Free Pascal:
Практика программирования

http://freepascal.ru/

20 сентября 2005 г.

Оглавление

Список таблиц

Введение

Часть I.

Общие вопросы

1. Язык Pascal

Язык Pascal в своем первоначальном виде был создан швейцарским ученым Никлаусом Виртом[1] в 1969–1970 гг. Предназначавшийся в первую очередь для обучения программированию, Pascal, тем не менее, быстро завоевал широкую популярность, поскольку в нем максимально воплотилась актуальнейшая на тот момент концепция *структурного программирования*.

За прошедшие десятилетия технологии и концепции программирования значительно изменились, и языки семидесятых сейчас могут вызвать лишь ностальгическую улыбку. Естественно, Pascal все это время развивался — его современные диалекты вполне адекватны новым веяниям. В целом можно выделить три основных ветви развития языка:

Линия Вирта: Создатель Pascal придерживается весьма негативного мнения относительно создания диалектов и считает, что существенно измененную версию языка следует считать новым языком и называть по новому. Его языки, являющиеся наследниками идей, заложенных в Pascal — это *Modula-2*[2], *Oberon* и *Oberon-2*.

Языки Вирта сочетают строгость и лаконичность языковых средств с большой гибкостью и выразительностью. Однако широкого распространения они не получили, став явлениями скорее академического плана.

Стандарты ANSI/ISO: Стандарт *ISO 7185 («Classic Pascal»)* в целом соответствует первичному Виртовскому языку. Стандарт *ISO 10206 («Extended Pascal»)* вводит различные расширения, как естественные — модульность, например, так и несколько странные — частично динамическая типизация. Стандарт на объектно-ориентированные расширения существует лишь в виде технического отчета и официального номера не имеет.

Как это часто бывает с ISO-стандартами[3], ISO Pascal разрабатывался непонятно кем и непонятно для кого, соответственно и о популярности говорить не приходится. Частично, правда, этот стандарт поддерживается компилятором GNU Pascal, который, однако, значительно менее известен, чем интересующий нас Free Pascal (оба проекта относятся к свободным программам, кроссплатформенные, и таким образом сравнение более-менее корректно).

Линия Borland: Фирма Borland, в общем-то, начала свою деятельность с компилятора *Turbo Pascal*, и по сей день его наследники остаются флагманским продуктом корпорации. В версии *4.0* появилась модульность, затем — *5.5* — объекты, а далее полностью объектно-ориентированный язык *Object Pascal*[4] и среда Delphi.

---

[1] Niklaus K. Wirth, род. в 1934 г.

[2] Первая версия — *Modula* — не была реализована.

[3] Например, ISO-кодировка кириллицы.

[4] В настоящее время фирма Borland переименовала язык, и последние версии Object Pascal называются *Delphi Language*, хотя отличия его от языка даже первых версий Delphi куда менее значительны, чем отличия языка Turbo Pascal ранних и поздних версий.

Поскольку популярность языка была прямой задачей корпорации Borland, в «раскрутку» были вложены значительные средства и силы. В результате эта ветвь развития языка стала не просто наиболее популяной, но фактически — промышленным стандартом.

Free Pascal Compiler ориентирован, главным образом, на линию Borland. Поддерживается полная совместимость с Turbo Pascal (насколько это вообще возможно для 32х-битного компилятора) и почти полная с Borland Delphi. Кроме того, FPC вводит некоторые собственные расширения языка. Одновременная поддержка нескольких диалектов языка во избежание конфликтов реализована как возможность выбора из нескольких *режимов компиляции* (или, если угодно — *режимов совместимости*).

TP Режим совместимости с Turbo Pascal 7.0.

FPC К языку Turbo Pascal добавлены специфические расширения, такие как возможность определения операций.

Delphi Режим совместимости с Borland Delphi (с версией Delphi 3 совместимость полная, с 5-й — частичная).

ObjFPC Соответствует предыдущему режиму с собственными расширениями.

GPC Частичная совместимость с GNU Pascal.

MacPas Режим совместимости с Mac Pascal.

Для нас в данной книге основным режимом будет *ObjFPC* как наиболее полный. Особенности тех или иных аспектов в других режимах будут даваться в виде замечаний.

1.1. Основные понятия

1.1.1. Символы языка

Комментарии

Комментарии в исходных текстах Free Pascal могут выделяться тремя различными способами, как и в Delphi:

- Блочный комментарий в круглых скобках со звездочками;
- Блочный комментарий в фигурных скобках;
- Строчной комментарий, начинающийся с двойного слэша.

```
(* Блочный комментарий 1 *)
{  Блочный комментарий 2  }
// Строчной комментарий
```

Все, что находится внутри комментария, компилятором игнорируется. Следует заметить, что *конец блочного комментария* в разных режимах определяется по

разному: если в режимах FPC и ObjFPC компилятор следит за вложенностью соответствующих скобок и концом комментария считает закрывающую скобку, парную начальной, то в остальных режимах концом комментария считается *первая* закрывающая скобка соответствующего вида, что означает запрет на вложенность однотипных коментариев.

Ключевые слова

Ключевые слова используются для обозначения различных конструкций языка, таких как исполняемые операторы, объявления переменных и т.д. Также специальные ключевые слова используются (наряду со *специальными символами*) для обозначения операций.

Ключевые слова языка не могут быть каким-либо образом переопредены, в частности — не могут выступать в качестве *идентификаторов*. Перечень ключевых слов дан в приложении А.

Согласно давней традициии в примерах исходных текстов ключевые слова выделяются жирным шрифтом.

