 A : MyInteger;
 B : Integer;
```

Будут *(фактически, с точки зрения компилятора)* иметь одинаковый тип *(`Integer`)*.

Описанным выше способом можно создать типы зависящие от платформы, используя псевдонимы типов можно определить типы для каждой платформы отдельно. Программист, который использует эти пользовательские типы не будет думать о размере исходного типа: он вообще может не знать на чём этот тип основан. Он также может использовать имена псевдонимов для полных имен типов. Например можно определить `system.longint` как `Olongint`, а затем переопределить как `longint`.

Псевдоним часто нужен для повторного введения типа:

```
Unit A;

Interface

Uses B;

Type
  MyType = B.MyType;
```

Эта конструкция часто видна при рефакторинге, при переобъявлении в модуле `B` типов из модуля `A`, для сохранения обратной совместимости интерфейса модуля `A`.

Другой случай более хитрый:

```
Type
  MyInteger = Type Integer;
```

Он не только создает новое имя для обозначения типа `Integer`, но фактически создает новый тип. То есть, две переменные:

```
Var
```

```
A : MyInteger;
B :  Integer;
```

Не будут иметь одинаковый тип *(с точки зрения компилятора)*. Тем не менее, эти два типа совместимы по присваиванию. А значит присваивание

```
A:=B;
```

будет действовать.

Разница видна при рассмотрении информации о типе:

```
If TypeInfo(MyInteger)<>TypeInfo(Integer) then
  Writeln('MyInteger и Integer различные типы');
```

Функция компилятора `TypeInfo` возвращает указатель на двоичную информацию о типе. Так как два типа `MyInteger` и `Integer` различны, будут генерироваться различные информационные блоки для этих типов, и указатели будут отличаться.

Разница типов имеет три последствия:

1. Так как они имеют разные `TypeInfo`, значит отличается и RTTI *(Run-Time Type Information)*.
2. Можно их использовать при прегрузке функций, то есть
   ```
   Procedure MyProc(A : MyInteger); overload;
   Procedure MyProc(A : Integer); overload;
   ```
   будет работать. Но это не будет работать если просто определить псевдоним типа *(как в первом случае)*.
3. Их можно использовать и при перегрузке операторов, то есть
   ```
   Operator +(A,B : MyInteger) : MyInteger;
   ```
   тоже будет работать.

Глава 4 Переменные

4.1 Определение

Переменные это области памяти которым явно задано имя и тип. При присвоении значений переменным, компилятор Free Pascal генерирует машинный код для помещения значения в ячейки памяти, отведенные для этой переменной. То, где эта переменная размещена, зависит от места его объявления:

- ***Глобальные переменные*** - переменные, объявленные в модуле или программе, но не в процедуре или функции. Они хранятся в глобальной области памяти и доступны в течение всего времени выполнения программы.
- ***Локальные переменные*** объявлены в процедуре или функции. Их значение храниться в стеке программы, то есть не в глобальной области.

Компилятор Free Pascal обрабатывает выделение памяти для этих ячеек прозрачно, хотя на расположение переменной можно повлиять с помощью ее объявления.

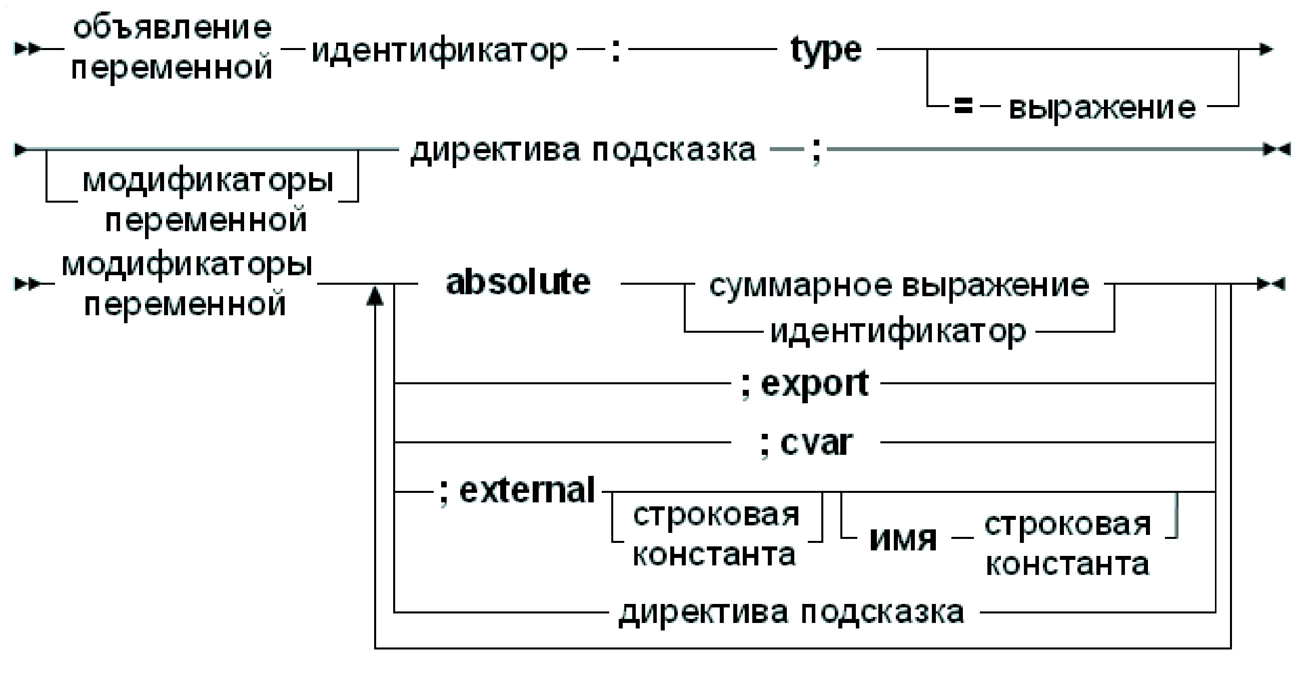
Компилятор Free Pascal также прозрачно обрабатывает чтение значений из и запись значений в переменные. Но даже это может быть явно обработано программистом при использовании свойств *(properties)*.

Переменные должны быть явно объявлены, когда они необходимы. Память ***не будет*** выделена, пока переменная не будет объявлена. Использование переменной-идентификатора (например, переменная цикла), которая ранее не была объявлена, является ошибкой, о которой сообщит компилятор.

4.2 Объявление

Переменные должны быть объявлены в секции объявления переменных модуля или процедуры или функции. *(см. 16.6 Область действия[269])* Выглядит это следующим образом:

Объявление переменной



Это означает, что допустимы следующие объявления переменных:

```
Var
  curterm1  : integer;
  curterm2  : integer; cvar;
  curterm3  : integer; cvar; external;
  curterm4  : integer; external name 'curterm3';
  curterm5  : integer; external 'libc' name 'curterm9';
  curterm6  : integer absolute curterm1;
  curterm7  : integer; cvar;  export;
  curterm8  : integer; cvar;  public;
  curterm9  : integer; export name 'me';
  curterm10 : integer; public name 'ma';
  curterm11 : integer = 1 ;
```

Различие между этими объявлениями следующие:

1. Первая форма (curterm1) определяет регулярную переменной.Компилятор управляет всем сам по себе.
2. Вторая форма (curterm2) объявляет также регулярную переменную, но определяет, что ассемблерное имя для этой переменной равняется имени переменной записаной в источнике.
3. Третья форма (curterm3) объявляет переменную, которая расположена внешне: компилятор предположит, что память расположена в другом месте, и что ассемблерное имя этой памяти совпадает с именем переменной, указаной в исходном коде. Имя можно не указывать (ассемблерное имя).
4. Четвертая форма абсолютно эквивалентна третьей, она объявляет переменную, которая сохранена внешне, и явно задает ее ассемблерное

имя. Если ключевое слово `cvar` не используется, имя должно быть указано явно.

5. Пятая форма - разновидность четвертой формы, только еще указано имя библиотеки, в которой зарезервирована память.
6. Шестая форма объявляет переменную (`curterm6`), и говорит компилятору, что она размещена в той же области памяти что и другая переменная (`curterm1`).
7. Седьмая форма объявляет переменную (`curterm7`), и говорит компилятору, что ассемблерная метка этой переменной должна быть такой же как и имя (чувствительной к регистру) переменной и должна быть экспортирована. то есть на нее можно ссылаться из других объектных файлов.
8. Восьмая форма (`curterm8`) эквивалентна седьмой: `'public'` - псевдоним для `'export'`.
9. Девятая и десятая форма эквивалентны: они определяют ассемблерное имя переменной.
10. Форма одинадцать объявляет переменную (`curterm11`) и инициализирует ее значением (**1** в вышеупомянутом случае).

Отметьте, что ассемблерные имена должны быть уникальными. Не возможно объявить или экспортировать **2** переменные с тем же самым ассемблерным именем.

4.3 Область видимости(контекст)

Переменные, так же, как и любой идентификатор, повинуются общим правилам области видимости. Кроме того, инициализированные переменные инициализируются, когда они вводят в область видимости:

- ***Глобально инициализированные переменные*** инициализируются однажды, при запуске программы.
- ***Локально инициализированные переменные*** инициализируются каждый раз, когда вызывается процедура *(функция)*.

Отметьте, что поведение для локальных инициализированных переменных отличается от того локальной типизированной константы. Локальная типизированная константа ведет себя как инициализированная переменная глобальной переменной.

4.4 Инициализированные переменные

По умолчанию переменные в Pascal не инициализируются после их объявления. Любое предположение, что они содержат **0** или любое другое значение по умолчанию, **ошибочно**: Они могут содержать мусор. Чтобы исправить это, существует понятие инициализированных переменных. Различие с нормальными переменными в том, что их объявление включает начальное значение, как может быть замечено в схеме из предыдущего раздела.

Учитывая объявление:

```
Var
  S : String = 'Это строка инициализации';
```

Значение следующей переменной будет инициализировано указанным значением. Следующее - еще лучший способ сделать это:

```
Const
  SDefault = 'Это строка инициализации';

Var
  S : String = SDefault;
```

Инициализация часто используется, чтобы инициализировать массивы и записи. Для массивов инициализированные элементы должны быть определены, окружены круглыми скобками, и разделены запятыми. Число инициализированных элементов должно быть точно таким же как число элементов в объявлении типа. Как пример:

```
Var
  tt : array [1..3] of string[20] = ('ikke', 'gij', 'hij');
  ti : array [1..4] of Longint = (1,3,5,0);
```

Для константных записей каждый элемент записи должен быть определен в форме `Поле : Значение`, разделенное точками с запятой, и окруженный круглыми скобками. Как пример:

```
Type
  Point = record
    X,Y : Real
    end;
Var
  Origin : Point = (X:0.0; Y:0.0);
```

Порядок полей в константной записи должен быть тем же самым как и в описании типа, иначе будет вызвана ошибка времени компиляции.

Замечание:

Нужно подчеркнуть, что ***инициализированные переменные*** инициализируются, когда они входят в область видимости, в отличии от типизированных констант, которые инициализируются в программе при запуске. Это - истино и для ***локально инициализированных переменных***. Они инициализируются всякий раз, при вызове подпрограмма. Любые изменения, которые произошли в предыдущем вызове подпрограммы, ***будут отменены***, потому что они снова инициализируются.

4.5 Инициализация переменных (по умолчанию)

Некоторые переменные должны быть инициализированы, поскольку они

содержат управляемые типы. Для переменных, объявленных в разделе var функций *(процедур)* или основной программы, это делается автоматически. Для переменных, которые выделяются в куче, это не так.

Для этого компилятор содержит внутреннюю функцию `Default`. Эта функция принимает **идентификатор типа** в качестве аргумента, и возвращает правильно инициализированную переменную **этого типа**. По сути, функция обнулит всю переменную.

Ниже приведен пример её использования:

```
type
  TRecord = record
    i: LongInt;
    s: AnsiString;
  end;

var
  i: LongInt;
  o: TObject;
  r: TRecord;
begin
  i := Default(LongInt); // 0
  o := Default(TObject); // Nil
  r := Default(TRecord); // ( i: 0; s: '')
end.
```

Случай, когда переменная выделяется в куче, более интересен:

```
type
  TRecord = record
    i: LongInt;
    s: AnsiString;
  end;

var
  i: ^LongInt;
  o: ^TObject;
  r: ^TRecord;
begin
  i:=GetMem(SizeOf(Longint));
  i^ := Default(LongInt); // 0
  o:=GetMem(SizeOf(TObject));
  o^ := Default(TObject); // Nil
  r:=GetMem(SizeOf(TRecord));
  r^ := Default(TRecord); // ( i: 0; s: '')
end.
```

Это функция работает для всех типов, **кроме** различных **типов файлов** *(или сложных типов, содержащих тип файла)*.

Примечание:

Это особенно полезно для значений по умолчанию `(Default)` для дженериков, потому, что тип переменных неизвестен во время объявления дженерика. Для получения дополнительной информации см. раздел 8.7 Инициализация по умолчанию 145

4.6 Потоконезависимые переменные

Для программы, которая использует потоки, переменные могут быть действительно глобальными, то есть такими же самыми для всех потоков, или локальными для потока: это означает, что каждый поток получает копию переменной. Локальные переменные *(определенные в процедуре)* всегда локальны для потока. Глобальные переменные, как правило, одинаковы для всех потоков. Глобальная переменная может быть объявлена локальной для потока, заменой ключевого слова `var` в начале блока объявления переменной на `Threadvar`:

```
Threadvar
   IOResult : Integer;
```

Если потоки не используются, переменная ведет себя как обычная переменная. При использовании потоков, делается копия для каждого из них (включая основной поток). Отметьте, что копия сделана с исходным значением переменной, ***не*** со значением переменной в то время, когда поток запущен.

Threadvars должен использоваться экономно: есть издержки для получения или установки значения переменной. Если возможно, рассмотрите использование локальных переменных; они всегда быстрее чем потоковые переменные.

Потоки не включены по умолчанию. Для получения дополнительной информации о программировании потоков, см. главу по потокам в Справочник программиста Free Pascal.

4.7 Свойства (Properties)

В глобальном блоке объявлений можно объявлять свойства, так же, как они могли бы быть определены в классе. Различие в том, что глобальное свойство не нуждается в экземпляре класса: есть только **1** экземпляр этого свойства. Кроме этого, глобальное свойство ведет себя как свойство класса. Спецификаторы чтения-записи для глобального свойства должны также быть регулярными процедурами, но не методами.

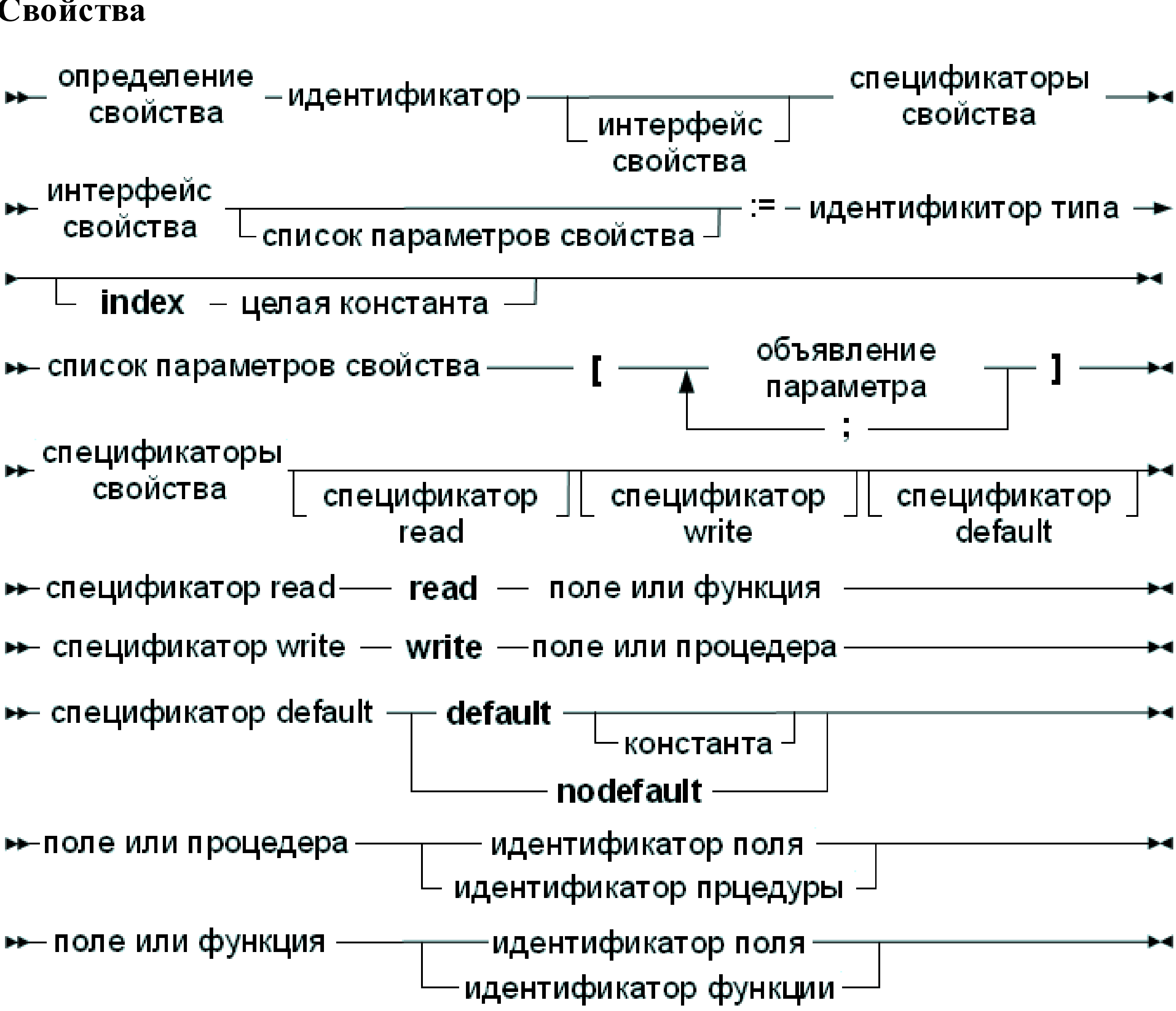
Понятие глобального свойства является специфическим для Free Pascal, и не

существует в режиме Delphi. Режим ObjFPC обязан работать со свойствами.

Понятие глобального свойства может использоваться, чтобы 'скрыть' расположение значения, или вычислить значение на лету, или проверить значения, которые записаны свойству.

Объявление выглядит следующим образом:

Свойства



Ниже приведен пример:

```
{$mode objfpc}
unit testprop;

Interface

Function GetMyInt : Integer;
Procedure SetMyInt(Value : Integer);
Property
  MyProp : Integer Read GetMyInt Write SetMyInt;
```

```
Implementation

Uses sysutils;

Var
  FMyInt : Integer;

Function GetMyInt : Integer;
begin
  Result:=FMyInt;
end;

Procedure SetMyInt(Value : Integer);
begin
  If ((Value mod 2)=1) then
   Raise Exception.Create('MyProp может содержать только чётные
значения');
  FMyInt:=Value;
end;

end.
```

Спецификаторы ***чтения-записи*** могут быть скрыты, объявляя их в другом модуле, который должен быть в списке `uses` используемых модулей. Это может использоваться, чтобы скрыть спецификаторы доступа для ***чтения-записи*** для программистов, так же, как если бы они были в разделе `private` класса *(обсуждены ниже)*. Для предыдущего примера это могло выглядеть следующим образом:

```
{$mode objfpc}
unit testrw;

Interface

Function GetMyInt : Integer;
Procedure SetMyInt(Value : Integer);

Implementation

Uses sysutils;

Var
  FMyInt : Integer;

Function GetMyInt : Integer;
begin
```

```
    Result:=FMyInt;
  end;

  Procedure SetMyInt(Value : Integer);
  begin
    If ((Value mod 2)=1) then
     Raise Exception.Create('Разрешены только чётные значения');
    FMyInt:=Value;
  end;

  end.
```

Модуль testprop был бы тогда похож на:

```
  {$mode objfpc}
  unit testprop;

  Interface

  uses testrw;

  Property
    MyProp : Integer Read GetMyInt Write SetMyInt;

  Implementation

  end.
```

Больше информации о свойствах может быть найдено в Глава 6 Классы 95.

Глава 5 Объекты

5.1 Объявление

Free Pascal поддерживает объектно-ориентированное программирование. Фактически, большая часть компилятора написана, с использованием объектов. Здесь мы представляем некоторые технические вопросы относительно объектно-ориентированного программирования во Free Pascal.

Объекты должны рассматриваться как особый вид записей. Запись содержит все поля, которые объявлены в определении объектов и указатели на методы, которые связаны с типом объекта.

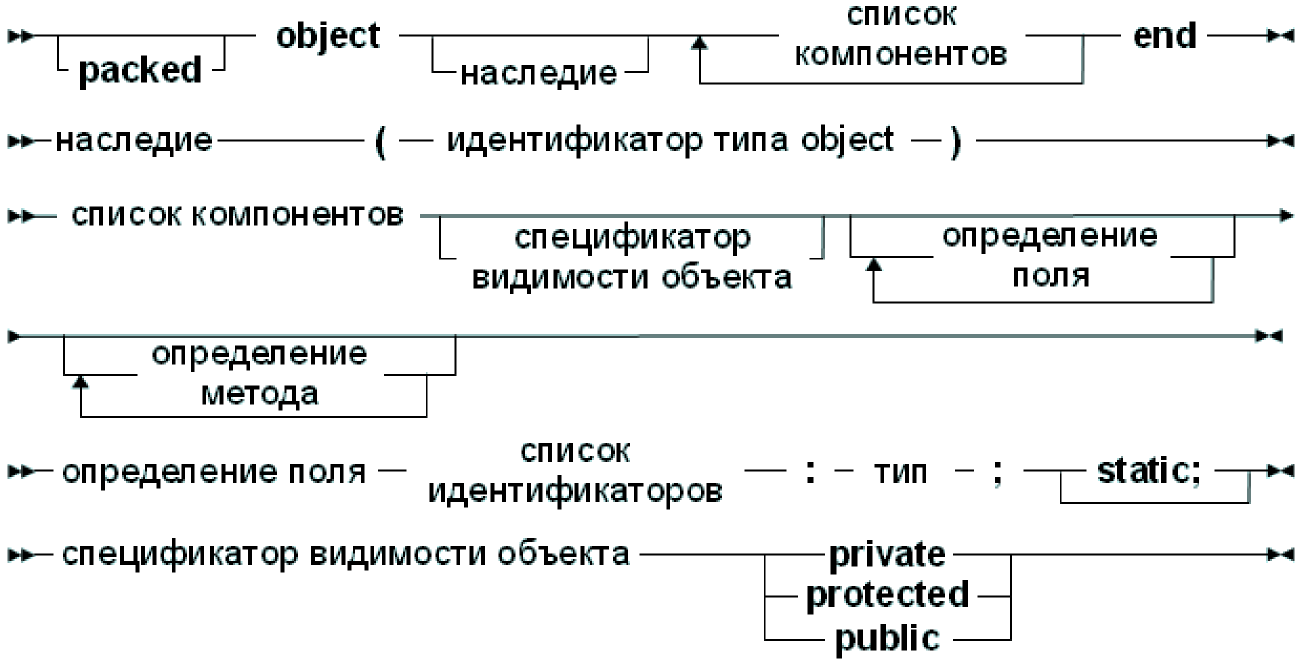
Объект объявлен, как была бы объявлена запись; за исключением того, что теперь, процедуры и функции могут быть объявлены как часть записи. Объекты могут "***наследовать***" поля и методы от "***родительских***" объектов. Это означает, что эти поля и методы могут использоваться , как будто они были включены в объекты, объявленные как "дочерний" объект.

Кроме того, вводится концепция видимости: поля, процедуры и функции могут быть объявлены как `public` *(публичные)*, `protected` *(защищенные)* и `private` *(частные)*. По умолчанию, поля и методы являются `public`, и доступны за пределами текущего блока.

Поля или методы, объявленные как `private`, доступны только в текущем модуле: их область видимости ограничена реализацией текущего модуля.

Объявление прототипа объекта выглядит следующим образом:

Типы объектов



Как видно далее, можно объявлять столько `private` и `public` блоков сколько нужно.

Ниже дано допустимое определение объекта:

```
Type
  TObj = object
  Private
    Caption : ShortString;
  Public
    Constructor init;
    Destructor done;
    Procedure SetCaption (AValue : String);
    Function GetCaption : String;
  end;
```

Объект содержит пару ***конструктор/деструктор***, и метод, для получения и установки заголовка. Поле `Caption` является частным для объекта: к нему нельзя получить доступ вне модуля, в котором объявлен `TObj`.

Замечание:

В режиме MacPas, ключевое слово `Object` заменяется ключевым словом `Class` для совместимости с другими компиляторами Pascal доступными для Mac. Это означает, что объекты не могут использоваться в режиме `MacPas`.

Замечание:

Free Pascal поддерживает также упакованные объекты. Они такие же, как и обычные объекты, только элементы *(поля)* объекта выровнены на границах байта, как и в упакованных записях. Объявление упакованного объекта похоже на объявление упакованной записи:

```
Type
  TObj = packed object
   Constructor init;
   ...
   end;
  Pobj = ^TObj;
 Var PP : Pobj;
```

Директива `{$PackRecords}` аналогично действует и на объекты.

5.2 Поля

Поля объекта походят на поля записей. Способ доступа к ним аналогичен способу обращения к полям записей: через идентификатор поля. Учитывая следующее объявление:

```
Type TAnObject = Object
  AField : Longint;
  Procedure AMethod;
  end;
Var AnObject : TAnObject;
```

следующее присвоение допустимо:

```
AnObject.AField := 0;
```

В методах к полям можно получить доступ, используя короткий идентификатор:

```
Procedure TAnObject.AMethod;
begin
  ...
  AField := 0;
  ...
end;
```

Или, можно использовать идентификатор `self`. Идентификатор `self` обращается к текущему экземпляру объекта:

```
Procedure TAnObject.AMethod;
begin
  ...
  Self.AField := 0;
  ...
```

```
end;
```

Нельзя получить доступ к полям, которые находятся в частных или защищенных разделах объекта, снаружи методов объектов. Если будет предпринята попытка получить к ним доступ, то компилятор будет жаловаться на неизвестный идентификатор.

Кроме того, можно использовать оператор `with` с экземпляром объекта, как и с записями:

```
With AnObject do
  begin
    Afield := 12;
    AMethod;
  end;
```

Это как если-бы к идентификаторам `Afield` и `Amethod` *(которые находятся между `begin` и `end`)* добавилось уточнение `AnObject`. Подробнее об этом в разделе 13.2.8 Оператор With[215].

5.3 Статические поля илм поля классов

Объект может содержать **классы** или **статические поля**: эти поля являются глобальными по отношению объекту, и действуют как глобальные переменные, но известны только в области видимости объекта. Различие между статическими переменными и переменными класса это режим, в котором они работают: Статическая переменная *(ключевое слово `static`)* будет работать всегда, ключевое слово `class` будет нужно в режиме `{$MODE DELPHI}`.

На них можно ссылаться из методов объекта, также можно ссылаться извне объекта, используя полное имя.

Для примера, выход следующей программы

```
{$mode delphi}
{$static on}
type
  cl=object
    l : longint; static;
    class var v : integer;
  end;

var
  cl1,cl2 : cl;
begin
  Writeln('Статический');
  cl1.l:=2;
  writeln(cl2.l);
  cl2.l:=3;
```

```
    writeln(cl1.l);
    Writeln(cl.l);
    Writeln('Класс');
    cl1.v:=4;
    writeln(cl2.v);
    cl2.v:=5;
    writeln(cl1.v);
    Writeln(cl.v);
  end.
```

будет следующим:

```
Статический
2
3
3
Класс
4
5
5
```

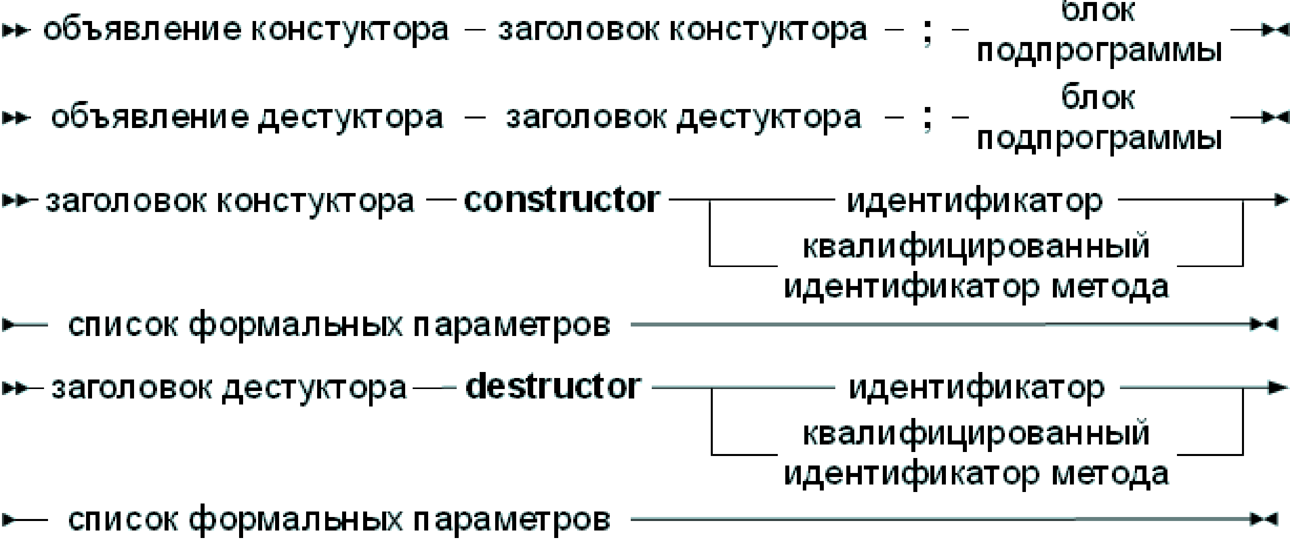
Обратите внимание, что последняя строка кода ссылается на сам тип объекта (`cl`), а не на экземпляр объекта (`cl1` или `cl2`).

5.4 Конструкторы и деструкторы

Как можно увидеть на синтаксической диаграмме объявления объекта, Free Pascal поддерживает ***конструкты*** и ***деструкторы***. При использовании объектов программист отвечает за вызов конструктора и деструктора явно.

Описание конструктора или деструктора выглядит следующим образом:

Конструкторы и дестукторы



Если объект использует виртуальные методы, обязательно ***требуется*** пара ***конструктор/деструктор***. Причина состоит в том, что для объекта с виртуальными методами, требуется некоторое внутреннее обслуживание: это обслуживание делается конструктором. *Указатель на VMT должен быть установлен*

При объявлении объектного типа должен использоваться простой идентификатор для имени конструктора или деструктора. При реализации конструктора или деструктора, должен использоваться полный идентификатор метода, то есть идентификатор вида `objectidentifier.methodidentifier`.

Free Pascal поддерживает также расширенный синтаксис процедур `New` и `Dispose`. В случае, если нужно выделить память для динамической переменной объектного типа, имя конструктора может быть передано при вызове `New`. `New` реализована как функция, которая возвращает указатель на экземпляр объекта. Рассмотрим следующее объявление:

```
Type
  TObj = object;
  Constructor init;
  ...
  end;
  Pobj = ^TObj;
Var PP : Pobj;
```

Тогда следующие три вызова эквивалентны:

```
pp := new (Pobj,Init);
```

и

```
new(pp,init);
```

и также

```
new (pp);
pp^.init;
```

В последнем случае компилятор выдаст предупреждение, что должен использоваться расширенный синтаксис New и Dispose, чтобы генерировать экземпляры объекта. Возможно проигнорировать это предупреждение, но использование расширенного синтаксиса лучшая практика программирования, для создания экземпляров объектов. Точно так же процедура Dispose принимает имя деструктора. Прежде, чем удалить объект из "кучи", будет вызван деструктор.

В связи с предупреждением компилятора, в следующей главе представлен подход Delphi для объектно-ориентированного программирования, и может считаться более естественным способом объектно-ориентированного программирования.

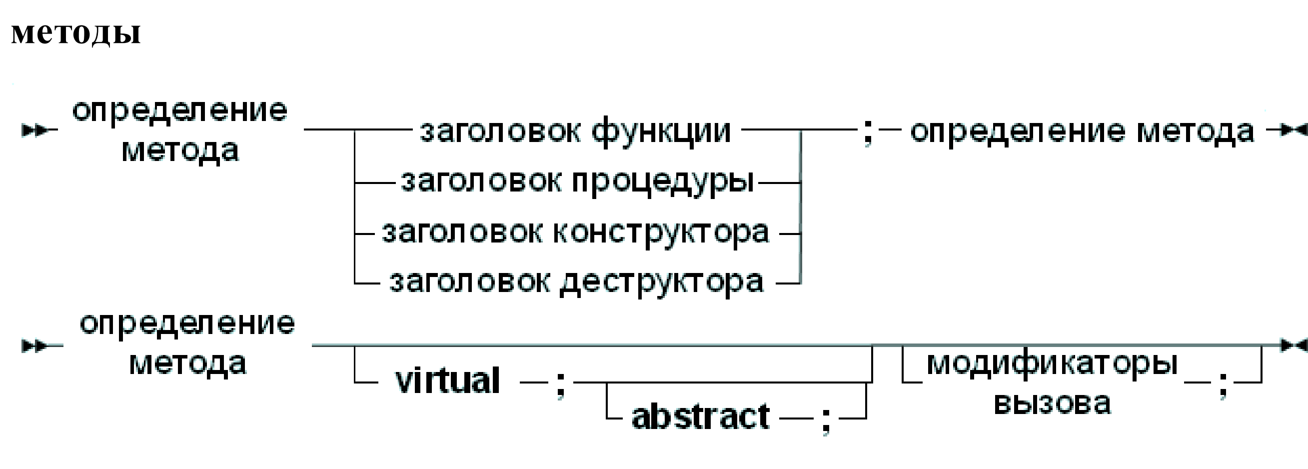
5.5 Методы

Методы объектов точно такие же как обычные процедуры или функции, только у них есть неявный дополнительный параметр: self. **Self** указывает на объект, с которым был вызван метод. При реализации методов, в заголовке функции должен быть указан полный идентификатор. При объявлении методов должен быть указан нормальный *(допустимый)* идентификатор.

5.5.1 Объявление

Объявление метода очень походит на нормальное объявление функции или процедуры с некоторыми дополнительными спецификаторами, как может быть замечено по следующей схеме, которая является частью объявления объекта:

методы



С точки зрения объявлений, объявление Метода *(Method definitions)* ето нормальное объявление функции или процедуры. В отличие от TP и Delphi,

поля могут быть объявлены после методов в том же блоке, то есть следующий код будет генерировать ошибку при компиляции в режиме Delphi или Turbo Pascal, но не в FPC:

```
Type
  MyObj = Object
   Procedure Doit;
   Field : Longint;
  end;
```

5.5.2 Вызов метода

Методы вызываются так же, как обычные процедуры, только они имеют идентификатор экземпляра объекта предшествующий им *(см. также Глава 13 Операторы[196])*. Чтобы определить, какой метод вызывается, необходимо знать тип метода. Мы рассмотрим различные типы в дальнейшем.

Статические методы

Статические методы это методы, которые были объявлены без ключевых слов abstract *(абстрактные)* или virtual *(виртуальные)*. При вызове статического метода, используется объявленный *(т.е. во время компиляции)* метод объекта. Для примера, рассмотрим следующие объявления:

```
Type
  TParent = Object
    ...
    procedure Doit;
    ...
    end;
  PParent = ^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;
    ...
    end;
  PChild = ^TChild;
```

Как видно, оба объекта родитель и ребенок имеют метод, называемый DoIt. Рассмотрим следующие объявления и вызовы:

```
Var
  ParentA,ParentB : PParent;
  Child  : PChild;
begin
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
```

```
      Child := New(PChild,Init);
      ParentA^.Doit;
      ParentB^.Doit;
      Child^.Doit;
```

Из трех вызовов DoIt, только последний вызовет TChild.Doit, а два других вызовут TParent.Doit. Это происходит потому, что для статических методов, компилятор во время компиляции, определяет, какие методы должны быть вызваны. Переменная ParentB имеет тип TParent, компилятор решает, что должен быть вызван метод TParent.Doit, даже если она будет создана как TChild. Могут быть случаи, когда вызываемый метод, должен зависеть от фактического типа объекта во время выполнения. Если да, метод не может быть статическим, он должен быть виртуальным методом.

Виртуальные методы

Чтобы исправить ситуацию в предыдущем разделе, создаются virtual *(виртуальные)* методы. Это делается достаточно просто, путем добавления модификатора virtual к объявлению метода. Потомки объекта могут переопределить метод с новой реализацией путем повторного объявления метода *(с тем же списком параметров)* с помощью ключевого слова virtual.

Возвращаясь к предыдущему примеру, рассмотрим следующую альтернативу объявлению:

```
Type
  TParent = Object
    ...
    procedure Doit;virtual;
    ...
    end;
  PParent = ^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;virtual;
    ...
    end;
  PChild = ^TChild;
```

Как видно, оба объекта родитель и ребенок имеют метод, называемый DoIt. Рассмотрим следующие объявления и вызовы:

```
Var
  ParentA,ParentB : PParent;
  Child   : PChild;

begin
   ParentA := New(PParent,Init);
```

```
ParentB := New(PChild,Init);
Child := New(PChild,Init);
ParentA^.Doit;
ParentB^.Doit;
Child^.Doit;
```

Теперь, будут вызваны различные методы, в зависимости от фактического типа объекта во время выполнения программы. Для ParentA, ничего не изменится, так как она создается как экземпляр класса TParent. Для Child, ситуация также не меняется: она вновь создается как экземпляр TChild.

Однако для ParentB, ситуация меняется: Даже если она была объявлена как TParent, она будет создана как экземпляр TChild. Теперь, когда программа работает, прежде чем вызывать DoIt, программа проверяет, какой фактический тип имеет ParentB, и только потом решает, какой метод должен быть вызван. Видя, что ParentB имеет тип TChild, будет вызван TChild.Doit. Код времени выполнения для проверки фактического типа экземпляра объекта вставляется компилятором во время компиляции.

Говорят что TChild.Doit override *(переопределяет)* TParent.Doit. Можно получить доступ к TParent.Doit изнутри TChild.Doit, с помощью ключевого слова inherited:

```
Procedure TChild.Doit;
begin
  inherited Doit;
  ...
end;
```

В приведенном выше примере, когда вызывается метод TChild.Doit, первое, что он делает, это вызов TParent.Doit. Ключевое слово inherited не может быть использовано в статических методах, только в виртуальных.

Для того, чтобы сделать это, компилятор хранит - для типа объекта - таблицы с виртуальными методами: VMT *(Virtual Method Table)*. Это просто таблица с указателями на каждый из виртуальных методов: каждый виртуальный метод имеет свои фиксированные места в этой таблице (индекс). Компилятор использует эту таблицу, чтобы найти фактический метод, который должен быть использован. Когда потомок объекта переопределяет метод, родительский метод будет переписан в VMT. Более подробную информацию о VMT можно найти в Справочник программиста Free Pascal.

Как было отмечено ранее, объекты, которые имеют VMT должны быть инициализирована с помощью конструктора: объектная переменная должна быть инициализирована с указателем на VMT фактического типа, с которым она была создана.

Абстрактные методы

Абстрактный метод представляет собой особый вид виртуального метода. Метод, который объявлен как `abstract` *(абстрактный)* не нуждается в реализации. Это задача для объектов-потомков, они должны переопределить и реализовать этот метод.

Из этого следует, что метод не может быть абстрактным, если он не виртуальный *(это видно из синтаксической схемы)*. Второе следствие состоит в том, что экземпляр объекта, который имеет абстрактный метод не может быть создан непосредственно.

Причина очевидна: **не существует метода, который компилятор может выполнить**. Метод, который объявлен `abstract` не имеет реализации. Она зависит только от потомков объектного типа, которые должны переопределить и реализовать этот метод. Продолжая наш пример, посмотрите на это:

```
Type
  TParent = Object
    ...
    procedure Doit;virtual;abstract;
    ...
    end;
  PParent=^TParent;
  TChild = Object(TParent)
    ...
    procedure Doit;virtual;
    ...
    end;

  PChild = ^TChild;
```

Как видно, оба объекта родитель и ребенок имеют метод, называемый `DoIt`. Рассмотрим следующие объявления и вызовы:

```
Var
  ParentA,ParentB : PParent;
  Child   : PChild;
 begin
   ParentA := New(PParent,Init);
   ParentB := New(PChild,Init);
   Child := New(PChild,Init);
   ParentA^.Doit;
   ParentB^.Doit;
   Child^.Doit;
```

Во-первых, на строке **3** будет сгенерирована ошибка компиляции, сообщая, что нельзя создавать экземпляры объектов с абстрактными методами: компилятор обнаружил, что `PParent` указывает на объект, который имеет абстрактный

метод. Закомментировав третью строчку, удастся скомпилировать программу.

Замечание:

Если абстрактный метод переопределен, родительский метод не может быть вызван с помощью `inherited`, так как не существует реализации родительского метода, компилятор обнаружит это, и будет генерировать ошибку на это, вот так:

```
testo.pp(32,3) Error: Abstract methods can't be called directly
testo.pp(32,3) Ошибка: Абстрактные методы не могут быть вызваны напрямую
```

Если абстрактный метод вызовут через некоторый механизм во время выполнения программы, то будет сгенерирована ошибка времени выполнения. *(ошибка* ***211****, если быть точным)*

Методы класса и статические методы

Методы класса или методы, объявленные директивой `static` *(статические)*, являются глобальными по отношению к объекту. В них недоступен неявный указатель `'self'`. Недоступны, так-же обычные методы *(их нельзя вызывать)* и поля объекта. Но могут быть использованы переменные класса.

Методы класса или статические методы являются регулярными, они могут быть назначены процедурным переменным.

Следующая программа демонстрирует это. Закомментированные операторы компилироваться не будет.

```
{$APPTYPE CONSOLE}
{$IFDEF FPC}{$MODE DELPHI}{$H+}{$ENDIF}
type
  TTest = object
    const Epsylon = 100;
    var f : integer;
    class var cv1,cv2:integer;
    procedure myproc;
    class procedure testproc;
    class procedure testproc2;static;
    procedure testproc3; static;
  end;

  procedure TTest.myproc;
  begin
    cv1:=0;
    f:=1;
  end;
```

```
  class procedure TTest.Testproc;
  begin
    cv1:=1;
    // f:=1;
  end;

  class procedure TTest.Testproc2;
  begin
    cv1:=2;
    // f:=1;
  end;

  procedure TTest.Testproc3;
  begin
    cv1:=3;
    // f:=1;
  end;

Var
  P : Procedure;
begin
  P:=@TTest.Myproc;
  P:=@TTest.Testproc;
  P:=@TTest.Testproc2;
  P:=@TTest.Testproc3;
end.
```

Раскоментируюте любой оператор и попытка компиляции приведёт к ошибке компиляции.