Идентификаторы

Идентификаторы предназначены для однозначного именования элементов программы, таких как переменные, типы, константы и т.д. Идентификаторы в языке Pascal могут состоять из букв латинского алфавита, цифр и знака подчеркивания ("_"), причем первый символ не может быть цифрой.

Специальные символы

Следующие символы и сочетания символов Free Pascal распознает как имеющие специальное значение: "`;`", "`.`", "`,`", "`:`", "`'`", "`^`", "`@`", "`$`", "`&`", "`%`", "`#`", "`( )`", "`[ ]`", "`(. .)`"[5], "`{ }`", "`(* *)`", "`//`", "`..`", "`+`", "`-`", "`*`", "`/`", "`**`", "`=`", "`<`", "`>`", "`<=`", "`>=`", "`<>`", "`:=`", "`+=`", "`-=`", "`*=`", "`/=`".

Конкретное значение этих симолов зависит от контекста.

Символьные константы

Символьные константы в языке Pascal записываются как сочетание строк, заключенных в апострофы (одиночные кавычки), и символов, указанных по их коду. Если строка должна содержать апостроф, его знак удваивается.

Указание символа по его коду формируется сочетанием знака "#" и *целочисленной беззнаковой константы.*

```
'Просто строка'
'Первая строка'#13#10'Вторая строка'  // Сочетание 13-го и 10-го
                                      // символов в DOS и Windows
                                      // обеспечивает перевод строки
#169' Vasya Pupkin'         // «© Vasya Pupkin» в кодировке Win-1251
'Pupkin''s string'          // «Pupkin's string»
```

[5] Пара скобок "`(. .)`" является синонимом "`[ ]`", так же как "`(* *)`" соответствует "`{ }`". Это связано с ограниченными возможностями старинных терминалов.

Числовые константы

- **Целые числа:**
  - *Без знака:*
    - Десятичные — представляются в виде последовательности цифр (“0” .. “9”);
    - Шестнадцатиричные — представляются в виде последовательности шестнадцатиричных «цифр» — к десятичным цифрам добавляются латинские буквы от “A” до “F”, регистр не имеет значения — предваряемой символом “$”.
    - Восьмеричные — представляются в виде последовательности цифр от “0” до “7”, предваряемой символом “&”.
    - Двоичные — последовательность двоичных цифр — “0” и “1” — предваряемая символом “%”.
  - *Со знаком* — представляют собой беззнаковую константу, предваряемую символом “+” или “–”.
- **Вещественные числа:**
  - *Обычное представление* — “*<знак><целая часть>.<дробная часть>*”. Знак и дробная часть — необязательны.
  - *«Научное» представление* — запись состоит из двух частей: *мантиссы*, представляющей собой вещественное число в обычном представлении[6], и порядка, представляющего собой *десятичное целое со знаком*. Части разделяются латинской буквой “E” (регистр не имеет значения).

Примеры числовых констант:

```
234       // Беззнаковое десятичное
$3F       // = 63; Шестнадцатиричное
&77       // = 63; Восьмеричное
%101001   // = 41; Двоичное
-1024     // Целое со знаком
12.5      // Вещественное
-1.003E+3 // = -1003.0
```

Директивы компилятора

Существует специальный вид комментариев, который хотя и не является частью программы, тем не менее не игнорируется компилятором, а напротив — управляет процессом компиляции. Такие инструкции называются *директивами компилятора* и имеют вид “{$... }” (пробелы между открывающей скобкой и знаком доллара недопустимы). Регистр в директивах значения не имеет.

Например, директива

```
{$MODE OBJFPC}
```

выбирает режим компиляции (см. выше) ObjFPC.

Подробное описание директив компилятора см. в главе 4, раздел 4.2.

[6] Нормализация записи, т.е. — приведение к такому виду, что мантисса имеет только одну, и не нулевую, цифру в целой части — не обязательна.

1.1.2. Выражения

Выражения в языке Pascal используются для вычисления значений, которые затем будут присвоены переменной, переданы в качестве параметра и т.д. Каждое выражение имеет свой тип, однако данный тип задается неявно — согласно правилам вычислений.

Синтаксис выражения проще всего определить рекурсивно — выражение может быть одной из следующих конструкций:

1. *Квалификатор* переменной, поля или свойства.
2. *Квалификатор* константы.
3. *Числовая* или *символьная константа*.
4. *Конструктор множества*.
5. Одно из ключевых слов: "`nil`" — нулевой *указатель*, "`true`" или "`false`" — булевы значения.
6. Вызов *функции*.
7. Явное *приведение типа* — *<тип>***(***<выражение>***)**.
8. Выражение в скобках — **(***<выражение>***)**.
9. Бинарная операция — *<выражение> <операция> <выражение>*.
10. Унарная операция — *<операция> <выражение>*.
11. Операция разыменования — *<адрес>*`^`, где «адрес» — выражение *адресного типа*.

Числовые и символьные константы описаны в подразделе 1.1.1, о функциях подробно пойдет речь в разделе 1.4, преобразования типов будут описаны в разделе 1.5, подраздел 1.5.6.

Булевы константы

Символы "`true`" и "`false`", означающие логические «истину» и «ложь» соответственно, в отличие от Delphi и Turbo Pascal, во Free Pascal являются ключевыми словами, а не идентификаторами. С точки зрения их использования в программе они, конечно же, остаются предопределенными константами, аналогично "`nil`".

Конструктор множества

Конструктор множества (см. подраздел 1.5.2) состоит из квадратных скобок, в которых находится список из элементов множества и *диапазонов элементов*, разделяемых запятыми — "**[***<список>***]**".

Диапазон элементов порядкового (см. подраздел 1.5.1) *типа* представляет собой начальное и конечное значения диапазона, между которыми расположен специальный символ — две точки подряд — "*<значение>* `..` *<значение>*". Помимо конструкторов множеств диапазоны используются в объявлениях статических массивов и в операторе "`case`" (см. 1.2.2 и 1.5.2).

Примеры конструкторов множеств:

```
[1, 3, 4 .. 6]
['A' .. 'Z']
[7]
[]                  // Пустое множество
```

Квалификаторы

Квалификаторами мы будем называть языковые структуры, которые позволяют обратиться к тому или иному элементу программы, когда *идентификатора* недостаточно (например, в данной области видимости он перекрыт другим, см. 1.1.3) — своего рода *идентифицирующие выражения*. Это могут быть:

- *Идентификаторы* сами по себе — тривиальный случай.
- *Вложенные идентификаторы*, разделяемые точкой. Пример — кострукция вида:

  <переменная-запись>.*<поле>*

  для обращения к конкретному полю записи. Точно также формируются квалификаторы полей, свойств и методов объектных типов, кроме того, таким образом можно указать некий элемент конкретного модуля.
- *Элементы массива по индексу*. Обращение к элементам массива имеет вид:

  <массив>[*<индекс(-ы)>*]
- *«Разыменование» адреса*. Выражение, получаемое посредством операции разыменования (см. выше), используется как переменная, т.е. может находиться и в левой части оператора присваивания.

Операции

Взятие адреса и разыменование. Унарная *операция взятия адреса* применима к *квалификатору* переменной, типизированной константы, подпрограммы, поля или метода и обозначается символом "`@`". По сути, она обратна операции *разыменования*.

```
@<квалификатор>
```

Результатом является адрес данного элемента, имеющий тип указатель, или процедурный (см. подразделы 1.5.1 и 1.5.1), если типизация адреса включена, и тип `Pointer` — если выключена. Типизация адреса изменяется директивой

```
{$TYPEDADDRESS ON/OFF} // или {$T+/-}
```

Операция разыменования напротив — позволяет получить из адресного выражения квалификатор. Тип элемента, указываемого квалификатором, зависит от типа адресного выражения. Операция разыменования обозначается знаком "`^`" *после* адресного выражения:

```
<адрес>^
```

Арифметические операции. К арифметическим относятся семь бинарных и две унарные операции.

- Бинарные: *сложение* — "`+`", *вычитание* — "`-`", *умножение* — "`*`", *деление* — "`/`", *целочисленное деление* — "`div`", *остаток* — "`mod`", а также операция *возведения в степень* — "`**`", которая, однако, определена в модуле `Math`, а не `System`.

  Операции "`div`" и "`mod`" применимы только к целым числам, остальные — как к целым, так и к вещественным. При этом результат имеет целый тип, если оба операнда целые, и вещественный, если хотя бы один операнд вещественный. Исключением является операция деления — ее результат всегда вещественный.

- Унарные операции: *обратный знак* — "`-`" и *тождественный знак* — "`+`". Применимы как к целым, так и к вещественным числам, результат имеет тип операнда.

Булевы операции. Как операнды, так и результат булевых операций относятся к булевым типам. Операций таких четыре:

- Бинарные: *логическое «И»* — "`and`", *логическое «ИЛИ»* — "`or`", *«исключающее ИЛИ»* — "`xor`".
- Унарная — *отрицание («НЕ»)* — "`not`".

Битовые операции. Группа операций, выполняющая действия непосредственно над битами данных. Эти операции определены только для целочисленных типов.

- Операции, соответствующие булевым: "`and`", "`or`", "`xor`" и "`not`". Выполняют те же действия, что и булевы, в отношении каждого бита операндов, принимая единицу за логическую «ИСТИНУ», а ноль — за «ЛОЖЬ».
- Операции *сдвига*: "`shl`" и "`shr`" — выполняют левый и правый сдвиг (не циклический) битов первого операнда на число позиций, указанное вторым операндом. Могут использоваться для быстрого умножения или целочисленного деления на степени двойки[7].

Строковые операции. Для строковых типов данных определена одна операция — *конкатенация*, или *сложение строк* — "`+`".

Операции над множествами. Для множеств в языке Pascal определены три операции: *объединение* — "`+`", *пересечение* — "`*`" и *разность* — "`-`".

Операции отношений. Все операции отношений в Pascal возвращают результат булева типа.

- *Равенство* — "`=`", определена для всех *простых типов*.

[7] Впрочем, при включенной оптимизации компилятор для умножения/деления на степени двойки сам выбирает именно команды сдвига.

- Неравенства: *больше* — “>”, *меньше* — “<”, *больше или равно* — “>=”, *меньше или равно* — “<=” и *не равно* — “<>”. Определены для числовых и строковых типов.
- *Принадлежность к множеству* — “in”. Первый операнд порядкового типа, второй — множество.

Объектно-ориентированные операции. Две операции: проверка *принадлежности к классу* — “is” и *приведения класса* — “as” мы выделим в отдельную группу и рассмотрим в главе 2, поскольку их описание требует использования понятий ООП.

Приоритет операций

В таблице 1.1 показан приоритет операций от высшего к низшему. Чтобы изменить порядок вычислений используются круглые скобки — “(...)”.