```
ocv.pp(32,6) Error: Only class methods, class properties and
class variables can be accessed in class methods
ocv.pp(32,6) Ошибка: В методах класса  доступны только методы
класса, свойства класса и переменные класса
```

5.6 Видимость

Для объектов существуют три спецификатора видимости: `private` *(частный)*, `protected` *(защищенный)* и `public` *(общественный)*. Если спецификатор видимости не определен, используется `public`. И методы и поля могут быть скрыты от программиста, для этого нужно поместить их в раздел `private`. Точное правило видимости следующее:

Private

Ко всем полям и методам, которые находятся в блоке `Private`, можно получить доступ только в модуле (то есть модуле или программе), который содержит определение объекта. К ним можно получить

доступ изнутри методов объекта или снаружи, например, с методов других объектов, или глобальных функций.

Protected

То же самое что и `Private`, кроме того, что члены `protected` раздела, также доступны для потомков типа, даже если они реализуются в других модулях.

Public

Общедоступные поля и методы всегда доступны, отовсюду. Поля и методы в `public` разделе ведут себя, как если бы они были частью обычного типа `record` *(записи)*.

Глава 6 Классы

В подходе Delphi к объектно-ориентированному программированию, все вращается вокруг концепции ***«классов»***. Класс может рассматриваться как указатель на объект, или как указатель на запись со связанными с ним методами.

Главная разница между объектами и классами в том, что объект будет помещен в стек, как и обычная запись, а класс всегда размещается в куче. В следующем примере:

```
Var
  A : TSomeObject; // Объект
  B : TSomeClass;  // Класс
```

Главное отличие в том, что переменная `A` будет занимать столько же места в стеке, сколько занимает объект (`TSomeObject`). С другой стороны, переменная `B` всегда будет иметь размер всего лишь указателя на стек. Фактические данные класса располагаются в куче.

Из этого следует еще одно отличие: класс нужно ***всегда*** инициализировать с помощью своего конструктора, тогда как для объекта такой необходимости нет. Вызов конструктора выделяет в куче необходимое количество памяти для данных экземпляра класса.

Примечание:

В предыдущих версиях Free Pascal, при использовании классов было необходимо выставлять в объявлениях модулей `uses objpas;` Начиная с версии **0.99.12** этого делать ***уже не нужно***. С этой версии модуль objpas будет загружаться автоматически, если используются опции `-MObjfpc` или `-MDelphi`, или в коде явно указаны соответствующие им директивы:

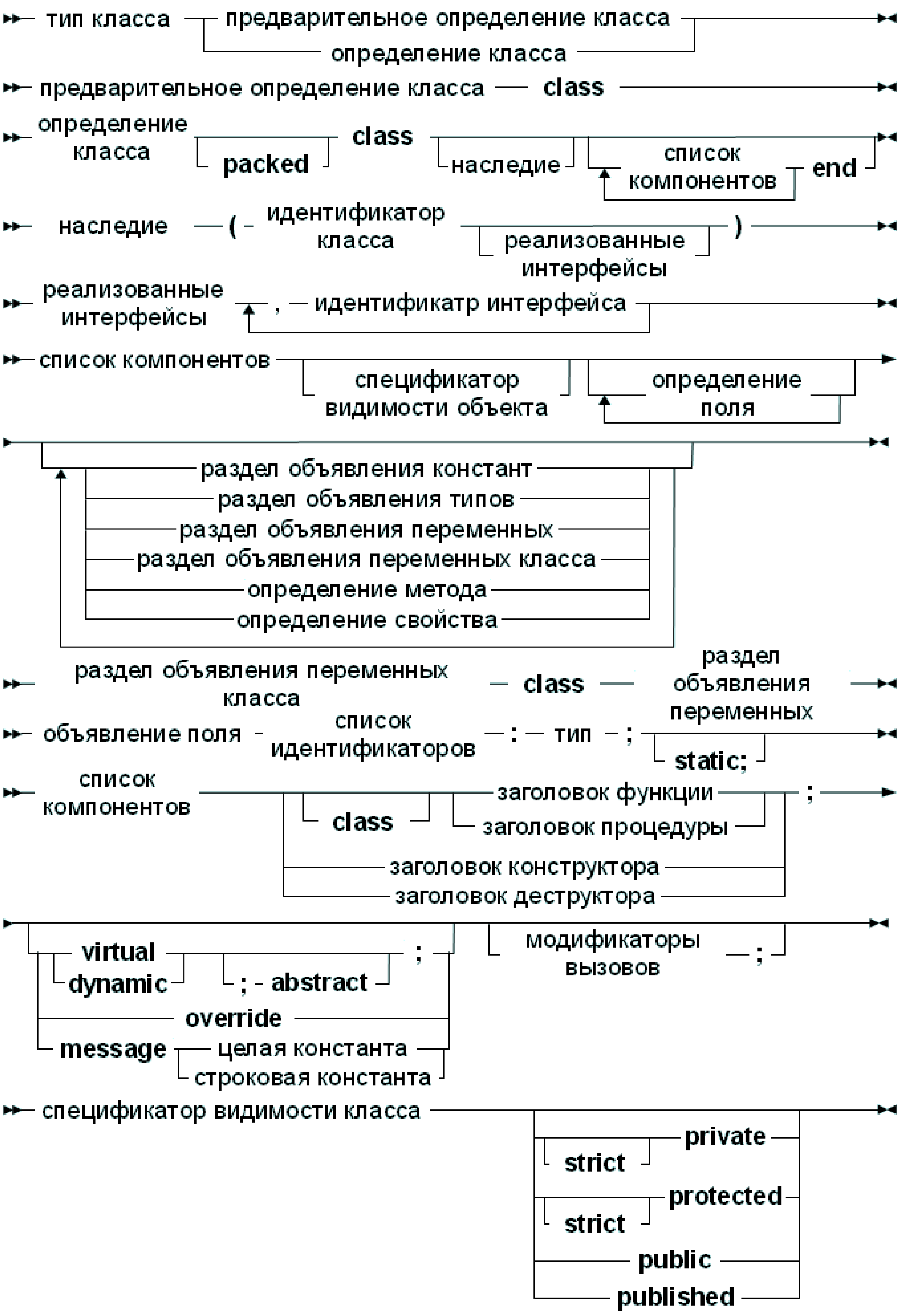
```
{$mode objfpc}
{$mode delphi}
```

Фактически, если компилятор встречает в объявлении модулей objpas, он выдает предупреждение.

6.1 Определения классов

Класс определяется следующим образом:

Тип класса



Примечание:

> В режиме MacPas, ключевое слово `Object` заменяется ключевым словом `class` для совместимости с другими Pascal компиляторами, имеющихся на Mac. Это означает, что в режиме MacPas, зарезервированное слово `class` *(класс)* на рисунке выше, может быть заменено зарезервированным словом `object` *(объект)*.

В объявлении класса, так как многие `private` *(частные)*, `protected` *(защищенные)*, `published` *(опубликованые)* и `public` *(публичные)* блоки могут быть использованы по мере необходимости: различные блоки можно повторять, и нет специального порядка, в котором они должны появляться.

Методы являются обычными декларациями функций или процедур. Как видно, объявление класса практически идентично декларации объекта. Реальная разница между объектами и классами заключается в способе их создания *(см далее в этой главе)*.

Видимость различных разделов выглядит следующим образом:

Private

Все поля и методы, из раздела `private`, могут быть доступны только в модуле *(т.е. unit-е)*, который содержит определение класса. Они могут быть доступны внутри методов класса или из за их пределами *(например, в других методах класса)*.

Strict Private

Все поля и методы, которые из раздела `strict private`, могут быть доступны только из методов самого класса. Другие классы или классы-потомки *(даже в том же модуле)* не могут получить доступ к `strict private` членам.

Protected

То же самое, что и `private`, кроме того, члены из раздела `protected` также доступны для потомком типа, даже если они используются в других модулях.

Public

раздел всегда доступен.

Published

То же самое, как в разделе `public`, но компилятор генерирует также информацию, необходимую для автоматического сохранения в потоке этих полей для этого класса, если компилятор в `{$M+}` состояния. Поля, определенные в разделе `published` должен быть типа класса. Свойства массива не могут быть в разделе `published`.

В синтаксической диаграмме, можно видеть, что класс может быть списоком реализуемых интерфейсов. Эта функция будет обсуждаться в следующей главе.

Классы могут содержать `Class methods` *(методы класса)*: это функции, которые не требуют экземпляра. Идентификатор `Self` действует в таких методах, но относится к самому классу указателя *(на VMT класса)*.

Примечание:

Как в функциях и указателях на типы, иногда необходимы `forward` *(предварительные)* определение класса. Предварительное определение класса - просто название класса, с ключевым словом `Class`, как показано в следующем примере:

```
Type
  TClassB = Class;
  TClassA = Class
    B : TClassB;
end;
  TClassB = Class
    A : TClassA;
end;
```

При использовании предварительного определения класса, класс должен быть определен в том же блоке, в том же разделе ***(interface/implementation)*** *(интерфейс/реализация)*. Но не обязательно должен быть определен в том же разделе `type`.

Кроме того, можно определить опорные классы:

Тип Опорный класс

►► class of — тип класса ——————————————————►◄

Тип опорного класса используются для создания экземпляров определенного класса, который пока не известен во время компиляции, но который указан во время выполнения. По сути, переменная ссылочного класса типа содержит указатель на определению указанного класса. Это может быть использовано для создания экземпляра класса, соответствующего определению, или проверить наследство. Следующий пример показывает, как это работает:

```
Type
  TComponentClass = Class of TComponent;
  Function CreateComponent(AClass: TComponentClass; AOwner:
TComponent): TComponent;
begin
  // ...
  Result:=AClass.Create(AOwner);
  // ...
end;
```

Этой функции может быть передана ссылка на класс любого класса, который наследуется от TComponent. Вызов ниже является допустимым:

```
Var
  C : TComponent;
begin
  C:=CreateComponent(TEdit,Form1);
end;
```

При вызове функции CreateComponent, C будет содержать экземпляр класса TEdit. Обратите внимание, что следующий вызов не будет компилироваться:

```
Var
  C : TComponent;
 begin
  C:=CreateComponent(TStream,Form1);
end;
```

потому что TStream не наследуется от TComponent, и AClass относится к классу TComponent. Компилятор проверит это во время компиляции, и выдаст ошибку.

Ссылки на классы также могут быть использованы для проверки наследования:

```
  TMinClass = Class of TMyClass;
  TMaxClass = Class of TMyClassChild;

Function CheckObjectBetween(Instance : TObject) : boolean;
begin
  If not (Instance is TMinClass) or ((Instance is TMaxClass)
          and (Instance.ClassType<>TMaxClass)) then
                  Raise Exception.Create(SomeError)
end;
```

Приведенный выше пример вызовет исключение, если экземпляр не является потомком TMinClass или является потомком TMaxClass.

Подробнее о экземпляре класса можно найти в следующем разделе 6.3 Экземпляр класса [102]

6.2 Обычные и статические поля

Классы могут иметь поля. В зависимости от того, как они определены, поля содержат данные, относящиеся к экземпляру класса или к классу в целом. Каким бы ни был способ, которым они были определены, поля должны соблюдать правила по ограничению видимости, как и любой другой член класса.

6.2.1 Обычные поля/переменные

Есть два способа объявить нормальное поле. Первый из них является классическим способом, аналогично определению объекта:

```
{$mode objfpc}
type
  cl=class
    l : longint;
  end;
var
  cl1,cl2 : cl;
begin
  cl1:=cl.create;
  cl2:=cl.create;
  cl1.l:=2;
  writeln(cl1.l);
  writeln(cl2.l);
end.
```

вывод будет иметь следующий вид

```
2
0
```

Пример показывает, что значения полей инициализируются нулём (или эквивалентом нуля для ***не порядковых*** *(ordinal)* типов: *пустая строка, пустой массив и так далее*).

Второй способ объявить поле *(доступно только в более поздних версиях Free Pascal)* использует блок `var`:

```
{$mode objfpc}
type
  cl=class
  var
    l : longint;
  end;
```

Он полностью эквивалентен предыдущему определение.

Примечание:

В версии компилятора **3.0**, компилятор может изменить порядок полей в памяти, если это приводит к большей согласованности мелких объектов. Это означает, что в некоторые, поля не обязательно появляться в том же порядке, как и в объявлении. RTTI сгенерированный для класса будет отражать это изменение.

6.2.2 Переменняе/поля класса

Подобно объектам, класс может содержать статические поля или переменные класса: эти поля или переменные являются глобальными по отношению к классу, и действуют как глобальные переменные, но видны только как часть класса. На них можно ссылаться из методов класса, но можно также ссылаться из вне класса, используя полное имя *(имя класса)*.

Опять же, есть два способа определения переменных класса. Первый из них равносилен тому, как это делается в объектах, используя статический модификатор:

Например, вывод следующей программы будет такой же, как вывод версии с использованием объекта:

```
{$mode objfpc}
type
  cl=class
    l : longint;static;
  end;
var
  cl1,cl2 : cl;
begin
  cl1:=cl.create;
  cl2:=cl.create;
  cl1.l:=2;
  writeln(cl2.l);
  cl2.l:=3;
  writeln(cl1.l);
  Writeln(cl.l);
end.
```

Вывод будет следующий:

```
2
3
3
```

Обратите внимание, что последняя строка кода ссылается на сам класс (`cl`), и не экземпляр класса (`cl1` или `cl2`).

Кроме **статического поля**, в классах может быть применён `class var`. Подобно тому, как поля могут быть определены в блоке переменных, переменные могут быть объявлены в блоке `var` класса:

```
{$mode objfpc}
type
  cl=class
  class var
    l : longint;
```

```
end;
```

Это определение эквивалентно предыдущему.

6.3 Экземпляр класса

Классы должны быть созданы с помощью одного из своих конструкторов *(конструкторов может быть несколько)*. Помните, что класс является указателем на объект в куче. Когда переменная некоторого класса объявлена, компилятор просто выделяет место для этого указателя, а не весь объект. Конструктор класса возвращает указатель на инициализированный экземпляр объекта в куче. Так, для инициализации экземпляра некоторого класса, можно было бы сделать следующее:

```
ClassVar := ClassType.ConstructorName;
```

Расширенный синтаксис `new` и `dispose` ***не может*** быть использован для создания и уничтожения экземпляра класса. Это конструкции зарезервированы для использования только с объектами. Вызов конструктора спровоцирует вызов метода виртуального класса `NewInstance`, который, *если не изменена реализация по умолчанию,* вызывает `GetMem`, чтобы выделить место, достаточное для хранения данных экземпляра класса, а затем очистить память.

После этого выполняется код конструктора. Конструктор имеет указатель на свои данные, в `Self`.

Замечание:

- Директива `{$PackRecords}` также влияет на классы, то есть выравнивание в памяти различных полей зависит от значения директивы `{$PackRecords}`.
- Так же, как для объектов и записей, может быть объявлен `packed class` *(упакованный класс)*. Это имеет тот же эффект, что и на объект, или запись, а именно, что элементы выровнены по границам **1** байт, ***то есть как можно ближе.***
- Функция `SizeOf(класс)` возвращает то же самое, что и `SIZEOF (Указатель)`, так как класс является указателем на объект. Чтобы получить размер данных экземпляра класса, используйте метод `TObject.InstanceSize`.
- Если во время выполнения конструктора происходит исключение, автоматически вызывается деструктор.

6.4 Уничтожение класса

Экземпляр класса всегда должен быть уничтожен с помощью деструктора. В отличии от конструктора, дестуктор определён однозначно: деструктор ***должен*** иметь имя `Destroy`, он должен переопределять деструктор `Destroy` объявленный в `TObject`, и не может иметь аргументы, и унаследованный деструктор всегда должен быть вызван.

Чтобы избежать вызова деструктора для экземпляра Nil, лучше вызвать метод Free класса TObject. Этот метод проверит, что Self не Nil, то вызывает Destroy. Если Self равно Nil, он просто выйдет.

Уничтожение экземпляр не освобождает ссылку на экземпляр *(переменная по прежнему будет ссылаться на уже не существующий класс)*:

```
Var
  A : TComponent;
begin
  A:=TComponent.Create;
  A.Name:='MyComponent';
  A.Free;
  Writeln('Переменной A - еще присвоен компонент: ',Assigned
(A));
end.
```

После вызова Free, переменная A не будет Nil, программа выведет:

```
Переменной A - еще присвоен компонент: TRUE
```

Чтобы убедиться, что переменная A очищена после вызова деструктора нужно вызвать функцию FreeAndNil из модуля SysUtils. Это будет вызывать Free, а затем запишет Nil в указатель на объект *(как переменная A в приведенном выше примере)*:

```
Var
  A : TComponent;
begin
  A:=TComponent.Create;
  A.Name:='MyComponent';
  FreeAndNil(A);
  Writeln('Переменной A - еще присвоен компонент: ',Assigned
(A));
end.
```

После вызова FreeAndNil, переменная A будет содержать Nil, вывод этой программы будет:

```
Переменной A - еще присвоен компонент: FALSE
```

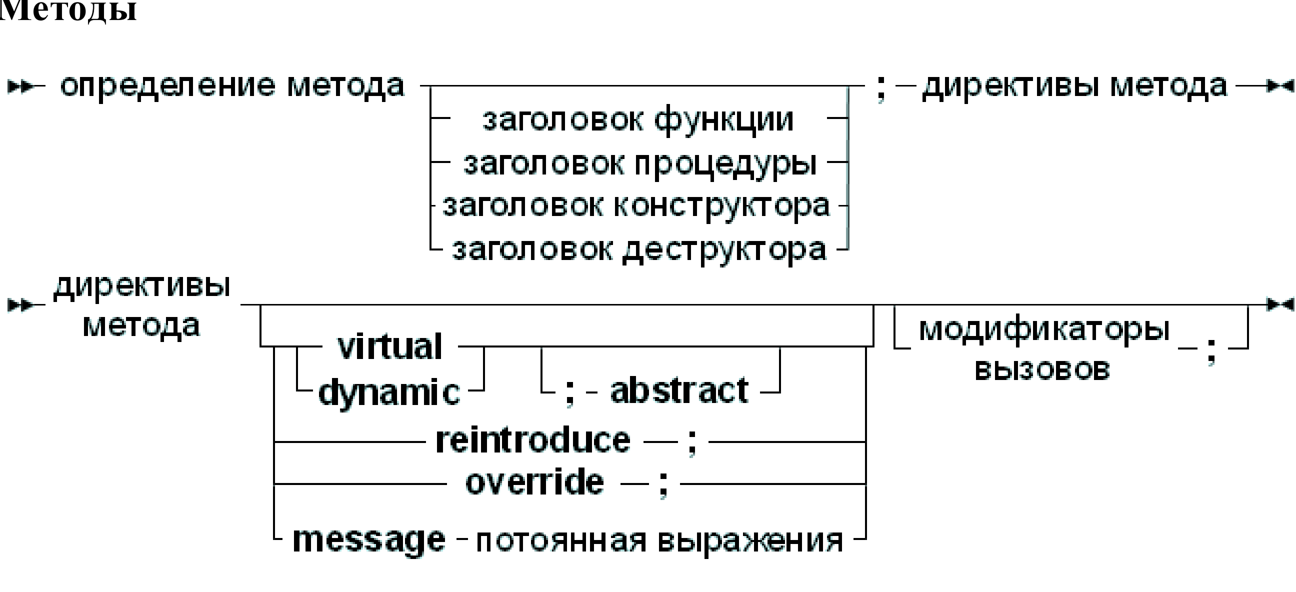
Примечание:

Если во время выполнения конструктора происходит исключение, деструктор вызывается автоматически.

6.5 Методы

6.5.1 Объявление

Объявление методов класса следует тем же правилам, что и в объявлении методов объектов:



Единственным отличием являются директивы `override` *(переопределение)*, `reintroduce` *(вернуть)* и `message` *(сообщение)*.

6.5.2 Вызов

Вызов метода класса ничем не отличается, от вызова метода для объекта. Ниже приведен допустимый вызов метода:

```
Var  AnObject : TAnObject;
begin
  AnObject := TAnObject.Create;
  ANobject.AMethod;
```

6.5.3 Виртуальные методы

Классы, так же, как объекты могут иметь виртуальные методы. Однако между ними существует разница. Для объекта, достаточно пере объявить метод в объекте потомке с ключевым словом `virtual`, чтобы заменить его. Для класса, ситуация иная: виртуальные методы ***должны быть*** перекрыты с ключевым словом `override`. Несоблюдение этого правила, начнет ***новую*** серию виртуальных методов, скрывая предыдущую. Ключевое слово `Inherited` не будет вызывать метод предка, если были использованы `virtual` *(виртуальные)* методы.

Следующий код является ***неправильным***:

```
Type
```

```
  ObjParent = Class
    Procedure MyProc; virtual;
  end;

  ObjChild  = Class(ObjPArent)
    Procedure MyProc; virtual;
  end;
```

Компилятор выдаст предупреждение:

```
Warning: An inherited method is hidden by OBJCHILD.MYPROC
Внимание: унаследованный метод скрыт OBJCHILD.MYPROC
```

Это код будет компилироваться, но с `Inherited` *(унаследованными)* методами будут происходить странные вещи.

Правильным объявлением классов, будет так:

```
Type
  ObjParent = Class
    Procedure MyProc; virtual;
  end;

  ObjChild  = Class(ObjPArent)
    Procedure MyProc; override;
  end;
```

Этот код будет компилироваться и запускаться без предупреждений или ошибок.

Если виртуальный метод действительно следует заменить методом с тем же именем, то нужно использовать ключевое слово `reintroduce`:

```
Type
  ObjParent = Class
    Procedure MyProc; virtual;
  end;
  ObjChild  = Class(ObjPArent)
    Procedure MyProc; reintroduce;
  end;
```

Метод `MyProc` больше не быдет виртуальным.

Для того, чтобы быть в состоянии сделать это, компилятор сохраняет для каждого класса - ***таблицу с виртуальных методов***:*VMT (Virtual Method Table)*. Это таблица с указателями на каждый из виртуальных методов: каждый виртуальный метод имеет фиксированное положение в этой таблице ***(индекс)***. Компилятор использует эту таблицу, чтобы определить, какой метод будет фактически использоваться во время выполнения. Когда объект потомка переопределяет метод, в VMT переписывается родительский метод. Более подробную информацию о VMT можно найти в Справочник программиста Free

Pascal.

Примечание:

Ключевое слово virtual *(виртуальный)* можно заменить на ключевое слово dynamic *(динамический)*:динамические методы ведут себя так же, как виртуальные методы. В отличие от Delphi, в FPC реализация динамических методов сделана также как и реализация виртуальных методов.

6.5.4 Методы класса

Методы класса определяются по ключевому слову Class перед объявлении процедуры или функции, как в следующем примере:

```
Class Function ClassName : String;
```

Методы класса могут не иметь экземпляра *(т.е. не указывать на экземпляр класса) (в этом случае указывается тип (название класса))*, но которые следуют правилам наследования класса. Они могут быть использованы для возврата информации о текущем классе, например, для регистрации класса или использования в фабрике классов. Поскольку экземпляр класса не может быть доступен, и не может вернуть информацию в подобных случаях.

Методы класса могут быть вызваны как внутри обычного метода, так и с использованием идентификатора класса *(без создания экземпляра класса)*:

```
Var
  AClass : TClass; // AClass имеет тип "type of class"
begin
  ..
  // ClassName - функция клааса TObject, возвращает строковое
имя класса
  if CompareText(AClass.ClassName,'TCOMPONENT')=0 then
  ...
```

Можно вызвать метод и из экземпляра класса *(метод* ClassNameIs - *метод (функция) класса, и он возвращает* True *если класс соответствует переданному типу (название типа - строковая константа))*:

```
Var
  MyClass : TObject;
begin
  ..
  if MyClass.ClassNameis('TCOMPONENT') then
  ...
```

Обратное невозможно: Внутри метода класса, **нельзя** обратится к Self *(точки таблицы VMT класса)*. **Никакие поля, свойства или обычные методы не доступны внутри *метода класса*.** Попытка обращения к

обычному свойству или методу *(из метода класса)* приведет к ошибке компиляции.

Обратите внимание, что методы класса объявленые как `virtual` *(виртуальные)* могут быть `overridden` *(переопределены)*.

В обычных свойствах, для записи или чтения можно использовать методы классов, но естественно, эти свойства будут иметь это значение во всех экземплярах класса, так как экземпляр не доступен в методе класса.

6.5.5 Конструктор и деструктор класса

Могут быть созданы как ***конструктор*** так и ***деструктор класса***. Они служат для создания экземпляра класса, некоторые переменные или свойства которого, должны быть инициализированы до того, как класс будет использован. Конструктор вызываются автоматически при запуске программы. ***Конструктор класса*** вызывается **до** секции `initialization` *(инициализации)* модуля, ***деструктор класса*** вызывается **после** секции `finalization` *(финализации)* этого модуля.

При использовании ***конструктора и деструктора класса*** нужно соблюдать некоторые ***меры предосторожности***:

- **Конструктор** должен быть назван `Create`, и не может иметь ***никаких параметров***.
- **Деструктор** должен быть назван `Destroy`, и не может иметь ***никаких параметров***.
- Ни **конструктор**, ни **деструктор** класса ***не может быть виртуальным***.
- **Конструктор и деструктор класса** вызывается независимо от использования класса: ***даже если класс никогда не используется***, конструктор и деструктор всё равно ***вызывается***.
- Порядок вызова ***конструкторов*** или ***деструкторов классов*** **не гарантируется**. Для вложенных классов единственным гарантированием порядка является то, что ***конструкторы вложенных классов*** вызываются **после** вызова конструктора класса, содержащего их, для деструкторов используется обратный порядок *(они вызываются* ***до*** *этого* ***деструктора****)*.

Рассмотрим пример программы:

```
{$mode objfpc}
{$h+}

Type
  TA = Class(TObject)
  Private
    Function GetA : Integer;
    Procedure SetA(AValue : integer);
  public
```

```
    Class Constructor create;
    Class Destructor destroy;
    Property A : Integer Read GetA Write SetA;
  end;

{Class} Function TA.GetA : Integer;
begin
  Result:=-1;
end;

{Class} Procedure TA.SetA(AValue : integer);
begin
  //
end;

Class Constructor TA.Create;
begin
  Writeln('Class constructor TA');
end;

Class Destructor TA.Destroy;
begin
  Writeln('Class destructor TA');
end;

Var
  A : TA;
begin
end.
```

Программа выведет следущее:

```
Class constructor TA
Class destructor TA
```

6.5.6 Стасический метод класса

FPC понимает и ***статические методы классов***: это методы класса, у которых есть ключевое слово `Static` после объявления метода. ***Эти методы ведут себя как обычные процедуры или функции.*** Это значит, что:

- ***Они не имеют параметра*** `Self`. А значит они не могут получить доступ к обычными свойствам, полям или методами.
- ***Они не могут быть виртуальными.***
- С ними можно обращаться как с обычными процедурными переменными или функциями.

Их используют, в основном, для включения методов в пространстве имен класса, и вместо процедур *(и функций)* в пространстве имен модуля. Обратите внимание, что они имеют доступ ко всем переменным класса, типам и т.д. Далее показано как это можно сделать:

```
{$mode objfpc}
{$h+}

Type
    TA = Class(TObject)
     Private
       class var myprivatea : integer;
     public
       class Function GetA : Integer;static;
       class Procedure SetA(AValue : Integer);static;
    end;

Class Function TA.GetA : Integer;
begin
    Result := myprivateA;
end;

Class Procedure TA.SetA(AValue : integer);
begin
    myprivateA := AValue;
end;

begin
    TA.SetA(123);
    Writeln(TA.MyPrivateA);
end.
```

Пример будет выводить `123`.

В коде реализации ***статического метода класса***, недоступен идентификатор `Self`. Метод ведет себя так, как будто ***он*** жёстко привязан к классу *(всем классам)*, а не к экземпляру класса, из которого он был вызван. В обычных методов класса, `Self` содержит класс, который вызвал метод. Следующий пример проясняет ситуацию:

```
Type
    TA = Class
      Class procedure DoIt; virtual;
      Class Procedure DoitStatic;static;
    end;

  TB = CLass(TA)
```

```
    Class procedure DoIt; override;
  end;

Class procedure TA.DOit;
begin
    Writeln('TA.Doit : ',Self.ClassName);
end;

Class procedure TA.DOitStatic;
begin
    Doit;
    Writeln('TA.DoitStatic : ',ClassName);
end;

Class procedure TB.DoIt;
begin
    Inherited;
    Writeln('TB.Doit : ',Self.ClassName);
end;

begin
    Writeln('С помощью статического метода:');
    TB.DoItStatic;
    Writeln('С помощью метода класса:');
    TB.Doit;
end.
```

При запуске, пример выведет:

```
С помощью статического метода:
TA.Doit : TA
TA.DoitStatic : TA
С помощью метода класса:
TA.Doit : TB
TB.Doit : TB
```

Для ***статического метода класса***, даже если он вызван с использованием класса TB, *(Self, если он доступен)* установлен в TA, *где и был определен статический метод класса*. Для ***метода класса***, класс *(Self)* устанавливается на фактический класс, используемый для вызова метода *(TB)*.

6.5.7 Методы обработки сообщений

Новое в методах обработки message *(сообщений)* класса. Указатели на методы обработки сообщений хранятся в специальной таблице, вместе с объявленными целыми или строковыми константами.Они в предназначены для облегчения программирования функций обратного вызова в некоторых GUI, таких как

Win32 или GTK. В отличии с Delphi, Free Pascal также может использовать ***строки*** в качестве идентификаторов сообщений. ***Методы обработки сообщений всегда виртуальные***.

При объявлении класса, методы обработки сообщений объявляются с ключевым словом Message с целочисленной константой *(или выражением)*.

Кроме того, они могут принимать только один var аргумент *(типизированный или нет)*:

```
Procedure TMyObject.MyHandler(Var Msg); Message 1;
```

Реализация метода обработки сообщений ничем не отличается от реализации обычного метода. Кроме того, можно вызвать метод обработки сообщения непосредственно, но этого ***не нужно*** делать. Вместо этого следует использовать метод TObject.Dispatch. Методы сообщений виртуальны, то есть они могут быть переопределены в классах потомках.

Метод TObject.Dispatch может быть использован для обработки сообщения (message). Сообщение *(аргумент метода)* объявлено в модуле system, первым должно быть поле типа Cardinal с идентификатором сообщения.

Например:

```
Type
  TMsg = Record
    MSGID : Cardinal;
    Data : Pointer;
Var
  Msg : TMSg;

MyObject.Dispatch (Msg);
```

В этом примере, метод Dispatch просматривает объект и всех его предков *(поиск начинается с объекта и продолжается вверх по дереву наследования класса)*, чтобы найти метод обработки сообщения с объявленым MSGID . Если такой метод найден, он вызывается с параметром Msg.

Если такой метод не найден, вызывается DefaultHandler. DefaultHandler представляет собой виртуальный метод TObject, который по умолчанию ничего не делает, но он может быть перекрыт, чтобы обеспечить ту обработку, которая может понадобиться. DefaultHandler объявлен следующим образом..:

```
procedure DefaultHandler(var message);virtual;
```

Кроме сообщения с идентификатором Integer, Free Pascal также поддерживает метод сообщения со строковым идентификатором.

```
Procedure TMyObject.MyStrHandler(Var Msg); Message 'OnClick';
```

В отличае от метода, где идентификатор - целое число, в этом случае

идентификатором является строка.

Метод TObject.DispatchStr может быть использован для вызова обработчика сообщений. *Этот метод похож на предыдущий* (Dispatch), *но в Msg первое поле - строковое.* Сообщение *(аргумент метода)* объявлено в модуле system, первым должно быть поле типа String с идентификатором сообщения. Например:

```
Type
  TMsg = Record
    MsgStr : String[10]; // Произвольное, длиной до 255 символов.
    Data : Pointer;
Var
  Msg : TMSg;

MyObject.DispatchStr (Msg);
```

В этом примере, метод DispatchStr просматривает объект и всех его предков *(поиск начинается с объекта и продолжается вверх по дереву наследования класса)*, чтобы найти метод обработки сообщения с объявленым MsgStr. Если такой метод найден, он вызывается с параметром Msg.

Если такой метод не найден, вызывается DefaultHandlerStr. DefaultHandlerStr представляет собой виртуальный метод TObject, который по умолчанию ничего не делает, но он может быть перекрыт, чтобы обеспечить ту обработку, которая может понадобиться. DefaultHandlerStr объявлен следующим образом..:

```
procedure DefaultHandlerStr(var message);virtual;
```

В дополнение к этому механизму *(кроме этого механизма)*, метод обработчика сообщения (строковым идентификатором) принимает параметр self:

```
Procedure StrMsgHandler(Data: Pointer; Self: TMyObject);
Message 'OnClick';
```

Если компилятор сталкивается таким методом, генерирует код, который загружает в параметр Self указатель на экземпляра объекта. Результатом этого является то, что можно передать Self в качестве параметра для метода.

Примечание:

Параметр Self должен иметь тип класса с определённым методом *(обработчика сообщения)*.

6.5.8 Использование наследования

В переопределяемом виртуальном методе часто бывает необходимо вызвать родительский метод. Это может быть сделано с помощью ключевого слова inherited. Ключевое слово inherited можно использовать и чтобы

вызвать любой метод родительского класса.

В простейшем случае:

```
Type
  TMyClass = Class(TComponent)
    Constructor Create(AOwner : TComponent); override;
  end;

Constructor TMyClass.Create(AOwner : TComponent);
begin
  Inherited;
  // Ещё код
end;
```

В приведенном выше примере, `Inherited` оператор вызовет *(конструктор)* `Create` из `TComponent`, передав ему `AOwner` в качестве параметра: параметры, которые были переданы текущему методу могут быть переданы и методу родителя. Параметры могут быть ***явно не указаны***: если ничего не указано, то компилятор передаст в метод те же аргументы, что и получил.

Второй случай немного сложнее:

```
Type
  TMyClass = Class(TComponent)
    Constructor Create(AOwner : TComponent); override;
    Constructor CreateNew(AOwner : TComponent; DoExtra :
Boolean);
  end;

Constructor TMyClass.Create(AOwner : TComponent);
begin
  Inherited;
end;

Constructor TMyClass.CreateNew(AOwner : TComponent; DoExtra :
Boolean);
begin
  Inherited Create(AOwner);
  // Работаем
end;
```

Метод `CreateNew` сначала вызывает `TComponent.Create` и передаст ему `AOwner` в качестве параметра. Он не вызовет `TMyClass.Create`.

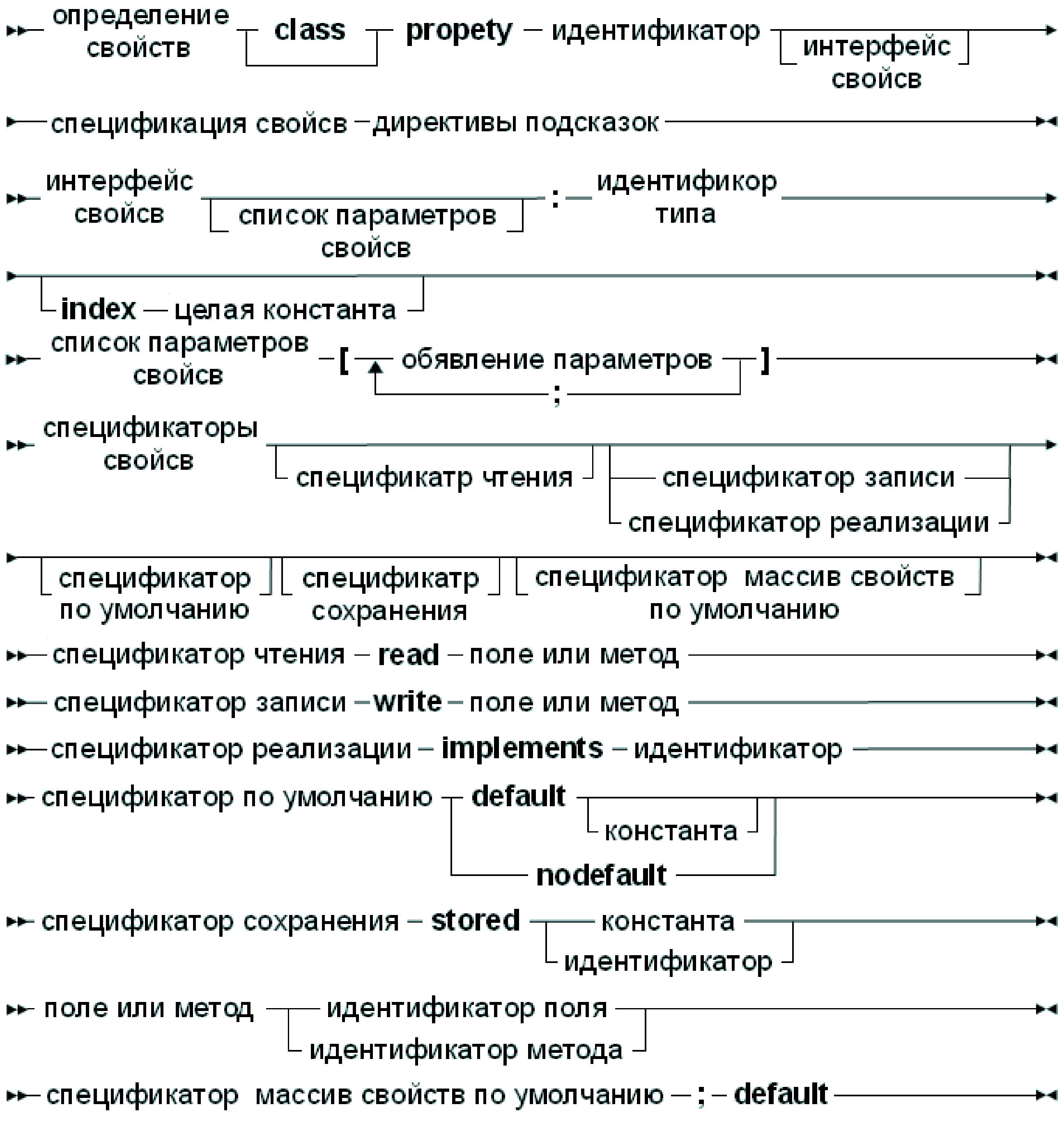
Хотя тут были приведены примеры использования конструкторов, использование `inherited` не ограничивается конструкторами, он может быть использован также для любой процедуры или функции или деструктора.

6.6 Свойства

6.6.1 Определение

Полем класса могут быть и ***свойства***. Свойство функционирует как и обычное поле, то есть его значение может быть прочитано или установлено, но она позволяет переориентировать доступ к полю с помощью функций и процедур. Они предоставляют средства, чтобы связать действие с присвоением или чтением из ***поля*** класса. Это позволяет, например, проверить, что значение является действительным для присвоении или при чтении, что позволяет конструировать значения на лету. Кроме того, свойства могут быть только для чтения или только для записи. Прототип объявления свойства выглядит следующим образом:

Свойства



`Read specifier` *(спецификатор чтения)* либо имя поля *(переменной)*, которое содержит свойство, либо имя функции *(метода)*, имеющее тот же тип что и тип свойства. В случае простого *(simple)* типа, эта функция не должна иметь аргумент. В случае массива свойств *(array property)*, функция должна иметь один аргумент типа индекса массива. В случае индексируемого свойства, она должно брать целое число в качестве аргумента.

`Read specifier` *(спецификатор чтения)* не является обязательным, и если его нет, то это свойство только для записи. Обратите внимание, что ***методы класса*** не могут быть использованы в качестве спецификаторов *(методов)* чтения.

`Write specifier` *(спецификатор записи)* - тоже не обязателный: Если его нет, то свойство только для чтения. `Write specifier` *(спецификатор записи)* - это либо имя поля *(переменной)*, либо имя процедуры *(метода)*, который принимает в качестве единственного аргумента переменную того же типа, что и свойство. В случае массива свойств *(array property)*, процедура должна принимать два аргумента: первый аргумент должен иметь тот же тип, что и индекс, второй аргумент должен быть того же типа, что и свойство. Также, в случае индексированного свойства, первый параметр должен быть целым числом.

Секция ***(private, published)***, в которой заключается указанная функция или процедура не имеет никакого значения. Однако, чаще это будет ***protected*** или ***private*** метод.

Например, учитывая следующее объявление:

```
Type
  MyClass = Class
    Private
    Field1 : Longint;
    Field2 : Longint;
    Field3 : Longint;
    Procedure  Sety (value : Longint);
    Function Gety : Longint;
    Function Getz : Longint;
    Public
    Property X : Longint Read Field1 write Field2;
    Property Y : Longint Read GetY Write Sety;
    Property Z : Longint Read GetZ;
    end;

Var
  MyClass : TMyClass;
```

Ниже приведены допустимые действия:

```
WriteLn ('X : ',MyClass.X);
WriteLn ('Y : ',MyClass.Y);
WriteLn ('Z : ',MyClass.Z);
MyClass.X := 0;
MyClass.Y := 0;
```

Но следующее присвоение будет генерировать ошибку:

```
MyClass.Z := 0;
```

потому *Z* является свойством только для чтения.

В приведенных выше действиях происходит то, что, когда читается значение, компилятор вызывает метод объекта `getNNN`, и используется результат вызова

этого метода. Когда задание присваивается *(свойству)*, компилятор передает значение , которое должно быть присвоено в качестве параметра для различных методов setNNN.

Из-за этого механизма свойства не могут быть переданы в качестве var аргументов в функции или процедуры, так как неизвестен адрес свойства *(по крайней мере, не всегда)*.

6.6.2 Индексированные свойства

Если свойство содержит index *(индекс)*, то читать и писать элементы можно также с помощью функций и процедур. Кроме значения свойства, эти функции требуют дополнительного параметра: целого числа *(индекса)*. Это позволяет читать или писать этими методами несколько свойств. Для этого, свойства должны иметь один и тот же тип. Ниже приведен пример свойств с индексом:

```
{$mode objfpc}
Type
  TPoint = Class(TObject)
  Private
    FX,FY : Longint;
    Function GetCoord (Index : Integer): Longint;
    Procedure SetCoord (Index : Integer; Value : longint);
  Public
    Property X : Longint index 1 read GetCoord Write SetCoord;
    Property Y : Longint index 2 read GetCoord Write SetCoord;
    Property Coords[Index : Integer]:Longint Read GetCoord;
  end;

Procedure TPoint.SetCoord (Index : Integer; Value : Longint);
begin
  Case Index of
   1 : FX := Value;
   2 : FY := Value;
  end;
end;

Function TPoint.GetCoord (INdex : Integer) : Longint;
begin
  Case Index of
   1 : Result := FX;
   2 : Result := FY;
  end;
end;

Var
```

```
    P : TPoint;

  begin
    P := TPoint.create;
    P.X := 2;
    P.Y := 3;
    With P do WriteLn ('X=',X,' Y=',Y);
  end.
```

Когда компилятор встречает присваивание полю X, то при вызове SetCoord в качестве первого параметра index *(1 в приведенном выше примере)* и в качестве второго параметра значение, которое нужно установить. И наоборот, при чтении значения X, компилятор вызывает GetCoord и передает ему index **1. Индексы могут быть только целые значения.**

6.6.3 Массив свойств

Можно также использовать и ***массив свойств***. Это свойства, которые имеют индекс, так же, как его имеет массив. Индекс может быть одномерным или многомерным. В отличии от массива *(статического или динамического)*, индекс ***массива свойств*** может быть не только порядкового типа, но но и любого другого типа.

Read specifier *(спецификатор чтения)* для ***массива свойств*** является методом *(функцией)*, которая имеет тотже тип возвращаемого значения, что и тип элемента свойства. Функция должна иметь в качестве единственного параметра переменную одного и того же типа, что и тип индекса. Для ***массива свойств***, нельзя указать поле котрое бы использовал read specifier.

Write specifier *(спецификатор записи)* для ***массива свойств*** является метод *(процедур)*, которая имеет два аргумента: первый аргумент имеет тот же тип, что и индекс, а второй аргумент является параметром того же типа, что и тип элемента свойства.

В качестве примера, смотри следующее объявление:

```
  Type
    TIntList = Class
    Private
      Function GetInt (I : Longint) : longint;
      Function GetAsString (A : String) : String;
      Procedure SetInt (I : Longint; Value : Longint;);
      Procedure SetAsString (A : String; Value : String);
    Public
      Property Items [i : Longint] : Longint Read GetInt Write
  SetInt;
      Property StrItems [S : String] : String Read GetAsString
  Write SetAsstring;
```

```
    end;

  Var
    AIntList : TIntList;
```

Тогда следующие операции будут справедливы:

```
  AIntList.Items[26] := 1;
  AIntList.StrItems['twenty-five'] := 'zero';
  WriteLn ('Item 26 : ',AIntList.Items[26]);
  WriteLn ('Item 25 : ',AIntList.StrItems['twenty-five']);
```

В то время как эти операции будет генерировать ошибку:

```
  AIntList.Items['twenty-five'] := 1;
  AIntList.StrItems[26] := 'zero';
```

Поскольку типы индексов неверны.

Массив свойств может быть многомерный:

```
  Type
    TGrid = Class
    Private
      Function GetCell (I,J : Longint) : String;
      Procedure SetCell (I,J : Longint; Value : String);
    Public
      Property Cellcs [Row,Col : Longint] : String Read GetCell
  Write SetCell;
    end;
```

Если есть размер **N**, то типы первых **N** аргументов методов Get и Set должны соответствовать типам индексов спецификаторов **N** в определении массива свойств.

6.6.4 Свойства по умолчанию

Массив свойств может быть объявлен как ***свойство по умолчанию***. Это означает, что не нужно указать имя свойства при присваивания или чтении. Еслибы в предыдущем примере определение свойства элементов было бы:

```
  Property Items[i : Longint]: Longint Read GetInt Write SetInt;
  Default;
```

Тогда присвоение

```
  AIntList.Items[26] := 1;
```

Было бы равносильно следующему сокращению.