Приоритет	Операции	Категория
1.	not @ + -	Унарные
2.	**	Возведение в степень
3.	* / div mod and shl shr as	Мультипликативные
4.	+ - or xor	Аддитивные
5.	= <> < > <= >= in is	Отношений

Таблица 1.1.: Приоритет операций

Заметим, что Free Pascal не поддерживает каких-либо соглашений о порядке вычисления операций равного приоритета, то есть выражение “a + b + c” может быть вычислено и как “(a + b) + c” и как “a + (b + c)”; кроме того, если в выражении участвуют функции, порядок их вызова также не определен. Если порядок важен, рекомендуется разделить вычисления на несколько операторов с использованием промежуточной переменной [1, Chapter 8, p. 67].

1.1.3. Структура исходных текстов

Проект[8] на Free Pascal состоит из *главного исходного файла* (*программы* или *библиотеки*) и *модулей*.

Структура исходного текста *программы* следующая:

1. Заголовок, состоящий из ключевого слова “program”, идентификатора и точки с запятой. Заголовок в данном случае необязателен.
2. Необязательная секция “uses”, в котором перечисляются используемые модули.
3. *Программный блок*, завершаемый точкой.

[8] Вообще говоря, термин «*проект*» в документации по Free Pascal Compiler не используется. Однако, для обозначения совокупности исходных текстов, компилируемых в единое целое, удобно использовать именно его, поскольку термин «*программа*» используется также для обозначения исходного текста, компилируемого в *исполняемый файл*, в отличие от *библиотеки*.

Секция используемых модулей состоит из ключевого слова "uses", за которым следует список используемых модулей через запятую, заканчивающийся точкой с запятой. Модули в списке представлены идентификаторами, к которым может быть добавлено явное указание исходного файла модуля в виде "**in** *<путь>*".

Программный блок[9] состоит из *области деклараций* и *блока исполняемых операторов*.

- Область деклараций состоит из необязательных секций, в произвольном порядке описывающих: *типы данных* — см. 1.5, *переменные и костанты* — 1.3, *подпрограммы* — 1.4 и *метки* — см. подраздел 1.2.8, посвященный оператору "goto".

  Также в области деклараций главного файла программы или библиотеки может присуствовать секция *экспорта* — см. главу 3, раздел 3.4.

- Блок исполняемых операторов начинается ключевым словом "begin" и завершается ключевым словом "end", между которыми расположены исполняемые операторы, непосредственно определяющие, что будет делать программа.

Исходный текст главного файла *библиотеки* тоже выглядит так, как описано выше, за исключением заголовка, который начинается ключевым словом "library" вместо "program". К тому же, надо заметить, секция экспорта в библиотеке практически необходима, тогда как в исходниках программы нужна весьма редко.

Модули

Модуль во Free Pascal — это отдельно компилируемая программная единица, содержащая описания подпрограмм, типов и т.д.

Исходник модуля выглядит следующим образом:

1. *Заголовок* — ключевое слово "unit", идентификатор модуля и точка с запятой.

   В отличие от программы или библиотеки, идентификатор должен совпадать с именем файла (без пути и расширения, естественно) без учета регистра. Кроме того, имя файла модуля должно быть в нижнем регистре, если операционная система регистр различает. Иначе нужный файл компилятором может быть не найден.

2. *Интерфейсная часть*, которая начинается ключевым словом "interface". За ним идет секция используемых модулей, а затем область деклараций. Объявленные здесь идентификаторы доступны из других модулей.

   Декларации интерфейсной части модуля отличаются от деклараций в главном файле и в части реализации тем, что для попрограмм описывается только заголовок, тогда как тело подпрограммы находится в части реализации.

3. *Часть реализации* начинается с ключевого слова "implementation", за которым следует необязательная секция используемых модулей[10] и область ло-

[9] Словосочетание «*программный блок*» следует воспринимать как единый термин — в таком качестве он еще стретится в дальнейшем.

[10] Обычно секция "uses" в части реализации не используется — достаточно той, что в интерфейсной части. Однако она может быть использована, скажем, для создания циклически ссылающихся

кальных деклараций: объявленные в этой части идентификаторы извне модуля недоступны, кроме подпрограмм, заголовки которых объявлены в интерфейсной части.

4. Далее возможны два варианта:

   - Старый вид модуля — ключевое слово "`begin`", за которым следует *код инициализации* модуля, т.е. — исполняемые операторы, которые выполняются единожды при загрузке программы.
   - Новый вид модуля — ключевое слово "`initialization`", за которым следует *код инициализации*, а затем ключевое слово "`finalization`" с последующим *кодом завершения*, который выполняется также один раз, но уже при завершении работы программы.

   Код инициализации/финализации в любом случае необязателен. Возможна финализирующая часть без инициализирующей и наоборот. В любом случае текст модуля завершается ключевым словом "`end`" с точкой.

Помимо того, что каждый модуль компилируется сам по себе и при изменении в одном из модулей проекта FPC перекомпилирует только его и те модули, которые от него зависят, есть еще такая прекрасная возможность как *«умная» компоновка (smarklink)*. Она позволяет включить в исполняемый файл только те элементы (подпрограммы, переменные и проч.), которые реально используются, а не все, описанные в модулях. Подробнее об этом будет сказано в главе 4, раздел 4.2.

Особняком стоят модули `System`, `ObjPas` и `MacPas`. Эти модули не нужно указывать в разделе "`uses`" — `System` автоматически подключается *всегда*, `ObjPas` — в режимах Delphi и ObjFPC, а `MacPas` — соответственно в режиме MacPas.

Области видимости

Чтобы избежать конфликтов между одинаковыми идентификаторами, язык Pascal предусматривает области видимости.

1. Область оператора "`with`" — см. описание данного оператора в подразделе 1.2.7.
2. Область подпрограммы: идентификаторы, объявленные внутри данной подпрограммы (между заголовком и "`begin`"). Внутри подпрограммы они имеют приоритет, снаружи же недоступны.

   Поскольку подпрограммы могут быть вложенными, их области видимости также вкладываются друг в друга.
3. Область класса: идентификаторы полей, методов и свойств класса[11] внутри реализации его методов имеют приоритет перед внешними, хотя и уступают локальным для данных методов.
4. Область модуля: идентификаторы, объявленные в данном модуле, имеют приоритет перед импортированными из модулей списка "`uses`".

друг на друга модулей, поскольку циклические ссылки в интерфейсной части компилятором запрещены.

Нельзя сказать, чтоб это было хорошей практикой: как правило, необходимость в циклических ссылках говорит о недочетах проектирования.

[11] То же самое относится и к `object`-типам.

5. Импортируемые идентификаторы имеют приоритет, обратный положению модуля в списке: идентификаторы последнего модуля перекрывают идентификаторы предыдущих. Последние остаются доступны посредством соответствующего *квалификатора*: имя модуля, точка, идентификатор.

Кроме того, следует помнить, что любой идентификатор доступен только *после* того как объявлен, и что в одной области видимости идентификаторы должны быть уникальными (единственное исключение — перегрузка подпрограмм, о которой пойдет речь в подразделе 1.4.5).

1.2. Исполняемые операторы

Прежде чем описывать исполняемые операторы языка Pascal хотелось бы сделать замечание: точка с запятой *";" не заканчивает оператор*, как в языках C или Java, а *отделяет операторы друг от друга*. Разница незаметная, но в некоторых случаях она имеет значение.

1.2.1. Простые операторы

Присваивание

Оператор присваивания выглядит следующим образом:

<Квалификатор> **:=** *<выражение>*

Здесь «квалификатор» — квалификатор переменной, поля или свойства, а «выражение» — произвольное выражение соответствующего (того же или *совместимого по присваиванию* — см. 1.5.6) типа.

Вызов процедуры

Вызов процедуры (или метода-процедуры) выглядит просто — соответствующий квалификатор и список *актуальных* параметров в круглых скобках. Параметры в списке разделяются запятыми.

*<Процедура>***(***<параметры>***)**

В случае использования *расширенного синтаксиса* (см. 1.4.5) таким образом могут вызываться и *функции*, возвращаемое значение при этом игнорируется.

1.2.2. Условное выполнение

Условный оператор — "`if`"

Синтаксис условного оператора следующий:

if *<условие>* **then** *<оператор-1>* **else** *<оператор-2>*

Ключевое слово “else” вместе со следующим за ним оператором необязательны.

«Условие» — произвольное выражение булева типа. Если оно истинно (= true), то выполняется «оператор-1», иначе (и при наличии ветви “else”) выполняется «оператор-2». «Оператор-1» и «оператор-2» — это единичные операторы, если нужно несколько операторов в уловной ветке, следует использовать *составной оператор* (см. далее).

Точка с запятой перед ключевым словом “else” *недопустима* — компилятор воспримет ее как окончание всего условного оператора.

Оператор выбора — “case”

Оператор выбора служит для описания множества вариантов действий вместо двух. Синтаксис его следующий:

```
case <выражение> of
     <значения-1> : <оператор-1>;
     <значения-2> : <оператор-2>;
     . . . . .
     <значения-N> : <оператор-N>
     else <оператор-иначе>
     end
```

Здесь «выражение» — произвольное выражение *порядкового типа*. «Значения-1,2...N» — некие наборы значений, заданные списками значений и диапазонов через запятую. Данные наборы обязательно константные, т.е. — должны быть вычислены на этапе компиляции. Если результат выражения входит в один из наборов, выполняется соответствующий оператор. Если ни одному набору он не соответствует, выполняется «оператор-иначе».

Ветка “else” необязательна. Перед “else” и перед “end” точка с запятой допустима[12].

1.2.3. Циклы

Цикл по индексной переменной

Цикл с использованием индексной переменной называется также циклом с определенным количеством итераций — исключая вариант досрочного выхода из цикла, их число дейтвительно заранее известно.

Синтаксис:

```
for <переменная> := <нач. знач.> to <кон. знач> do <оператор>
```

или

```
for <переменная> := <нач. знач.> downto <кон. знач> do <оператор>
```

Здесь «переменная» — квалификатор переменной порядкового типа. «Нач. знач.» и «кон. знач.» должны быть того же типа (или совместимы по присваиванию).

[12] Это следует считать своего рода исключением из обычной трактовки точки с запятой, однако в операторе “case” такое исключение не может привести к неоднозначностям.

Оператор «оператор» выполняется последовательно для всех значений, начиная с «нач. знач.», заканчивая «кон. знач.», в порядке возрастания (“to”) или убывания (“downto”).

Если конечное значение меньше начального в случае “to”, или конечное больше начального для “downto”, то «оператор» не выполняется ни разу. Если начальное и конечное значение равны, то «оператор» выполняется один раз.

В теле цикла изменение индексной переменной *недопустимо*. По окончанию цикла ее значение *неопределено*.

Цикл с предусловием

while *<условие>* **do** *<оператор>*

«Оператор» выполняется пока «условие» истинно. Если условие ложно изначально (при начале выполнения “while”), то цикл не выполняется ни разу.

Цикл с постусловием

repeat *<операторы>* **until** *<условие>*

Здесь «операторы» — произвольная последовательность операторов. Они выполняются до тех пор, пока «условие» не станет истинным. Если условие истинно с самого начала, то тело цикла тем не менее выполняется — проверка происходит уже позже.