```
  AIntList[26] := 1;
```

В классе допускается только одно свойство по умолчанию, но при объявлении класса можно переобъявить в потомке свойство по умолчанию.

6.6.5 Публикуемые (Published) свойства

Классы, скомпилированные с ключём {$M+} *(например, TPersistent из модуля classes)* может иметь раздел published *(опубликованно)*. Для методов, полей и свойств в этом разделе *(published)*, компилятор генерирует информацию RTTI *(Run Time Type Information)* , которая может быть использована для запроса определенных здесь методов, полей и свойств в published секции (ях). Модуль typinfo содержит необходимые процедуры, для запроса этой информации, и этот модуль используется в системе потоковой передачи в FPC в модуле classes.

RTTI генерируется независимо спецификатора чтения и записи: поля, функции и процедуры *(простые или индексированные)*.

Только типизированные поля класса могут быть *опубликованы*. Любое простое свойство, размер которого меньше или равна указатель, может быть объявлен published: **floats** *(вещественные)*, **integers** *(целые)*, **sets** *(множества) (с менее чем 32 различных элементов)*, **enumerateds** *(перечисления)*, **классы** или **динамические массивы** *(но **не** массив свойств)*.

Хотя информация о типе во время выполнения доступна для других типов, эти типы не могут быть использованы для определения свойств или полей в секции published. Информация присутствует, чтобы описать например аргументы процедур или функций.

6.6.6 Сохраняемая информация

Stored specifier *(спецификатор сохраненная)* должен быть либо ***логическое константа***, либо ***логическое поле*** класса, либо ***функция*** без параметров, возвращающая логический результат. Этот спецификатор ***не оказывает*** никакого результата на поведение класса. *Это вспомогательное средство для **потоковой системы***: stored specifier указан в RTTI для сгенерированного класса *(он может быть потоковым только если генерируется RTTI)*, и используется для определения того, следует ли свойству быть потоковым или нет: это экономит место в потоках *(файлах)*. Что не даёт возможности использовать директиву Stored для массива свойств.

Default specifier *(спецификатор по умолчанию)* может быть задан для порядковых типов и множеств. Он служит той же цели, что и stored specifier *(спецификатор сохранения)*: свойства, имеющие значение значения по умолчанию *(default)*, не будут записаны в поток с помощью потоковой системы. Значение по умолчанию сохраняется в RTTI для генерируемого класса. Обратите внимание, что

1. Значение по умолчанию ***автоматически не применяется*** к свойству, и программист отвечает за то, чтобы сделать это в конструкторе класса *(чтобы свойство имело значение по умолчанию в экземпляре класса)*.
2. Значение **2147483648** не может использоваться в качестве значения по

умолчанию, так как оно используется как nodefault.

3. ***Нельзя*** указать значения по умолчанию для массива свойств.

Nodefault specifier *(nodefault)* используется для указания того, что свойство ***не имеет*** значения по умолчанию. Значение этого свойства ***всегда*** записывается в поток при потоковых операциях.

6.6.7 Переопределение свойств

Свойства могут быть перекрыты в потомках класса, так же, как и методы. Разница заключается в том, что для свойства, всегда можно сделать *overriding (основным)*: свойства не нужно маркировать как *«виртуальное»*, т.е. они всегда могут быть перекрыты, (в этом смысле, свойства всегда *виртуальные*). Тип переопределенного свойства ***не должен*** быть таким же, как тип свойства родительского класса.

Так как свойство может быть перекрыто, ключевое слово inherited *(наследовать)* тоже может быть использовано для обозначения родительского свойства. Например, рассмотрим следующий код:

```
type
    TAncestor = class
      private
        FP1 : Integer;
      public
        property P: integer Read FP1 write FP1;
    end;

    TClassA = class(TAncestor)
      private
        procedure SetP(const AValue: char);
        function getP : Char;
      public
        constructor Create;
        property P: char Read GetP write SetP;
    end;

procedure TClassA.SetP(const AValue: char);
begin
    Inherited P:=Ord(AValue);
end;

procedure TClassA.GetP : char;
begin
    Result:=Char((Inherited P) and $FF);
end;
```

Класс `TClassA` переопределяет свойство `P` как *свойство* **символ** вместо **целого** *свойства*, но использует свойство родителя `P` для хранения значения.

Необходимо соблюдать осторожность при использовании *виртуальных **get/set*** процедур для свойств: при наследовании свойств применяются ***теже правила*** что и при наследовании методов. Рассмотрим следующий пример:

```
type
    TAncestor = class
      private
        procedure SetP1(const AValue: integer); virtual;
      public
        property P: integer write SetP1;
    end;

    TClassA = class(TAncestor)
      private
        procedure SetP1(const AValue: integer); override;
        procedure SetP2(const AValue: char);
      public
        constructor Create;
        property P: char write SetP2;
    end;

constructor TClassA.Create;
begin
    inherited P:=3;
end;
```

В этом случае при установке наследованного свойства `P`, будет вызываться метод `TClassA.SetP1` а не родительский `SetP1`.

Если при реализация класса, должен быть вызван родительский `SetP1`, то это должно быть ***явно указано*** *(обращением не к свойству, к самой функции)*:

```
constructor TClassA.Create;
begin
    inherited SetP1(3);
end;
```

6.7 Свойства класса

Свойства класса очень похожи на определения глобальных свойств. Они связаны с классом, а не с экземпляром класса.

Следствием этого является то, что для хранения значения свойства должен быть переменной класса, а не обычной переменной: нормальные поля хранятся в экземпляре класса, когда переменные класса, хранятся в типе класса.

(Переменные класса хранятся в сегменте данных, а обычные - в "куче")

Свойства класса могут иметь методы получения ***(Get)*** и установки ***(Set)***, но эти методы дожны быть тоже методами класса *(статическими)*.

Так могут быть определены методы и свойства класса:

```
TA = Class(TObject)
Private
  class var myprivatea : integer;
  class Function GetB : Integer;  static;
  class Procedure SetA(AValue : Integer); static;
  class Procedure SetB(AValue : Integer); static;
public
  Class property MyA : Integer Read MyPrivateA Write SetA;
  Class property MyA : Integer Read GetB Write SetB;
end;
```

Причиной этих требований является то, что свойство класса связано с конкретным классом *(в котором оно было определено)*, но не дочерним классом. Поскольку методы класса могут быть виртуальными, это позволило бы классам-потомкам переопределить метод, что делает его непригодным для доступа к свойствам класса.

6.8 Вложенные типы, константы и переменные

Определение класса может содержать раздел описания ***типов***, ***констант*** и ***переменных***. Разделы типов и констант действуют также, как такие же разделы в модуле или реализации метода/функции/процедуры. Переменные действует как обычные поля класса, кроме того что они находятся в разделе class var *(переменных класа)*, что означает, что они действуют как если бы они были определены на уровне модуля, в пределах пространства имен класса *(6.2 Обычные и статические поля[99])*.

Тем не менее, видимость этих разделов имеет значение: private и protected (строгие *(private strict)* или нет) константы, типы и переменные могут использоваться только там, где допускает их область видимости.

Типы public могут использоваться за пределами класса при использовании их полного имени:

```
type
  TA = Class(TObject)
  Public
    Type TEnum = (a,b,c);
    Class Function DoSomething : TEnum;
  end;
```

```
Class Function TA.DoSomething : TEnum;
begin
  Result:=a;
end;

var
  E : TA.TEnum;

begin
  E:=TA.DoSomething;
end.
```

В то время как

```
type
  TA = Class(TObject)
  Strict Private
    Type TEnum = (a,b,c);
  Public
    Class Function DoSomething : TEnum;
  end;

Class Function TA.DoSomething : TEnum;
begin
  Result:=a;
end;

var
  E : TA.TEnum;

begin
  E:=TA.DoSomething;
end.
```

Не будет компилироваться и вернет ошибку:

```
tt.pp(20,10) Error: identifier idents no member "TEnum"
tt.pp (20,10) Ошибка:Идентификатор "TEnum" неопределён
```

Глава 7 Интерфейсы

7.1 Определение

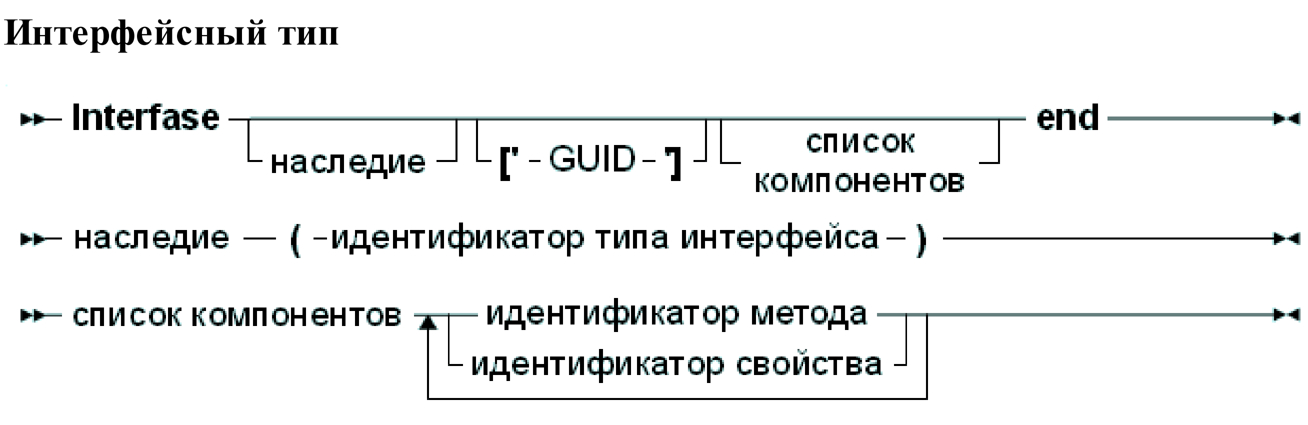
Начиная с версии **1.1**, FPC поддерживает интерфейсы. Интерфейсы являются альтернативой множественного наследования *(где класс может иметь несколько родительских классов)*, как реализовано, например, в C++. Интерфейс в основном это именованный набор методов и свойств: класс, который реализует интерфейс предоставляет все методы, как они перечислены в определении интерфейса. Класс должен реализовать ***все*** методы интерфейса: ***все или ничего.***

Из интерфейсов так же, как из классов также можно строить иерархии: *интерфейс-потомок* наследует *(от предка(ов) другого интерфейса)* все методы родительского интерфейса, а также явно указанные в его определении методы. Класс, реализующий интерфейс, должен тоже реализовать все элементы интерфейса, а также методы родительского интерфейса(ов).

Интерфейс может быть однозначно определён **GUID**. **GUID** *(аббревиатура **Globally Unique Identifier** - глобально уникальный идентификатор)* это **128**-битное целое число гарантированно уникальное *(конечно теоритически)*. В системе Windows, **GUID** интерфейса может и должен использоваться при использовании COM.

Определение интерфейса имеет следующий вид:

Интерфейсный тип



Наряду с этим определением необходимо отметить следующее:

- Интерфейсы могут быть использованы ***только*** в режимах `DELPHI` или `OBJFPC`.
- Спецификаторы ***не видны***. Все они являются `public` *(на самом деле, не имеет смысла делать их `private` или `protected`)*.
- Свойства объявленные в интерфейсе могут иметь только методы для

чтения и записи.

- В интерфейсах ***нет*** конструкторов и деструкторов. Экземпляры интерфейсов ***не могут*** быть созданы ***сами по себе***: вместо этого нужно создать экземпляр класса, реализующего интерфейс.
- В объявлении метода можно использовать ***только допустимые*** модификаторы. В определении интерфейса ***не могут*** присутствовать модификаторы `virtual`, `abstract` или `dynamic`, а следовательно и `override`.

Ниже приводятся примеры интерфейсов:

```
IUnknown = interface ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const iid : tguid;out obj) : longint;
  function _AddRef : longint;
  function _Release : longint;
end;
IInterface = IUnknown;

IMyInterface = Interface
  Function MyFunc : Integer;
  Function MySecondFunc : Integer;
end;
```

GUID, идентифицирующий интерфейс ***не является обязательным***.

7.2 Идентификация интерфейса: GUID

Интерфейс может быть определён с помощью **GUID** идентификатора. Это **128**-битное число, которое имеет текстовое представлении *(строковыми числами)*:

```
['{HHHHHHHH-HHHH-HHHH-HHHH-HHHHHHHHHHHH}']
```

Каждый символ `H` представляет собой шестнадцатеричное число ***(0-9**, **A-F)***. Формат **GUID** вмещает **8-4-4-4-12** знакомест. **GUID** представлен следующей записью, определённой в модуле objpas *(включается автоматически в режимах DELPHI или OBJFPC)*:

```
PGuid = ^TGuid;
TGuid = packed record
   case integer of
      1 : (
           Data1 : DWord;
           Data2 : word;
           Data3 : word;
           Data4 : array[0..7] of byte;
          );
      2 : (
           D1 : DWord;
           D2 : word;
           D3 : word;
           D4 : array[0..7] of byte;
          );
      3 : ( { uuid поля в соответствии с RFC4122 }
```

```
                time_low : dword;
                time_mid : word;
                time_hi_and_version : word;
                clock_seq_hi_and_reserved : byte;
                clock_seq_low : byte;
                node : array[0..5] of byte;
                );
end;
```

Константа типа TGUID может быть задана с помощью строковых чисел:

```
{$mode objfpc}
program testuid;

Const
  MyGUID : TGUID = '{10101010-1010-0101-1001-110110110110}';

begin
end.
```

Идентификаторы **GUID** используются только в Windows, при использовании COM-интерфейсов. Более подробно об этом в следующем разделе.

7.3 Реализация интерфейса

Класс реализующий интерфейс должен реализовать ***все методы*** интерфейса. Если метод интерфейса не реализован, то компилятор выдаст сообщение об ошибке. Например определение:

```
Type
  IMyInterface = Interface
    Function MyFunc : Integer;
    Function MySecondFunc : Integer;
  end;

  TMyClass = Class(TInterfacedObject,IMyInterface)
    Function MyFunc : Integer;
    Function MyOtherFunc : Integer;
  end;

Function TMyClass.MyFunc : Integer;
begin
  Result:=23;
end;

Function TMyClass.MyOtherFunc : Integer;
begin
  Result:=24;
end;
```

приведет к ошибке компиляции:

```
Error: No matching implementation for interface method "IMyIn-
terface.MySecondFunc:LongInt" found
Ошибка: Ненайдено реализации интерфейса для метода "IMyInter-
```

```
face.MySecondFunc: LongInt"
```

Как правило, имена методов, реализующих интерфейс, должны быть такими же как имена методов при определении интерфейса.

Тем не менее, можно использовать псевдонимы для методов, составляющих интерфейс: то есть, компилятору можно показать, что метод интерфейса осуществляется с помощью метода класса с другим именем. Это делается следующим образом:

```
Type
  IMyInterface = Interface
    Function MyFunc : Integer;
  end;

  TMyClass = Class(TInterfacedObject,IMyInterface)
    Function MyOtherFunction : Integer;
    Function IMyInterface.MyFunc = MyOtherFunction;
  end;
```

Это объявление говорит компилятору, что метод интерфейса `MyFunc` в интерфейсе `IMyInterface` реализован методом в классе `MyOtherFunction` классом `TMyClass`.

7.4 Делегация Интерфейса

Иногда методы интерфейса реализуются с помощью объекта хелпера *(или делегата)*, или экземпляр класса получил указатель на интерфейс который нужно использовать. Это нужно, например, когда интерфейс должен быть добавлен к последовательности абсолютно не связанных классов: необходимая функциональность интерфейса добавляется в отдельный класс, и каждый из этих классов использует экземпляр класса-помощника для реализации функциональности.

В таком случае, можно указать компилятору, что интерфейс не поддерживается самим объектом, но находится во вспомогательном классе или интерфейсе. Это можно сделать с помощью модификатора свойства `implements`.

Если класс имеет указатель на желаемый интерфейс, нужно указать компилятору что, когда запрашивается интерфейс `IMyInterface`, следует использовать ссылку на поле:

```
type
  IMyInterface = interface
    procedure P1;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
  private
    FMyInterface: IMyInterface; // тип интерфейс
  public
    property MyInterface: IMyInterface
       read FMyInterface implements IMyInterface;
```

```
  end;
```

Интерфейс не должен обязательно быть полем, может быть использован любой способ чтения.

Если интерфейс реализован с помощью объекта-делегата, *(вспомогательный объект, который и реализует интерфейс)*, то используется ключевое слово `implements`:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
  end;

  // NOTE: Interface must be specified here  Примечание:
Интерфейс должен быть указан здесь
  TDelegateClass = class(TObject, IMyInterface)
  private
    procedure P1;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
  private
    FMyInterface: TDelegateClass; // тип класса
    property MyInterface: TDelegateClass
      read FMyInterface implements IMyInterface;
  end;
```

Следует отметить, что в отличии от Delphi, класс-делегат должен явно указать на интерфейс: компилятор не будет искать методы в классе делегате, он будет просто проверить, что класс-делегат реализует указанный интерфейс.

Нельзя смешивать отождествленные методы и делегирование интерфейса. Это значит, что нельзя реализовать часть интерфейса отождествлёнными методами, а часть через делегирование. В следующем примере сделана попытка реализовать `IMyInterface` частично отождествлёнными методами *(P1)*, а частично через делегирование. Компилятор не примет код:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FI : IMyInterface;
  protected
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
    property MyInterface: IMyInterface  read FI implements
IMyInterface;
  end;
```

Компилятор выдаст ошибку:

```
Error: Interface "IMyInterface" can't be delegated by
"TMyClass", it already has method resolutions
Ошибка: Интерфейс "IMyInterface" не может быть делегирована
"TMyClass", он уже имеет отождествлённый метод
```

Однако, можно реализовать один интерфейс посредством отождествлённых методов, а другой посредством делегирования:

```
{$interfaces corba}
type
  IMyInterface = interface
    procedure P1;
  end;

  IMyInterface2 = interface
    procedure P2;
  end;

  TMyClass = class(TInterfacedObject,
                   IMyInterface, IMyInterface2)
    FI2 : IMyInterface2;
  protected
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  public
    property MyInterface: IMyInterface2
       read FI2 implements IMyInterface2;
  end;
```

7.5 Интерфейсы и COM

Для интерфейсов Windows, которые используются для COM, должно быть соглашение о вызове `stdcall`, оно не является соглашением по умолчанию для Free Pascal, его надо ***явно*** указать.

В COM ***нет*** свойств. Есть только методы. Таким образом, при определении свойств как части интерфейса *(interface)*, следует помнить, что свойства можно использовать только в программе на Free Pascal: другие программы для Windows не будут понимать свойства.

7.6 CORBA и другие интерфейсы

COM - не единственная архитектура, где используются интерфейсы. Интерфейсы есть и в CORBA, и в UNO *(OpenOffice API используются интерфейсы)*, и в Java. Эти языки не знают интерфейса `IUnknown`, который используется в качестве основы для всех интерфейсов в COM. Поэтому было бы неплохо, если бы интерфейс автоматически наследовался от `IUnknown`, если не был указан родительский интерфейс. Таким образом, директива

{$INTERFACES} была специально введена в Free Pascal: она определяет, что интерфейс объявлен без родителя. Более подробную информацию о директиве можно найти в Справочник программиста Free Pascal.

Обратите внимание , что COM - интерфейсы, по умолчанию, подсчитывает ссылки, т.к. они происходят от IUnknown.

CORBA интерфейсы объявлены с помощью обычной строки, таким образом они совместимы по присваиванию со строками, а не с TGUID. Компилятор автоматически не подсчитывает ссылки на CORBA- интерфейсы, поэтому программист несет ответственность за "ссылочную бухгалтерию".

7.7 Подсчет ссылок

Все COM интерфейсы используют подсчет ссылок. Это означает, что всякий раз, когда интерфейс присваивается переменной, обновляется и счетчик ссылок. Когда переменная выходит из области видимости, счётчик ссылок автоматически уменьшается. Когда счетчик ссылок достигает нуля, экземпляр класса, реализующий интерфейс *(как правило)*, освобождается.

Однако соблюдайте осторожность при использовании этого механизма. Компилятор может создать временную переменную при вычислении выражения, и присвоить ей интерфейс, и только затем присвоить её переменной результата. Не делайте предположений о количестве временных переменных и когда они будут освобождены - это может отличается *(и действительно отличается)* от того, как другие компиляторы *(например Delphi)* обрабатывает выражения с интерфейсами. Например, преобразование типа тоже является выражением:

```
Var
  B : AClass;

begin
  // ...
  AInterface(B.Intf).testproc;
  // ...
end;
```

Если интерфейс intf подсчитывает ссылки. То когда компилятор вычисляя B.Intf создает временную переменную. Эта переменная будет освобождена, когда процедура закончит выполнение: поэтому нельзя освобождать экземпляр B до выхода из процедуры, т.к при уничтожении временной переменной, экземпляр B попытается новь освободить память.

Глава 8. Дженерики

8.1 Введение

Дженерики *(шаблоны, обобщения)* - это шаблоны для создания других типов. Это могут быть классы, объекты, интерфейсы и даже функции, массивы, записи. Это понятие заимствовано из C++, где оно глубоко интегрировано в язык. Начиная с версии **2.2**, Free Pascal тоже официально поддерживает дженерики или шаблоны. Они реализуются как своего рода макрос, который хранится в модуле, генерируемом компилятором, и который воспроизводится, как только специализируется класс дженерика.

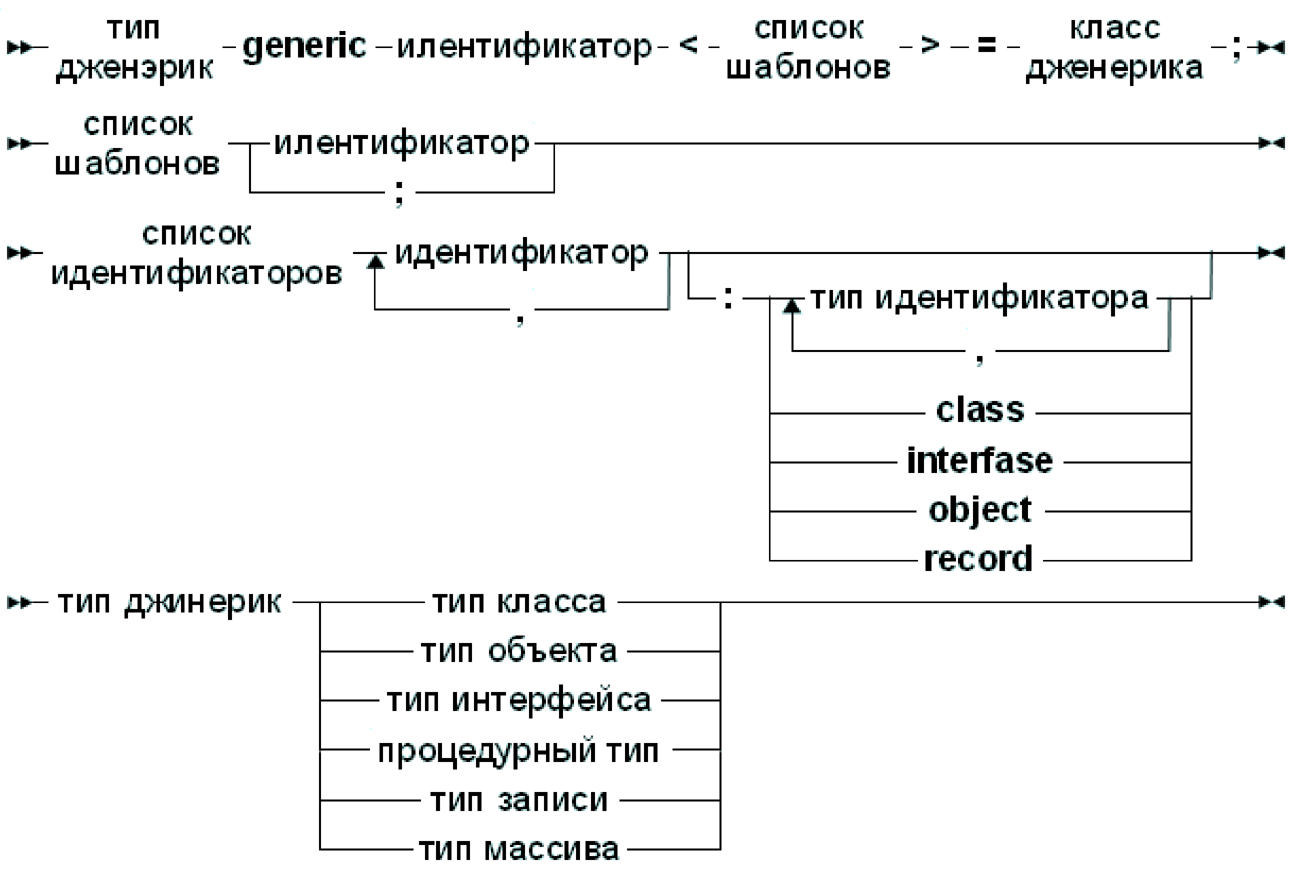
Создание и использование дженериков является двухшаговым процессом.

1. **Дженерик определяется как новый тип:** это шаблон кода, макрос, который может быть воспроизведен компилятором на более поздней стадии.
2. **Специализация типа дженерика:** определяется другой тип, который является конкретной реализацией типа дженерика: компилятор воспроизводит макрос, который был сохранен при определении типа дженерика.

8.2 Определение дженерика классов

Объявление ***дженерика*** очень похоже на определение типа, за исключением того, что он должен содержать список типов заполнителей *(шаблонов)*, как показано на следующей синтаксической диаграмме:

Дженерик классов



Для классов, объектов, процедурных типов и расширенных записей, объявление дженерика должно сопровождаться его реализацией. Это то же самое, что и обычная реализация класса, кроме того что, имя идентификатора шаблона, должен быть именем типа *(класса, записи оди простого типа)*.

Таким образом, объявление типа дженерика очень похоже на объявление обычного типа, но в объявлении присутствует неопределённый тип. Неопределённые типы перечислены в списке, и они неопределены , пока класс не специализируется.

Ниже приведено допустимое определение дженерика:

```
Type
    generic TList<_T>=class(TObject)
      type public
        TCompareFunc = function(const Item1, Item2: _T):
Integer;
      var public
        data : _T;
      procedure Add(item: _T);
      procedure Sort(compare: TCompareFunc);
    end;
```

Класс может быть реализован следующим образом:

```
procedure TList.Add(item: _T);
begin
    data:=item;
end;

procedure TList.Sort(compare: TCompareFunc);
begin
    if compare(data, 20) <= 0 then halt(1);
end;
```

В этом объявлении и реализации есть некоторые особенности:

1. Заполнитель `_T` будет заменен именем уточнённого типа, когда дженерик будет специализирован. Идентификатор `_T` не может использоваться для чего - нибудь другого, кроме заполнителя типа. Что значит, что следующий код неверен:

```
procedure TList.Sort(compare: TCompareFunc);
Var
    _t : integer;

begin
    //  какие-то действия
end;
```

2. Блок локальных типов содержит единственный тип `TCompareFunc`. Обратите внимание, что фактический тип не известен при определения дженерика: определение содержит ссылку на заполнитель `_T`. Все остальные ссылки на идентификатор должны быть известны, когда определяется класс дженерика, а ***не*** когда дженерик специализируется.
3. Блок локальных переменных определяется следующим образом:

```
    generic TList<_T>=class(TObject)
      type public
        TCompareFunc = function(const Item1, Item2: _T):
Integer;
    Public
      data : _T;
      procedure Add(item: _T);
      procedure Sort(compare: TCompareFunc);
    end;
```

Дженериками могут быть определены не только классы, но и другие типы:

```
{$mode objfpc}
{$INTERFACES CORBA}
type
    generic PlanarCoordinate<t> = record
      x,y : t;
    end;
```

```
    TScreenCoordinate = specialize PLanarCoordinate<word>;
    TDiscreteCoordinate = specialize PlanarCoordinate<integer>;
    TRealCoordinate = specialize PlanarCoordinate<extended>;

    generic TDistanceFunction<t> = function (x,y : t) :
Extended of object;

    TScreenDistance = specialize TDistanceFunction<word>;
    TDiscreteDistance = specialize TDistanceFunction<integer>;
    TRealDistance = specialize TDistanceFunction<Extended>;

    generic TArray<t> = array of t;

    TMyIntegerArray = specialize TArray<integer>;

    generic IList<_T> = Interface
      Function GetItem(AIndex : Integer) : _T;
      Procedure SetItem(AIndex : Integer; AValue : _T);
      Function GetCount : Integer;
      Property Items [AIndex : Integer] : _T Read GetItem Write
 SetItem;
      Property Count : Integer Read GetCount;
    end;

    generic TList<_T>=class(TObject, specialize IList<_T>)
    public type
      TCompareFunc = function(const Item1, Item2: _T): Integer;
      Function GetItem(AIndex : Integer) : _T;
      Procedure SetItem(AIndex : Integer; AValue : _T);
      Function GetCount : Integer;
    Public
      data : _T;
      procedure Add(item: _T);
      procedure Sort(compare: TCompareFunc);
    end;

    generic TPointSet<t> = array of specialize
PlanarCoordinate<t>;

    TScreenPointSet = specialize TPointSet<word>;
    TDiscretePointSet = specialize TPointSet<integer>;
    TRealPointSet = specialize TPointSet<extended>;
```

Примечание:

Несколько слов о зоне видимости. Типы шаблонов `T` или `_T` доступны в качестве `strict private` *(строгих частных)* типов. Это означает, что эти типы будут недоступны в классах-потомках, пока они не будут сделаны доступными с помощью некоторых специальных механизмов *(переопределения поля как `protected` (защищенного) или `private` (частного))*, что и показано в следующем примере:

```
generic TList<_T>=class(TObject)
public type
    TItemType = _T;
end;
```

8.3 Специализация дженерика класса

После того, как дженерик определен, он может использоваться для создания других типов: это как переопределение типов, с использованием шаблона, заполненного фактическими определениями типов.

Что может быть сделано в любом блоке `Type` для определения типов. Специализированный тип выглядит следующим образом:

Специализация типа

▸▸– тип специализации – specialize – идентификатор – < - список идентификаторов типов – > –▸◂

▸▸– список идентификаторов типов –┬– идентификатор –┬–▸◂ (идентификаторы разделяются символом ,)

Которое является очень простым определением. Учитывая объявление `TList` в предыдущем разделе, будет действительно следующее определение типа:

```
Type
    TPointerList = specialize TList<Pointer>;
    TIntegerList = specialize TList<Integer>;
```

В версии **3.0** Free Pascal, в объявлении переменной, тоже может быть использовано ключевое слово `specialize`:

```
Var
    P : specialize TList<Pointer>;
```

Типы в операторе `specialize`, должны быть известны *(на момент специализации)*, за исключением другого определения типа дженерика. Исходя из определения двух классов дженериков:

```
type
```

```
    Generic TMyFirstType<T1> = Class(TMyObject);
    Generic TMySecondType<T2> = Class(TMyOtherObject);
```

Недопустима следующая специализация:

```
type
    TMySpecialType = specialize TMySecondType<TMyFirstType>;
```

Так как тип `TMyFirstType` - это обобщенный тип, и он не полностью определен. Компилятор будет жаловаться:

```
Error: Generics cannot be used as parameters when specializing generics
Ошибка: Дженерики не могут быть использованы в качестве параметров для специализации
```

Но разрешено следующее:

```
type
    TA = specialize TMyFirstType<Atype>;
    TB = specialize TMySecondType<TA>;
```

Потому что `TA` уже полностью определен, на момент специализации `TB`.

Тем не менее, ключевое слово `specialize` можно использовать при определении дженерика типов, как показано в приведенном примере:

```
generic TList<_T>=class(TObject, specialize IList<_T>)
```

и

```
generic TPointSet<t> = array of specialize PlanarCoordinate<t>;
```

В этих определениях дженерика специализация осуществляется только тогда, когда специализируется сам дженерик, и все типы становятся известны.

Замечание:

Начиная с версии **3.0**, можно сделать определение дженерика классов заранее *(forward definition)*. В предыдущих версиях компилятор генерировал ошибку, если предварительное объявление класса было позже чем специализация дженерика. Это значит, что теперь возможно следующее:

```
{$mode objfpc}
Type
    TMyClass = Class;

    // Другие объявления

    TMyClass = specialize TList<T>;
```

8.4 Ограничения дженериков

Диаграмма в разделе 8.1 Введение[132], показывает, что список шаблонов для типа может иметь дополнительные спецификаторы для типов. Это особенно

полезно для типов объектов: если тип шаблона должен быть унаследован от определенного класса, то это может быть указано в списке шаблонов:

```
{$mode objfpc}
{$h+}
uses sysutils, classes;

Type
    generic TList<_T : TComponent> = class(TObject)
    public
      Type TCompareFunc = function(const Item1, Item2: _T):
Integer;
    Public
      data : _T;
      procedure Add(item: _T);
      procedure Sort(compare: TCompareFunc);
    end;
```

Учитывая приведенное выше описание, следующий код будет скомпилирован:

```
TPersistentList = specialize TList<TComponent>;
```

Но не будет компилироваться это

```
TPersistentList = specialize TList<TPersistent>;
```

Компилятор вернёт ошибку:

```
Error: Incompatible types: got "TPersistent" expected "TCompon-
ent"
Ошибка: Несовместимые типы: получен "TPersistent" вместо
ожидаемого "TComponent"
```

Можно сгруппировать вместе несколько типов:

```
Type
    generic TList<Key1,Key2 : TComponent; Value1 : TObject> =
class(TObject)
```

Кроме того, можно указать более одного идентификатора типа для ограничений типа класса и интерфейса. Если указан класс, то компилятор определяет подойдёт или нет такой дженерик для использования как шаблона для типа.

```
Type
    generic TList<T: TComponent, IEnumerable> = class(TObject)
```

Класс используемый для специализации `T` должен наследоваться от `TComponent` и реализовывать интерфейс `IEnumerable`.

Если указан интерфейс, то шаблон для типа должен реализовать этот интерфейс, но он может быть и потомком этого интерфейса:

```
Type
    generic TGenList<T: IEnumerable> = class(TObject)
```

```
    IMyEnum = Interface (IEnumerable)
      Procedure DoMy;
    end;

    TList = specialize TGenList<IMyEnum>;
    TSomeList = Specialize TGenList<TList>;
```

Можно указать несколько интерфейсов, в этом случае класс должен реализовать все перечисленные интерфейсы: можно смешать одно имя класса с несколькими именами интерфейсов.

Если не действуют ограничения для типа, то компилятор будет считать, что типы шаблонов ***не совместимы*** по присваиванию.

Это особенно важно, когда дженерик содержит перегруженные методы. Учитывая следующее обобщение для типов:

```
type
    generic TTest<T1, T2> = class
      procedure Test(aArg: LongInt);
      procedure Test(aArg: T1);
      procedure Test(aArg: T2);
    end;
```

При специализации написанное выше будет компилироваться, если `T1` и `T2` это два различных типа, и ни один не является `LongInt`. Такое выражение должно компилироваться:

```
T1 = specialize TTest<String, TObject>;
```

Но следующих два выражения не компилируются:

```
T2 = specialize TTest<String, String>;
```

или

```
T2 = specialize TTest<String, Longint>;
```

8.5 Совместимость с Delphi

Поддержка дженериков в FPC несколько отличается от Delphi. В этом разделе будут показаны основные отличия.

8.5.1 Элементы синтаксиса

На синтаксических диаграммах показан синтаксис для режима ObjFPC. В режиме Delphi ключевые слова `specialize` и `generic` не используются, это показано в следующем примере:

```
Type
    TTest<T> = Class(TObject)
```

```
      Private
        FObj : T;
      Public
        Property Obj : T Read FObj Write FObj;
      end;

      TIntegerTest = TTest<Integer>;
```

В отличии от режима Objfpc, имя шаблона типов должно повторяться при определении методов.