Прерывание цикла

В теле цикла возможно использование двух дополнительных операторов для внеочередного прерывания текущей итерации или всего цикла:

“**break**” — прерывает цикл — управление передается оператору, следующему за оператором цикла.

“**continue**” — прерывает текущую итерацию — управление передается на конец тела цикла.

1.2.4. Специальные операторы

Составной оператор

Составной оператор служит для объединения некоторой последовательности операторов в один, что используется, например, в условных операторах, операторах “for” и “while”.

begin *<операторы>* **end**

Конструкция “begin ... end” также называется операторными скобками и, кроме обозначения составного оператора, служит для выделения исполняемого блока (см. подраздел 1.1.3).

Пустой оператор

Пустой оператор никак не обозначается и ничего не делает. Это абстрактное понятие введено для возможности передачи управления на конец составного оператора или тела цикла “repeat”.

В результате становится допустимой точка с запятой перед “end”, “until” и т.д.[13] Кроме того, между исполняемыми операторами можно вставлять сколько угодно точек с запятой (это не относится к другим случаям использования данного знака).

1.2.5. Обработка исключений

Обработка исключений — довольно объемная тема, кроме того, как и некоторые другие расширения, впервые появившиеся в Borland Delphi, стоит несколько в стороне от классических конструкций языка Pascal. Поэтому ее мы рассмотрим отдельно в главе 3, раздел 3.1. Пока лишь отметим три оператора:

- “try/except” и “try/finally” формируют *защищенные блоки*;
- “raise” генерирует *исключение*.

1.2.6. Ассемблерный оператор

```
asm
 <инструкции ассемблера>
end
```

Ассемблерный оператор позволяет вставить в программу на языке Pascal фрагмент ассеблерного кода. Надо заметить, что ассемблерные вставки могут помешать созданию *кроссплатформенных* программ. Подробно работа со встроенным и внешним ассемблером описывается в главе 3, раздел 3.6.

1.2.7. Оператор “with”

```
with <квалификатор> do <оператор>
```

Данный оператор создает специальную *область видимости*, где подчиненные идентификаторы элемента, указанного через «квалификатор» — это может быть запись, объект, класс и т.д., непосредственно доступны и имеют приоритет перед прочими.

На примере:

```
var
  X : Integer;
  P : record
      X, Y : Integer
      end;
```

[13] И теперь от этой абстракции не избавишься — если “goto” в pascal-программах встречается редко, то точка с запятой перед “end” — на каждом шагу. См. также сноску 12 на стр. 18.

```
begin
 X := 10;         // Переменная X
 with P do
      begin
      X := 20;    // поле P.X
      P.Y := 100 // по квалификатору поля
                 // доступны как и ранее
      end
end.
```

1.2.8. Метки и оператор "goto"

Оператор "goto" осуществляет немедленный переход к оператору, отмеченному *меткой*. Метки в Pascal — идентификаторы или целочисленные константы без знака, помещаемые перед исполняемым оператором через двоеточие. Все метки должны быть объявлены заранее.

Пример:

```
var
  First, Last, Summa, I : Integer;

label
  Loop;

begin
 ReadLn ('Начальное значение: ', First);
 ReadLn ('Конечное значение: ', Last);
 Summa := 0;
 I := First;
Loop : Summa = Summa + I;
 I := I + 1;
 if I <= Last
    then goto Loop;
 WriteLn (Summa)
end.
```

Пример, бузусловно, надуманный — безусловный переход использован для организации цикла, подобного "repeat". Однако очень трудно найти пример, где использование "goto" сколь-нибудь оправдано, и невозможно — где такое использование необходимо[14].

Использование оператора безусловного перехода резко затрудняет чтение программы человеком, а следовательно — поиск ошибок, дальнейшее развитие и сопровождение. Концепция структурного программирования подразумевает полный отказ от этого оператора, однако язык Pascal был создан еще в то время, когда эта концепция только формировалась и не была общепризнанной.

Free Pascal Compiler по умолчанию работает в режиме без поддержки "goto", включить поддержку можно директивой компилятора

[14] Не-необходимость может быть строго доказана. Более того: любой алгоритм *может* быть записан при помощи только, например, операторов присваивания, составного и цикла с предусловием, но это уже из разряда теоретических ухищрений — читать такое будет еще хуже, чем «вермишель» из "goto".

```
{$GOTO ON}
```

1.3. Переменные и константы

1.3.1. Переменные

Переменная, вообще говоря, это область памяти, которой сопоставлен идентификатор, и для которой определен некий тип данных.

Секция описаний переменных начинается с ключевого слова “var”, за которым следует один или несколько списков переменных с указанием типа. Например:

```
var
  I, J, K : Integer; // тип задан идентификатором
  R : record         // тип задан конструктором
      X, Y : Integer
      end;
  Q : QWord absolute R;
```

Объектные типы (“object”, “class” и “interface”) могут быть заданы только идентификатором.

Ключевое слово “absolute” указывает компилятору не выделять отдельную область памяти под переменную, а разместить ее по адресу переменной или типизированной константы[15], идентификатор которой следует за ключевым словом. В примере выше переменная Q содержит те же данные, что и переменная R, но трактует их не как запись, а как единое 64х-битное целое.

Переменные, определенные внутри подпрограмм называют *локальными*, а определенные в основной области описаний модуля или программы — *глобальными*. Впрочем то же относится к прочим элементам программы.

Начальное значение переменных неопределено. В большинстве реализаций глобальные переменные инициализируются нулевыми байтами, а локальные содержат всякий мусор, однако рассчитывать на это нельзя. Впрочем, что касается Free Pascal, то компилятор выдает предупреждение при попытке использовать значение неинициализированной явно переменной.

Впрочем, в современных диалектах Pascal переменные можно инициализировать при описании. Синтаксис в этом случае соответствует описанию типизированных констант[16] (см. ниже), с той лишь разницей, что описание находится в секции “var”, а не “const”.

```
var
  A : Integer = 0;
```

1.3.2. Константы

Основное назначение констант — дать некоторым значениям читаемые идентификаторы. Во-первых, программа становится понятней, во-вторых, изменить та-

[15] Существует также возможность указания адреса целочисленной константой, однако учитывая тот факт, что FPC компилирует программы для защищенного режима, трудно спрогнозировать, куда этот адрес будет указывать во время выполнения.

[16] И результат тот же, не считая того, что на инициализированные переменные директива компилятора {$WRITEABLECONST OFF} не действует — они всегда доступны для изменения.

кое значение достаточно в одном месте — в определении константы, и в-третьих, уменьшается количество ошибок по причине опечаток, точнее — они распознаются на этапе компиляции.

Секция констант начинается с ключевого слова “const”, за которым следуют определения вида “*<идентификатор>* **=** *<значение>*;” — *простые* константы, или “*<идентификатор>* **:** *<тип>* **=** *<значение>*;”. Значение задается *константным выражением* или (для типизированных структурированного типа) *конструктором* константы.

Простые константы могут быть любого *простого* типа и типа-множества. Впрочем, при использовании их в выражениях, они подставляются «как есть», само значение нигде не хранится.

Типизированные константы, кроме простых типов и множеств, могут быть структурированного типа — записи и массивы. Конструктор константы-массива представляет собой список значений через запятую, заключенный в круглые скобки. Конструктор записи — перечень пар «идентификатор – значение» для каждого поля записи через точку с запятой, и также в круглых скобках.

Примеры:

```
const
  A = 3455;
  B = 324.2343;
  C = 'GTYHbdfff';
  D = 'X';
  E : Integer = 0;
  I : record Re, Im : Double end = (Re : 0.0; Im : 1.0);
  S : array [0..2] of string = ('First', 'Second', 'Third');
```

Заметим, что константа `D` в дальнейшем может трактоваться и как строка, и как символ.

Значение типизированных констант, в отличие от простых, в процессе работы программы может изменяться — по сути, это инициализированные переменные. Однако, такое поведение можно отменить директивой `{$WRITEABLECONST OFF}` Та же директива с ключом “`ON`”, соответственно, вновь разрешит изменение для тех констант, что будут описаны после нее.

Константные выражения

Константными называются выражения, которые могут быть вычислены на этапе компиляции. Они не могут ссылаться на значения каких-либо переменных, типизированных констант, или вызывать функции (кроме некоторых, определеных в модуле `System`) — см. таблицу 1.2. Причем, функции `High()` и `Low()` в константных выражениях допустимы только примененные к порядковому типу или статическому массиву.

1.4. Подпрограммы

Подпрограммы во Free Pascal бывают трех видов:

- *Процедуры* — не возвращают никакого значения, вызываются как оператор;

Категория	Функции	См. также
Порядковые	`Ord()`, `Chr()`, `Pred()`, `Succ()`, `Odd()`	1.5.1
Специальные	`SizeOf()`, `High()`, `Low()`	1.5, 1.5.1, 1.5.2
Математические	`Pi()`, `Abs()`, `Sin()`, `Cos()`, `ArcTan()`, `Exp()`, `Ln()`, `Frac()`, `Int()`, `Trunc()`, `Round()`, `Sqr()`, `Sqrt()`	

Таблица 1.2.: Функции, допустимые в константных выражениях

- *Функции* — возвращают значение, вызываются из выражений;
- *Операции, определяемые программистом* — возвращают значение, вызываются из выражений при использовании соотвествующих операций.

Общий синтаксис подпрограмм таков: *заголовок* (зависит от вида); *программный блок* (см. 1.1.3), составляющий *тело* подпрограммы.

Заголовок процедуры:

procedure *<идентификатор>* **(***<параметры>***);** *<модификатор(-ы)>***;**

Заголовок функции:

function *<идентификатор>* **(***<параметры>***) :** *<тип>***;** *<модификатор(-ы)>***;**

Заголовок определения операции (варианты):

operator *<символ>* **(***<параметры>***) :** *<тип>***;** *<модификатор(-ы)>***;**
operator *<символ>* **(***<параметры>***)** *<возврат>* **:** *<тип>***;** *<модификатор(-ы)>***;**

«Модификаторы» указывают тип вызова и некоторые другие характеристики подпрограммы, и во всех случаях не обязательны. «Тип» — идентификатор типа возвращаемого значения.

1.4.1. Параметры

Параметры характеризуются своим типом и способом передачи. В списке *формальные параметры* разделяются точкой с запятой.

Free Pascal предусматривает четыре способа передачи параметров:

- *Параметры-значения* — в списке "*<идентификатор>* **:** *<тип>*" — данные полностью помещаются в стек или в регистры.
- *Параметры-переменные* — "**var** *<идентификатор>* **:** *<тип>*" — в стек или в регистры помещаются не данные, а их адрес в памяти. Фактическим параметром в этом случае может быть только квалификатор переменной или поля.

- *Параметры-константы* — “**const** *<идентификатор>* : *<тип>*” — если размер данных меньше или равен размеру указателя, то в стек или регистры помещаются данные, иначе — их адрес, но любые изменения такого параметра в теле функции запрещены. При этом фактический параметр может быть и выражением: если его размер больше размера указателя, вычисленные данные будут размещены в куче, а в стек передан указатель на них.
- *Выходные параметры* — “**out** *<идентификатор>* : *<тип>*” — полностью соответствуют параметрам-переменным, за исключением того, что в качестве таких параметров можно передавать неинициализированные переменные, и компилятор не выдаст никакого предупреждения.