```
  Type
      TTest<T> = Class(TObject)
      Private
        FObj : T;
      Public
        Procedure DoIt;
        Property Obj : T Read FObj Write FObj;
      end;

  Procedure TTest<T>.DoIt;
  begin
  end;
```

Это требование связано с возможностью перегрузки дженерика, упомянутую в следующем разделе.

8.5.2. Ограничения для записей

В режиме Delphi, ограничения для типа `record` также позволяют использовать простые типы:

```
  Type
  generic TList<_T : record> = class(TObject)
      public
        Type TCompareFunc = function(const Item1, Item2: _T):
  Integer;
      Public
        data : _T;
        procedure Add(item: _T);
        procedure Sort(compare: TCompareFunc);
      end;

  TIntList = TList<Integer>;
```

8.5.3 Перегрузка типов

Режим Delphi позволяет перегрузку дженерика. Это означает, что можно объявить один и тот же класс дженерика с различными списками типа шаблонов. Поэтому возможны следующие объявления:

```
Type
    TTest<T> = Class(TObject)
    Private
      FObj : T;
    Public
      Property Obj : T Read FObj Write FObj;
    end;

    TTest<T,S> = Class(TObject)
    Private
      FObj1 : T;
      FObj2 : S;
    Public
      Property Obj1 : T Read FObj1 Write FObj1;
      Property Obj2 : S Read FObj2 Write FObj2;
    end;
```

8.5.4 Соглашение о пространствах имен

В режиме Delphi, дженерики не мешают пространству имен переменных, это значит что будет скомпилирован следующий код:

```
Type
    TTest<T> = Class(TObject)
    Private
      FObj : T;
    Public
      Property Obj : T Read FObj Write FObj;
    end;

Var
    TTest : Integer;
```

Но это не работает для констант и функций.

8.5.5 Соглашение об области действия

Режим Delphi позволяет использовать специализированный дженерик без его явного объявления. А значит возможно следующее:

```
var
```

```
    t: TTest<LongInt>;
begin
    t := TTest<LongInt>.Create;
end;
```

В режиме objfpc, тип обязательно должен быть объявлен.

8.6 Совместимость типов

Специализация дженерика всегда приводит к новому типу. Эти типы совместимы по присваиванию, если используют один и тот же шаблон типов.

Возьмем следующее определение дженерика:

```
{$mode objfpc}
unit ua;

interface

type
    Generic TMyClass<T> = Class(TObject)
      Procedure DoSomething(A : T; B : INteger);
    end;

Implementation

Procedure TMyClass.DoSomething(A : T; B : Integer);
begin
    // какой-то код
end;

end.
```

И следующие специализации:

```
{$mode objfpc}
unit ub;

interface

uses ua;

Type
    TB = Specialize TMyClass<string>;

implementation

end.
```

Идентичную специализацию, но описанную в другом модуле:

```
{$mode objfpc}
unit uc;

interface

uses ua;

Type
    TB = Specialize TMyClass<string>;

implementation

end.
```

Код ниже будет скомпилирован:

```
{$mode objfpc}
unit ud;

interface

uses ua,ub,uc;

Var
    B : ub.TB;
    C : uc.TB;

implementation

begin
    B:=C;
end.
```

Типы `ub.TB` и `uc.TB` совместимы по присваиванию. Не важно, что эти типы описаны в различных модулях. Можно их описать и в одном модуле:

```
{$mode objfpc}
unit ue;

interface

uses ua;

Type
    TB = Specialize TMyClass<string>;
    TC = Specialize TMyClass<string>;
```

```
Var
    B : TB;
    C : TC;

implementation

begin
    B:=C;
end.
```

Каждая специализация дженерика классов, с теми же типами в качестве параметров является новым, особым типом, но эти типы совместимы по присваиванию, если типы шаблонов, используемых для их специализации равны.

Если специализация является типом с другим шаблоном, типы по-прежнему различны и не совместимы по присваиванию. А значит следующий код не будет компилироваться:

```
{$mode objfpc}
unit uf;

interface

uses ua;

Type
    TB = Specialize TMyClass<string>;
    TC = Specialize TMyClass<integer>;

Var
    B : TB;
    C : TC;

implementation

begin
    B:=C;
end.
```

При компиляции возникнет ошибка:

```
Error: Incompatible types: got "TMyClass<System.LongInt>" expected "TMyClass<System.ShortString>"
Ошибка: Типы несовместимы: получен "TMyClass <System.LongInt>" вместо"TMyClass <System.ShortString>"
```

8.7 Инициализация по умолчанию

При написании методов дженерика, иногда переменная, тип которой неизвестен во время объявления дженерика, должна быть инициализирована. В этом случае можно применить инициализацию по умолчанию *(Default)* *(раздел [4.5 Инициализация переменных (по умолчанию)](75))*. Рассмотрим пример дженерика:

```
type
    generic TTest<T> = class
      procedure Test;
    end;
```

Следующий код будет правильно инициализировать переменную `myt` во время специализации:

```
procedure TTest.Test;
var
    myt: T;
begin
    // При специализации класса будет присвоено правильное
значение по умолчанию
    myt := Default(T);
end;
```

8.8 Несколько слов об области действия

Следует подчеркнуть, что при объявляется дженерика классов все идентификаторы, за исключением заполнителей шаблона, должны быть известны. Это работает в двух случаях. В первом случае, все типы должны быть известны, то есть, идентификатор типа с тем же именем должен существовать. Следующий код выдаст ошибку:

```
{$mode objfpc}
unit myunit;

interface

type
    Generic TMyClass<T> = Class(TObject)
      Procedure DoSomething(A : T; B : TSomeType);
    end;

Type
    TSomeType = Integer;
    TSomeTypeClass = specialize TMyClass<TSomeType>;

Implementation
```

```
Procedure TMyClass.DoSomething(A : T; B : TSomeType);
begin
    // Какой-то код
end;

end.
```

Приведенный выше код приведет к ошибке, так как тип TSomeType не известен, когда анализируется объявление:

```
home: >fpc myunit.pp
myunit.pp(8,47) Error: Identifier not found "TSomeType"
myunit.pp(11,1) Fatal: There were 1 errors compiling module,
stopping

home: >fpc myunit.pp
myunit.pp(8,47) Ошибка: Не найден идентификатор "TSomeType"
myunit.pp(11,1) Неутранимая (ошибка): Была 1 ошибка при
компиляции модуля, остановка
```

Второй случай, когда это видно, состоит в следующем. Пример модуля:

```
{$mode objfpc}
unit mya;

interface

type
    Generic TMyClass<T> = Class(TObject)
      Procedure DoSomething(A : T);
    end;

Implementation

Procedure DoLocalThings;
begin
    Writeln('mya.DoLocalThings');
end;

Procedure TMyClass.DoSomething(A : T);
begin
    DoLocalThings;
end;

end.
```

Компилятор не позволит собрать этот модуль, так как процедура DoLocalThings не видна, когда тип дженерика специализируется:

```
Error: Global Generic template references static symtable
Ошибка: Глобальный дженерик содержит модель со ссылкой на статическую таблицу символов
```

Если модуль изменить так, что описание процедуры DoLocalThings переместиться в секцию интерфейса, модуль будет компилироваться. При использовании этого дженерика в программе:

```
{$mode objfpc}
program myb;

uses mya;

procedure DoLocalThings;
begin
    Writeln('myb.DoLocalThings');
end;

Type
    TB = specialize TMyClass<Integer>;

Var
    B : TB;

begin
    B:=TB.Create;
    B.DoSomething(1);
end.
```

Несмотря на то, что дженерики действуют как макрос, который обрабатывается во время специализации, ссылка на DoLocalThings будет учитываться, когда определяется TMyClass, а не тогда, когда определяется TB. Это объясняет, результаты работы компилятора:

```
home: >fpc -S2 myb.pp
home: >myb
mya.DoLocalThings
```

Такое поведение продиктовано соображениями безопасности и необходимостью:

1. Программист, специализирующий класс, не знает, какие локальные процедуры будут использованы, поэтому он не может случайно "переопределить" их.
2. Программист, специализирующий класс, не знает какие локальные процедуры будут использованы, поэтому он не может выполнить *(реализовать)* их, так как он ещё не знает параметров.
3. Если реализуемые процедуры используются как в приведенном выше примере, то они не могут ссылаться наружу модуля. Они должны быть в другом модуле целиком, и программист не имеет возможности узнать, что он должен включать их до специализации его класса.

8.9 Перегрузка операторов и дженерики

Перегрузка операторов *(глава [Глава 15 Перегрузка операторов](#)[249])* и дженерики тесно связаны между собой. Представьте себе обычный класс, который имеет следующее определение:

```
{$mode objfpc}
unit mya;

interface

type
    Generic TMyClass<T> = Class(TObject)
      Function Add(A,B : T) : T;
    end;

Implementation

Function TMyClass.Add(A,B : T) : T;
begin
    Result:=A+B;
end;

end.
```

Когда компилятор воспроизводит макрос дженерика, сложение должно быть возможным. Для такой специализации:

```
TMyIntegerClass = specialize TMyClass<integer>;
```

что не является проблемой, так как метод Add может быть определен:

```
Procedure TMyIntegerClass.Add(A,B : Integer) : Integer;
begin
    Result:=A+B;
end;
```

Компилятор знает, как сложить два целых числа, так что этот код будет скомпилирован без проблем. Но следующий код:

```
Type
    TComplex = record
      Re,Im : Double;
    end;

Type
    TMyIntegerClass = specialize TMyClass<TComplex>;
```

не скомпилируется, если сложение двух типов `TComplex` не определено. Это может быть сделано с помощью операторов действий над записями:

```
{$modeswitch advancedrecords}
```

```
uses mya;

Type
    TComplex = record
      Re,Im : Double;
      class operator +(a,b : TComplex) : TComplex;
    end;

class operator TComplex.+ (a,b : TComplex) : TComplex;
begin
    Result.re:=A.re+B.re;
    Result.im:=A.im+B.im;
end;

Type
    TMyComplexClass = specialize TMyClass<TComplex>;

begin
    // код
end.
```

В настоящее время, из-за ограничений реализации, они *(операции с записями)* не будут работать с использованием глобального оператора, то есть следующий код пока не работает:

```
uses mya;

Type
    TComplex = record
      Re,Im : Double;
    end;

operator + (a,b : TComplex) : TComplex;
begin
    Result.re:=A.re+B.re;
    Result.im:=A.im+B.im;
end;

Type
    TMyComplexClass = specialize TMyClass<TComplex>;

begin
    // код
end.
```

Поддержка этой конструкции ожидается в будущих версиях Free Pascal.

Глава 9 Расширенные записи

9.1 Описание

Расширенные записи эквивалентны объектам и классам *(меньше)*: они являются записями, которые имеют методы и свойства. Как и объекты, если запись определена в качестве переменной она выделяются в стеке. Им не нужно иметь конструктор. Расширенные записи имеют ряд ограничений в сравнению с объектами и классами, они не поддерживают наследование и полиморфизм. Невозможно создать потомка записи от записи *(хотя она может быть усилена с помощью записей-помошникав (helpers), больше об этом в главе 10.1 Определение*[155] *(о помощников-записях))*.

Почему же тогда введены расширенные записи? Они были введены Delphi 2005 для поддержки одной из особенностей, вносимых NET. Delphi изменил старый стиль TP объектов, и добавил в .NET расширенные записи. Free Pascal стремится быть Delphi совместим, таким образом расширенные записи разрешаются в Free Pascal, только в режиме совместимости с Delphi.

Если расширенные записи необходимы, нужно использовать режим ObjFPC:

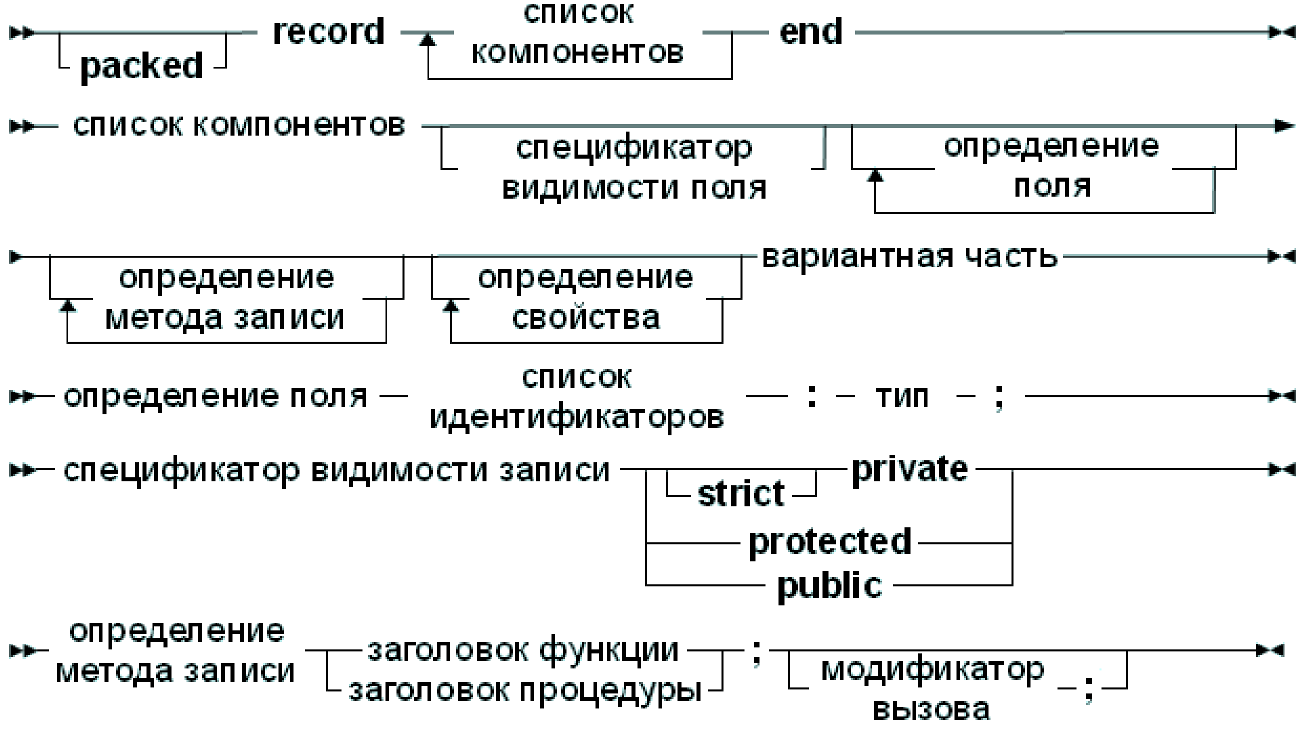
```
{$mode objfpc}
{$modeswitch advancedrecords}
```

Совместимость не является единственной причиной введения расширенных записей. Есть некоторые практические причины для использования методов или свойств в записях:

1. Это больше соответствует объектно-ориентированному подходу к программированию: тип *(расширенные записи)* содержит методы, которые работают с ним.
2. В отличие от процедурного подхода, собрав все операции, которые работают с записью в самой записи, позволяет IDE, чтобы показать доступные методы записи, когда предоставляет варианты завершения кода.

Расширенная запись объявляется сходно с объявлением объекта или класса:

Тип расширенная запись



Ограничения в сравнении с классами или объектами видны из диаграммы синтаксиса:

- Записи ***не наследуются*** *(inheritance)*.
- Раздела `published` ***не существует***.
- Не определены ***конструктор*** и/или ***деструктор***.
- Методы класса *(если их можно так назвать)* требуют ключевого слова `static`.
- Методы ***не могут быть*** *virtual (виртуальными)* или *abstract (абстрактные)* - это следствие того, что нет наследования.

За исключением этого определение напоминает определение класса или объекта.

Ниже приведены несколько примеров допустимых определений расширенных записей:

```
TTest1 = record
  a : integer;
  function Test(aRecurse: Boolean): Integer;
end;

TTest2 = record
private
  A,b : integer;
public
  procedure setA(AValue : integer);
```

```
    property SafeA : Integer Read A Write SetA;
  end;

  TTest3 = packed record
  private
    fA,fb : byte;
    procedure setA(AValue : Integer);
    function geta : integer;
  public
    property A : Integer Read GetA Write SetA;
  end;

  TTest4 = record
   private
     a : Integer;
   protected
     function getp : integer;
   public
     b : string;
     procedure setp (aValue : integer);
     property p : integer read Getp Write SetP;
   public
   case x : integer of
     1 : (Q : string);
     2 : (S : String);
   end;
```

Обратите внимание, что можно указать видимость для членов записи. Это очень полезно, при создании интерфейса к библиотеке C: реальные поля могут быть объявлены скрытыми, а более pascal подобные свойства могут быть открыты, и действовать как фактические поля. Определение записи `TTest3` показывает, что директива `packed` *(упакована)* тоже может использоваться в расширенных записях. Расширенные записи имеют структуру памяти такую-же, как и обычные записи: методы и свойства ***не являются*** частью структуры записей в памяти.

Определение записи `TTest4` в приведенном выше примере показывает, что расширенная запись может иметь и вариантную часть. Как и в случае с обычной записью, вариантная часть должна быть последней. Она ***не может*** содержать методы.

9.2 Энумераторы расширенной записи

Расширенные записи могут иметь энумератор *(перечислитель)*. С этой целью, должна быть определена функция, возвращающая как результат расширенную запись ***enumerator***:

```
type
  TIntArray = array[0..3] of Integer;

  TEnumerator = record
  private
    FIndex: Integer;
    FArray: TIntArray;
    function GetCurrent: Integer;
  public
    function MoveNext: Boolean;
    property Current: Integer read GetCurrent;
  end;

  TMyArray = record
    F: array[0..3] of Integer;
    function GetEnumerator: TEnumerator;
  end;

function TEnumerator.MoveNext: Boolean;
begin
  inc(FIndex);
  Result := FIndex < Length(FArray);
end;

function TEnumerator.GetCurrent: Integer;
begin
  Result := FArray[FIndex];
end;

function TMyArray.GetEnumerator: TEnumerator;
begin
  Result.FArray := F;
  Result.FIndex := -1;
end;
```

После этого определения, следующий код будет скомпилирован и перечислит все элементы в F:

```
var
  Arr: TMyArray;
  I: Integer;
begin
  for I in Arr do WriteLn(I);
end.
```

Тот же эффект может быть достигнут с помощью оператора перечисления:

```
type
```

```
    TIntArray = array[0..3] of Integer;

    TEnumerator = record
    private
      FIndex: Integer;
      FArray: TIntArray;
      function GetCurrent: Integer;
    public
      function MoveNext: Boolean;
      property Current: Integer read GetCurrent;
    end;

    TMyArray = record
      F: array[0..3] of Integer;
    end;

  function TEnumerator.MoveNext: Boolean;
  begin
    inc(FIndex);
    Result := FIndex < Length(FArray);
  end;

  function TEnumerator.GetCurrent: Integer;
  begin
    Result := FArray[FIndex];
  end;

  operator Enumerator(const A: TMyArray): TEnumerator;
  begin
    Result.FArray := A.F;
    Result.FIndex := -1;
  end;
```

Это позволит также исполнить код.

Глава 10 Хелперы для классов, записей и типов

10.1 Определение

Хелперы для классов, записей и типов могут быть использованы для добавления методов к существующим классам, записям или простым типам, не делая наследование класса и без повторного объявления записи.

Для записи или простого типа, хелпер действует так, как будто запись или простой тип является классом, методы объявляются аналогично. Внутри методов, `Self` будет ссылаться на значение записи или простого типа.

Для классов эффект аналогичен вставке метода в таблицу методов класса *(VMT)*. Если объявление хэлпера находится в текущей области кода, то методы и свойства хэлпера могут быть использованы, как если бы они были частью объявления класса или записи.

Синтаксическая схема для хелпера класса, записи или типа представлена ниже:

Тип хелпера

На диаграмме показано, что определение хелпера выглядит очень похоже на обычное определение класса. Он просто объявляет некоторые дополнительные конструкторы, методы, свойства и поля для класса: класс, запись или простой тип, для которого хелпер является расширением, указывается после ключевого слова `for`.

Так как перечислитель для класса получается с помощью обычного метода, хелперы класса также могут быть использованы для переопределения перечислителя для класса.

Как видно из синтаксической диаграммы, можно создать потомков хелпера: хелперы могут образовывать собственную иерархию, что позволяет переопределять методы родительского хелпера. Они имеют спецификаторы видимости, так же, как записи и классы.

Как и в экземпляре класса, идентификатор Self в методе хелпера для класса относится к экземпляру класса *(а не к экземпляру хелпера)*. Для записи Self относится к записи.

Ниже приведен простой хелпер класса для класса TObject, который обеспечивает альтернативный вариант стандартного метода ToString.

```
TObjectHelper = class helper for TObject
    function AsString(const aFormat: String): String;
end;

function TObjectHelper.AsString(const aFormat: String): String;
begin
    Result := Format(aFormat, [ToString]);
end;

var
    o: TObject;
begin
    Writeln(o.AsString('Название объекта: %s'));
end.
```

Замечание:

'helper' является только модификатором для ключевых слов 'class' или 'record'. Это означает, что первый член класса или записи не может быть назван как 'helper'. Член класса или записи можно назвать helper *(хелпером)*, но он не может быть первым, если его не экранировать символом '&', также как и для всех идентификаторов, которые соответствуют ключевым словам.

10.2 Ограничения для классов хелперов

Не все методы или свойства класса могут быть расширены с помощью хелперов. Есть некоторые ограничения на возможности:

- **Деструкторы** или **деструкторы классов** ***не допускаются***.
- **Конструкторы классов** ***не допускаются***.
- Хелперы класса ***не могут*** наследоваться от хелперов **записей** и **расширеных записей**.
- ***Не допускается*** определения полей. Как и полей классов.
- Свойства, ссылающиеся на поле ***не допускаются***. На самом деле это является следствием предыдущего пункта.
- Абстрактные методы ***не допускаются***.
- Виртуальные методы класса ***не могут*** быть перекрыты *(overridden)*. Можно их скрыть дав им то же имя или можно их перегрузить с помощью директивы overload.

- При перегрузке методов класса в хелпере классов, в отличие от обычных процедур или методов, должен явно использоваться спецификатор `overload`. Если перегрузка не используется метод, который расширяется с помощью хелпера, будет скрыт методом хелпера *(как для обычных классов)*.

Ниже показано как в предыдущем примере выполнена модификация метода `ToString` с помощью его перегрузки:

```
TObjectHelper = class helper for TObject
    function ToString(const aFormat: String): String; overload;
end;

function TObjectHelper.ToString(const aFormat: String): String;
begin
    Result := Format(aFormat, [ToString]);
end;

var
    o: TObject;
begin
    Writeln(o.ToString('Название объекта %s'));
end.
```

10.3 Ограничения на хелперы записей

Записи не предлагают те же возможности, что и классы. Это отражается на возможностях при создании хелперов для записей. Список ограничений на хелперы записей таков:

- Хелпер записей ***не может*** быть использован для расширения классов. Следующий код не скомпилируется:

```
TTestHelper = record helper for TObject
end;
```

- Внутри объявления хелпера, методы и поля расширенной записи для определениях свойств, недоступны. Но в реализации они, конечно, доступны. Это означает, что следующий код не будет компилироваться:

```
TTest = record
    Test: Integer;
end;

TTestHelper = record helper for TTest
    property AccessTest: Integer read Test;
end;
```

- Хелперы записей могут получить доступ только к общедоступным *(public)* полям *(в случае использования расширенных записей со спецификаторами видимости)*.

- Наследование хелперов записей допускается только в режиме ObjFPC; В режиме Delphi, не допускается.
- Хелперы записей наследуются только от других хелперов записей, а не от хелперов классов.
- В отличие от хелперов классов, потомок хелпера записи должен расширять один и тот же тип записей.
- В режиме Delphi, нельзя вызвать родительский *(inherited)* метод расширенной записи. Однако в режиме ObjFPC это сделать можно. Следующему коду для компиляции необходим режим ObjFPC:

```
type
    TTest = record
      function Test(aRecurse: Boolean): Integer;
    end;

    TTestHelper = record helper for TTest
      function Test(aRecurse: Boolean): Integer;
    end;

function TTest.Test(aRecurse: Boolean): Integer;
begin
    Result := 1;
end;

function TTestHelper.Test(aRecurse: Boolean): Integer;
begin
    if aRecurse then
      Result := inherited Test(False)
    else
      Result := 2;
end;
```

10.4 Особенности хелперов простых типов

Для простых типов, правила почти такие же, как и для записей, плюс есть некоторые дополнительные требования:

- Поддержку хелперов типов необходимо активировать с помощью переключателя режимов работы `modeswitch`:
  ```
  {$modeswitch typehelpers}
  ```
  Это переключатель включен по умолчанию только в режиме Delphi и DelphiUnicode.
- В режиме Delphi *(и DelphiUnicode)*, для более строгой совместимости с Delphi, вместо хелперов типов должны быть использованы хелперы записей.
- В режимах ObjFPC и MacPas можно использовать хелперы типов, но

должен быть использован modeswitch TypeHelpers.

- Следующие типы не поддерживаются:
  - Все типы файлов *(Text, file of ...)*
  - Процедурные переменные
  - Запрещены типы такие как записи, классы, классы Objective C, классы C++, объекты и интерфейсы. Для классов должны быть использованы хелперы классов. Это означает, что следующий код не скомпилируется:

    ```
    TTestHelper = type helper for TObject
    end;
    ```

    Однако, все другие простые типы поддерживаются.
- Хелперы типов могут реализовать конструкторы.
- Наследование хелперов записей допускается только в режиме ObjFPC; В режиме Delphi, не допускается.
- Хелперы типов могут наследоваться только от других хелперов типов, а не из хелперов классов или записей.
- Потомок хелпера типа должен иметь тот же тип.

Примеры использования:

```
{$mode objpas}
{$modeswitch typehelpers}

type
    TLongIntHelper = type helper for LongInt
      constructor create(AValue: LongInt);
      class procedure Test; static;
      procedure DoPrint;
    end;

constructor TLongIntHelper.create(AValue: LongInt);
begin
    Self:=Avalue;
    DoPrint;
end;

class procedure TLongIntHelper.Test;
begin
    Writeln('Тест');
end;

procedure TLongIntHelper.DoPrint;
begin
    Writeln('Значение:',Self);
end;

var
```

```
    i: LongInt;
begin
    I:=123;
    i.Test;
    $12345678.Test;
    LongInt.Test;
    I:=123;
    i.DoPrint;
    $12345678.DoPrint;
end.
```

10.5 Замечание по видимости и времени жизни хелперов записей и типов

Для классов, время жизни экземпляра класса явно управляется программистом. Поэтому ясно, что означает параметр Self и когда он действителен.

Записи и другие простые типы находятся в стеке, что означает, что они выходят из области видимости, когда функция, процедура или метод, в котором они используются, заканчивается.

В сочетании с тем, что методы хелпера имеют тип, совместимый с методами класса, и они могут быть использованы в качестве обработчиков событий, но это может привести к неожиданным ситуациям: указатель данных *(Data)* в методе хелпера устанавливается на адрес переменной.

Рассмотрим следующий пример:

```
{$mode objfpc}
{$modeswitch typehelpers}
uses
    Classes;

type
    TInt32Helper = type helper for Int32
      procedure Foo(Sender: TObject);
    end;

procedure TInt32Helper.Foo(Sender: TObject);
begin
    Writeln(Self);
end;

var
    i: Int32 = 10;
    m: TNotifyEvent;
begin
```

```
    m := @i.Foo;
    WriteLn('Данные: ',PtrUInt(TMethod(m).Data));
    m(nil);
end.
```

Этот код выдаст что-то вроде *(фактическое значение поля `Data` может отличаться)*:

```
Данные: 6848896
10
```

Переменная `i` все еще находится в области видимости, когда `m` вызывается.

Но изменение кода на

```
{$mode objfpc}
{$modeswitch typehelpers}
uses
    Classes;

type
    TInt32Helper = type helper for Int32
      procedure Foo(Sender: TObject);
    end;

procedure TInt32Helper.Foo(Sender: TObject);
begin
    Writeln(Self);
end;

Function GetHandler  :TNotifyEvent;
var
    i: Int32 = 10;
begin
    Result:=@i.foo;
end;

Var
    m: TNotifyEvent;
begin
    m := GetHandler;
    WriteLn(PtrUInt(TMethod(m).Data));
    m(nil);
end.
```

Выдаст:

```
140727246638796
0
```

Реальное значение будет зависеть от архитектуры, но дело в том, что i больше не находится в области видимости, делая вывод своего значения неопределенным, и возможно даже приводя к нарушениям доступа и сбоям в программе.

10.6 Наследование

Как было отмечено в предыдущем разделе, можно создать потомков хелпера классов. Так как в текущей области может быть использован только последний класс хелпера, необходимо наследовать его от другого, если необходимо использовать методы обоих хелперов. Более подробно об этом в следующем разделе.

Потомок хелпера класса может расширять другой класс, чем его родитель. Ниже приводится допустимый вспомогательный класс для TMyObject:

```
TObjectHelper = class helper for TObject
    procedure SomeMethod;
end;

TMyObject = class(TObject)
end;

TMyObjectHelper = class helper(TObjectHelper) for TMyObject
    procedure SomeOtherMethod;
end;
```

TMyObjectHelper наследуется от TObjectHelper, он не расширяет класс TObject, однако он расширяет только класс TMyObject.

Так как записи не могут наследоваться, очевидно что потомки хелперов записей могут расширять только ту же запись.

Примечание:

Для обеспечения максимальной совместимости с Delphi, создание потомков записей хелперов в режиме Delphi невозможно.

10.7 Использование

После того, как хелпер класса определен, его методы могут быть использованы всякий раз, когда хелпер класса находится в области видимости. Это означает, что, если он определен в отдельном модуле, то этот модуль должен находиться в пункте uses везде, где используются методы хелпера класса.

Рассмотрим следующий модуль:

```
{$mode objfpc}
```

```
{$h+}
unit oha;

interface

Type
    TObjectHelper = class helper for TObject
      function AsString(const aFormat: String): String;
    end;

implementation

uses sysutils;

function TObjectHelper.AsString(const aFormat: String): String;
begin
    Result := Format(aFormat, [ToString]);
end;

end.
```

Тогда будет скомпилирован следующий код:

```
Program Example113;

uses oha;

{ Программа для демонстрации области видимости хелпера. }

Var
    o : TObject;

begin
    O:=TObject.Create;
    Writeln(O.AsString('Объект "O" в виде строки : %s'));
end.
```

Но, если создан второй модуль *(ohb)*:

```
{$mode objfpc}
{$h+}
unit ohb;

interface

Type
    TAObjectHelper = class helper for TObject
      function MemoryLocation: String;
```

```
    end;

implementation

uses sysutils;

function TAObjectHelper.MemoryLocation: String;
begin
    Result := format('%p',[pointer(Self)]);
end;

end.
```

И он добавлен после первого модуля в пункте `uses`:

```
Program Example113;

uses oha,ohb;

{ Программа для демонстрации области видимости хелпера. }

Var
    o : TObject;

begin
    O:=TObject.Create;
    Writeln(O.AsString('Объект "O" в виде строки : %s'));
    Writeln(O.MemoryLocation);
end.
```

Тогда компилятор будет жаловаться, что он не знает метод `'AsString'`. Это происходит потому, что компилятор перестает искать хелпер класса, как только встречает первый хелпер класса. Так как модуль ohb расположен последним в пункте uses, компилятор будет использовать только `TAObjectHelper` как хелпер класса.

Решение состоит в том, чтобы повторно реализовать модуль ohb:

```
{$mode objfpc}
{$h+}
unit ohc;

interface

uses oha;

Type
    TAObjectHelper = class helper(TObjectHelper) for TObject
```

```
      function MemoryLocation: String;
    end;

implementation

uses sysutils;

function TAObjectHelper.MemoryLocation: String;
begin
    Result := format('%p',[pointer(Self)]);
end;

end.
```

А после замены модуля ohb на ohc, пример будет компилироваться и работать, как ожидалось.

Обратите внимание, что включить модуль с хелпером класса один раз в проекте недостаточно. Модуль должен быть включен всякий раз, когда хелпер класса необходим.

Глава 11 Классы Objective-Pascal

11.1 Введение

В Mac OS X используется среда разработки *(frameworks)* с языком программирования Objective-C. Что бы реализовать написанные на этом языке системные интерфейсы, предлагается вариант Object Pascal осуществляемый компилятором Free Pascal, который предлагает туже функциональность что и Objective-C. Этот вариант называется Objective-Pascal.

В зависимости от конструкций Objective-C, компилятор позволяет использовать переключатели режимов. Есть два вида языка Objective-C, различающихся по номеру версии: Objective-C 1.0 и Objective-C 2.0.

Особенности языка Objective-C 1.0 могут быть включены использованием `modeswitch` в исходном файле:

```
{$modeswitch objectivec1}
```

или с помощью параметра командной строки компилятора `-Mobjectivec1`.

Особенности языка Objective-C 2.0 могут быть включены аналогичного с помощью `modeswitch`:

```
{$modeswitch objectivec2}
```

или параметром командной строки `-Mobjectivec2`.

Особенности языка Objective-C 2.0 являются надстройкой над языком Objective-C 1.0, и поэтому переключатель `{$modeswitch objectivec2}` автоматически подразумевает и использование Objective-C 1.0. Программы, использующие функции языка Objective-C 2.0 будет работать только на Mac OS X 10.5 и более поздних версиях.

То что Objective-C использует переключатель режимов, а не синтаксис, означает, что можно использовать комбинации режима с синтаксисом *(fpc, objfpc, tp, delphi, macpas)* Обратите внимание, что директива переключателя `{$Mode }` сбрасывает `{$modeswitch }`, поэтому последняя директива должна быть расположена после него.

11.2 Объявление классов Objective-Pascal

Классы Objective-C или -Pascal объявляются как и классы объектов Pascal, но они используют ключевое слово `objcclass`:

Классы Objective-C

Как можно видеть, синтаксис эквивалентен синтаксису Object Pascal, с некоторыми расширениями.

Для того, чтобы использовать классы Objective-C, существует внешний модификатор: он указывает компилятору, что класс реализован во внешнем объектном файле или библиотеке, и что определение предназначено для импорта. Ниже приведен пример внешнего определения класса Objective-C:

```
NSView = objcclass external(NSResponder)
private
```

```
    _subview  : id;
public
    function initWithFrame(rect : NSRect): id;
      message 'initWithFrame:';
    procedure addSubview(aview: NSView);
      message 'addSubview:';
    procedure setAutoresizingMask(mask: NSUInteger);
      message 'setAutoresizingMask:';
    procedure setAutoresizesSubviews(flag: LongBool);
      message 'setAutoresizesSubviews:';
    procedure drawRect(dirtyRect: NSRect);
      message 'drawRect:';
end;
```

Как можно видеть, определение класса не сильно отличается от определения класса Object Pascal; Присутствует только заметно больше директив сообщений: каждый метод Objective-C или Objective-Pascal должен иметь имя сообщения, связанное с ним. В приведенном выше примере, для определения класса не было указано никакое внешнее имя, а это означает, что идентификатор Pascal используется в качестве имени класса Objective-C. Однако, поскольку Objective-C не столь строг в соглашениях по именованию, псевдоним иногда должен быть указан для имени класса Objective-C, который не подчиняется правилам идентификаторов Pascal.

Следующий пример определяет класс Objective-C, который реализован на языке Pascal:

```
MyView = objcclass(NSView)
public
    data : Integer;
    procedure customMessage(dirtyRect: NSRect);
      message 'customMessage';
    procedure drawRect(dirtyRect: NSRect); override;
end;
```

Отсутствие ключевого слова `external` указывает компилятору, что методы должны быть реализованы позже в исходном файле: он будет рассматриваться так же, как обычный класс объекта Pascal. Обратите внимание на наличие директивы `override`: в Objective-C все методы являются виртуальными. В Object Pascal, переопределение виртуального метода должно быть сделано через директиву `override`. Это было распространено на классы Objective-C: что позволяет компилятору проверить правильность определения.

Если класс не реализует метод протокола *(подробнее об этом в следующем разделе)*, ожидается `message` или `override`: все методы являются виртуальными, и либо запускается новый метод *(или повторно введённый)*, или существующий отменяется. Только в случае метода, который является частью протокола, этот метод может быть определен без `message` или `override`.

Обратите внимание, что объявление класса Objective-C может указывать на родительский класс, а может и не указывать. В Object Pascal, если у класса нет родителя то класс автоматически будет потомком TObject. В Objective-C, это не так: новый класс будет начинать иерархию классов. Тем не менее, Objective-C имеет класс, который содержит полный набор функций общего корневого класса: NSObject, который можно рассматривать как эквивалент TObject в Object Pascal. Он имеет и другие корневые классы, но в целом, классы Objective-Pascal должны быть потомками от NSObject. Новый корневой класс в любом случае, должен реализовать NSObjectProtocol - так же, как это делает сам класс NSObject.

И, наконец, классы Objective-Pascal могут обладать свойствами, но эти свойства могут использоваться только в коде Pascal: компилятор в настоящее время не экспортирует свойства что-бы использовать их в Objective-C.

11.3 Формальное объявление

Object Pascal имеет концепцию forward объявлений. Objective-C развивает эту концепцию немного дальше: он позволяет объявить класс, который определен в другом модуле. Это названо в Objective-Pascal ***'Формальным объявлением'***. Из синтаксической диаграммы, допустимо следующее объявление:

```
MyExternalClass = objcclass external;
```

Это ***формальное объявление***. Оно говорит, что компилируемый класс MyExternalClass является классом Objective-C, но таким образом не объявлены члены класса. Тип может быть использован в остальной части модуля, но его использование ограничено распределением памяти *(в определении поля или параметра метода)* и назначением *(подобно указателю)*.

Как только встречается определение класса, компилятор обеспечит совместимость типов.

Следующий модуль использует формальное объявление:

```
unit ContainerClass;
{$mode objfpc}
{$modeswitch objectivec1}

interface

type
    MyItemClass = objcclass external;

    MyContainerClass = objcclass
      private
       item: MyItemClass;
```

```
      public
       function getItem: MyItemClass; message 'getItem';
    end;

implementation

function MyContainerClass.getItem: MyItemClass;
begin
    result:=item; // Присвоение возможно.
end;

end.
```

Второй модуль может содержать объявление фактического класса:

```
unit ItemClass;

{$mode objfpc}
{$modeswitch objectivec1}

interface

type
    MyItemClass = objcclass(NSObject)
      private
        content : longint;
      public
        function initWithContent(c: longint): MyItemClass;
            message 'initWithContent:';
        function getContent: longint;
            message 'getContent';
  end;

implementation

function MyItemClass.initWithContent(c: longint): MyItemClass;
begin
    content:=c;
    result:=self;
end;

function MyItemClass.getContent: longint;
begin
    result:=content;
end;
```

```
end.
```

Если оба модуля используются в программе, компилятор знает, что это за класс и может проверить правильность некоторых присвоений:

```
Program test;

{$mode objfpc}
{$modeswitch objectivec1}

uses
    ItemClass, ContainerClass;

var
    c: MyContainerClass;
    l: longint;
begin
    c:=MyContainerClass.alloc.init;
    l:=c.getItem.getContent;
end.
```

11.4 Распределение и освобождение экземпляров

Синтаксическая диаграмма классов Objective-C показывает, что понятие конструктора и деструктора не поддерживается в Objective-C. Новые экземпляры создаются в два этапа:

1. Вызываем метод `'Alloc'` *(отправляем сообщение `'Alloc'`)*: Это метод класса `NSObject`, и он возвращает указатель на область памяти для нового экземпляра. Использование `alloc` является соглашением в Objective-C.
2. Отправляем сообщение `'initXXX'`. По соглашению, все классы имеют один или несколько методов `'InitXXX'`, который инициализирует все поля экземпляра. Этот метод возвращает конечный указатель на экземпляр, который может быть и `Nil`.

Следующий код демонстрирует это:

```
var
  obj: NSObject;
begin
  // Первое выделение памяти
  obj := NSObject.alloc;
  // Последующая инициализация.
  obj := obj.init;
  // Всегда проверяйте результат!!
  if (Obj = Nil) then
    // Некоторые ошибки;
```

По соглашению, метод `initXXX` вернет `Nil`, если не удалось

инициализировать некоторые поля, поэтому крайне важна проверка результата функции.

Аналогично, не существует выделенных методов-деструкторов. По соглашению, метод `dealloc` выполняет очистку экземпляров. Этот метод может быть переопределен для выполнения любой необходимой очистки. `Destroy` никогда не должен вызываться напрямую, вместо него должен вызываться метод `release`: Все экземпляры в Objective-C осуществляют подсчет ссылок, и `release` будет вызывать `dealloc`, только если счетчик ссылок достигает нуля.

11.5 Определения протокола

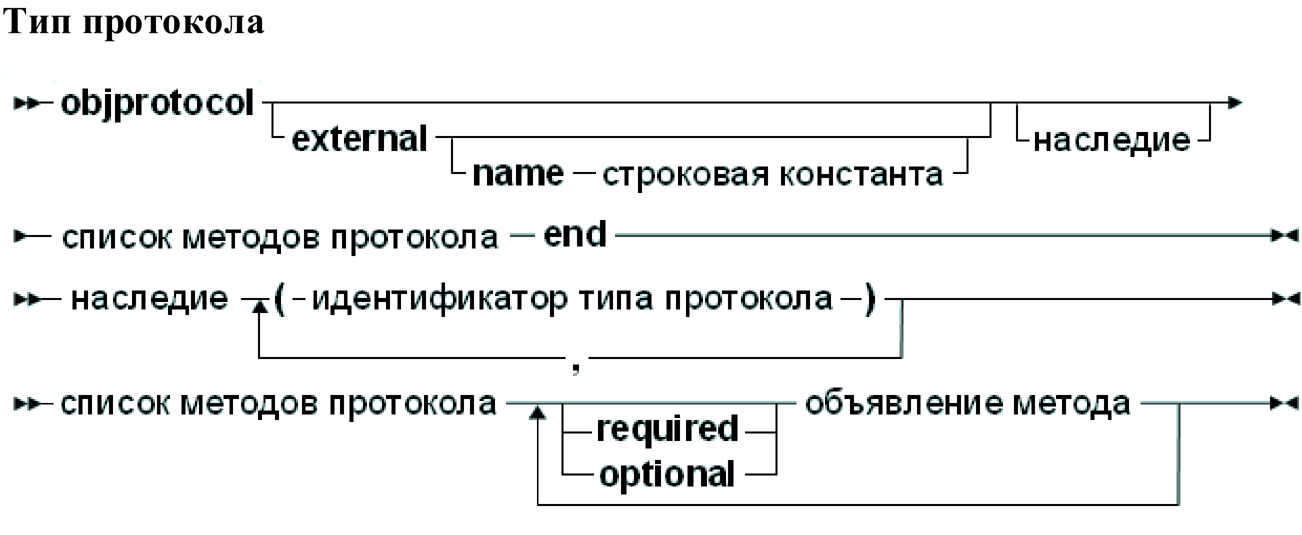
В Objective-C, протоколы играют туже роль что и интерфейсы в Object Pascal, но есть некоторые отличия:

- **Методы протокола могут быть отмечены как необязательные**, то есть класс, реализующий протокол может принять решение не реализовывать эти методы.
- **Протоколы могут наследоваться от нескольких других протоколов.**

В классах Objective-C можно указать, какие протоколы они реализуют при определении класса, это можно увидеть на синтаксической диаграмме классов для Objective-C.

На следующей диаграмме показано, как объявить протокол. Он начинается с ключевого слова `objcprotocol`:

Тип протокола



Как и в случае классов Objective-Pascal, спецификатор external сообщает компилятору, что объявление будет импортировать протокол, определенный в другом месте. К методам Objective-Pascal применяются те же правила, что к методам в объявлении класса. Исключением является то, что должен присутствовать спецификатор `message` *(сообщение)*.

Спецификаторы required и optional не являются обязательными перед последовательностью объявлений метода. Если ничего не указано, то предполагается описатель required. Ниже приводится определение протокола:

```
type
    MyProtocol = objcprotocol
        // Требуется по умолчанию
        procedure aRequiredMethod;
          message 'aRequiredMethod';
      optional
        procedure anOptionalMethodWithPara(para: longint);
          message 'anOptionalMethodWithPara:';
        procedure anotherOptionalMethod;
          message 'anotherOptionalMethod';
      required
        function aSecondRequiredMethod: longint;
          message 'aSecondRequiredMethod';
    end;

    MyClassImplementingProtocol = objcclass(NSObject,
MyProtocol)
      procedure aRequiredMethod;
      procedure anOptionalMethodWithPara(para: longint);
      function aSecondRequiredMethod: longint;
    end;
```

Обратите внимание, что в объявлении класса, опущен спецификатор message. Компилятор *(во время выполнения)* может вывести его из определения протокола.

11.6 Категории

Подобно хелперам классов в Object Pascal, Objective-C имеет ***категории***. Категории позволяют расширить классы без фактического создания потомков этих классов. Тем не менее, категории Objective-C обеспечивают больше функциональных возможностей, чем хелпер класса:

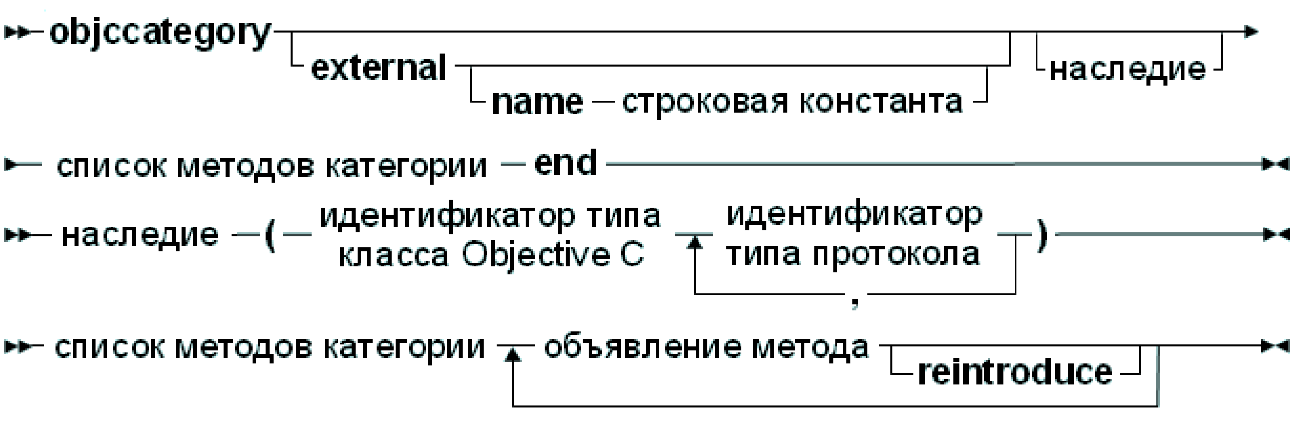
1. В Object Pascal, только один хелпер класса может находиться в области видимости класса. В Objective-C, в области видимости определенного класса могут находиться несколько категорий.
2. В Object Pascal, метод хелпера не может изменить существующий метод, присутствующий в исходном классе *(но он может скрывать метод)*. В Objective-C, категория может заменять существующие методы в другом классе, а не только добавлять новые. Поскольку в Objective-C все методы являются виртуальными, это также означает, что этот метод будет изменен для всех классах, которые наследуются от класса, в котором был заменен

метод *(если они не переопределят его)*.

3. Хелперы Object Pascal не могут быть использованы для добавления интерфейсов к существующим классам. В отличие от этого, категория Objective-C может также реализовывать протоколы.

Определение категории очень напоминает определение протокола класса Objective-C, и начинается с ключевого слова `objccategory`:

Тип категории



Для определенных внешне категорий допустима возможность псевдонима: Objective-C 2.0 позволяет пустое имя категории. Обратите внимание, что модификатор `reintroduce` используется, когда заменяется существующий метод, а не тогда когда добавляется новый метод.

При замене метода и вызове *"`inherited`"* не будет вызываться оригинальный метод класса, но вместо него будет вызвана реализацию метода родительского класса.

Ниже приведен пример определения категории:

```
MyProtocol = objcprotocol
    procedure protocolmethod; message 'protocolmethod';
end;

MyCategory = objccategory(NSObject,MyProtocol)
    function hash: cuint; reintroduce;
    procedure protocolmethod; // от MyProtocol.
    class procedure newmethod; message 'newmethod';
end;
```

Обратите внимание, что это объявление заменяет метод `Hash` каждого класса , который наследуется от `NSObject` *(если класс специально не переопределяет его)*.

11.7 Пространство имён и идентификаторы

В Object Pascal, каждый идентификатор должен быть уникальным в пространство имен модуля. В Objective-C, это может быть не так, и каждый идентификатор типа должен быть уникальным только в своем роде: внутри классов, протоколов, категорий, полей или методов *(имена, скажем, класса и протокола могут совпадать, но должны быть разными имена классов или протоколов)*. Это показано при определении базового протокола и класса Objective-C: и протокол и класс называется NSObject.

При импорте классы и протоколы Objective-C в Objective-Pascal, имена Objective-Pascal типов должны соответствовать правилам Objective-Pascal, и поэтому должны иметь различные имена. Точно так же имена, которые являются допустимыми идентификаторами в Objective-C могут быть зарезервированными словами в Object Pascal. Они также должны быть переименованы при импорте.

Для того, чтобы это стало возможным, модификаторы external и message позволяют задать имена: это имя типа или метода, вместо имён в Objective-C:

```
NSObjectProtocol = objcprotocol external name 'NSObject'
    function _class: pobjc_class; message name 'class';
end;

NSObject = objcclass external (NSObjectProtocol)
    function _class: pobjc_class;
    class function classClass: pobjc_class; message 'class';
end;
```

11.8 Селекторы

Селектор в Objective-C можно рассматривать как эквивалент процедурного типа в Object Pascal.

В отличии с процедурного типа, Objective-C имеет только один тип селектора: SEL. Он определяется в модуле ObjC - который автоматически включается в пункте uses любого модуля, собранного с modeswitch objectivec1.

Чтобы присвоить значение переменной типа SEL, должен использоваться метод objcselector:

```
{$modeswitch objectivec1}
var
    a: SEL;
begin
    a:=objcselector('initiWithWidth:andHeight:');
    a:=objcselector('myMethod');
end.
```

Модуль ObjC содержит методы, для манипулирования и использования селектора.

11.9 Тип id

Тип `id` обособлен в Objective-C/Pascal. Он напоминает, тип `pointer` в Object Pascal, за исключением того, что это настоящий класс. Его присвоения совместимы с экземплярами каждого из типов `objcclass` и `objcprotocol`, в обоих направлениях:

1. Переменные любого типа *(objcclass/objcprotocol)* могут быть присвоены переменной типа `id`.
2. Переменные типа `id` могут быть присвоены переменным любого определенного типа *(objcclass/objcprotocol)*.

Явного приведения типов не требуется для любого из этих присвоений.

Кроме того, любой метод Objective-C , объявленный в `objcclass` или `objccategory`, который находится в области видимости можно вызвать используя идентификатор переменной типа `id`.

Если во время выполнения экземпляр `objcclass`, хранящийся в переменной `id` типа, не отвечает на отправленное сообщение, программа будет завершена с ошибкой времени выполнения: так же, как механизм вызова variants под MS-Windows.

При наличии нескольких методов с тем же Pascal именем, компилятор будет использовать стандартную логику разрешения перегрузки, чтобы выбрать наиболее подходящий метод. Он будет вести себя так, как будто все методы `objcclass/objccategory` в области видимости были объявлены как глобальные `overload` процедуры *(функции)*. Кроме того, компилятор выведет сообщение об ошибке, если не сможет определить какой перегруженный метод необходим для вызова.

В таких случаях список всех методов, которые могут быть использованы для реализации вызова будет печататься в качестве подсказки.

Чтобы устранить эту ошибку вы должны использовать явное приведение типа, чтобы сообщить компилятору, какой метод `objcclass` необходимо вызвать.

11.10 Перечисления в классах Objective-C

Быстрое перечисление *(enumeration)* в Objective-C представляет собой конструкцию, которая позволяет перечислять элементы в контейнере Cocoa в общем виде. Оно осуществляется с использованием цикла `for-in` в Objective-C.

На Objective-Pascal это было переведено с использованием существующего

механизма цикла for-in. Поэтому функция ведет себя одинаково на обоих языках. Обратите внимание, что требуется активизация переключателя режима Objective-C 2.0.

Ниже приведен пример использования цикла for-in:

```
{$mode delphi}
{$modeswitch objectivec2}
uses
    CocoaAll;

var
    arr: NSMutableArray;
    element: NSString;
    pool: NSAutoreleasePool;
    i: longint;
begin
    pool:=NSAutoreleasePool.alloc.init;
    arr := NSMutableArray.arrayWithObjects(
      NSSTR('Один'),
      NSSTR('Два'),
      NSSTR('Три'),
      NSSTR('Четыре'),
      NSSTR('Пять'),
      NSSTR('Шесть'),
      NSSTR('Семь'),
      nil);

    i:=0;
    for element in arr do
      begin
        inc(i);
        if i=2 then continue;
        if i=5 then break;
        if i in [2,5..10] then halt(1);
        NSLog(NSSTR('элемент: %@'),element);
      end;
    pool.release;
end.
```

Глава 12 Выражения

Выражения используются в присваиваниях или проверках. Выражения продуцируют значения определенного типа. Выражения строятся с двумя компонентами: операторы и их операнды. Обычно операторы является бинарными, т.е. требует двух операндов. Бинарные операторы происходят всегда между операндами *(например `X/Y`)*. Иногда оператор унарный, т.е. он требует только один аргумент. Унарный оператор всегда происходит перед операндом *(например `-X`)*.

При использовании нескольких операндов в выражении, учитывается их приоритет *(правила приоритета показаны в таблице 12.1)*.

Таблица 12.1: Приоритет операторов

Оператор	Приоритет	Категория
`Not, @`	Самый высокий *(первый)*	Унарные операторы
`* / div mod and shl shr as << >>`	Второй	Операторы умножения
`+ - or xor`	Третий	Операторы сложения
`= <> < > <= >= in is`	Самый низкий *(последний)*	Операторы отношения

При определении приоритета, используются следующие правила:

1. **В операциях с неравными приоритетом операнды принадлежат к оператору с наивысшим приоритетом.** Например, в выражении `5*3+7`, умножение имеет больший приоритет, чем сложение, так что он выполняется в первую очередь. Результатом будет **22**.
2. **Если в выражении используются круглые скобки, их содержимое вычисляется в первую очередь.** Таким образом, `5*(3+7)` будет равно **50**.

Примечание:

Не гарантируется порядок *(слева-направо)* вычислния выражений одного приоритета. Вы ***не можете*** делать предположений о порядке вычисления выражений. Компилятор будет решать, какие выражение вычислить в первую очередь, основываясь на правилах оптимизации.

Таким образом, в следующем выражении:

```
a := g(3) + f(2);
```

`f(2)`, может быть выполнено, раньше чем `g(3)`. Такое поведение заметно отличается от Delphi или Turbo Pascal.

Если одно выражение ***должно быть*** выполнено перед другим, необходимо разделить выражения используя временные результаты:

```
e1:=g(3);
a:=e1+f(2);
```

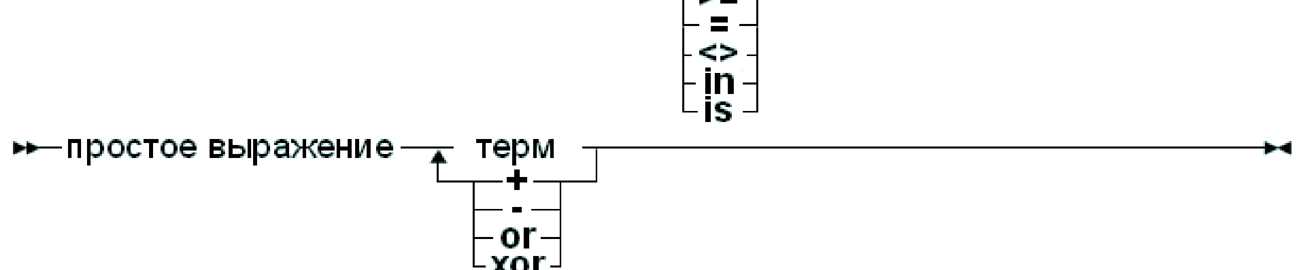
Примечание:

Оператор экспоненты `(**)` доступен для перегрузки, но не определено ни по одному из стандартных типов Pascal *(вещественных и/или целых чисел)*.

12.1 Синтаксис выражений

Выражения состоят из простых выражений в которых используются операторы отношения. В простых выражениях имеется ряд ***термов*** *(что такое* ***терм****, поясняется ниже)*, к которым добавляется оператор дополнения.

Выражения



Ниже приведены допустимые выражения:

```
GraphResult <> grError
(DoItToday=Yes) and (DoItTomorrow=No);
```

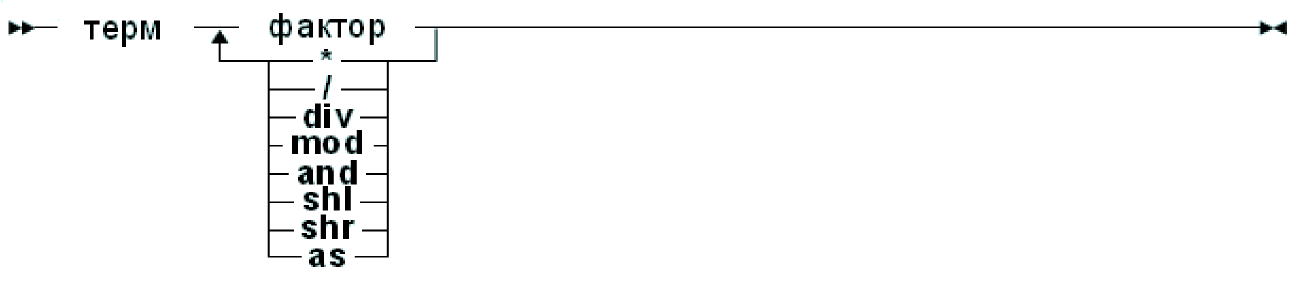
```
DayinWeekend
```

И вот некоторые простые выражения:

```
A + B
-Pi
ToBe or NotToBe
```

Термы состоят из факторов, соединённых операторами умножения.

Термы



Вот некоторые допустимые термы:

```
2 * Pi
A Div B
(DoItToday = Yes) and (DoItTomorrow = No);
```

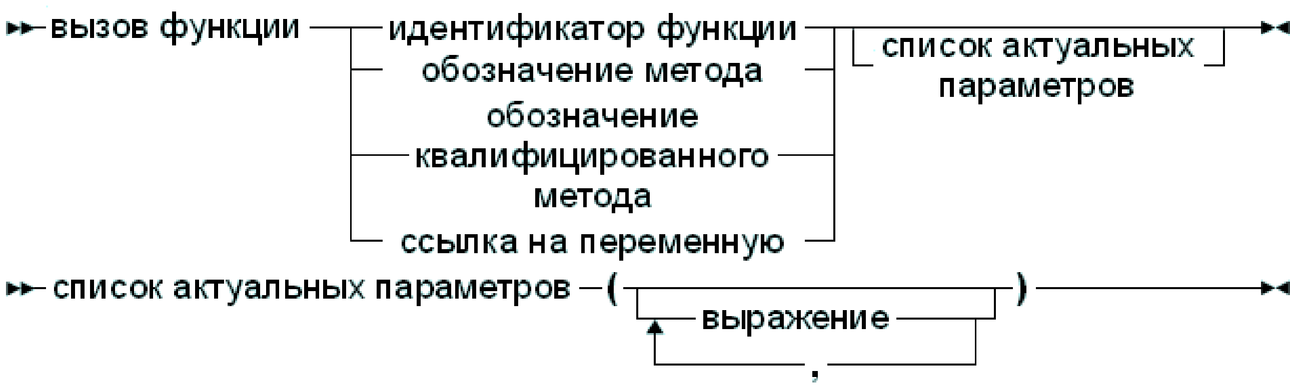
Факторы и другие конструкции:

Факторы

12.2 Вызов функций

Вызов функций являются частью выражений *(хотя, используя расширенный синтаксис можно вызвать функция как оператор)*. Они построены следующим образом:

Вызов функций



Переменная-ссылка может быть *(содержать адрес)* переменной процедурного типа. Имя метода может использоваться только внутри других методов этого объекта. Для использования имени метода вне объекта его надо ***квалифицировать***. Функции вызывается со списком фактических параметров, которые соответствуют списку параметров с которыми объявлена функция. Это значит, что

1. Количество фактических параметров должно быть равно числу объявленных параметров *(если не используются значения параметров по умолчанию)*.
2. Типы параметров должны быть совместимы. Для фактических и объявленных переменных-параметров, типы должны быть такими-же.

Если не будет найдено ни одно совпадение с объявленной функцией, то компилятор выдаст ошибку. Ошибка зависит и от того, перегружена функция или нет *(объявлено несколько функций с одинаковыми именами, но разным списком параметров)*.

Существуют случаи, когда компилятор не выполнит вызов функции в выражении. Это случай, когда в режиме Delphi или Turbo Pascal нужно присвоить значение переменной процедурного типа, как показано в следующем примере:

```
Type
  FuncType = Function : Integer;
Var
  A : Integer;

Function AddOne : Integer;
begin
  A:=A+1;
  AddOne:=A;
end;

  Var F:FuncType;
```

```
    N : Integer;
  begin
    A:=0;
    F:=AddOne; {Присваивание адреса AddOne переменной F, Не
  вызывается функция AddOne}
    N:=AddOne; {N:=1!!}
  end.
```

В приведенном выше листинге, присваивание функции AddOne переменной F не будет вызывать функцию. А присвоение переменной N вызовет функцию AddOne.

Иногда желательно вызвать функцию принудительно, например, в рекурсии. Это может быть сделано путем добавления скобок к имени функции:

```
function rd:char;
var
  c:char;
begin
  read(c);
  if(c='\')then c:=rd();
  rd:=c;
end;

var
  ch:char;
begin
  ch:=rd;
  writeln(ch);
end.
```

Приведённая выше функция будет читать символ и распечатать его. Если введён обратный слеш, то читается второй символ.

Проблема с этим синтаксисом проявляет следующая конструкция:

```
If F = AddOne Then DoSomethingHorrible;
```

Что должен сделать компилятор сравнить адреса F и AddOne, или он должен вызвать обе функции, и сравните результат? В режиме FPC и objfpc он рассматривает процедурную переменную как указатель. Таким образом, компилятор выдаст ошибку несоответствия типов, так как результат вызова функции AddOne целый а F является указателем.

Как же, следует проверить, является ли F указателем на функцию AddOne? Для этого нужно использовать оператор адреса @:

```
If F = @AddOne Then WriteLn ('функции равны');
```

Левая часть логического выражения является адресом. Правая часть также, и поэтому компилятор сравнит два адреса. Для сравнения значений, которые возвращают обе функции, вызвать функцию на которую указывает переменная

F можно добавленем пустого списка параметров:

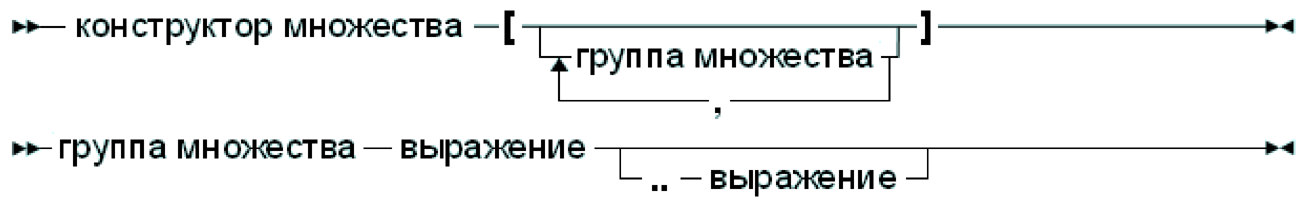
```
If F()=Addone then WriteLn ('функции возвращают одинаковые значения');
```

Заметим, что поведение, как в последнем примере, не совместимо с синтаксисом Delphi. Включение режима Delphi позволит вам использовать синтаксис Delphi.

12.3 Конструкторы множеств

Конструктор множеств состоит из констант и выражений множественного типа. По сути это так-же как определяется тип, только нет идентификатора для определения множества. Конструктор множества состоит из списка выражений разделенных запятыми и заключенный в квадратные скобки.

Конструкторы множеств



Все группы множества и элементы множества должны быть одного и тог-же порядкового типа. Пустое множество обозначается [], и оно может быть присвоено любой переменной типа множество. Группа множества с диапазоном [A..Z] сочетает все значения в диапазоне множества элементов. Допустимы следующие конструкторы множеств:

```
[today,tomorrow]
[Monday..Friday,Sunday]
[2,3*2,6*2,9*2]
['A'..'Z','a'..'z','0'..'9']
```

Примечание:

Если первый элемент диапазона имеет порядковый номер больший чем второй, результирующее множество будет пустым, например множество ['Z'..'A'] означает ***пустое множество***. По этому нужно быть осторожным при определении диапазона.

12.4 Приведение типов значений

Иногда необходимо изменить тип выражения или части выражения, чтобы оно было совместимо по присваиванию. Это делается с помощью приведения

типов значений. Синтаксическая схема приведения типов значений выглядит следующим образом:

Приведение типов

```
►►─приведение значения ─ идентификатор типа ─(─ выражение ─)──────►◄
```

Приведение типов не может быть использована с левой строны от действия присвоения, в качестве преобразования типа значения. Вот некоторые допустимые приведения типов:

```
Byte('A')
Char(48)
boolean(1)
longint(@Buffer)
```

В общем случае, размер типа выражения и размер типа приведённого типа должны быть одинаковыми. Однако, для порядковых типов *(байт, символ, слово, логическое значение, энумератор)* это не так, они могут использоваться как взаимозаменяемые. То есть следующий пример будет работать, хотя размеры не совпадают.

```
Integer('A');
Char(4875);
boolean(100);
Word(@Buffer);
```

Это поведение совместимо с Delphi или Turbo Pascal.

12.5 Приведения типов переменной

Переменная может рассматриваться как единственный фактор в выражении. Поэтому она может быть приведёна к любому типу, при условии, что тип того-же размера, что и исходной тип переменной.

Нельзя приводить переменную целых типов к вещественным и наоборот. Лучше в этом случае пользоваться стандартными функциями для изменения типа.

Обратите внимание, что переменные совместимых типов должны быть с обеих сторон от присваивания, т.е. следующее приведение типов допустимо:

```
Var
  C : Char;
  B : Byte;
begin
  B:=Byte(C);
  Char(B):=C;
```

```
end;
```

Переменные содержащие указатель *(pointer)* совместимы с процедурными типами, но не указателями на метод *(класса)*.

Приведённая переменная - переменная указанного типа, это означает, что можно использовать ***квалификатор***:

```
Type
  TWordRec = Packed Record
    L,H : Byte;
  end;

Var
  P : Pointer;
  W : Word;
  S : String;
begin
  TWordRec(W).L:=$FF;
  TWordRec(W).H:=0;
  S:=TObject(P).ClassName;
```

12.6 Приведение невыровненых типов

Приведение типов может быть `Unaligned` *(невыровненым)* в выражении или переменной. Это не реальное приведение, а скорее указание компилятору, что выражение может быть смещено *(то есть не выровнено по адресам памяти)*. Некоторые процессоры не обеспечивают прямой доступ к невыровненым структурам данных, и поэтому должен иметь доступ к данным побайтно.

Ключевое слово `unaligned` при приведении типа выражения сигнализирует компилятору, что он должен получить доступ к данным побайтно.

Обратите внимание, что компилятор предполагает, что все поля/элементы упакованных структур данных ***невыровненны***.

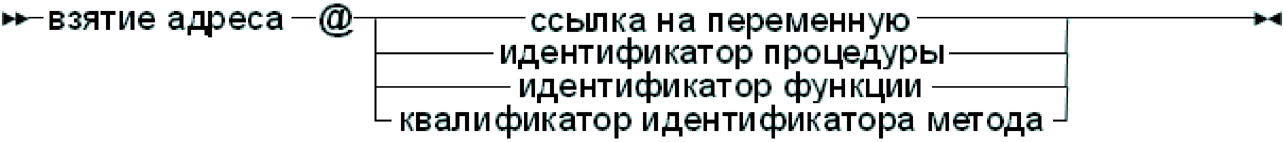
Пример:

```
program me;

Var
  A : packed Array[1..20] of Byte;
  I : LongInt;
begin
  For I:=1 to 20 do
    A[I]:=I;
  I:=PInteger(Unaligned(@A[13]))^;
end.
```

12.7 Оператор @

Оператор взятия адреса *(@)* возвращает адрес переменной, процедуры или функции. Он используется следующим образом:

Действие *(фактор)* взятие адреса



Если переключатель $T включен, оператор @ возвращает типизированный указатель. Если переключатель $T выключен, то оператор взятия адреса возвращает нетипизированный указатель, который является совместим по присваиванию со всеми типами указателями. Указатель типа ^T, где T является переменной на которую ссылается указатель. Следующий пример будет скомпилирован

```
Program tcast;
{$T-} { @ возвращает нетипизированный указатель}

Type
  art = Array[1..100] of byte;
Var
  Buffer : longint;
  PLargeBuffer : ^art;
begin
 PLargeBuffer := @Buffer;
end.
```

Изменение {$T-} на {$T+} изменит поведение компилятора. При компиляции будет ошибка несоответствие типов.

По умолчанию, оператор возвращает адрес как нетипизированный указатель: применяя оператор взятия адреса к идентификатору метода функции или процедуры даст указатель на точку входа этого метода. Результат возвращается как нетипизированный указатель.

Это означает, что следующий код будет работать:

```
Procedure MyProc;
begin
end;

Var
  P : PChar;
begin
```

```
    P:=@MyProc;
  end;
```

Если значение должно быть присвоено переменной процедурного типа, должен быть использован оператор взятия адреса.Такое поведение можно изменить с помощью переключателей -Mtp или -MDelphi, которые делают синтаксис более совместимым с Delphi или Turbo Pascal.

12.8 Операторы

Операторы классифицированы в соответствии с типом выражения с которыми они работают. Мы будем обсуждать их по этому типу.

12.8.1 Арифметические операторы

Арифметические операторы применяются в арифметических операциях, то есть в выражениях, содержащих целые или вещественные числа. Есть два вида операторов: Бинарные и унарные арифметические операторы. Бинарные операторы перечислены в таблице (12.2), унарные операторы перечислены в таблице (12.3).

Таблица 12.2: Бинарные арифметические операции

Оператор	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление
Div	Целочисленное деление
Mod	Остаток от деления

Все операторы принимают в качестве операндов вещественные и целые выражения, кроме Div и Mod, которые принимают только целые выражения .

Для бинарных операторов, если ***оба операнда*** являются ***целыми*** выражениями типа ***результат*** будет ***целым***. Если ***один из операндов*** будет выражением ***вещественного*** типа, то ***результат вещественный***.

Результат операции деления (/) всегда вещественный.

Таблица 12.3: Унарные арифметические операторы

Оператор	Операция
+	Идентичность знака
–	Изменение знака

Для унарных операторов, тип результата всегда равен типу выражения. Если второй аргумент равен нулю, оператор деления *(/)* и остаток от деления *(Mod)* во время выполнения может привести к ошибкам программы, .

Знак результата оператора Mod будет такой же, как знак операнда левой части. Оператор Mod эквивалентен следующей операции:

```
I mod J = I - (I div J) * J
```

Но он *(оператор)* выполняется быстрее, чем выражение - заменитель.

12.8.2 Поразрядные логические операторы

Логические операторы применяются к отдельным битам порядковых выражений. Логические операторы требуют операнды целого типа, и результат тоже целый. Все возможные логические операторы перечислены в таблице (12.4).

Таблица 12.4: Логические операторы

Оператор	Операция
not	Побитовая инверсной *(унарный)*
and	Побитовая И
or	Побитовая ИЛИ
xor	Побитовое XOR *(исключающее ИЛИ)*
shl	Поразрядный сдвиг влево
shr	Поразрядный сдвиг вправо
<<	Поразрядный сдвиг влево *(то же, что и SHL)*
>>	Поразрядный сдвиг вправо *(то же, что и SHR)*

Допустимы следующие логические выражения:

```
A shr 1   { то же что и A div 2, но быстрее}
Not 1     { эквивалентно -2 }
Not 0     { эквивалентно -1 }
Not -1    { эквивалентно 0   }
B shl 2   { то же что и B * 4 для целых чисел }
```

```
1 or 2    { эквивалентно 3 }
3 xor 1   { эквивалентно 2 }
```

12.8.3 Логические операторы (однобитовые)

Логические операторы можно рассматривать как логические операции с одним битом. Поэтому операции SHL и SHR безсмыслены. ***Логические операторы*** могут иметь операнды только логического типа, значение результата всегда логического типа *(один бит)*. Все возможные операторы перечислены в таблице (12.5)

Таблица 12.5: Логические операторы

Оператор	Операция
not	Логическое отрицание *(унарная)*
and	Логическое И
or	Логическое ИЛИ
xor	Логическое XOR

Примечание:

По умолчанию, логические выражения вычисляются с оценкой ***короткого замыкания***. Это означает, что вычисление прекращается как только становится понятен результат, и возвращается результат. Например, в следующем выражении:

```
B := True or MaybeTrue;
```

Компилятор не будет вычислять на значении MaybeTrue, так как очевидно, что выражение ***всегда*** будет истинным.

В результате этой стратегии, если MaybeTrue является функцией, ***она не будет вызвана***! *(Это может привести к непонятному поведению. Например при использовании в свойствах)*

12.8.4 Строковый оператор

Существует только один строковый оператор: + *(конкатенации)*. Он соединяет содержимое двух строк *(или символов)*. Нельзя использовать + для конкатенации строк оканчивающихся нулём *(PChar)*. Допустимы следующие операции со строками:

```
'Thisis'+'VERY'+'easy!'
Dirname+'\'
```

Недопустима:

```
Var
  Dirname : PChar;
...
Dirname := Dirname + '\';
```

Поскольку Dirname является строкой с завершающим нулем.

Обратите внимание, что если все строки в выражении являются короткими строками, результирующая строка также короткая. Таким образом, может произойти обрезание: нет автоматического перехода к AnsiString.

Если все строки в строковом выражении имеют тип AnsiString, то результат будет иметь тип AnsiString.

Если выражение смешивает типы AnsiString и ShortString, результатом будет тип AnsiString.

Значение переключателя {$H} может быть использован для управления типом строковых констант; по умолчанию они представляют собой короткие строки *(и ограничивается до **255** символов)*.

12.8.5 Операторы действий над множествами

Над множествами могут быть выполнены следующие операции: ***объединение, разность, симметричная разность, включение и пересечение***. Элементы могут быть добавлены или удалены в/из множества операторами Include или Exclude. Операторы действий с множествами, перечислены в таблице (12.6).

Таблица 12.6: Операторы действий над множествами

Оператор	Действие
+	Объединение
-	Разность
*	Пересечение
><	Симметричная разность
<=	Содержит
include	Включает элемент в множество
exclude	Исключает элемент из множества
in	Проверить множество на наличие элемента

Тип множеств, участвующих в операциях над множествами должен быть

одинаковым, иначе будет сгенерирован ошибка компилятора.

Следующая программа показывает примеры некоторых операций над множествами:

```
Type
  Day = (mon,tue,wed,thu,fri,sat,sun);
  Days = set of Day;

Procedure PrintDays(W : Days);
Const
  DayNames : array [Day] of String[3] =
('mon','tue','wed','thu','fri','sat','sun');
Var
  D : Day;
  S : String;
begin
  S:='';
  For D:=Mon to Sun do
    if D in W then
      begin
      If (S<>'') then S:=S+',';
      S:=S+DayNames[D];
      end;
  Writeln('[',S,']');
end;

Var
  W : Days;
begin
   W:=[mon,tue]+[wed,thu,fri];      // результат операции [mon,
tue,wed,thu,fri]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed];          // результат операции [mon,
tue]
   PrintDays(W);
   W:=[mon,tue,wed]-[wed,thu];      // результат операции [mon,
tue]
   PrintDays(W);
   W:=[mon,tue,wed]*[wed,thu,fri];  // результат операции [wed]
   PrintDays(W);
   W:=[mon,tue,wed]><[wed,thu,fri]; // результат операции [mon,
tue,thu,fri]
   PrintDays(W);
end.
```

Как можно заметить, ***объединение*** эквивалентна ***бинарному ИЛИ***, ***пересечение***

эквивалентна ***бинарному И***, ***симметричная разность*** эквивалентна операции ***XOR***.

Операции Include и Exclude эквивалентны + *(объединение)* или - *(разность)* с множеством из одного элемента. Таким образом,

```
Include(W,wed);
```

эквивалентно

```
W:=W+[wed];
```

и

```
Exclude(W,wed);
```

эквивалентно

```
W:=W-[wed];
```

Операция In как результат возвратит True, если левый операнд *(элемент)*, входит во множество указанное как правый операнд, иначе будет False.

12.8.6 Операторы отношения

Операторы отношения приведены в таблице (12.7)

Таблица 12.7: Операторы сравнения

Оператор	Действие
=	Равно
<>	Не равно
<	Строго меньше
>	Строго больше
<=	Меньше или равно
>=	Больше или равно
in	Элемент входит во множество

Как правил, левый и правый операнды должны быть одного и того же типа. Есть некоторые исключения, когда компилятор может обрабатывать смешанные выражения:

1. Целые *(Integer)* и вещественный *(real)* типы могут быть смешаны в выражениях отношения.
2. Если оператор перегружен и перегруженная версия существует, типы аргументов совпадают с типами в выражении.
3. Могут быть смешаны типы ShortString, AnsiString и WideString .

Сравнение строк производится на основе представления кодов их символов.

При сравнении указателей, сравниваются адреса, на которые они указывают. Это верно и для указателей типа PChar. Для сравнения строки и PChar *(строки оканчивающейся нулём)*, необходимо использовать функцию StrComp из модуля strings. Оператор in возвращает True, если левый операнд *(который должен иметь один и тот же тип, что и элементы множества, а эти элементы должны быть в диапазоне **0..255**)* является элементом множества, которое является правым операндом, иначе возвращается False.

12.8.7 Операторы действий над классами

Операторы действий с классами немного отличаются от операторов перечисленных выше, в том смысле, что они могут быть использованы только выражениях с классами. Есть только два оператора действий над классами, что можно видеть в таблице (12.8).

Таблица 12.8: Операторы действий над Классами

Оператор	Действие
is	Проверки типа класса
as	Приведение типов

Выражение, содержащий оператор is возвращает результат логического типа. Оператор is может быть использован только со ссылкой на класс или экземпляра класса. Использование этого оператора выглядит следующим образом:

```
Object is Class
```

Это выражение полностью эквивалентно

```
Object.InheritsFrom(Class)
```

Если Object является Nil, будут возвращено False.

Ниже приведен пример:

```
Var
  A : TObject;
  B : TClass;
begin
  if A is TComponent then ;
  If A is B then;
end;
```

Как оператор as выполняет приведение типа. Результат этого выражения, имеет тип класса:

```
Object as Class
```

Это эквивалентно следующему:

```
If Object=Nil then
    Result:=Nil
  else if Object is Class then
    Result:=Class(Object)
  else
    Raise Exception.Create(SErrInvalidTypeCast);
```

Обратите внимание , что если объект равен `nil`, то, оператор `as` не генерирует исключение.

Ниже приведены некоторые примеры использования `as` оператора:

```
Var
  C : TComponent;
  O : TObject;
begin
  (C as TEdit).Text:='Some text';
  C:=O as TComponent;
end;
```

В качестве операторов `as` и `is` также работают на COM - интерфейсы.

Они могут быть использованы для проверки реализует ли интерфейс, применение `is` и `as` показано в следующем примере:

```
{$mode objfpc}

uses
  SysUtils;

type
  IMyInterface1 = interface
    ['{DD70E7BB-51E8-45C3-8CE8-5F5188E19255}']
    procedure Bar;
  end;

  IMyInterface2 = interface
    ['{7E5B86C4-4BC5-40E6-A0DF-D27DBF77BCA0}']
    procedure Foo;
  end;

  TMyObject = class(TInterfacedObject, IMyInterface1,
IMyInterface2)
    procedure Bar;
    procedure Foo;
  end;
```

```
procedure TMyObject.Bar;
begin

end;

procedure TMyObject.Foo;
begin

end;

var
  i: IMyInterface1;
begin
  i := TMyObject.Create;
  i.Bar;
  Writeln(BoolToStr(i is IMyInterface2, True)); // Напечатает
true
  Writeln(BoolToStr(i is IDispatch, True)); // Напечатает false

  (i as IMyInterface2).Foo;
end.
```

Кроме того, оператор `is` может быть использован для проверки, что класс реализует интерфейс, а оператор `as` приводит пременную типа интерфейс обратно к классу:

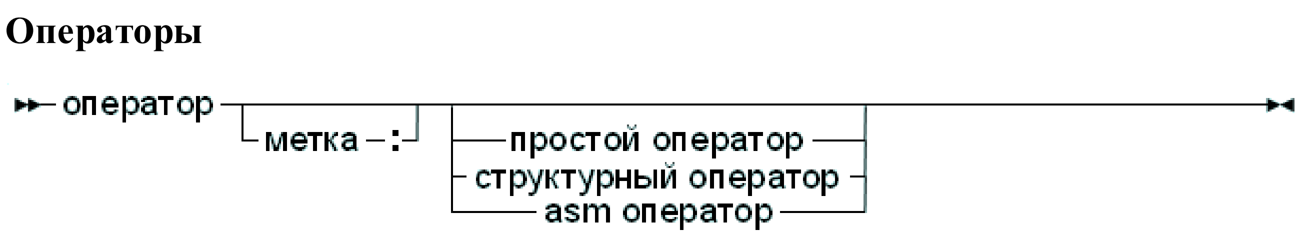
```
{$mode objfpc}
var
  i: IMyInterface;
begin
  i := TMyObject.Create;
  Writeln(BoolToStr(i is TMyObject,True)); // Напечатает true
  Writeln(BoolToStr(i is TObject,True)); // Напечатает true
  Writeln(BoolToStr(i is TAggregatedObject,True)); //
Напечатает false
  (i as TMyObject).Foo;
end.
```

Хотя интерфейсы должны быть COM-интерфейсами, но приведённый обратно к классу будет работать только, если интерфейс наследуется от класса Object Pascal. Он не будет работать для интерфейсов, полученных из системы с помощью COM.

Глава 13 Операторы

Сердце каждого алгоритма - это предпринимаемые действия. Эти действия определяются операторами программы или модуля. Возле каждого оператора может быть метка, для перехода *(в определенных пределах)* на него *(этот оператор)* с помощью др. оператора *(Goto)*. Это можно увидеть на следующей синтаксической диаграмме:

Операторы

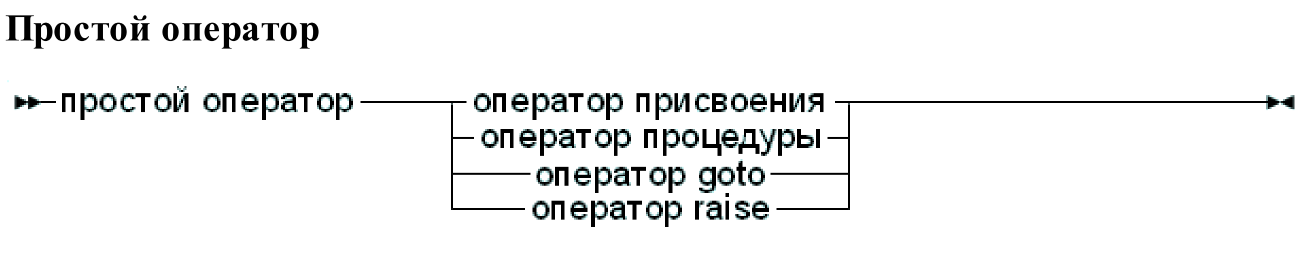


Метка может быть допустимым идентификатором или целым числом.

13.1 Простые операторы

Простой оператор не может быть разложен на отдельные операторы. Есть четыре *(вида)* простых оператора:

Простой оператор

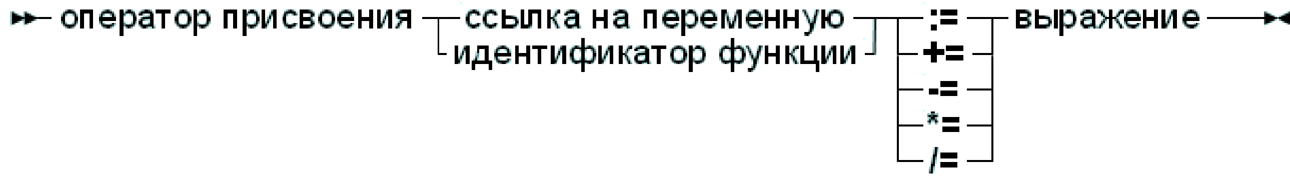


Про ***оператор raise***, будет объяснено в главе 17.1 Оператор Raise [273].

13.1.1 Оператор присвоения

Оператор присвоения - присваивает переменной значение выражения или другой переменной, заменяя предыдущее значение которое имеет переменная:

Присвоение



В дополнение к стандартному Pascal оператор присваивания *(:=)*, который просто заменяет значение переменной на значение результата выражения справа от оператора присваивания, Free Pascal поддерживает и некоторые конструкции в C-стиле. Все поддерживаемые конструкции приведены в таблице (13.1).

Таблица 13.1: Во Free Pascal разрешены конструкции C

Присваивание	Результат
`a += b`	Добавляет `b` к `a` и сохраняет результат в `a`.
`a -= b`	Вычитает `b` из `a` и сохраняет результат в `a`.
`a *= b`	Умножает `a` на `b` и сохраняет результат в `a`
`a /= b`	Делит `a` на `b` и сохраняет результат в `a`.

Для того чтобы эти конструкции работали, должен быть установлен переключатель `-Sc` в командной строке.

Примечание:

Эти конструкции применяются для удобства присвоения, они не генерируют разный код. Вот некоторые примеры допустимых операторов присваивания:

```
X := X+Y;
X += Y;       { То же, что и X:= X + Y, необходим
переключатель -Sc в командной строке}
X /= 2;       { То же, что и X:= X / 2, необходим
переключатель -Sc в командной строке}
Done := False;
Weather := Good;
MyPi := 4* Tan(1);
```

Имейте в виду, что разыменования типизированного указателя указывающего на заданный тип указывает на результат *(область в памяти, где расположено значение)*, справедливы следующее присвоения:

```
Var
```

```
   L : ^Longint;
   P : PPChar;
begin
   L^:=3;
   P^^:='A';
```

Обратите внимание на двойное разыменования во втором присваивании.

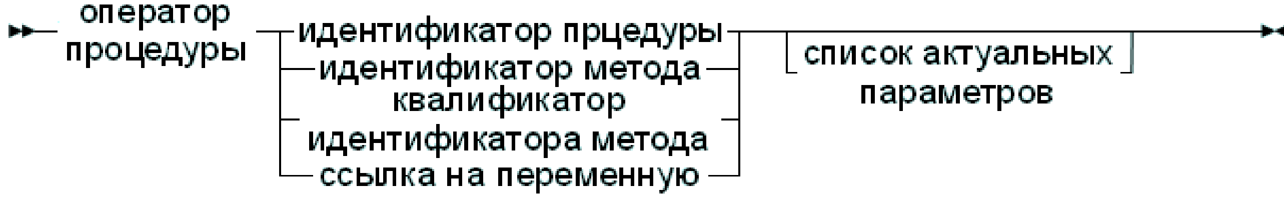
13.1.2 Оператор вызова процедуры

Оператор вызова процедуры обращается к подпрограммам *(вызывает процедуры и функции (расширенный вызов))*. Существуют различные возможности для вызовов процедур:

- Обычный вызов процедуры.
- Вызов метода Объекта ***(поное** (с указанным квалификатором)* ***или нет)***.
- Вызов процедуры через переменную процедурного типа.

Все эти вызовы показаны на следующей диаграмме:

Оператор вызова процедуры



Компилятор Free Pascal будет искать процедуру с тем же именем, как указано в ***операторе вызова процедуры*, а также** со списком объявленных параметров, который быдет соответствовать списку фактических параметров. Допустимыми являются следующие операторы вызова процедуры:

```
Usage;
WriteLn('Pascal является простым языком!');
Doit();
```

Примечание:

При поиске функции, которая совпадает со списком параметров вызова, типы параметров должны быть **совместимы по присваиванию** для параметров значений и констант, и должны **точно совпадать** для параметров, передаваемых по ссылке.

13.1.3 Оператор Goto

Free Pascal поддерживает оператор Goto *(оператор перехода)*. Его

синтаксический прототип

Оператор Goto

►► оператор goto — goto — метка ———————————————— ►◄

При использовании оператора Goto, нужно иметь в виду следующие:

1. Метка перехода *(куда переходить)* должна быть определена в том же блоке, что и оператор Goto.
2. Переходы из цикла и в цикл могут иметь странные последствия.
3. Для того, чтобы использовать оператор Goto, необходимо использовать переключатель компилятора -SG или директиву {$GOTO ON}.

Примечание:

В стандарте ISO, режиме macpas или с modeswitch "nonlocalgoto", компилятор также позволяет нелокальные переходы.

Использование оператора Goto считаются плохой практикой и его следует избегать, насколько это возможно. Всегда можно заменить конструкцию с оператором goto, на конструкцию без goto, хотя эта конструкция может быть не столь же понятна. Например, следующая конструкция с оператором goto разрешена.

```
label
  jumpto;
...
Jumpto :
  Statement;
...
Goto jumpto;
...
```

13.2 Структурные операторы

Структурированные операторы могут быть разбиты на более мелкие простые операторы, которые выполняются **циклически**, **по условию** или **последовательно**:

Структурные операторы

Условный оператор может быть двух вариантов:

Условный оператор

Оператор цикла может быть трёх вариантов:

Оператор цикла

Следующие разделы посвящены каждому из этих операторов.

13.2.1 Составной оператор

Составные операторы представляют собой группу операторов, разделенных точкой с запятой, и заключенных в ключевые слова `begin` и `end`. Последний оператор - перед ключевым словом `end` - не нуждается в точке с запятой, хотя это допустимо. Составной оператор представляет собой способ группировки операторов, выполняемых последовательно. Они рассматриваются как один оператор в тех случаях, когда синтаксис Pascal ожидает один оператор, например, в операторе `If...Then...Else`.

Составной оператор

```
►► состанной оператор — begin ┬ оператор ┬ end ——————————►◄
                              └─── ; ────┘
```

13.2.2 Оператор Case

Free Pascal поддерживает оператор `case` *(выбора)*. Его синтаксическая схема

Оператор Выбора *(Case)*

Константы, в различных частях ***case*** *(альтернатив)* должны быть известны во время компиляции, и могут быть следующих типов: перечисления, порядкового типа *(за исключением логического (`boolean`))*, и символьного или строкового типа. Выражение *(после служебного слова `case`)* должно быть такого типа, или произойдет ошибка компиляции. Все индексы ***case*** должны иметь одинаковый тип.

Компилятор подсчитает выражение *(после служебного слова `case`)*. Если одно из значений индекса ***case*** совпадает со значением выражения, выполняется оператор следующий за константой. После этого программа продолжает выполняться после `end` оператора `case`.

Если ни один из ***case*** индексов не соответствует значению выражения, выполняется список инструкций после ключевых слов `else` или `otherwise`. Это может быть и пустой список операторов. Если отсутствует `else` часть и ни один ***case*** индекс не соответствует значению выражения, программа продолжает выполняться после `end` оператора `case`.

В операторе `case` могут быть составные операторы (то есть блоки `begin..end`).

Примечание:

В отличие от Turbo Pascal, дублирование ***case*** индекса не допускаются в Free Pascal, поэтому следующий код вызовет ошибку компиляции:

```
Var i : integer;
...
Case i of
 3 : DoSomething;
 1..5 : DoSomethingElse;
end;
```

Компилятор выдаст ошибку *Duplicate case label (Продублирован case индекс)* при компиляции этого, потому что **3** также появляется *(неявно)* в диапазоне `1..5`. Это дублирует синтаксис Delphi.

Free Pascal позволяет использовать строки в качестве индексов ***case***, и в этом случае `case` переменная *(или выражение)* также должна быть строкой. При использовании переменной строкового типа в `case` и различные индексы сравниваются с учётом регистра.

```
Case lowercase(OS) of
 'windows',
 'dos'    : WriteLn ('Платформа Microsoft');
 'macos',
 'darwin' : Writeln('Платформа Apple');
 'linux',
 'freebsd',
 'netbsd' : Writeln('Общественная платформа');
else
  WriteLn ('Другая платформа');
end;
```

`Case` со строками эквивалентна последовательности операторов `if then else`, при этом никаких оптимизаций не выполняются.

Тем не менее, диапазоны допустимы, и эквивалентны

```
if (value >= beginrange) and (value <= endrange) then
  begin
  end;
```

13.2.3 Оператор If..then..else

Синтаксический шаблон `If..then..else..`

Оператор *if then*

```
►►─оператор if─if─выражение─then─оператор─┬──────────────────┬──►◄
                                          └─else─оператор─┘
```

Выражение между ключевыми словами if и then должно иметь результат типа Boolean. Если выражение имеет значение True, то выполняется оператор *(простой или составной)* следующий за ключевым словом then.

Если выражение имеет значение False, то тогда выполняется оператор *(простой или составной)* после ключевого слова else, если он присутствует.

Некоторые моменты, нужно учесть:

- Помните о том, что логическое выражение по умолчанию будет вычислено по схеме короткого замыкания[189], это значит, что вычисление будет остановлен в точке, где результат уже определён.
- Кроме того, перед ключевым словом else, точка с запятой *(;)* не допускается, но все операторы могут быть вложенными операторами. Во вложенной конструкции if..then..else, может возникнуть некоторая двусмысленность, какое else с каким if в паре.
- Правило гласит, что ключевое слово else соответствует первому же ключевым словом if *(в обратном порядке)*, уже не соответствует ключевому слову else.

Например:

```
If exp1 Then
  If exp2 then
    Stat1
else
  stat2;
```

Несмотря на внешний вид, этот код синтаксически эквивалентен

```
If exp1 Then
   begin
     If exp2 then
        Stat1
     else
        stat2
   end;
```

а не

```
{ НЕ ЭКВИВАЛЕНТНО }
If exp1 Then
   begin
     If exp2 then
        Stat1
```

```
    end
else
    stat2;
```

Если необходима последняя конструкция, то должны присутствовать ключевые слова begin и end. Если сомневаетесь, их лучше добавить.

Следующий оператор допустим:

```
If Today in [Monday..Friday] then
     WriteLn ('Должно быть работать тяжелее')
else
     WriteLn ('Возьмите выходной день.');
```

13.2.4 Оператор For..to/downto..do

Free Pascal поддерживает построение циклов for. Цикл for используется в случае, если нужно что-то вычислять фиксированное число раз. Синтаксический прототип выглядит следующим образом:

Оператор *for*

```
►►─оператор for ─for─параметр цикла ─:= ─начальное значения ┬─to─────┬─►
                                                            └─downto─┘
►─конечное значение ─do ─оператор ──────────────────────────────────────►◄
►►─параметр цикла ─ идентификатор переменной ───────────────────────────►◄
►►─начальное значение ─ выражение ──────────────────────────────────────►◄
►►─конечное значение ── выражение ──────────────────────────────────────►◄
```

Этот оператор *(for)* может быть составным. Когда компилятор встречает оператор for, управляющая переменная инициализируется начальным значением, и сравнивается с конечным значением.Что будет дальше, зависит от того, используется ли to или downto:

- В случае если используется to, и начальное значение **больше** конечного, то оператор не выполнится ни разу.
- В случае если используется downto, и начальное значение **меньше** конечного, то оператор не выполнится ни разу.

После этой проверки выполняется оператор после do. После выполнения оператора, управляющая переменная увеличивается или уменьшается на единицу, в зависимости от того, используется to или downto. Переменная управления должна быть порядкового типа, другие типы не могут быть использованы в качестве счетчиков цикла.

Примечание:

Free Pascal всегда вычисляет верхнюю границу перед инициализацией переменной счетчика начальным значением.

Примечание:

Не допускается изменять *(т.е. присваивать)* значение переменной цикла внутри цикла.

Приведенные ниже циклы допустимы:

```
For Day := Monday to Friday do Work;
For I := 100 downto 1 do WriteLn ('Обратный счёт: ',i);
For I := 1 to 7*dwarfs do KissDwarf(i);
```

Код ниже будет генерировать ошибку:

```
For I:=0 to 100 do
  begin
  DoSomething;
  I:=I*2;
  end;
```

потому что переменной цикла **нельзя** присваивать значение внутри цикла.

Если оператор *(после do)* является составным, то могут быть использованы системные процедуры `Break` и `Continue` для выхода из цикла или начала новой итерации оператора `for`. Обратите внимание, что `Break` и `Continue` не зарезервированные слова, и поэтому ***могут быть перегружены***.

13.2.5 Оператор For..in..do

Начиная с версии Free Pascal **2.4.2**, поддерживает конструкцию цикла `for..in`. Цикл `for..in` используется в случае, если нужно что-то вычислять фиксированное число раз с перечислимой переменной. Синтаксический прототип выглядит следующим образом:

Оператор *for*

```
►►─оператор for in ──for─параметр цикла─in─перечисление──────────►◄
►►─параметр цикла─идентификатор переменной──────────────────────►◄
►►─перечисление─┬тип перечисления┬──────────────────────────────►◄
                └──выражение─────┘
```

Этот оператор *(for)* может быть составным. ***Перечислимое выражение*** должно принимать фиксированное число элементов: ***переменная цикла*** будет

принимать значение каждого из этих элементов, и каждый раз будет выполнен оператор после ключевого слова do.

Перечислимое выражение может быть одним из пяти вариантов:

1. **Идентификатор типа перечисление.** Цикл будет производиться над всеми элементами типа перечисления. ***Переменная цикла*** должна быть типа перечисления.
2. **Значение типа множество.** Цикл будет произведён со всеми элементами в множестве, ***переменная цикла*** должна быть основана на типе множество.
3. **Значение типа массив.** Цикл будет произведён над всеми элементами массива, а ***переменная цикла*** должна иметь тот же тип что элемент массива. *Как частный случай, строка рассматривается как массив символов.*
4. **Экземпляр класса enumeratable *(перечисления)*.** Это экземпляр класса, который поддерживает интерфейсы IEnumerator и IEnumerable. В этом случае тип ***переменной цикла*** должен быть такой-же как тип значения возвращаемого IEnumerator.GetCurrent.
5. **Любой тип, для которого определен оператор перечисления.** Оператор должен возвращать класс перечисления *(enumerator)*, который реализует интерфейс IEnumerator. Тип ***переменной цикла*** должен быть как тип возвращаемого значения перечислителем класса GetCurrent.

Простейший случай цикла for..in использует переменную перечисляемого типа:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday, friday,
saturday,sunday);

Var
  d : TWeekday;
begin
  for d in TWeekday do writeln(d);
end.
```

Этот код выведет на экран все дни недели.

Выше приведённая конструкция for..in эквивалентна следующей конструкции for..to:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday, friday,
saturday,sunday);

Var
  d : TWeekday;
begin
  for d:=Low(TWeekday) to High(TWeekday) do writeln(d);
end.
```

Второй случай цикла `for..in`, когда перечислимое выражение представляет собой множество, а затем цикл будет выполняться каждый раз для каждого элемента множества:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday, friday,
saturday,sunday);

Var
  Week : set of TWeekDay = [monday, tuesday, wednesday,
thursday, friday];
  d : TWeekday;
begin
  for d in Week do writeln(d);
end.
```

Этот код выведет на экран все дни недели. Обратите внимание, что переменная `d` того же типа, что и множество *(Week)*.

Выше приведённая конструкция `for..in` эквивалентна следующей конструкции `for..to`:

```
Type
  TWeekDay = (monday, tuesday, wednesday, thursday, friday,
saturday,sunday);

Var
  Week : set of TWeekDay
       = [monday, tuesday, wednesday, thursday, friday];
  d : TWeekday;

begin
  for d:=Low(TWeekday) to High(TWeekday) do
       if d in Week then writeln(d);
end.
```

Третий вариант цикла `for..in`, когда перечислимое выражение является массивом:

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');

Var
  S : String;
begin
  For s in a do Writeln(s);
end.
```

Этот код выведет на экран все дни недели т эквивалентен

```
var
  a : Array[1..7] of string
    = ('monday','tuesday','wednesday','thursday',
       'friday','saturday','sunday');
Var
  i : integer;
begin
  for i:=Low(a) to high(a) do Writeln(a[i]);
end.
```

Тип string эквивалентен array of char, по этому строка может быть использована в цикле for..in. Этот код выведет на экран все буквы алфавита, каждая буква с новой строки:

```
Var
  c : char;
begin
 for c in 'abcdefghijklmnopqrstuvwxyz' do writeln(c);
end.
```

Четвертый вариант цикла for..in использует классы. Класс реализует интерфейс IEnumerable, который определяется следующим образом:

```
IEnumerable = interface(IInterface)
  function GetEnumerator: IEnumerator;
end;
```

Фактический тип не обязательно должен возвращать GetEnumerator, но должен реализовать интерфейс IEnumerator, вместо этого, он может быть классом, который реализует методы IEnumerator:

```
IEnumerator = interface(IInterface)
  function GetCurrent: TObject;
  function MoveNext: Boolean;
  procedure Reset;
  property Current: TObject read GetCurrent;
end;
```

Свойство Current и метод MoveNext должен присутствовать в классе, возвращаемом методом GetEnumerator. Фактический тип свойства Current не обязательно должен быть TObject. Встретив for..in цикл с экземпляром класса, в качестве in операнда, компилятор проверит каждое из следующих условий:

- Является ли класс в перечислимом выражении реализующим метод GetEnumerator
- Является ли результат метода GetEnumerator классом с методом: Function MoveNext : Boolean
- Является ли результат метода GetEnumerator классом со свойством только для чтения:

```
Property Current : AType;
```

Тип свойства должен соответствовать типу переменной цикла цикла `for..in`.

Интерфейсы `IEnumerator`, `IEnumerable` не должны быть фактически объявлены в перечислимом классе: компилятор обнаруживает, присутствуют ли эти интерфейсы с помощью описанных выше проверок. Интерфейсы определены только для совместимости с Delphi и не используются внутри. *(Также было бы невозможно обеспечить соблюдение их правильности).*

Модуль Classes содержит ряд классов, которые содержат перечислители:

TFPList
Перечисляет все указатели в списке.

TList
Перечисляет все указатели в списке.

TCollection
Перечисляет все элементы коллекции.

TStringList
Перечисляет все строки в списке.

TComponent
Перечисляет все дочерние компоненты, принадлежащие компоненту.

Этот код выведет на экран все дни недели.

```
{$mode objfpc}
uses classes;

Var
  Days : TStrings;
  D : String;
begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    For D in Days do Writeln(D);
  Finally
    Days.Free;
  end;
```

```
end.
```

Обратите внимание, что компилятор нуждается в безопасности типов: объявление D как целого числа приведет к ошибке компиляции:

```
testsl.pp(20,9) Error: Incompatible types: got "AnsiString" ex-
pected "LongInt"
testsl.pp (20,9) Ошибка: Типы несовместимы: получил
"AnsiString" ожидая "LongInt"
```

Приведенный выше код эквивалентен следующему:

```
{$mode objfpc}
uses classes;

Var
  Days : TStrings;
  D : String;
  E : TStringsEnumerator;
begin
  Days:=TStringList.Create;
  try
    Days.Add('Monday');
    Days.Add('Tuesday');
    Days.Add('Wednesday');
    Days.Add('Thursday');
    Days.Add('Friday');
    Days.Add('Saturday');
    Days.Add('Sunday');
    E:=Days.getEnumerator;
    try
      While E.MoveNext do
        begin
          D:=E.Current;
          Writeln(D);
        end;
    Finally
      E.Free;
    end;
  Finally
    Days.Free;
  end;
end.
```

Обе программы приведут к одному результату.

Пятая и последняя возможность использовать цикл `for..in` его можно использовать для перечисления практически любого типа, с помощью оператора `enumerator` *(перечислитель)*. Оператор `enumerator` должен

возвращать класс, который имеет ту же сигнатуру, что и IEnumerator в подходе выше. Следующий код будет определять enumerator *(перечислитель)* для типа Integer:

```
Type

TEvenEnumerator = Class
  FCurrent : Integer;
  FMax : Integer;
  Function MoveNext : Boolean;
  Property Current : Integer Read FCurrent;
end;

Function TEvenEnumerator.MoveNext : Boolean;
begin
  FCurrent:=FCurrent+2;
  Result:=FCurrent<=FMax;
end;

operator enumerator(i : integer) : TEvenEnumerator;
begin
  Result:=TEvenEnumerator.Create;
  Result.FMax:=i;
end;

var
  I : Integer;
  m : Integer = 4;
begin
  For I in M do Writeln(i);
end.
```

Цикл выведет на экран все отличные от нуля четные числа меньше или равные перечислим. *(как в примерах 2 и 4).*

Необходимо соблюдать осторожность при определении операторов Enumerator: компилятор найдет и использовать первый же доступный оператор Enumerator для перечисляемого выражения. Для классов это означает, что метод GetEnumerator даже не рассматривается. Следующий код будет определять оператор перечислитель, который извлекает объекты из StringList:

```
{$mode objfpc}
uses classes;

Type
  TDayObject = Class
    DayOfWeek : Integer;
```

```
    Constructor Create(ADayOfWeek : Integer);
  end;

  TObjectEnumerator = Class
    FList : TStrings;
    FIndex : Integer;
    Function GetCurrent : TDayObject;
    Function MoveNext: boolean;
    Property Current : TDayObject Read GetCurrent;
  end;

Constructor TDayObject.Create(ADayOfWeek : Integer);
begin
  DayOfWeek:=ADayOfWeek;
end;

Function TObjectEnumerator.GetCurrent : TDayObject;
begin
  Result:=FList.Objects[Findex] as TDayObject;
end;

Function TObjectEnumerator.MoveNext: boolean;
begin
  Inc(FIndex);
  Result:=(FIndex<FList.Count);
end;

operator enumerator (s : TStrings) : TObjectEnumerator;
begin
  Result:=TObjectEnumerator.Create;
  Result.Flist:=S;
  Result.FIndex:=-1;
end;

Var
  Days : TStrings;
  D : String;
  O : TdayObject;

begin
  Days:=TStringList.Create;
  try
    Days.AddObject('Monday',TDayObject.Create(1));
    Days.AddObject('Tuesday',TDayObject.Create(2));
    Days.AddObject('Wednesday',TDayObject.Create(3));
```

```
    Days.AddObject('Thursday',TDayObject.Create(4));
    Days.AddObject('Friday',TDayObject.Create(5));
    Days.AddObject('Saturday',TDayObject.Create(6));
    Days.AddObject('Sunday',TDayObject.Create(7));
    For O in Days do Writeln(O.DayOfWeek);
  Finally
    Days.Free;
  end;
end.
```

Приведенный выше код выведет на экран название дня недели за неделю.

Если класс не поддерживает перечисление, компилятор выдаст сообщение об ошибке , когда он встретит цикл `for...in`.

Примечание:

Как и в цикле `for..to`, не разрешается изменять *(т.е. присваивать)* значение ***переменной цикла*** внутри цикла.

13.2.6 Оператор Repeat..until

Оператор `repeat` выполняет список операторов, пока не будет достигнуто определенное условие. Список выполняться *(в любом случае)* хотя-бы один раз. Синтаксический прототип оператора `repeat..until` отображён на схеме

Оператор *repeat*

Этот оператор будет выполнять операторы между `repeat` и `until` до момента, когда `выражение` *(expression)* не примет значение `True`. Поскольку `выражение` вычисляется ***после*** выполнения операторов, они выполняются хотя-бы один раз.

Помните о том, что логическое `выражение` *(expression)* будет вычисляться по умолчанию посхеме короткого замыкания [189], а это означает, что вычисления будет остановлены в точке, где результат определён.

Ниже приведены допустимые операторы `repeat`

```
repeat
  WriteLn ('I =',i);
  I := I+2;
until I>100;
```

```
repeat
 X := X/2
until x<10e-3;
```

Обратите внимание, что последний оператор перед ключевым словом until не нужно заканчивать точкой с запятой, но она допускается.

Для выхода из цикла или начала новой итерации оператора repeat..until могут быть использованы системные процедуры Break и Continue. Обратите внимание, что Break и Continue не зарезервированные слова, и поэтому ***могут быть перегружены***.

13.2.7 Оператор While..do

Оператор while используется для выполнения оператора *(простого или составного)*, пока выполняется определенное условие. В отличии от цикла repeat, оператор может никогда не выполниться.

Синтаксический прототип оператора while..do отображён схеме

Оператор *while*

```
▸▸─оператор while ─ while ─ выражение ─ do ─ оператор ──────────────▸◂
```

Это позволит выполнять оператор до тех пор, пока выражение имеет значение True. Так как выражение вычисляется ***перед*** выполнением оператора, может быть что оператор **не выполняется вообще**. Оператор может быть и составным.

Помните о том, что логическое выражение *(expression)* будет вычисляться по умолчанию посхеме короткого замыкания [189], а это означает, что вычисления будет остановлены в точке, где результат определён.

Ниже приведены допустимые операторы while

```
I := I+2;
while i<=100 do
  begin
  WriteLn ('I =',i);
  I := I+2;
  end;

X := X/2;
while x>=10e-3 do X := X/2;
```

Они соответствуют примерам циклов для операторов repeat.

Если оператор *(после do)* является составным, то могут быть использованы системные процедуры `Break` и `Continue` для выхода из цикла или начала новой итерации оператора `while`. Обратите внимание, что `Break` и `Continue` не зарезервированные слова, и поэтому ***могут быть перегружены***.

13.2.8 Оператор With

***Оператор* `with`** служит для доступа к элементам записи, объекта или класса, без указания каждый раз полного имени этого элемента. Синтаксис оператора `with`

Оператор *with*

```
►► оператор with ── with ─┬─ ссылка на переменную ─┬─ do ─ оператор ──────►◄
                          └────────── , ───────────┘
```

Переменная ссылка должна быть переменной типа записи, объекта или класса. В операторе with,любая ссылка на переменную или метод проверяется, чтобы увидеть что это поле, метод записи, объекта или класса. Если да, то поле доступно и к нему можно обратится, или можно вызвать метод. Примем во внимание определение:

```
Type
  Passenger = Record
    Name : String[30];
    Flight : String[10];
  end;

Var
  TheCustomer : Passenger;
```

Тогда следующие присвоения полностью эквивалентны:

```
TheCustomer.Name   := 'Michael';
TheCustomer.Flight := 'PS901';
```

и

```
With TheCustomer do
  begin
    Name := 'Michael';
    Flight := 'PS901';
  end;
```

Оператор

```
With A,B,C,D do Statement;
```

эквивалентен

```
With A do
 With B do
  With C do
   With D do Statement;
```

Это говорит о том, что переменные ищутся *от последнего к первому*, то есть, когда компилятор встречает ссылку на переменную, он сначала проверить, это поле или метод в последней переменной *(ссылке)*. Если поля нет, то он проверит предпоследнюю *(ссылку)*, и так далее. Следующий пример демонстрирует это;

```
Program testw;
Type AR = record
      X,Y : Longint;
     end;
     PAR = ^Ar;

Var S,T : Ar;
begin
  S.X := 1;S.Y := 1;
  T.X := 2;T.Y := 2;
  With S,T do WriteLn (X,' ',Y);
end.
```

Эта программа выведет

```
2 2
```

Таким образом, `X`, `Y` в операторе `Writeln` совпадают с полями переменной записи `T`.

Примечание:

Использовании оператора `with` с указателем или классом, не допускается, для изменения самого указателя или класса в блоке `with`.Используя определения предыдущего примера, проиллюстрируем о чём идёт речь:

```
Var p : PAR;

begin
  With P^ do
   begin
     // Некоторые операции
     P:=OtherP;
     X:=0.0;  // Будет использовано не то поле X  !!
   end;
```

Указатель не может быть изменен по тому что адрес хранится компилятором во временном регистре. Изменение указателя не изменит временный адрес. Это же верно и для классов.

13.2.9 Операторы Исключения

Free Pascal поддерживает исключения. Исключения обеспечивают удобный способ программирования механизмов обработки ошибок и восстановления после ошибок, и тесно связаны с классами. Поддержка исключений объясняется в главе Глава 17 Исключения[273].

13.3 Оператор Asm

Оператор ассемблера позволяет вставлять код ассемблера прямо в код Pascal.

Оператор Ассемблера

```
►► оператор asm ── asm ── код ассемблера ── end ──┬─────────────────┬──►◄
                                                  └ список регистров ┘
►► список регистров ── [ ─┬ строковая константа ┬─ ] ────────────────►◄
                          └──────── , ──────────┘
```

Более подробную информацию о блоках ассемблера можно найти в Справочник программиста Free Pascal. В списке регистр используется для обозначения регистров, которые изменяются при помощи инструкции ассемблере в блоке ассемблере. **Список регистров** используется для обозначения, какие ***регистры компилятору следует сохранить*** ибо они будут меняться. Компилятор возвращает определенные результаты в регистрах. Если регистры модифицируются в операторе ассемблера, компилятор иногда сообщает об этом. Регистры обозначаются их именами, для процессора Intel *(i386)*, например 'EAX', 'ESI' и т.д. В качестве примера, рассмотрим следующий код на ассемблере:

```
asm
  Movl $1,%ebx
  Movl $0,%eax
  addl %eax,%ebx
end ['EAX','EBX'];
```

Это говорит компилятору, что он должен сохранить и восстановить регистры EAX и EBX, когда он встретит этот оператор.

Free Pascal поддерживает различные стили синтаксиса ассемблера. По умолчанию, для платформы 80386 и Compatibles предполагается синтаксис AT&T. Стиль ассемблера по умолчанию, может быть изменен переключателем {$asmmode xxx} в коде Pascal, или опцией -R командной строки. Более подробно об этом можно найти в Справочник программиста Free Pascal.

Глава 14 Использование функций и процедур

Free Pascal поддерживает использование функций и процедур. Он поддерживает

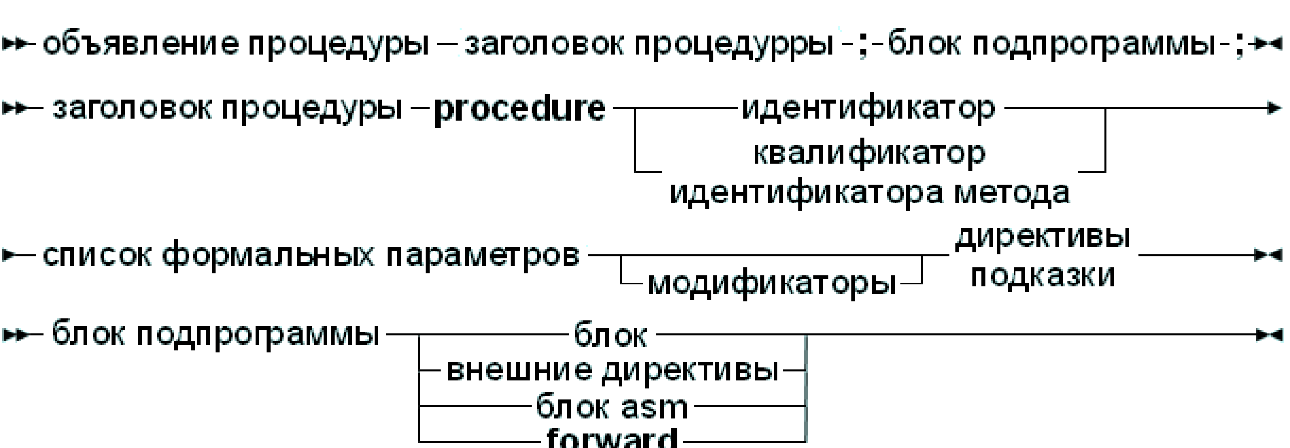
- ***Перегрузка функций***, т.е. функций с одинаковыми именами, но разными списками параметров.
- ***Const параметры***.
- ***Открытые массивы*** *(т.е. массивы без границ)*.
- Переменным количеством аргументов *(как в C)*.
- Конструкция - возврата из функции *(процедуры)*, как и в C, с использованием ключевого слова Exit.

Примечание:

Во многих последующих пунктах слова procedure *(процедура)* и function *(функция)* будет использоваться как взаимозаменяемые. Объявление действительное для обоих, за исключением случаев, когда указано иное.

14.1 Объявление процедуры

Объявление процедуры определяет идентификатор *(имя)* и связывает его с блоком кода. Процедура может быть вызвана ***оператором вызова процедуры*** 198.



Смотрите раздел 14.4 Список параметров 220 для определения списка параметров. Объявление процедуры, содержит блок в котором реализуется действие процедуры. Ниже показано допустимое объявление процедуры:

```
Procedure DoSomething (Para : String);
begin
  Writeln ('Параметр: ',Para);
```

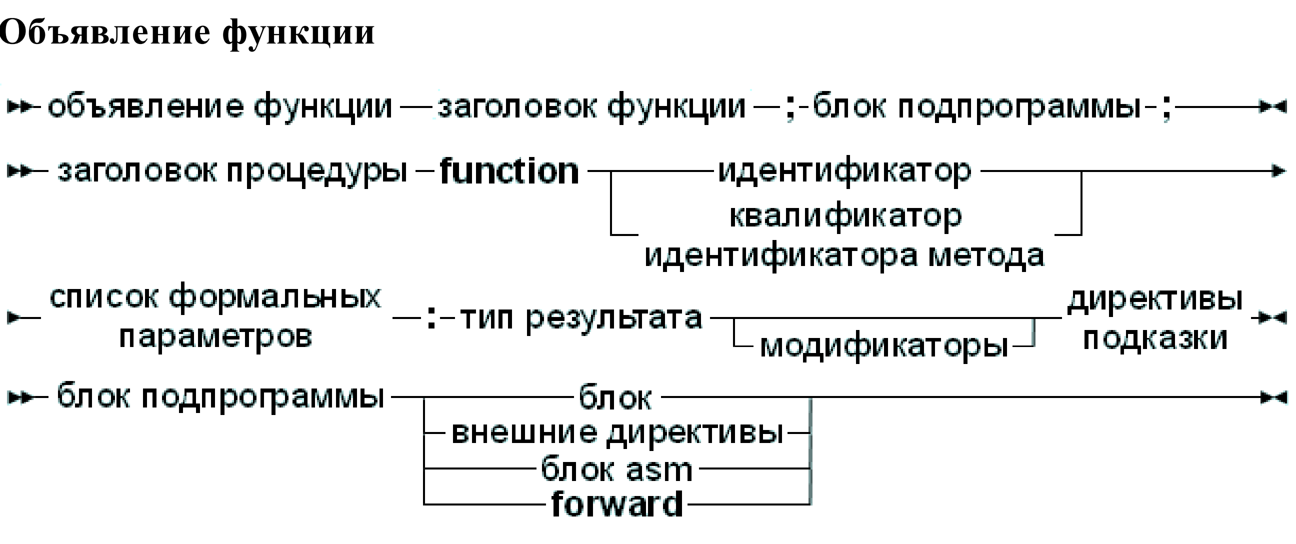
```
    Writeln ('Параметр в верхнем регистре: ',Upper(Para));
  end;
```

Обратите внимание, что процедура может вызывать сама себя *(рекурсия)*.

14.2 Объявление функции

Объявление функции определяет идентификатор *(имя)* и связывает его с блоком кода. Блок кода должен вернуть результат. Функция может быть вызвана в ***выражении***, или как процедура *(**оператором вызова процедуры*** 198*)*, если расширенный синтаксис включен.

Объявление функции



Результат функции может быть любого ранее объявленного типа. В отличие от Turbo Pascal, где можно было вернуть только простые типы.

14.3 Результат функции

Результат функции можно вернуть, установив переменную результат *(Result)*: это может быть идентификатор *(имя)* функции или, *(только в режимах ObjFPC или Delphi)* специальный идентификатор Result:

```
Function MyFunction : Integer;
begin
  MyFunction:=12; // Возвращаем 12
end;
```

В режиме Delphi или ObjFPC, код может также быть таким:

```
Function MyFunction : Integer;
begin
  Result:=12;
end;
```

Как расширение синтаксиса в режимах Delphi или ObjFPC поддерживается

специальная расширенная процедура Exit *(выхода)*:

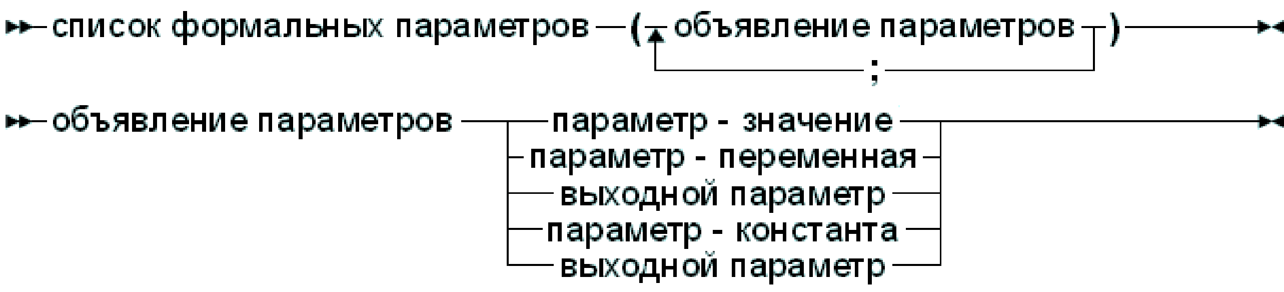
```
Function MyFunction : Integer;
begin
  Exit(12);
end;
```

Вызов Exit устанавливает результат функции *(в значение в скобках)* и переходит к концу блока объявления функции. Её можно рассматривать как эквивалент инструкции C return .

14.4 Список параметров

Для передачи функциям *(или процедурам)* параметров, они должны быть объявлены в списке формальных параметров этой функции *(или процедуры)*. Список формальных параметров это идентификаторы *(имёна)* параметров, которые могут быть использованы внутри этой процедуры или функции.

Параметры



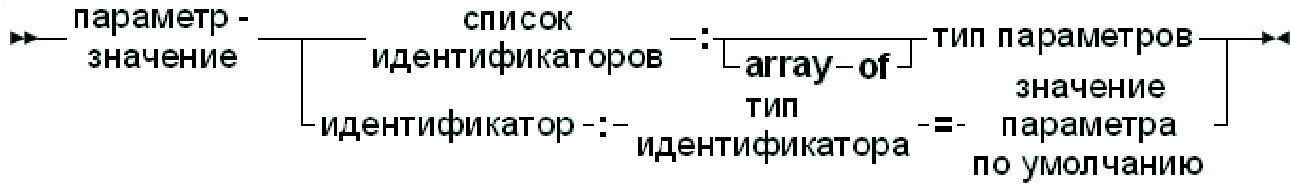
Параметры const *(константы)*, out *(выходные)* и var *(переменные)* могут быть и untyped *(нетипизированные)*, ***(если не имеют имени типа)***.

Начиная с версии Free Pascal **1.1** поддерживаются значения по умолчанию *(default)* для значений параметров const и var, но ***только*** простых типов. Компилятор должен быть в режиме OBJFPC или DELPHI, чтобы применять значения по умолчанию *(default)*.

14.4.1 Параметры-значения

Параметры-значения объявлены следующим образом:

Параметры-значения



Когда параметры объявлены как ***параметры-значения***, то процедура получает ***копию*** параметров, когда проходит её вызов. Любые изменения этих параметров являются локальными внутри блока процедуры, и не распространяются обратно в вызывающий блок.

Когда надо вызвать процедуру с ***параметрами-значениями*** нужно присвоить *(при вызове)* параметры совместимых типов. Это означает, что типы могут точно не совпадать, но должны быть преобразованы в фактические параметры совместимых типов. Код преобразования вставляется самим компилятором.

Необходимо соблюдать осторожность при использовании ***параметров-значений***: параметры-значений интенсивно используют стек, особенно если они больших размеров. Общий размер всех параметров в списке формальных параметров должен быть меньше **32**K для портирования *(для Intel размер ограничен **64**K)*.

В качестве ***параметров-значений*** могут быть использованы открытые массивы. Для получения дополнительной информации об использовании открытых массивов смотрите раздел 14.4.5 Параметр- открытый массив [226].

Для параметров простых типов *(т.е. не структурированных типов)*, могут быть указаны значения по умолчанию *(default)*. Здесь может быть нетипизированная константа. Если при вызове функции нет какого-то параметра, при вызове функции будет принято значение по умолчанию. Для динамических массивов или других типов, которые можно рассматривать как эквивалент указателя, значение по умолчанию всегда `Nil`.

Следующий пример выведет на экране **20**:

```
program testp;
Const
  MyConst = 20;

Procedure MyRealFunc(I : Integer = MyConst);
begin
  Writeln('Функция получила: ',I);
end;

begin
  MyRealFunc;
```

```
end.
```

14.4.2 Параметры-переменные

Параметры переменные объявляются следующим образом:

Параметр переменная

Если параметр объявлен как ***параметр-переменная***, процедура *(или функция)* сразу же получает доступ к переменной *(параметру)*, потому что параметр *(или параметры, если они идут блоком)* передается как ссылка. Процедура получает указатель на переданную переменную и использует этот указатель для доступа к значению переменной. Это значит что изменения, внесенные в параметр, будет распространяться и обратно в вызывающий блок. Этот механизм может быть использован для передачи значений обратно процедурам. По этому, вызывающий блок должен передать параметр ***точно*** такого же типа, как и объявленный параметр. Если этого не будет, то компилятор выдаст ошибку.

Параметры-переменные *(var) (как и параметры-константы (const))* могут быть и untyped *(нетипизирована)*. В этом случае переменная не имеет типа, значит несовместима с другими типами. Тем не менее, здесь может быть использован оператор взятия адреса [186], в функцию может быть передан параметр, который сам untyped *(нетипизированн)*. Если для присвоения используется untyped *(нетипизированный)* параметр или ему *(параметру)* должно быть присвоено значение, необходимо использовать приведение типов [183].

Переменная файлового типа должна быть передана как ***параметр-переменная***.

Открытый массив тоже может быть передан как параметр-переменная. Для получения дополнительной информации об использовании открытых массивов смотрите раздел 14.4.5 Параметр- открытый массив [226].

Примечание:

- Обратите внимание, что значения по умолчанию **не поддерживаются** для ***параметров-переменных***. Это бессмысленно, так как теряется смысл возможность передать значение обратно.

- Результат функции *(Result)* рассматривается внутри как ***параметр-переменная*** и может иметь ненулевое *(или ненулевое)* начальное значение. Это особенно важно для управляемых типов.

14.4.3 Выходные (Out) параметры

Out параметры *(выходные параметры)* объявляются следующим образом:

Out параметры

```
>>- выходной параметр -- out -- список идентификаторов --+--------------------------------------------+-><
                                                         |_ : _+__________+_ тип идентификатора _|
                                                               |_ array - of _|
```

Out параметр предназначен что-бы передавать значения **обратно из функции**: значение передается по ссылке. Начальное значение параметра отбрасывается, и не может использоваться.

Если переменная нужна для передачи данных в функцию и получения их из функции, то необходимо использовать параметр-переменную [222]. Если нужно ***только*** извлечь значение может быть использован out-параметр.

Само собой разумеется, значения по умолчанию ***не поддерживаются*** для out-параметров.

Отличие out параметров от параметров переданных по ссылке *(var и const)* очень мала *(про управляемые типы смотри ниже)*: out параметр дает информацию о том, что будет с аргументам при передаче в процедуры: коммилятор знает, что переменная не должна быть инициализированы перед вызовом. Следующий пример иллюстрирует это:

```
Procedure DoA(Var A : Integer);
begin
  A:=2;
  Writeln('A содержит ',A);
end;

Procedure DoB(Out B : Integer);
begin
  B:=2;
  Writeln('B содержит',B);
```

```
end;

Var
  C,D : Integer;
begin
  DoA(C);
  DoB(D);
end.
```

Процедуры DoA и DoB делают одно и то же. Но объявление DoB дает больше информации для компилятора, что позволяет ему обнаружить, что D не должна быть инициализирована до вызова. В то время как параметр A в DoA может принимать и возвращать значения, компилятор замечает, что C не инициализирована до вызова процедуры DoA.

```
home: >fpc -S2 -vwhn testo.pp
testo.pp(19,8) Hint: Variable "C" does not seem to be
initialized
testo.pp(19,8) Подсказка: Переменная "C", кажется, не быть
инициализирована
```

Таким образом лучше использовать out параметры, когда нужно только вернуть значения.

Примечание:

Out параметры поддерживаются только в режимах Delphi и ObjFPC. Для других режимов out тоже является **допустимым** идентификатором.

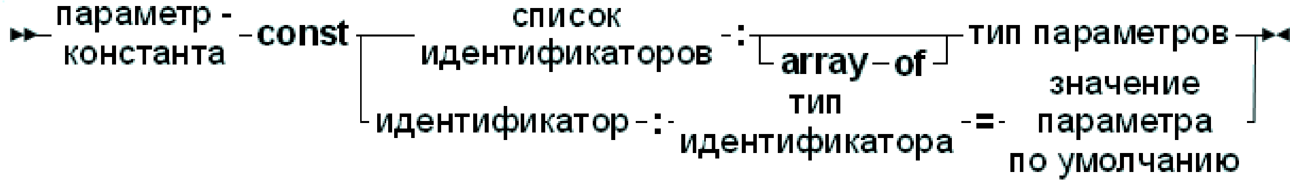
Примечание:

Для управляемых типов *(подсчет ссылок)*, несет некоторые накладные расходы при использовании Out параметров: компилятор должен быть уверен, что значения инициализированы правильно *(т.е. имеет счетчик ссылок отличный от ноля (**0**))*. Эта инициализация обычно производится вызывающей стороной.

14.4.4 Параметры-константы

В дополнение к параметрам-переменным [222] и параметрам-значениям [220] Free Pascal также поддерживает и ***параметры-константы*** *(const)*. **Параметр-константа** может быть определен следующим образом:

Параметры const



Определение параметра как константы дает компилятору указание, что параметр ***не будет*** изменен вызывающей подпрограммой. Что позволяет компилятору выполнять оптимизацию, которую он не мог бы иначе выполнить, а также он выполняет определёные проверки кода внутри подпрограммы, например: ***он запрещает присвоения параметру***. Например ***параметр-константа*** не может быть передан другой функции, которая требует ***параметра-переменной***: компилятор тоже может это проверить. В основном ***параметр-константы*** уменьшает размер стека, а значит повышает производительность *(копировать параметры не требуется)*, что нужно делать при передаче параметра по значению...

Примечание:

В отличие от Delphi, ***нельзя*** делать предположений о том, как ***const-параметры*** передаются в вызывающую процедуру. В частности, предположение, что параметры большого размера передаются по ссылке, не правильно. Для этого нужно использовать параметр типа `constref` *(ссылка на константу)*, доступный начиная с версии компилятора **2.5.1**.

Исключением является соглашение о вызовах `stdcall`: применяемый для совместимости со стандартами COM, где большие параметры-константы передаются по ссылке.

Примечание:

Учтите, что определение `const` представляет собой договор между программистом и компилятором. Программист, сообщает компилятору, что содержимое ***параметра-константы*** не будет изменено, при выполнении процедуры, а ***не компилятор***, говорит программисту, что параметр не будет изменен.

Это **очень важно** и видно при использовании типов со счётчиком ссылок *(refcounted)*. Для этих типов, увеличение или уменьшение счетчика ссылок *(неявное)* не делается при использовании `const`. Это часто позволяет компилятору опустить неявные `try/finally` блоки для этих подпрограмм.

В качестве примера такого побочного эффекта, приведён следующий код, результатом которого будет неожиданный результат:

```
Var
  S : String = 'Какая то информачия';

Procedure DoIt(Const T : String);
begin
  S:='Ещё какая-то информация';
  Writeln(T);
end;

begin
  DoIt(S);
end.
```

Он выведет

```
Ещё какая-то информация
```

Такое поведение является особенностью конструкции.

Параметры-константы могут быть и нетипизированными *(untyped)*. Для получения дополнительной информации о нетипизированных параметрах смотрите раздел 14.4.2 Параметры-переменные [222]

Что касается значений параметра, ***параметры-константы*** могут получить значение по умолчанию *(default)*.

Открытые массивы тоже могут быть переданы в качестве ***параметров-констант***. Для получения дополнительной информации об использовании открытых массивов смотрите раздел 14.4.5 Параметр- открытый массив [226].

14.4.5 Параметр- открытый массив

Free Pascal поддерживает ***параметры- открытые массивы***, т.е. процедура

может быть объявлена с массивом неопределённой длинны в качестве параметра, как и в Delphi. ***Параметры- открытые массивы*** будут доступны в процедуре или функции как массив, который объявлен с начальным индексом **0**, а индекс последнего элемента `High(параметр)`. Например, объявление параметра

```
Row : Array of Integer;
```

будет эквивалентно

```
Row : Array[0..N-1] of Integer;
```

Где `N` - фактический размер массива, передаваемый функции. `N-1` может быть вычислено как `High(Row)`.

Однако, если передается пустой массив, то `High(параметр)` возвращает **-1**, а `low(параметр)` возвращает **0**.

Параметры- открытые массивы могут быть переданы ***по значению***, ***по ссылке*** или в качестве ***параметра-константы***. В последнем случае подпрограмма получает указатель на фактический массив. В первом случае, он получает копию массива. В функцию *(или процедуру)*, открытые массивы могут быть переданы ***только*** функциям *(или процедурам)*, которые также объявлены с ***параметрами- открытыми массивами***, а ***не*** функциям *(или процедурам)*, у которые ***параметры- массивы фиксированной длины***. Ниже приведен пример функции с использованием открытого массива:

```
Function Average (Row : Array of integer) : Real;
Var
  I : longint;
  Temp : Real;
begin
  Temp := Row[0];
  For I := 1 to High(Row) do Temp := Temp + Row[i];
  Average := Temp / (High(Row)+1);
end;
```

Начиная с FPC **2.2**, можно передавать ***частичные*** массивы в функцию, которая требует ***открытый массив*** *как* ***параметр***. Это может быть сделано путем определения диапазона массива, который должен быть передан в открытый массив.

Принимая во внимание объявление

```
Var
  A : Array[1..100];
```

следующий вызов вычислит и выведет среднее **100** элементов:

```
  Writeln('Среднее 100-а элементов: ',Average(A));
```

А эти вызовы будет вычислять и выводить среднее значение первой и второй половины *(массива)*:

```
  Writeln('Среднее первых 50-ти элементов: ',Average(A
[1..50]));
  Writeln('Среднее последних 50-ти элементов: ',Average(A
[51..100]));
```

14.4.6 Массив констант

Free Pascal в режиме Object Pascal или Delphi, поддерживает конструкцию `Array of Const` *(массив констант)* для передачи параметров в подпрограмму.

Это частный случай конструкции `Open array` *(открытого массива)*, так он может передать любое выражение в функцию *(или процедуру)*. Тип результата выражения должен быть простым: структура не может быть переданы как аргумент. Это означает, что могут быть переданы порядковые и вещественные типы, строки, а также указатели на классы и интерфейсы.

Элементы `array of const` преобразуются в специальную вариантную запись:

```
Type
  PVarRec = ^TVarRec;
  TVarRec = record
     case VType : Ptrint of
       vtInteger    : (VInteger: Longint);
       vtBoolean    : (VBoolean: Boolean);
       vtChar       : (VChar: Char);
       vtWideChar   : (VWideChar: WideChar);
       vtExtended   : (VExtended: PExtended);
       vtString     : (VString: PShortString);
       vtPointer    : (VPointer: Pointer);
       vtPChar      : (VPChar: PChar);
       vtObject     : (VObject: TObject);
       vtClass      : (VClass: TClass);
       vtPWideChar  : (VPWideChar: PWideChar);
       vtAnsiString : (VAnsiString: Pointer);
       vtCurrency   : (VCurrency: PCurrency);
       vtVariant    : (VVariant: PVariant);
       vtInterface  : (VInterface: Pointer);
       vtWideString : (VWideString: Pointer);
       vtInt64      : (VInt64: PInt64);
       vtQWord      : (VQWord: PQWord);
   end;
```

Внутри тела процедуры аргумент `array of const` эквивалентен открытому массиву `TVarRec`:

```
Procedure Testit (Args: Array of const);
```

```
Var I : longint;
begin
  If High(Args)<0 then
    begin
    Writeln ('Нет аргументов');
    exit;
    end;
  Writeln ('Got ',High(Args)+1,' arguments :');
  For i:=0 to High(Args) do
    begin
    write ('Аргумент ',i,' имеет тип ');
    case Args[i].vtype of
      vtinteger    :
        Writeln ('Целое, Со значением :',args[i].vinteger);
      vtboolean    :
        Writeln ('Boolean, Со значением :',args[i].vboolean);
      vtchar       :
        Writeln ('Символ, Со значением : ',args[i].vchar);
      vtextended   :
        Writeln ('Вещественное, Со значением : ',args[i]
.VExtended^);
      vtString     :
        Writeln ('ShortString, Со значением :',args[i]
.VString^);
      vtPointer    :
        Writeln ('Указатель,, Со значением : ',Longint(Args[i]
.VPointer));
      vtPChar      :
        Writeln ('PChar,, Со значением : ',Args[i].VPChar);
      vtObject     :
        Writeln ('Объект, С именем : ',Args[i]
.VObject.Classname);
      vtClass      :
        Writeln ('Ссылка на класс, С именем :',Args[i]
.VClass.Classname);
      vtAnsiString :
        Writeln ('AnsiString, Со значением :',AnsiString(Args
[I].VAnsiString);
    else
        Writeln ('(Неизвестное) : ',args[i].vtype);
    end;
    end;
end;
```

В код этой процедуры можно передать произвольный массив элементов:

```
    S:='AnsiString 1';
    T:='AnsiString 2';
    Testit ([]);
    Testit ([1,2]);
    Testit (['A','B']);
    Testit ([TRUE,FALSE,TRUE]);
    Testit (['String','Иная строка']);
    Testit ([S,T])  ;
    Testit ([P1,P2]);
    Testit ([@testit,Nil]);
    Testit ([ObjA,ObjB]);
    Testit ([1.234,1.234]);
    TestIt ([AClass]);
```

Если процедура объявлена с модификатором cdecl, то компилятор будет передать массив как откомпилированный под C. Он, эмулирует C конструкцию с переменным числом аргументов, как показано в следующем примере:

```
program testaocc;
{$mode objfpc}

Const
  P : PChar = 'пример';
  Fmt : PChar =
        'Этот %s показывает использование printf для вывода
чисел (%d) и строк'#10;
// Объявление стандартной функции printf:

procedure printf (fm : pchar; args : array of const);cdecl;
external 'c';
begin
 printf(Fmt,[P,123]);
end.
```

Заметим, что это не верно для Delphi, поэтому код опирающийся на эту функцию не будет переносим.

Примечание:

Обратите внимание , что не существует поддержки аргументов QWord в array of const. Это нужно для совместимости с Delphi, и компилятор будет игнорировать проверку диапазона в режиме Delphi.

14.5 Управляение типами со счетчиком ссылок

Некоторые типы *(UnicodeString, AnsiString, интерфейсы, динамические массивы)* обрабатываются особым образом: данные этих типов имеют счетчик

ссылок, который увеличивается или уменьшается в зависимости от того, как много существует ссылок на эти данные.

Квалификаторы параметров вызова функций *(или процедур)* влияют на счетчик управляющий ссылками на типы:

- ничего *(без квалификатора) (передача по значению)*: счетчик ссылок на параметр увеличивается на входе и уменьшается на выходе.
- `out`: при вызове подпрограммы счетчик ссылок на значение уменьшается на **1**, а переменной присваивается ***пустое значение*** *(как правило `Nil`, но на эту деталь реализации не следует полагаться)*.
- `var` со счетчиком ссылок ничего не происходит. Передается ссылка на *исходную* переменную, и ***изменение*** или ***чтение*** параметра имеет точно такой же эффект, как ***изменение/чтение*** исходной переменной.
- `const` этот случай немного сложнее.Со счетчиком ссылок ***ничто не происходит***, потому что здесь вы можете передать не-значение. Например, вы можете передать класс реализующий интерфейс, а не сам интерфейс вызывающий класс, и класс может быть неожиданно освобожден.

Следующий пример демонстрирует эту опасность:

```
{$mode objfpc}

Type
  ITest = Interface
    Procedure DoTest(ACount : Integer);
  end;

  TTest = Class(TInterfacedObject,ITest)
    Procedure DoTest(ACount : Integer);
    Destructor destroy; override;
  end;

Destructor TTest.Destroy;
begin
  Writeln('Вызывается Destroy');
end;

Procedure TTest.DoTest(ACount : Integer);
begin
  Writeln('Тестируем ',ACount,' : счётчик ссылок: ',RefCount);
end;

procedure DoIt1(x: ITest; ACount : Integer);
begin
  // Счетчик ссылок увеличивается
```

```
    x.DoTest(ACount);
    // И уменьшается
  end;

  procedure DoIt2(const x: ITest; ACount : Integer);
  begin
    // Счетчик ссылок не меняется.
    x.DoTest(ACount);
  end;

  Procedure Test1;
  var
    y: ITest;
  begin
    y := TTest.Create;
  ..// Счётчик ссылок в этот момент равен 1.
    y.DoTest(1);
    // При входе в DoIT счётчик ссылок увеличивается и
  уменьшается при выходе.
    DoIt1(y,2);
    // Ещё один счетчик ссылок.
    y.DoTest(3);
  end;

  Procedure Test2;
  var
    Y : TTest;
  begin
    Y := TTest.Create; // На объект еще нет счетчика ссылок
    // В этой точке счётчик ссылок равен 0.
    y.DoTest(3);
    // Счетчик ссылок остаётся нулевым.
    DoIt2(y,4);
    Y.DoTest(5);
    Y.Free;
  end;

  Procedure Test3;
  var
    Y : TTest;
  begin
    Y := TTest.Create; // На объект еще нет счетчика ссылок
    // В этой точке счётчик ссылок равен 0.
    y.DoTest(6);
    // Счетчик ссылок остаётся нулевым.
```

```
    DoIt1(y,7);
    y.DoTest(8);
  end;

  begin
    Test1;
    Test2;
    Test3;
  end.
```

Этот пример выведет:

```
Тестируем 1 : счётчик ссылок: 1
Тестируем 2 : счётчик ссылок: 2
Тестируем 3 : счётчик ссылок: 1
Вызывается Destroy
Тестируем 3 : счётчик ссылок: 0
Тестируем 4 : счётчик ссылок: 0
Тестируем 5 : счётчик ссылок: 0
Вызывается Destroy
Тестируем 6 : счётчик ссылок: 0
Тестируем 7 : счётчик ссылок: 1
Вызывается Destroy
Тестируем 8 : счётчик ссылок: 0
```

Как можно видеть, в `test3`, счетчик ссылок уменьшается с **1** до **0** в конце вызова `Doit`, в результате экземпляр будет освобожден до возвращения из подпрограммы.

Следующая небольшая программа демонстрирует счётчик ссылок, используемый в строках:

```
{$mode objfpc}
{$H+}

// Вспомогательная функция для извлечения счётчика ссылок.
function SRefCount(P : Pointer) : integer;
Type
  PAnsiRec = ^TAnsiRec;
  TAnsiRec = Record
    CodePage    : TSystemCodePage;
    ElementSize : Word;
  {$ifdef CPU64}
  { align fields  }
    Dummy       : DWord;
  {$endif CPU64}
    Ref         : SizeInt;
    Len         : SizeInt;
  end;
```

```
begin
  if P=Nil then
    Result:=0
  else
    Result:=PAnsiRec(P-SizeOf(TAnsiRec))^.Ref;
end;

Procedure ByVar(Var S : string);
begin
  Writeln('Счётчик ссылок на Переменную : ',SRefCount(Pointer
(S)));
end;

Procedure ByConst(Const S : string);
begin
  Writeln('Счётчик ссылок на Константу : ',SRefCount(Pointer
(S)));
end;

Procedure ByVal(S : string);
begin
  Writeln('Счётчик ссылок на Значение : ',SRefCount(Pointer
(S)));
end;

Function FunctionResult(Var S : String) : String;
begin
  Writeln('Аргумент  функции, со счётчиком ссылк : ',SRefCount
(Pointer(S)));
  Writeln('Результат функции, со счётчиком ссылк : ',SRefCount
(Pointer(Result)));
end;

Var
  S,T : String;

begin
  S:='Некоторая строка';
  Writeln('Контанта          : ',SrefCount(Pointer(S)));
  UniqueString(S);
  Writeln('Однозначный       : ',SRefCount(Pointer(S)));
  T:=S;
  Writeln('После присвоения : ',SRefCount(Pointer(S)));
  ByVar(S);
  ByConst(S);
  ByVal(S);
  UniqueString(S);
  T:=FunctionResult(S);
  Writeln('После фанкции     : ',SRefCount(Pointer(S)));
end.
```

14.6 Перегрузка функций

Перегрузка функций означает, что функция определена более одного раза, но с разными списками формальных параметров. Списки параметров должны отличаться, типом хотя бы одного элемента. Когда компилятор встречает вызов функции, он просматривает параметры функции, чтобы решить, какую из определенных функций он должен вызывать. Это полезно, когда одна и та же функция должна быть определена для разных типов. Например, в RTL, процедура `Dec` определена следующим образом:

```
...
Dec(Var I : Longint;decrement : Longint);
Dec(Var I : Longint);
Dec(Var I : Byte;decrement : Longint);
Dec(Var I : Byte);
...
```

Когда компилятор встречает вызов функции `Dec`, он сначала ищет какую функцию он должен использовать. Поэтому он проверяет параметры в вызове функции, и смотрит, какие функции определены, выбирая ту список параметров которой совпадает. Когда компилятор находит такую функцию, она и вызовется. Если такой функции нет, генерируется ошибка компиляции.

Функции, имеющие модификатор `cdecl` не могут быть перегружены. (С технической точки зрения, компилятор встречая этот модификатор, изменяет имя функции).

До версии компилятора **1.9**, *перегруженные* функции необходимо было искать в том же модуле. Теперь компилятор будет продолжать поиск в других модулях, если присутствует ключевое слово overload, и если он не находит соответствующую версию *перегруженной* функции в одном модуле,

Если нет ключевого слова `overload`, то все перегруженные версии функций должны находиться в том же модуле, методы в классе должны быть определены в том же классе, *(т.е. компилятор не будет искать перегруженные методы в родительских классах, если не указано ключевого слова `overload`).*

14.7 Forward объявление подпрограмм

Функция может быть объявлена без её последующей его реализации, если за ней следует `forward`. Реализация этой функции должна быть далее в этом же модуле. Эта функция может быть вызвана сразу после ***предварительного*** объявления, как если бы она была уже реализована. Ниже приведен пример ***предварительного*** объявления.

```
Program testforward;

Procedure First (n : longint); forward;
```

```
Procedure Second;
begin
  WriteLn ();
  First (1);
end;

Procedure First (n : longint);
begin
  WriteLn (,n);
end;

begin
  Second;
end.
```

Функция может быть ***предварительно*** объявлена только один раз. В модуле ***не нужно*** делать ***предварительное*** объявленной функции *(или процедуры)*, которая уже была объявлена в интерфейсной части этого модуля.Декларация интерфейса считается `forward` декларации. Компиляция следующего модуля сгенерирует ошибку:

```
Unit testforward;

interface

Procedure First (n : longint);
Procedure Second;

implementation

Procedure First (n : longint); forward; //!!!!!!

Procedure Second;
begin
  WriteLn ('В процедуре second вызывается процедура first...');
  First (1);
end;

Procedure First (n : longint);
begin
  WriteLn ('Процедера First получила : ',n);
end;

end.
```

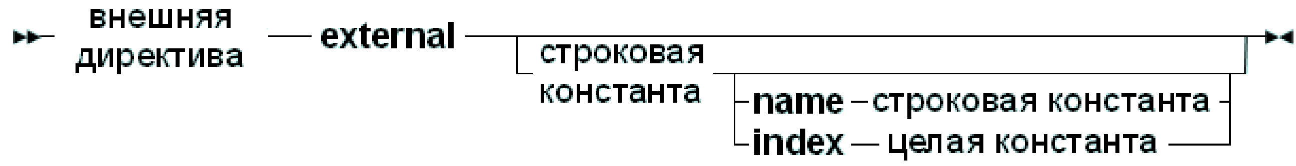
И наоборот, функции, объявленные в секции интерфейса ***не должны*** быть

предварительно объявлены в секции реализации. Так как они ***уже были объявлены***.

14.8 Внешние (external) функции

Модификатор `external` используется для объявления функции, которая находится во внешнем объектном файле. Это позволяет использовать функцию *(или процедуру)* в каком-либо коде, содержащую реализацию в объектном файле, на момент компиляции они должны быть связаны.

Директива external



Она заменяет, блок кода функции *(или процедуры)*. В качестве примера:

```
program CmodDemo;
{$LINKLIB c}

Const P : PChar = 'Это интересно!';

Function StrLen(P: PChar): Longint;cdecl; external name
'strlen';
begin
  WriteLn ('Длина (',p,') : ',StrLen(p));
end.
```

Примечание:

Параметры объявленные функции C *(у Вас в программе)* должна точно соответствовать параметрам в C объявлении *(в коде C)*.

Если модификатор `external` сопровождается строковой константой:

```
external 'lname';
```

Она говорит компилятору, что функция находится в библиотеке lname. Компилятор будет автоматически связать эту библиотеку с программой.

Имя функции в библиотеке тоже может быть указано:

```
external 'lname' name 'Fname';
```

Сообщим компилятору, что функция находится в библиотеке lname, под именем Fname. Компилятор автоматически свяжет эту библиотеку с

программой, и использует правильное имя функции. Под Windows и os/2 может быть использована и следующая форма:

```
external 'lname' Index Ind;
```

Сообщаем компилятору, что функция находится в библиотеке lname, с индексом Ind. Компилятор будет автоматически связать эту библиотеку с программой, и использовать правильный индекс функции.

Наконец, директиву external можно использовать, для указания имени внешний функции:

```
external name 'Fname';
{$L myfunc.o}
```

Сообщаем компилятору, что функция имеет имя Fname. Правильная библиотека или объектный файл *(в данном случае* myfunc.o*)* также должны быть указаны, гарантируя, что функция Fname будет найдена на стадии компоновки.

14.9 Функции на ассемблере

Функция *(или процедура)* может быть полностью реализована на языке ассемблера. Чтобы это указать, используется ключевое слово assembler:

Функции на ассемблере

блок asm – **assembler** – ; – раздел объявления – оператор asm

---

В отличие от Delphi, ключевое слово assembler нужно использовать для указания функции на *ассемблере*. Для получения дополнительной информации о функциях на ассемблере, смотрите главу об использовании ассемблера в Справочник программиста Free Pascal.

14.10 Модификаторы

Функция *(или процедура)*, при объявлении, может содержать модификаторы. Здесь перечислены разные возможности:

Модификаторы

Free Pascal не поддерживает все модификаторы Turbo Pascal *(хотя и разбирает их для совместимости)*, но поддерживает ряд дополнительных модификаторов. Они используются, в основном, для *Ассемблера* или ссылок на объектные файлы C.

14.10.1 alias

Модификатор `alias` позволяет программисту указать другое имя для процедуры или функции. Это в основном полезно ***для ссылки на эту процедуру*** из конструкций на языке ассемблера или из другого объектного файла. В качестве примера рассмотрим следующую программу:

```
Program Aliases;

Procedure Printit;alias : 'DOIT';
begin
  WriteLn ('Печатает (Псевдоним : "DOIT")');
end;
begin
  asm
    call DOIT
  end;
end.
```

Примечание:

Код с указанным псевдонимом *(alias)* вставляется прямо в ассемблерный код, и ***он чувствителен к регистру***.

Модификатор alias ***не делает*** символ публичным для других модулей, если процедура не будет объявлена в интерфейсной части модуля, или не используется модификатор public, чтобы заявить его, как публичный. Рассмотрим следующий пример:

```
unit testalias;

interface

procedure testroutine;

implementation

procedure testroutine;alias:'ARoutine';
begin
  WriteLn('Hello world');
end;

end.
```

Этот модуль сделает процедуру testroutine доступной и поместит её во внешнем объектном файле под названием ARoutine.

Примечание:

Директива alias считается устаревшей. Вместо неё используйте директиву public name. Смотрите раздел 14.10.12 public [245].

14.10.2 cdecl

Модификатор cdecl используют для объявления функции, которая использует соглашение о вызовах языка C. Его необходимо использовать при обращении к функциям, помещенных в объектный файл генерируемый с помощью стандартного компилятора C, которые нужно использовать и для подпрограмм Pascal, в качестве функций обратного вызова для стандартных библиотек C.

Модификатор cdecl позволяет использовать в коде функцию C. Внешние C функции, реализация которых содержит объектный файл, должны быть связаны. Как пример:

```
program CmodDemo;
{$LINKLIB c}
```

```
Const P : PChar = 'Это интересно!';

Function StrLen(P: PChar): Longint;cdecl; external name
'strlen';
begin
  WriteLn ('Длина (',p,') : ',StrLen(p));
end.
```

При компиляции этого кода, и ссылки на C-библиотеки, функцию `StrLen` можно вызывать по всей программе. Директива `external` сообщает компилятору, что функция постоянно находится во внешнем объектном файле *(или библиотеке)* под именем strlen *(смотри* *14.8 Внешние (external) функции* 237*).*

Примечание:

> Параметры объявленные функции C *(у Вас в программе)* должна точно соответствовать параметрам в C объявлении *(в коде C).*

Для функций, которые ***не являются внешними***, но объявленные с использованием `cdecl`, внешнего связывание ***не требуют***. Эти функции имеют некоторые ***ограничения***, например не могут использоваться конструкции `array of const` *(из-за разницы использования стека)*. Однако модификатор `cdecl` позволяет использовать функции написанные на C, как функции обратного вызова для подпрограмм, что подразумевает что они используют соглашение о вызовах cdecl.

14.10.3 export

Модификатор `export` используется для экспорта имен при создании общей библиотеки или исполняемой программы. Это означает, что символ будет находиться в открытом доступе, и может быть импортирован из других программ. Для получения дополнительной информации об этом модификаторе, обратитесь к разделу ***"Создание библиотек"*** см. Справочник программиста Free Pascal.

14.10.4 inline

Код процедуры объявленной `inline` *(встроенная)* копируются без изменения в то место, где она должна вызываться. Это приводит к тому, что нет фактического вызова процедуры, просто её код копируется туда, где она необходима, это обеспечивает более быструю скорость выполнения, при многократном использовании функции *(или процедуры)*. Очевидно, что встраивание больших функций не имеет смысла.

По умолчанию `inline` процедуры ***не допускается***. Код `inline` можно

использовать только когда есть переключатель командной строки `-Si` или директива `{$inline on}`.

Примечание:

1. `inline` только указание компилятору. Это ***не означает*** что все `inline` вызовы *встраиваются* автоматически; иногда компилятор может решить, что функция просто не может быть *встроена*, или что конкретный вызов функции не может быть *встроен*. Если это так, то компилятор выдаст предупреждение.
2. В старых версиях Free Pascal, *встроенный* код не мог быть экспортирован из модуля. Это означало, что при вызове `inline` процедуры из другого модуля, выполниться ***обычный вызов*** процедуры. Процедуры `inline` действительно *встраиваемы* ***только*** внутри модуля. Начиная с версии **2.0.2**, `inline` работает и из модуля.
3. Рекурсия во *встроенных* функциях ***не допускается***. т.е. не допускается *встроенная* функция, вызывающая сама себя.

14.10.5 interrupt

Ключевое слово `interrupt` нужно для объявления кода, который будет использоваться как обработчик прерываний. При вызове этой процедуры, все регистры будут сохранены, а при выходе восстановлены, таким образом выполнится возврат из прерывания или ловушки *(вместо обычного возврата из подпрограммы инструкции)*.

На платформах, где нет возврата из прерывания, будет выполнен нормальный код выхода из подпрограммы. Для получения дополнительной информации о сгенерированном коде, смотри Справочник программиста Free Pascal.

14.10.6 iocheck

Ключевое слово `iocheck` используется, чтобы объявить подпрограмме, которая вызывает генерацию кода проверки результата ввода/вывода *(всякий раз, когда нужно)* в пределах блока *(где включен переключатель)* `{$IOChecks ON}`.

В результате компилятор вставит код проверки ввода/вывода для процедуры, если она находится в пределах блока `{$IOChecks ON}`.

Этот модификатор предназначен для внутренних процедур RTL, а не для использования в коде приложения.

14.10.7 local

Модификатор `local` нужен компилятору для оптимизации функции: локальная функция не может быть в интерфейсной секции блока: она всегда находится в

секции реализации модуля. Значит, эта функция не может быть экспортированы из библиотеки.

В Linux, директива `local` приводит к некоторой оптимизации. В операционной системе Windows, директива не имеет никакого эффекта. Она была введена для совместимости с Kylix.

14.10.8 noreturn

Модификатор `noreturn` используется, для сообщения компилятору что это процедура *(не возвращает результата)*. Эта информация может использоваться компилятором, чтобы избежать предупреждений о неинициализированных переменных или не установленных результатах.

В следующем примере, компилятор не выдаст предупреждение о том , что результат в функции `F` может быть ***не установлен***:

```
procedure do_halt;noreturn;
begin
  halt(1);
end;

function f(i : integer) : integer ;
begin
  if (i<0)  then
    do_halt // Если выполнится эта ветвь, результат неопределён
  else
    result:=i;
end;
```

14.10.9 nostackframe

Модификатор `nostackframe` используется, чтобы сообщить компилятору что он не должен генерировать кадр стека для этой процедуры или функции. По умолчанию, кадр стека ***всегда*** генерируется для каждой процедуры или функции.

Нужно быть очень осторожным при использовании этого модификатора: большинству процедур или функций нужен стек. В частности, он необходим для отладки.

14.10.10 overload

Модификатор `overload` сообщает компилятору, что функция перегружена. Он используется для совместимости с Delphi, и для Free Pascal, все функции и процедуры могут быть перегружены без этого модификатора.

Существует ***только один случай***, когда модификатор `overload` является обязательным: если функция должна быть перегружена, и находится в другом модуле. Обе функции должны быть объявлены с модификатором `overload`: модификатор `overload` говорит компилятору, что он должен продолжать искать перегруженные версий функций в других модулях.

Следующий пример это иллюстрирует. Возьмем первый модуль:

```
unit ua;

interface

procedure DoIt(A : String); overload;

implementation

procedure DoIt(A : String);
begin
  Writeln('ua.DoIt получил ',A)
end;

end.
```

И второй модуль, содержит перегруженную версию:

```
unit ub;

interface

procedure DoIt(A : Integer); overload;

implementation

procedure DoIt(A : integer);

begin
  Writeln('ub.DoIt получил ',A)
end;

end.
```

Следующая программа использует оба модуля:

```
program uab;

uses ua,ub;

begin
  DoIt('Некоторая строка');
```

```
end.
```

Когда компилятор ищет объявление DoIt, он начнёт с модуля ub. Без директивы overload, компилятор выдаст ошибку несоответствие типа аргумента:

```
home: >fpc uab.pp
uab.pp(6,21) Error: Incompatible type for arg no. 1:
Got "Constant String", expected "SmallInt"

home: >fpc uab.pp
uab.pp(6,21) Ошибка: Несовместимый тип аргумента номер. 1:
Получила "Строковую константу", ожидая "SmallInt"
```

Директивы overload *(в обоих месте местах)*, говорят компилятору, что нужно продолжать поиски перегруженной версии функций для согласования списка параметров. Обратите внимание, что ***оба*** описания должны иметь модификатор overload; ***недостаточно*** иметь модификатор ***только*** в модуле ub. Это делается для предотвращения нежелательной *перегрузки*: программист, который реализовал модуль ua, должен объявить процедуру, как перегружаемую.

14.10.11 pascal

Модификатор pascal используется для объявления функции, которая использует классическое Pascal соглашение о вызовах *(передача параметров слева направо)*. Для получения более подробной информации о Pascal вызовах, смотрите Справочник программиста Free Pascal.

14.10.12 public

Ключевое слово public используется для глобального объявления функции в модуле. Это полезно, если функция ***не должна*** быть доступна из файла модуля *(т.е. другой модуль/программа с использованием модуля* ***не видит*** *функцию)*, но должна быть доступна. В качестве примера:

```
Unit someunit;

interface

Function First : Real;

Implementation

Function First : Real;
begin
  First := 0;
end;
```

```
Function Second : Real; [Public];
begin
  Second := 1;
end;

end.
```

Если другая программа или модуль использует этот модуль, он ***не может*** использовать функцию Second, так как она ***не объявлена*** в интерфейсной части. Тем не менее, можно получить доступ к функции Second на уровне языка ассемблера, используя его "искаженное" имя *(смотрите Справочник программиста Free Pascal)*.

Модификатор public также может сопровождаться директивой name, чтобы указать имя для вызова процедуры на ассемблере, следующим образом :

```
Unit someunit;

interface

Function First : Real;

Implementation

Function First : Real;
begin
  First := 0;
end;

Function Second : Real; Public name 'second';
begin
  Second := 1;
end;

end.
```

При использовании символа нужно учитывать регистр букв, его надо использовать как в public name, то есть имя будет second, строчными буквами.

14.10.13 register

Ключевое слово register используется для совместимости с Delphi. В версии компилятора с **1.0.x**, эта директива не имеет никакого влияния на генерируемый код. В версии **1.9.X**, эта директива тоже поддерживается. Первые три аргумента передаются в регистрах EAX, ECX и EDX.

14.10.14 safecall

Модификатор `safecall` имеет близкое сходство с модификатором `stdcall`. Он загружает параметры в стек *(справа налево)*. Кроме того, вызываемая процедура сохраняет и восстанавливает все регистры.

Более подробную информацию об этом модификаторе можно найти в Справочник программиста Free Pascal, в главе о связывании и разделе о механизме вызова.

14.10.15 saveregisters

Модификатор `saveregisters` сообщает компилятору, что все регистры процессора должны быть сохранены до вызова этой функции. Какие регистры процессора сохраняются, зависит от CPU.

14.10.16 softfloat

Модификатор `softfloat` имеет смысл только на архитектуре процессора ARM.

14.10.17 stdcall

Модификатор `stdcall` загружает параметры в стек *(справа налево)*, также он выравнивает параметры по умолчанию.

Более подробную информацию об этом модификаторе можно найти в Справочник программиста Free Pascal, в главе о связывании и разделе о механизме вызова.

14.10.18 varargs

Этот модификатор может использоваться ***только вместе*** с модификатором `cdecl`, для внешних процедур C. Он указывает что процедура принимает переменное число аргументов ***после последней объявленной переменной***. Эти аргументы передаются без какой-либо проверки типа. Это эквивалентно использованию конструкции `array of const` для процедур `cdecl`, без объявления этой конструкции. Квадратные скобки аргументов-переменных не нужно использовать, когда используется эта форма объявления.

Следующие два способа объявления ссылок на функции библиотеки C:

```
Function PrintF1(fmt : pchar); cdecl; varargs; external 'c'
name 'printf';
Function PrintF2(fmt : pchar; Args : Array of const); cdecl;
external 'c' name 'printf';
```

Но они должны по-разному вызываться:

```
PrintF1('%d %d\n',1,1);
PrintF2('%d %d\n',[1,1]);
```

14.11 Неподдерживаемые модификаторы Turbo Pascal

Модификаторы, существующие в Turbo Pascal, но не поддерживаемые во Free Pascal, приведены в таблице (14.1).

Таблица 14.1: Неподдерживаемые модификаторы

Модификатор	**Почему не поддерживается**
Near	Free Pascal является **32**-битным компилятором.
Far	Free Pascal является **32**-битным компилятором.

При обнаружении этих модификаторов компилятор выдаст предупреждение и проигнорирует их.

Глава 15 Перегрузка операторов

15.1 Введение

Free Pascal поддерживает перегрузку операторов. Это означает, что можно определить действие оператора в зависимости от типа, таким образом разрешить использование этих типов в математических выражениях.

Определение действия оператора во многом напоминает определение функции или процедуры, но есть некоторые ограничения при определении, что будет показано далее.

Перегрузка операторов - мощный инструмент определения; однако эти-же результаты могут быть получены с помощью вызова обычных функций. При использовании перегрузки операторов, важно иметь в виду, что несоблюдение некоторых неявные правил может привести к неожиданным результатам. Что будет показано.

15.2 Объявление оператора

Определение действия оператора во многом напоминает определение функции:

Объявление оператора.

Список параметров для оператора сравнения или арифметического оператора всегда должнен содержать два параметра, за исключением унарного минуса, где необходим только один параметр. Тип результата оператора сравнения должно быть логическим *(Boolean)*.

Примечание:

При компиляции в режиме Delphi или Objfpc, результат идентификатора может быть игнорирован. К результату можно обращаться через стандартный символ Result *(Результат)*.

Если результат отбрасывается и компилятор не находится ни в одном из этих режимов, произойдет ошибка синтаксиса.

Блок оператора содержит необходимые операторы для определения операции. Он может содержать сколь угодно большие куски кода; они выполняется всякий раз, когда в выражении встречается эта операция. Результат блока операторов должен быть определён **всегда**; ошибки не проверяются компилятором, код должен позаботиться о обработке всех возможных случаях ошибок времени выполнения, если они возможны.

Далее будут рассмотрены три типа определений оператора. В качестве примера, для определения перегруженных операторов в этой главе будет использоваться следующий тип:

```
type
```

```
complex = record
  re : real;
  im : real;
end;
```

Этот тип будет использоваться во всех примерах.

Исходники библиотеки времени выполнения содержат два модуля, которые в значительной степени используют перегрузку операторов:

ucomplex
Этот модуль содержит всё для вычисления комплексных чисел.

matrix
Этот модуль содержит всё для вычисления матриц.

15.3 Операторы присваивания

Оператор присваивания определяет действие присвоения переменной одного типа другой *(переменной)*. Тип результата должен совпадать с типом переменной в левой части оператора присваивания, единственный параметр оператора присваивания должен иметь тот же тип, что и выражение справа от оператора присваивания.

Эта система может быть использована, чтобы объявить новый тип, и определить присвоение для этого типа. Например, чтобы иметь возможность присваивать вновь определенный тип 'Complex'

```
Var
    C,Z : Complex; // Определённый ранее тип complex
begin
    Z:=C;  // присвоение одной переменной типа complex другой.
end;
```

Должен быть определен следующий оператор присваивания:

```
Operator := (C : Complex) z : complex;
```

Для того, чтобы присвоить тип real сложному типу *(complex)* можно действовать следующим образом:

```
var
    R : real;
    C : complex;
begin
    C:=R;
end;
```

Должен быть определен следующий оператор присваивания:

```
Operator := (r : real) z : complex;
```

Таким образом определяется оператор присваивания (:=) с переменной типа real

в правой части *(аргумент оператора)* и complex в левой части *(результат оператора)* выражения.

Пример реализации может быть следующим:

```
operator := (r : real) z : complex;
begin
    z.re:=r;
    z.im:=0.0;
end;
```

В данном примере, идентификатор результата *(z)* используется для хранения результата выполнения операции. При компиляции в режимах Delphi или ObjFPC, допускается использование идентификатора Result, или может быть заменен на z *(как идентификатор результата)*, эти два подхода эквивалентны (как показано выше).

```
operator := (r : real) z : complex;
begin
    Result.re:=r;
    Result.im:=0.0;
end;
```

Оператор присваивания используется и для преобразования типов *(из одного типа в другой)*. Компилятор будет просматривать все перегружена операторы присваивания, пока не найдёт соответствия типов с левой и правой стороны выражения. Если такой оператор не найден, генерируется ошибка ***'несоответствие типов'*** *('type mismatch')*.

Замечание:

Оператор присваивания не является коммутативным; компилятор не меняет типов двух аргументов *(источника и назначения)*. Другими словами, если используется приведенное выше определение оператора присваивания, следующее ***не*** возможно:

```
var
    R : real;
    C : complex;
begin
    R:=C;
end;
```

Если обратное присвоение должно быть возможно, то оператор присваивания также должен быть определен. *(Это не так для вещественных и комплексных чисел.)*

Замечание:

Оператор присваивания также используется для неявного преобразования типов. Это может иметь нежелательные последствия. Рассмотрим следующие определения:

```
operator := (r : real) z : complex;
function exp(c : complex) : complex;
```

Тогда следующее присваивание даст несоответствие типов:

```
Var
    r1,r2 : real;
begin
    r1:=exp(r2);
end;
```

Несоответствие происходит потому, что компилятор будет сталкиваться с определением функции `exp` с комплексным аргументом. Компилятор неявно преобразует r2 в complex, поэтому он может использовать описанную выше функцию exp. Результатом этой функции является тип `complex`, который ***не может*** быть назначен на r1, поэтому компилятор выдаст сообщение об ошибке *'несоответствия типов'*. Компилятор не будет дальше смотреть на следующую функцию exp , которая имеет уже правильные аргументы.

Можно избежать этой проблемы, путём уточнения

```
r1:=system.exp(r2);
```

При выполнении явного приведение типов, компилятор будет пытаться выполнить неявное преобразование, если присутствует оператор присваивания. Что означает:

```
Var
    R1 : T1;
    R2 : T2;
begin
  R2:=T2(R1);
```

Будет осуществляться оператором

```
Operator := (aRight: T1) Res: T2;
```

Оператор будет переопределен, и затем он будет использоваться для объявленного типа вместо оператора по умолчанию.

Обратное не верно: В случае обычного присваивания, компилятор не будет рассматривать явные операторы присваивания.

Учитывая следующие определения:

```
uses
    sysutils;
```

```
type
    TTest1 = record
      f: LongInt;
    end;
    TTest2 = record
      f: String;
    end;
    TTest3 = record
      f: Boolean;
    end;
```

Можно создать операторы присваивания:

```
operator := (aRight: TTest1) Res: TTest2;
begin
    Writeln('Неявное TTest1 => TTest2');
    Res.f := IntToStr(aRight.f);
end;

operator := (aRight: TTest1) Res: TTest3;
begin
    Writeln('Неявное TTest1 => TTest3');
    Res.f := aRight.f <> 0;
end;
```

Но можно также определить тип оператора:

```
operator Explicit(aRight: TTest2) Res: TTest1;
begin
    Writeln('Явное TTest2 => TTest1');
    Res.f := StrToIntDef(aRight.f, 0);
end;

operator Explicit(aRight: TTest1) Res: TTest3;
begin
    Writeln('Явное TTest1 => TTest3');
    Res.f := aRight.f <> 0;
end;
```

Таким образом, следующий код

```
var
    t1: TTest1;
    t2: TTest2;
    t3: TTest3;
begin
    t1.f := 42;
    // Неявное
```

```
    t2 := t1;
    // теоретически явная, но будет использоваться неявная
операция,
    // потому что никакого явного оператора не определено
    t2 := TTest2(t1);
    // следующий код не будет компилировать,
    // т.к. не определён ни один оператор присваивания
    //(явное присваивание здесь использоваться не будет)
    //t1 := t2;
    // Явное
    t1 := TTest1(t2);
    // Первое преобразование явное (TTest2 => TTest1),
    // следующее преобразование неявное (TTest1 => TTest3)
    t3 := TTest1(t2);
    // Неявное
    t3 := t1;
    // Явное
    t3 := TTest3(t1);
end.
```

будет выводить:

```
Неявное TTest1 => TTest2
Неявное TTest1 => TTest2
Явное TTest2 => TTest1
Явное TTest2 => TTest1
Неявное TTest1 => TTest3
Неявное TTest1 => TTest3
Явное TTest1 => TTest3
```

15.4 Арифметические операторы

Арифметические операторы по сути бинарный оператор. Возможные операции:

умножение

чтобы умножить два аргумента нестандартного типа, оператор умножения ***(*)*** должен быть перегружен.

деление

для того, чтобы разделить два аргумента нестандартного типа, оператор деления ***(/)*** должен быть перегружен.

сложение

чтобы сложить два аргумента нестандартного типа, оператор сложения ***(+)*** должен быть перегружен.

вычитание

чтобы вычесть аргумент нестандартного типа из др. аргумента, оператор вычитания ***(-)*** должны быть перегружен.

возведение в степень *(exponentiation)*

Для возведения аргумента в степень другого аргумента нестандартного типа, оператор возведения в степень ***(**)*** должен быть перегружен.

унарный минус

используется, чтобы взять отрицательное значение аргумента следующий за ним *(это оператор одного аргумента)*.

симметричная разность

Для вычисления симметрической разности 2 структур, оператор ***(><)*** должен быть перегружен.

Арифметический оператор принимает два параметра, за исключением унарного минуса, для который требуется только один параметр. Первый параметр должен иметь тип, который соответствует аргументу в левой части *(до)* оператора, второй параметр должен быть того типа, который находится справа *(после)* от арифметического оператора. Тип результата, должен соответствовать типу результаты арифметической операции.

Для компиляции следующего кода

```
var
  R : real;
  C,Z : complex;

begin
  C:=R*Z;
end;
```

Нужно определить оператор умножения так:

```
Operator * (r : real; z1 : complex) z : complex;
begin
  z.re := z1.re * r;
  z.im := z1.im * r;
end;
```

Как можно видеть, первый аргумент оператора имеет тип `real`, а второй имеет тип `complex`. Тип результата является `complex`.

Умножение и сложение вещественных и комплексных чисел коммутативные *(можно переставлять аргументы)* операции. Компилятор, однако, об этом не имеет ни малейшего представления, так что даже если оператор умножения вещественного на комплексное число определен, то компилятор не будет использовать это определение для умножения комплексного числа на вещественное, он использует определённый оператор для аргументов указанных только в таком порядке. Для ***коммутативности*** необходимо определить обе операции *(оператор с разными комбинациями аргументов)*.

Таким образом, учитывая приведенное выше определение оператора умножения, компилятор не примет следующий порядок следования

аргументов:

```
var
  R : real;
  C,Z : complex;
begin
  C:=Z*R;
end;
```

Поскольку типы Z и R ***не совпадают*** с типами в определении оператора.

Причина такого поведения заключается в том, что вполне возможно, умножение ***не всегда коммутативно***. Например , умножение матрицы (n, m) на матрицу (m, n) приведет к матрице (n, n), в то время как результат умножение матрицы (m, n) на матрицу (n, m) представляет собой матрица (m, m), т.е. операция не может быть одинаковой во всех случаях.

15.5 Операторы сравнения

Оператор сравнения может быть перегружен, чтобы сравнить два различных или два одинаковых типа, не являющихся стандартными. Тип результата оператора сравнения всегда будет boolean.

Могут быть перегруженными следущие операторы сравнения:

равно
(=) Чтобы определить, что два аргумента равны.

неравно
(<>) Чтобы определить, что два аргумента различны.

меньше
(<) Чтобы определить, что один аргумент меньше другого.

больше
(>) Чтобы определить, что один аргумент больше другого.

больше или равно
(>=) Чтобы определить, что один аргумент больше или равен другому.

меньше или равно
(<=) Чтобы определить, что один аргумент больше или равен другому.

Если нет отдельного оператора ***неравно (<>)***, чтобы оценить ***на неравенство*** выражение, компилятор использует оператор ***равно (=)***, и логически отрицает результат операции. Обратное не верно: если нет оператора ***"равно"***, а оператор ***"неравно"*** существует, то компилятор **не будет** использовать его, чтобы оценить выражение, содержащее оператор ***равно (=)***.

В качестве примера, следующий оператор позволяет сравнивать два комплексных числа:

```
operator = (z1, z2 : complex) b : boolean;
```

приведенное выше определение позволяет сделать сравнение следующего вида:

```
Var
  C1,C2 : Complex;

begin
  If C1=C2 then
    Writeln('C1 и C2 равны');
end;
```

Определение оператора сравнения требует двух параметров, с типами, которые определены при объявлении оператора. Компилятор не применяет коммутативности *(не переставляет аргументы)*: если аргументы двух различных типов, то необходимо определить два оператора сравнения.

В случае комплексных чисел, например, необходимо определить два оператора сравнения: один с аргументом комплексного типа впереди, потом вещественного типа, второй оператор - наоборот *(сначала вещественного, потом комплексного типа)*.

С учетом определения операторов

```
operator = (z1 : complex; r : real) b : boolean;
operator = (r : real; z1 : complex) b : boolean;
```

возможны следующие два сравнения:

```
Var
  R,S : Real;
  C : Complex;
begin
  If (C=R) or (S=C) then
   Writeln ('Ok');
end;
```

Обратите внимание, что порядок аргументов вещественных и комплексных типов в двух сравнениях, меняется.

15.6 Оператор In

Начиная с версии **2.6** Free Pascal, оператор `in` тоже может быть перегружен. Первый аргумент `in` оператора должен быть операндом слева от ключевого слова `in`. Следующий пример показывает как оператор `in` перегружен для записей:

```
{$mode objfpc}{$H+}

type
  TMyRec = record A: Integer end;

operator in (const A: Integer; const B: TMyRec): boolean;
begin
```

```
    Result := A = B.A;
  end;

  var
    R: TMyRec;
  begin
    R.A := 10;
    Writeln(1 in R); // false
    Writeln(10 in R); // true
  end.
```

Оператор `in` может быть перегружен и для других *(не порядковых)* типов, как показано в следующем примере:

```
{$mode objfpc}{$H+}

type
  TMyRec = record A: Integer end;

operator in (const A: TMyRec; const B: TMyRec): boolean;
begin
  Result := A.A = B.A;
end;

var
  S,R: TMyRec;
begin
  R.A := 10;
  S.A:=1;
  Writeln(S in R); // false
  Writeln(R in R); // true
end.
```

Глава 16 Программы, модули, блоки

Программа на Pascal может состоять из модулей, называемых units. `Unit` *(модуль)* может быть использован для группировки кусков кода, или что-бы отдать кому-то код, без указания источников. Программы и модули состоят из блоков кода, которые являются смесью операторов, процедур, объявления переменных и типов.

16.1 Программы

Программа на Pascal состоит из заголовка программы *(после ключевого слова* `program`*)*, затем может быть `uses` *(с перечислением подключаемых модулей)*, далее должен быть ***программный блок*** *(блок* `begin..end`*)*.

Программы

```
►► программа –заголовок программы–;–┬──────────┬–блок–.–────────►◄
                                    └пункт uses┘
►► заголовок программы–program–идентификатор–┬──────────────────────────┬►◄
                                             └(–параметры программы–)─┘
►► параметры программы–список идентификаторов ───────────────────►◄
►► пункт uses–uses–┬–идентификатор–┬───────────────────────┬┬–;───►◄
                   │               └in–строковый литерал┘│
                   └─────────────────,─────────────────────┘
```

Заголовок программы нужен для обратной совместимости, и игнорируется компилятором.

Пункт `uses` служит для идентификации всех модулей, которые необходимы программе. Все идентификаторы, объявленные в интерфейсной части модулей, перечисленных в пункте `uses`, добавляются к идентификаторов самой программы. Модуль system не должен быть в этом списке, так как он всегда загружается компилятором.

Порядок, в котором перечислены модули является значимым, он определяет, в каком порядке они инициализируются. Модули инициализируются в том же порядке, как они появляются в пункте `uses`. Идентификаторы просматриваются в ***обратном порядке***, то есть, когда компилятор ищет идентификатор, то он сначала смотрит в последнем модуле указанном в `uses`, потом в предпоследнем, и так далее. Это ***очень важно*** когда, два модуля объявляют различные типы с тем же идентификатором *(именем)*.

Компилятор будет искать *скомпилированные* или *исходные* версий всех модулей перечисленных в `uses`. Если имя файла модуля было явно указано,

используя ключевое слово in, модуль берется из указанного файла:

```
program programb;

uses unita in '..\unita.pp';
```

Исходный файл модуля unita ищется в родительском каталоге programb.

Когда компилятор ищет файлы модулей, он добавляет расширение .ppu к имени модуля. В операционных системах Linux, где имена файлов чувствительны к регистру, при поиске модуля используется следующий механизм:

1. Сначала модуль ищется когда все буквы имени в неизменном регистре.
2. Модуль ищется когда все буквы имени строчные.
3. Модуль ищется когда все буквы имени заглавные.

Кроме того, если имя модуля больше, чем **8** символов, компилятор сначала ищет модуль с именем этой длинны, а затем усекает имя до **8** символов и снова ищет. Из соображений совместимости, это верно и на платформах, которые поддерживают длинные имена файлов.

Обратите внимание, что по указанной выше схеме, выполняется поиск в каждом каталоге путей поиска.

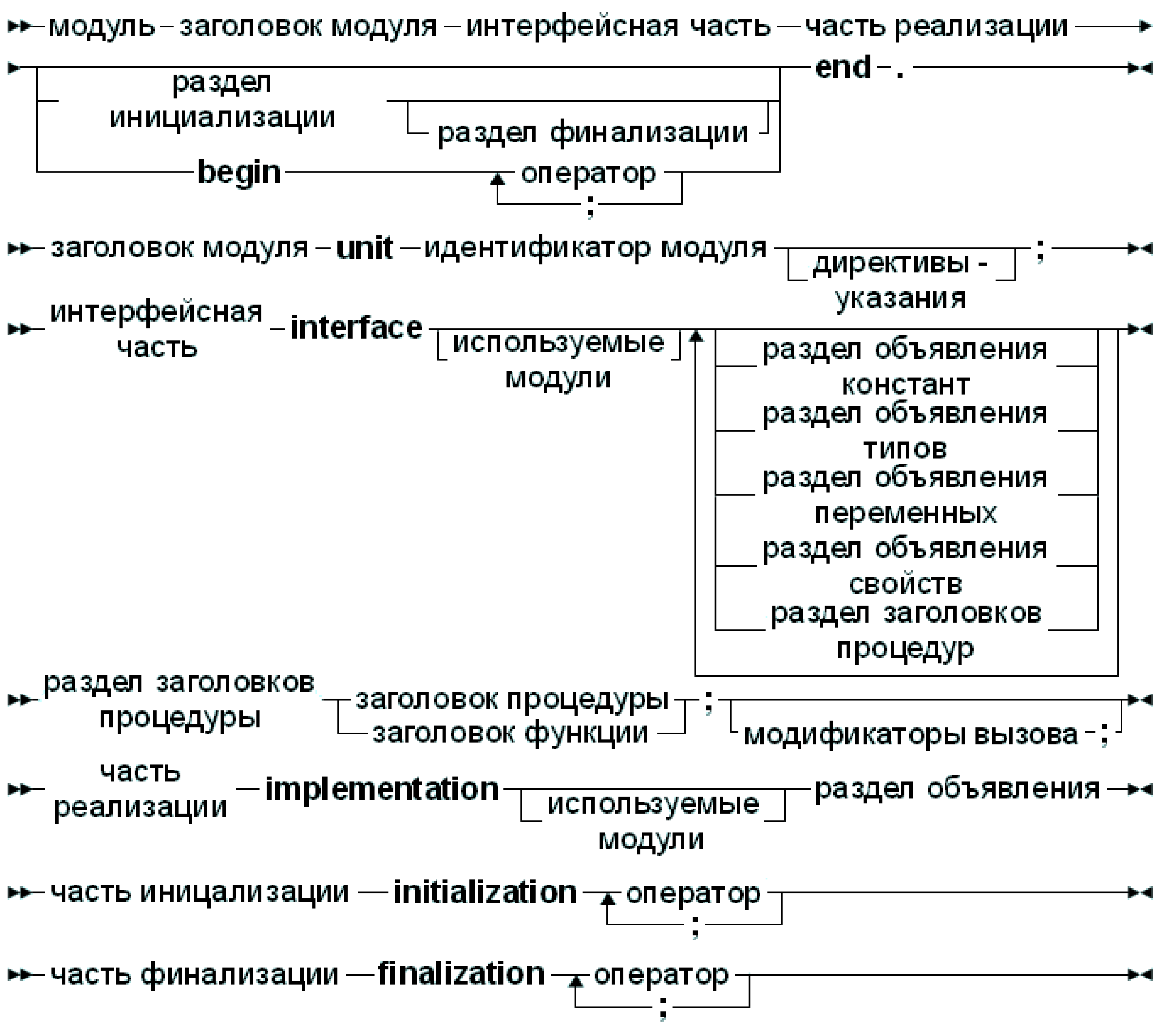
Программный блок содержит операторы, которые ***последовательно*** будут выполняться при запуске программы. Однако, это могут быть не первые операторы програмного блока: код ***инициализации модулей*** может тоже содержать операторы, ***которые выполняются до кода программы***.

Структура ***блока программы*** обсуждается ниже.

16.2 Модули

Модуль содержит набор объявлений, процедур и функций, которые могут использоваться программой или другим модулем. Синтаксис модуля выглядит следующим образом:

Модули



Как видно из синтаксической диаграммы, модуль ***всегда*** состоит из частей `interface` *(интерфейса)* и `implementation` *(реализации)*. И ***необязательных*** блоков `initialization` *(инициализации)* и `finalization` *(финализации)*, содержащие код, выполняющийся соответственно при запуске программы, и при её остановке.

Оба части *интерфейса* и *реализации* могут быть пустыми, но должны быть указаны ключевыми словами `interface` и `implementation`. Ниже приводится правильное определение модуля;

```
unit a;

interface

implementation

end.
```

Интерфейсная часть объявляет все идентификаторы *(имена)*, экспортируемые из модуля. Они могут быть объявлением константы, типы, переменные, процедуры и функции. *Интерфейсная* часть не может содержать исполняемый код: допускается ***только*** объявления. Ниже приведена ***правильная*** *интерфейсная* часть:

```
unit a;

interface

uses b;

Function MyFunction:SomeBType;

Implementation
```

Тип SomeBType определяется в модуле b.

Все функции и методы, объявленные в *интерфейсной* части, должны быть реализованы в части *реализации* модуля, за исключением объявленных внешних функций или процедур. Если объявленный метод или функция не реализована, компилятор выдаст сообщение об ошибке, например:

```
unit unita;

interface

Function MyFunction:Integer;

implementation

end.
```

Приведет к следующей ошибке:

```
unita.pp (5,10)Error:Forward declaration not solved "MyFunc-
tion:SmallInt;"
unita.pp (5,10) Ошибка: Предварительная декларация не
реализована "MyFunction:SmallInt;"
```

Часть *реализации* в основном предназначена для реализации процедур и функций объявленных в *интерфейсной* части. Тем не менее, она тоже может содержать собственные объявления: объявленные внутри части *реализации* ***не доступны*** за пределами модуля.

Части initialization *(инициализации)* и finalization *(финализации)* модуля ***не обязательны***.

Блок *инициализации* используется для инициализации некоторых переменных или выполнения кода, необходимый для правильного функционирования модуля. Части *инициализации* модулей выполняются в порядке, в котором

компилятор применяет модули при компиляции программы. Они выполняются до первого оператора программы.

Части *финализации* модулей выполняются в порядке ***обратном*** порядку *инициализации*. Они используются для очистки ресурсов, выделенных в части инициализации модуля, или в процессе работы программы. Часть *финализации* ***всегда выполняется*** в случае нормального завершения программы: независимо от того что в коде программы достигается окончательное `end`, или потому, что была где-то выполнена инструкция `Halt`.

Если программа останавливается во время выполнения блоков *инициализации* одного из модулей, будут завершены ***только*** модули, ***которые уже были инициализированы***. В отличии от Delphi, в Free Pascal блок `finalization` может присутствовать ***без*** блока `initialization`. Это означает, что следующий код будет скомпилирован во Free Pascal, но не в Delphi.

```
Finalization
  CleanupUnit;
end.
```

Раздел *инициализации* сам по себе *(то есть без доработки)* можно заменить блоком операторов. То есть, следующее:

```
Initialization
  InitializeUnit;
end.
```

полностью эквивалентно

```
Begin
  InitializeUnit;
end.
```

16.3 Namespaces: Уточнение модуля

Из диаграммы синтаксиса для модуля видно что, имя модуля может содержать ***уточнение***. Это означает что модули могут быть организованы в ***пространстве имен***.

Тогда, следующее определение модуля правильно:

```
unit a.b;

interface

Function C : integer;

implementation

Function C : integer;
begin
```

```
    Result:=1;
  end;

end.
```

Модуль можно использовать так:

```
program d;

uses a.b;

begin
  Writeln(c);
end.
```

При использовании символических имён, ***уточнение модуля*** всегда имеют приоритет над символическими именами внутри модуля.

Тогда следующие определения модулей:

```
unit myunit;

interface

var
  test: record
    a: longint;
  end;

implementation

initialization
  test.a:=2;
end.
```

и

```
unit myunit.test;

interface

var
  a: longint;

implementation

initialization
  a:=1;
end.
```

Программа ***всегда*** обращается к переменной `a` в модуле `myunit.test`, как `myunit.test.a`:

```
uses
   myunit, myunit.test; //!!!

begin
  Writeln('myunit.test.a : ',myunit.test.a);
end.
```

Она выведет:

```
myunit.test.a: 1
```

Изменение порядка объявления модулей не изменит *уточнённого обращения* к переменной.

```
uses
   myunit.test, myunit; //!!! Порядок изменён

begin
  Writeln('myunit.test.a : ',myunit.test.a);
end.
```

будет всегда выводить

```
myunit.test.a : 1
```

Аналогично, следующая программа таким образом `myunit.test.a` обращается к переменной `a` в модуле `myunit.test` *(т.к. в модуле `myunit` нет переменной `a`)*:

```
uses
   myunit.test, myunit;

begin
  Writeln('a : ',a);
end.
```

она выведет:

```
a : 1
```

Аналогично, следующая программа таким образом `test.a` обращается к переменной *(внутри записи)* `test.a` в модуле `myunit`:

```
uses
   myunit.test, myunit;

begin
  Writeln('test.a : ',test.a);
end.
```

выведет

```
test.a : 2
```

16.4 Зависимость модулей

Когда программа использует модуль *(скажем unitA)*, а этот модуль тоже использует модуль *(скажем unitB)* , то программа косвенно зависит также от unitB. Это означает, что компилятор должен иметь доступ к unitB при попытке компиляции программы. Если модуль ***не присутствует*** во время компиляции, то возникает ошибка.

Обратите внимание, что ***косвенная зависимость*** программы ещё ***не определяет*** что идентификаторы модуля ***являются доступными***. Для того, чтобы иметь доступ к идентификаторам модуля, модуль должен находиться в пункте `uses` программы или модуля, где необходимы эти идентификаторы.

Модули могут быть ***взаимно зависимы***, то есть они могут ссылаться друг на друга при использовании *(uses)*. Это допускается, при условии, что по крайней мере одна из ссылок сделана в разделе `implementation` *(реализации)* модуля. Это справедливо для взаимозависимых ***модулей*** *(косвенным образом)*.

Компилятор выдаст ошибку, если оба модуля *используют (uses)* друг друга в разделе `interface`. Следующее объявление не допускается:

```
Unit UnitA;
interface
Uses UnitB;
implementation
end.


Unit UnitB
interface
Uses UnitA;
implementation
end.
```

А это, разрешено

```
Unit UnitA;
interface
Uses UnitB;
implementation
end.


Unit UnitB
implementation
Uses UnitA;
end.
```

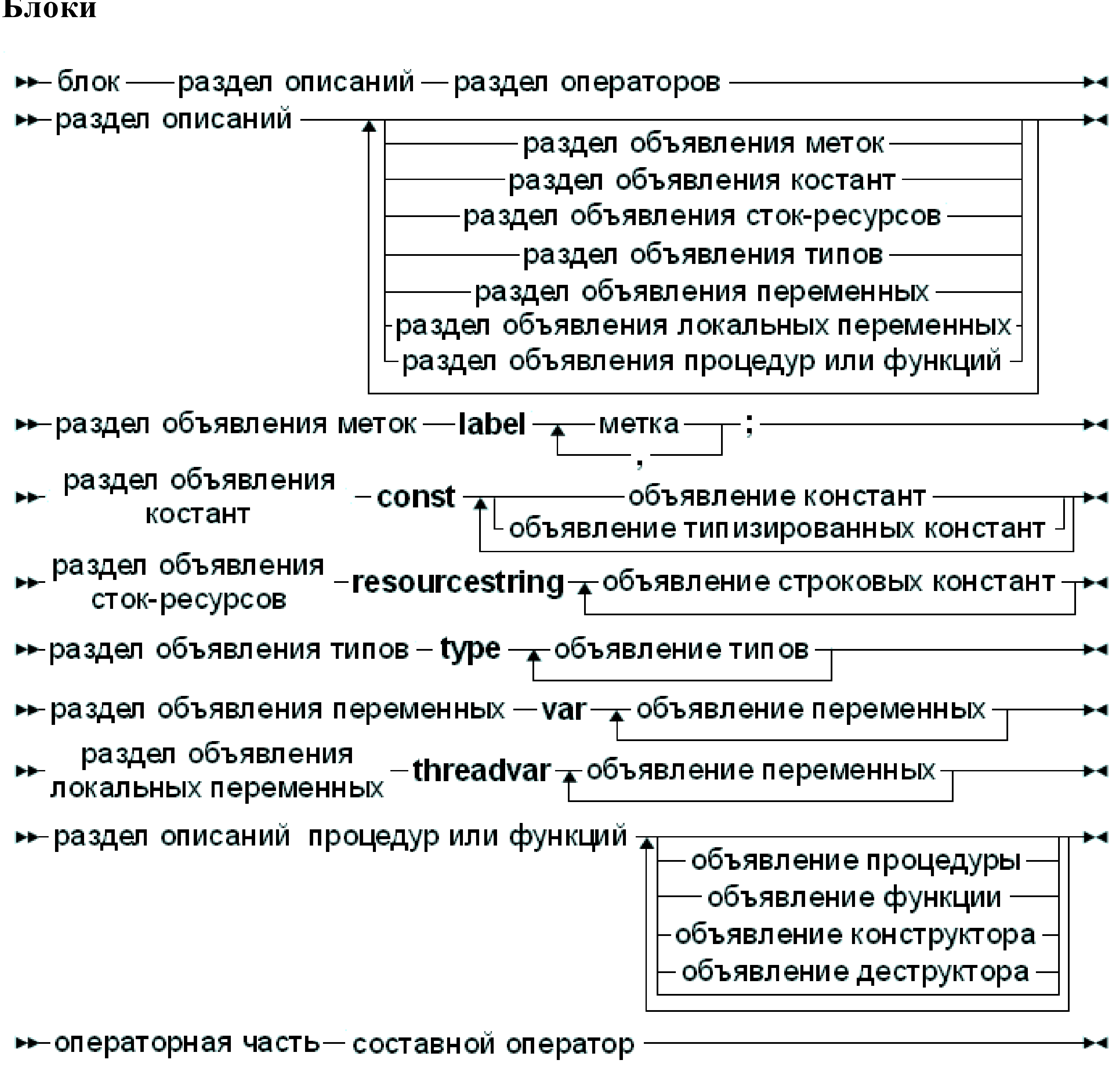
Поскольку UnitB использует UnitA только в разделе *реализации*.

В общем, плохая идея, иметь взаимозависимые модули, даже если это только в разделах реализации!!! *(это, по возможности, следует избегать)*

16.5 Блоки

Модули и программы состоят из ***блоков***. Блок состоит из объявления меток, констант, типов, переменных, функций и процедур. Блоки могут быть вложенными определенным образом, то есть, объявление процедуры или функции могут иметь блоки сами по себе. Блоки могут быть вложены некоторым способом, *т.е. объявление процедур или функций тоже может иметь блоки*. Блок выглядит следующим образом:

Блоки



Метки, которые могут быть использованы для идентификации операторов в блоке, ***объявление меток*** - часть блока. Метка может определить только один

оператор.

Если ***константы*** будут применены только в этом блоке, они должны быть объявлены в части *(этого блока)* ***объявления констант***.

Если ***переменные*** будут применены только в этом блоке, они должны быть объявлены в части *(этого блока)* ***объявления переменных***.

Если ***типы*** будут применены только в этом блоке, они должны быть объявлены в части *(этого блока)* ***объявления типов***.

И наконец, если ***функции*** или ***процедуры*** будут применены только в этом блоке, они должны быть объявлены в части *(этого блока)* ***объявления функций/процедур***.

Части этих четырёх объявлений могут быть смешаны *(порядок не определён)*, но вы не можете использовать идентификаторы *(или ссылки)*, которые еще не были объявлены.

Только после того, как были сданы все объявления, можно переходить к ***операторной части***. Эта часть содержит какие-либо действия, которые должен выполнить блок. Все идентификаторы, объявленные до ***операторной части*** части могут быть использованы в этом ***операторной части***.

16.6 Область действия

При объявлении идентификатора он действителен до конца блока, в котором он объявлен. Диапазон, в котором действует идентификатор - является ***областью действия*** идентификатора. Точная ***область действия*** идентификатора зависит от того, как и где он был определён.

16.6.1 Область действия блока

Область действия переменной, декларированной в части объявлений блока, вступает в силу с момента объявления и до конца блока. Если блок содержит другой блок, в котором повторно объявлен идентификатор, то внутри этого блока, будет действительна другое объявление. При выходе из вложенного блока, снова действительна первоначальное объявление. Рассмотрим следующий пример:

```
Program Demo;
Var X:Real;
{ X является вещественной переменной}
Procedure NewDeclaration
Var X:Integer; {Переобъявляем X как целое}
begin
//X:=1.234;{При попытке компиляции, даст сообщение об ошибке}
X:=10; {Правильное присвоение}
```

```
end;
{Здесь и далее X снова вещественное}
begin
X:=2.468;
end.
```

В этом примере, внутри процедуры, переменная X будет целой. Она имеет свое собственное место для хранения, независимое от переменной X вне процедуры.

16.6.2 Область действия записи

Идентификаторы, определенные внутри поля записи, действительны в следующих местах:

1. До конца определения записи.
2. Поле с указанием *(designators)* переменной записи данного типа.
3. Идентификаторы внутри оператора `with` , который работает на переменную записи данного типа.

16.6.3 Область действия класса

Идентификатор компонента *(один из элементов в списке компонента класса)* действует в следующих местах:

1. С точки объявления до конца определения класса.
2. Во всех потомках класса этого типа, если он не находится в `private` части объявления класса.
3. Во всех блоках объявления метода этого класса и блоках потомков класса.
4. В операторе `with`, для ссылки на операторы и переменные определённые в данном классе.

Обратите внимание, что метод с указанием *(designators)* тоже считается идентификатором.

16.6.4 Область действия модуля

Все идентификаторы в *интерфейсной* части модуля действительны с точки объявления и до конца модуля. Кроме того, идентификаторы определены в программах или модулях, которые используют модули в пункте `uses`.

Идентификаторы *косвенно зависимых* модулей ***не доступны***. Идентификаторы, объявленные в части *реализации* модуля действительны с точки объявления и до конца модуля.

Модуль system автоматически используется во всех модулях и программах. Поэтому его идентификаторы всегда известны, в каждой Pascal программе, библиотеке или модуле.

Правила области действия модуля подразумевают, что идентификатор модуля

может быть переопределен. Для того, чтобы иметь доступ к идентификатору другого модуля, который был переопределён в текущем модуле, перед идентификатором нужно указать имя модул *(через точку)*, как показано в следующем примере:

```
unit unitA;

interface

Type
  MyType=Real;

implementation
end.


Program prog;
Uses UnitA;
{ Переопределим MyType}
Type MyType=Integer;
Var
  A:Mytype; {Будет целым}
  B:UnitA.MyType {Будет вещественным}
begin
end.
```

Это полезно, когда переобъявляются идентификаторы модуля system *(а надо использовать первоначальные)*.

16.7 Libraries (Библиотеки)

Free Pascal поддерживает создание динамических библиотек (DLL под Win32 и OS/2), которые потом будут источником процедур и функций, для создания библиотеки используется ключевое слово `Library`.

Библиотека может быть как отдельная единица или как часть программы:

Библиотеки

```
►► библиотека - заголовок библиотеки -;─┬──────────────────────┬─ блок -.─►◄
                                        └ используемые модули ┘
►► заголовок библиотеки — library — идентификатор ─────────────────────────►◄
```

По умолчанию, функции и процедуры, которые объявлены и реализованы в библиотеке не доступны для программисту, который хочет использовать эту библиотеку.

Для того, чтобы использовать функции или процедуры, из библиотеки, они должны быть экспортированы в части экспорта:

Часть экспорта

```
▸▸─использование экспорта ─ exports ─ список экпорта ─ ; ──────────────▸◂

▸▸─ список экпорта ────────────┬─ ввод экпорта ─┬──────────────────▸◂
                               └────── , ───────┘

▸▸─ввод экпорта ─ идентификатор ─┬──────────────────────────┬┬───────────────────────────┬─▸◂
                                 └ index ─ целая константа ─┘└ name ─ строковая константа ┘
```

Под Win32, в раздел экспорта могут быть добавлены индексы. Индексы должны быть положительные числа большие или равные **1**, и меньше `MaxInt`.

Экспортная запись может иметь имя (спецификатор). Если оно присутствует, имя-спецификатор *(чувствительное к регистру)*, с помощью которого функция будет экспортироваться из библиотеки.

Если ни одина из этих конструкций не присутствует, функция или процедура будет экспортироваться с точным именем, указным в разделе экспорта.

Глава 17 Исключения

Исключения предоставляют удобный способ обработки ошибок и механизмы устранения ошибок, и тесно связаны с классами. Исключения поддерживаются как конструкции трёх видов:

Raise

Raise - вызывает исключение. Это обычно делается, чтобы сигнализировать наличие ошибки. Также можно использовать, чтобы прервать выполнение и немедленно вернуться к известной точке исполняемого файла.

Try ... Except

Блок Try служит для отлова исключений, сгенерированных в его рамках. Если исключение происходит, то выполняется часть Except. Она должна предоставить код ***«восстановления после исключения»***.

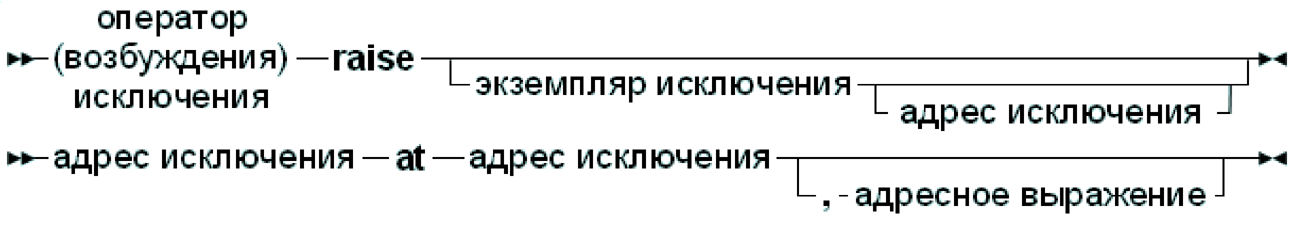
Try ... Finally

В случае возникновения исключения в блоке Try, последовательное исполнение блока будет прервана и выполнен блок Finally. *(Блок **Finally** будет выполнен **всегда**, и в случае нормального исполнения, и в случае возникновения исключения)*. Как правило, они служат для очистки памяти или закрыть файлы, а так-же для освобождения ресурсов в случае возникновения исключения. Кроме того компилятор генерирует неявные блоки Try...Finally вокруг процедур, чтобы согласовать память.

17.1 Оператор Raise

Оператор Raise выглядит следующим образом:

Оператор *Raise*



Этот оператор вызывает исключение. Если он используется, экземпляр исключения должен быть инициализирован как экземпляр любого класса, который является типом допустимым Raise. Адрес исключения и адрес области можно опустить. Если они не указаны, компилятор сообщит адрес сам по себе. Если экземпляр исключения опущен, то текущее исключение будет вновь инициировано. Эта конструкция может быть использована только в блоке обработки исключений *(см 17.2 Операторы try...except* [275]*)*.

Примечание:

После обработки исключения, управление ***не возвращается*** в точку, вызвавшую исключение. Управление передаются сразу-же, если очередной оператор блока try...finally или try...except *(в блоке try)* и ***не возвращается***, **обработка исключения разрушает стэк**. Если такой оператор не найден *(оператор try...except)*, будет сгенерирована ошибка библиотеки времени выполнения Free Pascal - **217** *(см также раздел 17.5 Классы исключений[279])* . Процедура обработки исключения выводит адрес обрабатываемого исключения.

Как пример: Приведена функция деления, если знаменатель равен нулю, то инициируется исключение EDivException

```
Type EDivException = Class(Exception);
Function DoDiv (X,Y : Longint) : Integer;
begin
  If Y=0 then
    Raise EDivException.Create ('Произойдёт деление на ноль');
  Result := X Div Y;
end;
```

В модуле Sysutils библиотеки RTL определены Классы исключений (Exception). *(раздел 17.5 Классы исключений[279])*

Примечание:

Хотя класс Exception используется в качестве базового класса для исключений по всему коду, это всего лишь негласное соглашение: класс может быть любого типа, и не должен быть потомком класса Exception.

Конечно, большая часть кода использует неписаное соглашение и класс исключения происходит от Exception.

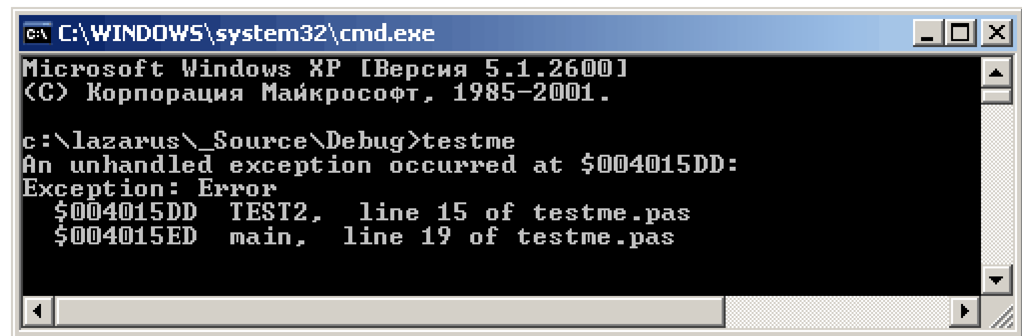
Следующий код показывает, как пропустить процедуру формирования отчета об ошибках стека, показанной в обработчик исключений:

```
{$mode objfpc}
uses sysutils;

procedure error(Const msg : string);
begin
  raise exception.create(Msg) at
    get_caller_addr(get_frame),
    get_caller_frame(get_frame);
end;

procedure test2;
begin
```

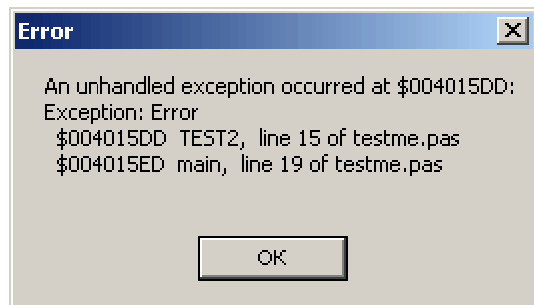
```
    error('Error');
  end;

  begin
    test2;
  end.
```

Программа, при запуске, покажет следущее:

```
An unhandled exception occurred at $00000000004015DD :
Exception : Error
  $00000000004015DD line 15 of testme.pas
  $00000000004015ED line 19 of testme.pas
Произошло необработанное исключение в $00000000004015DD :
Исключение : Ошибка
  $00000000004015DD в строке 15 в файле testme.pas
  $00000000004015ED в строке 19 в файле testme.pas
```



или

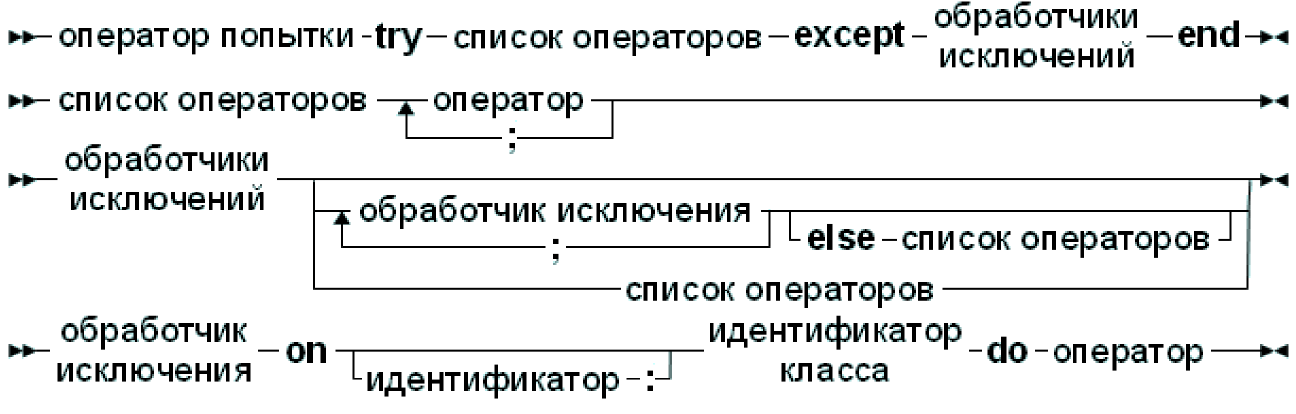


Строка **15** находится в процедуре `Test2`, а не `Error`, которая на самом деле вызвала исключение.

17.2 Операторы try...except

Блок ***try...except*** имеет следущий вид:

Оператор *try..except*



Если исключение не инициируется во время выполнения `списка операторов`, то все операторы в списке будут выполняться последовательно до `end`, за исключением блока `except`, который будет пропущен,

Если `исключение` происходит во время выполнения `списка операторов`, поток выполнения будет прерван и управление будет передано на `except` блок. Операторы между местом, где произошло исключение и до блока `except` - игнорируются.

В блоке `обработки исключения`, проверяется объект исключения, и если есть обработчик исключения, данного типа *(класса)* или родительского класса объекта исключения, то выполняются операторы после соответствующего `do`. Используется первый же подходящий объект, данного класса. После того , как блок - `do` был выполнен, программа продолжается с конца блока *(служебное слово `end`)*.

Идентификатор *(имя, после служебного слова **on**)* в операторе обработки исключений не является обязательным, и если он есть, то он объявляет объект исключения *(можно указать имя объекта, а можно только его тип)*. Он может быть использован для манипуляций с объектом исключения в коде обработки исключений. То, как организует программист блок после слова `do`, зависит от его потребностей.

Если ни один из обработчиков `on` не соответствует типу объекта исключения, то выполняется список операторов после `else`. Если этот список не найден *(отсутствует сл. слово `else`)*, то обработка исключения передаётся на более высокий уровень *(неявный `raise`)*. Этот процесс позволяет обрабатывать только некоторые исключения *(указанные в блоке `try...except`)*.

Если исключение начало обрабатываться, то объект исключения ***автоматически*** разрушается вызовом деструктора `destroy` *(самому вызывть **ненадо**)* и упрвление передается инструкции, следующей за блоком `try...except`. (*Если вызов стандартных процедур `Exit`, `Break` или `Continue` выводит управление из обработчика, объект исключения также разрушается*).

В качестве примера, рассмотрим предыдущую функцию DoDiv *(из предыдущего примера)*:

```
Try
  Z := DoDiv (X,Y);
Except
  On EDivException do Z := 0;
end;
```

Если Y будет равным нулю, то код функции DoDiv вызовет исключение. Когда это произойдёт, будет создано исключение, и его обработчик установит Z значение **0**. Если исключение не происходит, то программа выполнится, игнорируя блок except. Для того, что-бы применить восстановление после ошибки, нужно использовать блок. Блок try...finally, гарантирует, что будут выполнены операторы после сл. слова finally, даже если произойдёт исключение.

17.3 Операторы try...finally

Оператор ***try..finally*** имеет следующий вид:

Оператор *try...finally*

```
►► оператор попытки - try - список операторов - finally - обязательные операторы - end ►◄
►► обязательные операторы - список операторов ►◄
```

Если исключение не происходит внутри списка операторов, то программа работает, как будто ключевые слова try, finally и end отсутствуют, есть или нет exit: перед тем, как выйти, **в любом случае**, будет выполнено то, что находится в блоке, между ключевыми словами finally и end.

Если происходит исключение, программа немедленно передает управление из точки, где исключение произошло в ближайший блок *(оператор)* finally. *(если нет обработчика исключения **except**)*

Все операторы после ключевого слова finally, будет выполнены, а затем исключение будет автоматически **передано на более высокий уровень**. Любые операторы между местом, где возникло исключением и началом блока finally, пропускаются.

В качестве примера рассмотрим следущую процедуру:

```
Procedure Doit (Name : string);
Var F : Text;
begin
    Assign (F,Name);
    Rewrite (name);
```

```
      Try
       // ... Обработка файла ...
      Finally
        Close(F);
      end;
```

Если произойдёт исключение, то будет прервано нормальное выполнение процедуры и файл закрыт функцией Close. Если исключение не произошло, то будут выполнены все операции с файлом, после чего файл закрыт функцией Close.

Обратите внимание, что прежде, чем выйти по Exit, будет выполнен блок finally.

Дополнение предыдущего примера:

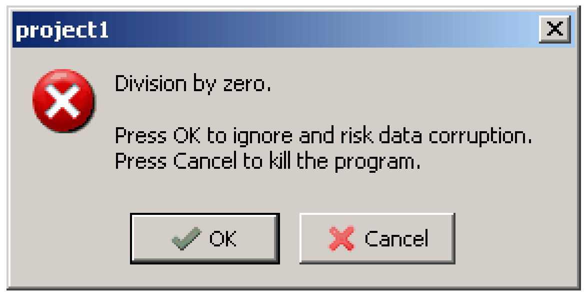
```
Procedure Doit (Name : string);
Var
      F : Text;
      B : Boolean;
begin
      B:=False;
      Assign (F,Name);
      Rewrite (name);
      Try
        // ... Обработка файла ...
        if B then exit; // Немедленно прекратить обработку
        // Дополнительная обработка файлов
      Finally
        Close(F);
      end;
```

Файл будет закрыт, даже если обработка завершается преждевременно с использованием функции Exit.

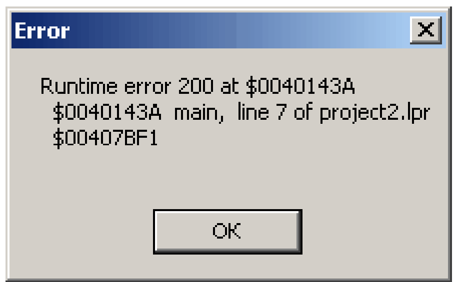
17.4 Обработка вложенных исключений

Возможна вставка блоков Try...Except и Try...Finally друг в друга. Выполнение программы будет осуществляться в соответствии с **LIFO** *(последний пришёл, первым обслужен)*. Принцип: Код блока Try...Except или Try...Finally встреченный последним, будет выполнен первым. Не перехваченное исключение или исключение для которого нет обработчика будет передаваться, ***до бесконечности***.

Если в программе не предусмотрена обработка какого-нибудь исключения, то оно обрабатывается глобальным обработчиком. Он обеспечивает стандартную реакцию на возникшее исключение – выводит предупреждение на экран с кратким описанием причины, вызвавшее исключение.



Если произойдёт необработанное исключение, то будет сгенерирована ошибка времени выполнения **217**. При использовании модуля SysUtils *(когда создана консольная программа)*, будет использован обработчик по умолчанию, окно которого будет отображаться сообщение объект исключения и адрес, где произошло исключение, после чего программа завершится инструкцией `Halt`.



17.5 Классы исключений

Модуль SysUtils содержит много классов обработки исключений. Он также определяет базовый класс обработки исключений:

```
Exception = class(TObject)
private
  fmessage : string;
  fhelpcontext : longint;
public
  constructor create(const msg : string);
  constructor createres(indent : longint);
  property helpcontext : longint read fhelpcontext write
fhelpcontext;
  property message : string read fmessage write fmessage;
end;
ExceptClass = Class of Exception;
```

И использует это объявление что-бы определить целый ряд обработчиков исключений, например:

```
{ математические исключения }
EIntError = class(Exception);
EDivByZero = class(EIntError);
ERangeError = class(EIntError);
EIntOverflow = class(EIntError);
EMathError = class(Exception);
```

SysUtils модуль также устанавливает обработчик исключений. Если исключение не обрабатывается в другом блоке обработки исключений, то этот обработчик вызывается библиотекой времени выполнения. В основном, он выводит адрес исключения и сообщение объекта `Exception`, после чего выходит с кодом **217**. Если объект исключения не является потомком класса `Exception`, то выводится имя класса вместо сообщения исключения.

Рекомендуется использовать класс `Exception` или его потомки, для создания других классов исключений для оператора raise, так как может быть использован конструктор сообщений объекта исключения.

Глава 18 Использование ассемблера

Free Pascal поддерживает использование ассемблера в коде, но встроенный макроассемблер отличается от MASM *(ассемблер макросов)*. Чтобы получить более подробную информацию о синтаксисе ассемблера процессора и его ограничениях смотрите Справочник программиста Free Pascal.

18.1 Операторы Ассемблера

Ниже приведен пример включения ассемблера в код на Pascal.

```
...
Statements;
...
Asm
  //здесь код на ассемблере
  ...
end;
...
Statements;
```

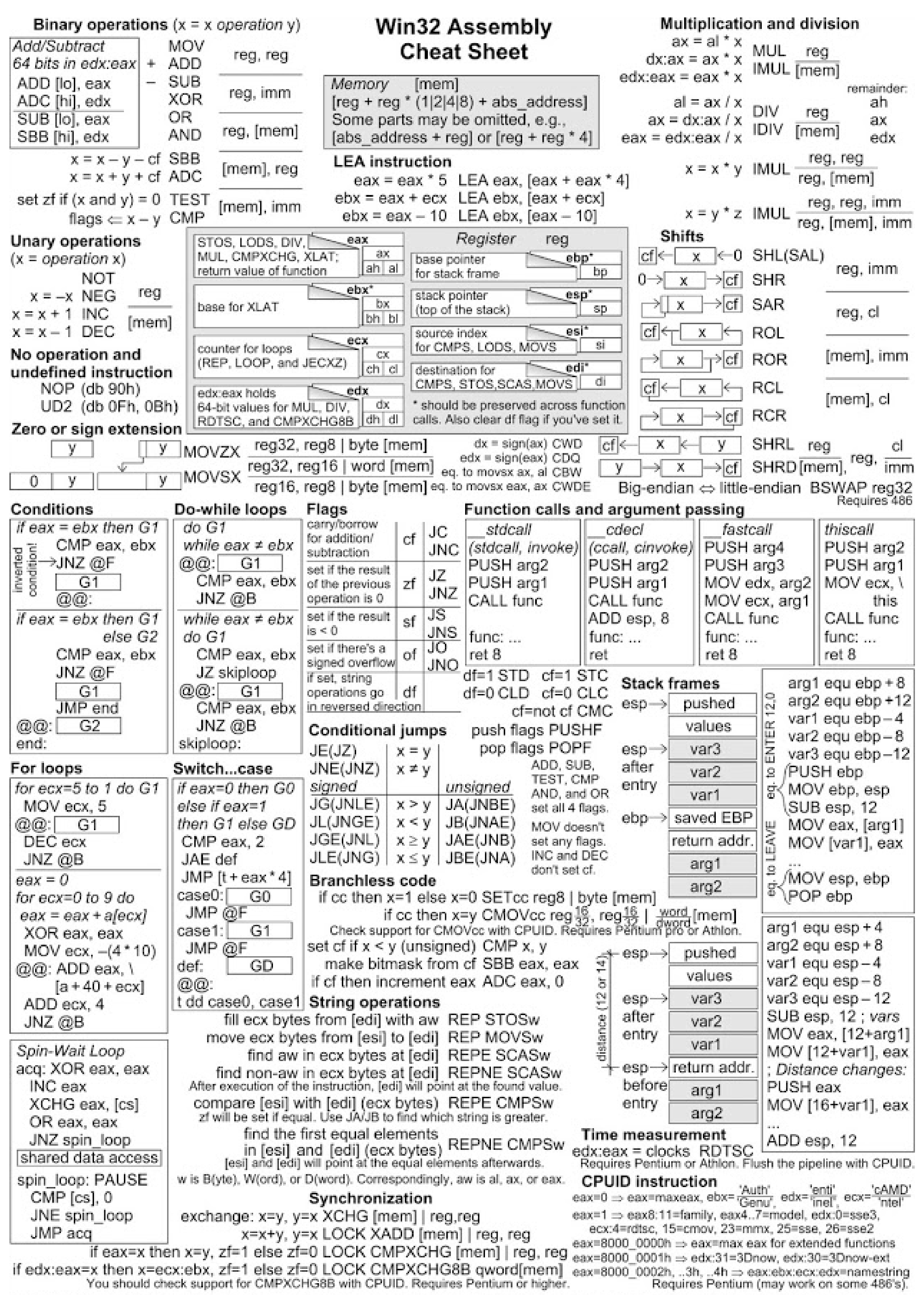
Инструкции ассемблера меж ключевых слов Asm и end будет вставлены в код генерируемый компилятором. Условные конструкции могут использоваться и в коде ассемблера, компилятор распознает их, и относиться к ним, как и к любым другим условным конструкциям.

18.2 Процедуры и функции Ассемблера

Процедуры и функции ассемблера объявляются с помощью директив Assembler *(ассемблера)*. Это позволяет генератору кода, сделать ряд оптимизаций генерации кода.

Генератор кода не генерирует кадр стека *(входа и выхода код для обычной процедуры)*, если она не содержит локальные переменные и параметры. В случае функций, порядковые значения *(ordinal)* должны быть возвращены в аккумуляторе. В случае значений с плавающей запятой, они зависят от целевого процессора и вариантов эмуляции.

18.3 Приложение



- * -

*= 196

- / -

/= 196

- : -

:= 196

- + -

+= 196

- - -

-= 196

- A -

Asm 217

- B -

begin 200
Boolean 26, 28
Break 204, 213, 214
Byte 26
ByteBool 26, 28

- C -

Cardinal 26
case 201
Char 26
Comp 32
Continue 204, 213, 214
Currency 32

- D -

deprecated 16
do 204, 205, 214
Double 32
downto 204

- E -

else 201, 202
end 200
experimental 16
Extended 32

- F -

False 28
finalization 261
for 204, 205
For..in..do 205
For..to/downto..do 204

- G -

Goto 198

- I -

IEnumerable 205
IEnumerator 205
if 202
If..then..else.. 202
implementation 261
in 205
initialization 261
Int64 26
Integer 26
interface 261

- L -

label 18
LongBool 26, 28
LongInt 26
LongWord 26

- O -

of 201
otherwise 201

- P -

platform 16
program 260

- Q -

QWord 26

- R -

Real 32
repeat 213
Repeat..until 213

- S -

ShortInt 26
Single 32
SmallInt 26

- T -

TCollection 205
TComponent 205
TFPList 205
then 202
TList 205
to 204
True 28
TStringList 205

- U -

unimplemented 16
until 213

- W -

while 214
while..do 214
With 215
Word 26
WordBool 26, 28

- Z -

булевы типы 28
вещественные типы 32
восьмеричный формат 17
двоичная нотация 17
десятичный формат 17
зарезервированные слова 10, 12
зарезервированные слова Object Pascal 13
зарезервированным слова Free Pascal 13
идентификаторы 10, 15
исключения 217
ключевые слова Turbo Pascal 12
комментарии 11
константы 10
метка 18, 196
модификаторы 14
область действия идентификатора 269
Оператор Goto 198
Оператор вызова процедуры 198
Оператор присвоения 196
Оператор цикла 199
Операторы 10, 196
основные типы 25
перечислимые типы 26
перечислимый тип 29
Простой оператор 196
простые типы 25
разделители 10
символы 10
символьная строка 19
строка 19
Структурированные операторы 199
тип 25
тип диапазон 31
типы перечислений 29
Токены 10
Условный оператор 199
целочисленные типы 26
целые типы 26
шестнадцатеричный формат 17