Несколько параметров одного типа и одного способа передачи подряд могут быть объединены в группу через запятые. Например, следующие два объявления эквивалентны:

```
procedure Dummy (A : Byte; B : Byte; const C : Word; const D : Word);
procedure Dummy (A, B : Byte; const C, D : Word);
```

Параметры, описываемые как “var”, “const” или “out” *могут не иметь типа.* В этом случае, работа с ними производится посредством приведения типов, через указатель или через локальную переменную, объявленную с ключевым словом “absolute”. Поскольку актуальный параметр может быть любого типа, следует позаботиться о передаче размера в явном виде.

Для параметров-значений и параметров-констант можно указать *значение по умолчанию* — “*<идентификатор>* : *<тип>* = *<значение>*”. В этом случае при вызове подпрограммы данный параметр можно не указывать — будет передано это значение. Однако такая возможность существует только в режимах Delphi и ObjFPC. Значения по умолчанию допустимы только для простых типов и множеств и должны быть константными выражениями.

Открытые массивы

Существует возможность создания процедур и фукнций с переменным количеством однотипных параметров. *Параметры – открытые массивы* объявляются следующим образом:

<Идентификатор> : **array of** *<Тип>*

Такие параметры могут использовать любой способ передачи: параметры-значения, “var”, “const” или “out”. В дальнейшем, в качестве такого параметра можно передавать произвольный массив элементов указанного типа, как статический, так и динамический, а в случае параметров-констант или параметров-значений можно также передать список значений через запятые, заключенный в квадратные скобки.

В теле процедуры к элементам открытого массива обращаться можно так же, как и к элементам обычного массива. Для определения верхней границы (наибольшего индекса) используется функция `High()`. Для опредения нижней границы можно использовать функцию `Low()`, однако в текущей реализации открытые массивы рассматриваются как начинающиеся с нулевого элемента (zero-based), таким образом, `Low()` всегда возвращает `0`.

Продемострируем на примере:

```
procedure Dummy (const A : array of Integer);
 var
   I : Integer;
 begin
 for I := 0 to High(A) do
     WriteLn (A[I])
 end;

begin
Dummy ([1,2,12])
end.
```

array of const

В режимах компиляции Delphi и ObjFPC еще бóльшую свободу предоставляет следующая конструкция:

<Идентификатор> **: array of const**

В отличие от открытых массивов, в данном случае параметр может передаваться только как параметр-значение или параметр-константа. При вызове используются квадратные скобки со списком значений, значения могут быть любого типа из следующих: булево значение, целое число, вещественное число, строка символов, единичный символ, указатель, объект, класс.

В теле подпрограммы данный параметр трактуется как массив записей приблизительно[17] следующего типа:

```
type
  TVarRec = record
            case VType : Longint of
                 vtInteger    : (VInteger    : Longint);
                 vtBoolean    : (VBoolean    : Boolean);
                 vtChar       : (VChar       : Char);
                 vtExtended   : (VExtended   : PExtended);
                 vtString     : (VString     : PShortString);
                 vtPointer    : (VPointer    : Pointer);
                 vtPChar      : (VPChar      : PChar);
                 vtObject     : (VObject     : TObject);
                 vtClass      : (VClass      : TClass);
                 vtAnsiString : (VAnsiString : Pointer);
                 vtWideString : (VWideString : Pointer);
                 vtInt64      : (VInt64      : PInt64);
            end;
```

1.4.2. Выход из подпрограмм

Нормальным выходом считается случай, когда подпрограмма выполняется до своего завершающего “end”. Кроме того, в теле подпрограммы могут присутствовать и другие точки выхода, для чего служит ключевое слово “exit”, что также является *штатным*, хотя и преждевременным выходом.

[17] Определение типа TVarRec взято из документации [1, Chapter 10, p. 92] — реально в модуле ObjPas находится более сложное и расширенное определение, которое, однако, сути не меняет.

Также выполнение подпрограммы может быть прервано с возникновением *исключительной ситуации*. См. главу 3, раздел 3.1.

Возвращаемое значение

Free Pascal поддерживает три способа указания возвращаемого значения *функции*:

- Значение присваивается идентификатору функции (a-la Turbo Pascal). Данный способ применим во всех режимах компиляции.
- Значение присваивается псевдопеременной "`result`" (a-la Delphi). Данный способ поддерживается только в режимах ObjFPC и Delphi.
- Значение указывается ключевым словом "`exit`" с параметром в круглых скобках: "**`exit (`***`<значение>`***`)`**". Данный вариант также доступен во всех режимах.

Конструкция "`result`" может использоваться не только в левой части оператора присваивания, но и в выражениях в теле функции. В случае идентификатора функции, если она не имеет параметров, такое использование синтаксически неотличимо от рекурсивного вызова.

Чтобы допустить использование идентификатора функции как переменной в режимах FPC и ObjFPC принято следующее соглашение: идентификатор функции без скобок в теле функции соответсвует текущему присвоенному значению, а чтобы использовать рекурсию, требуется указать скобки, даже для функций, не имеющих параметров. В других режимах любое использование идентификатора функции, кроме как в левой части присваивания, трактуется как рекурсивный вызов.

1.4.3. Модификаторы

Модификаторы во Free Pascal являются *ключевыми словами*.

"**`alias : '`*`<имя>`*`'`**" задает имя-синоним, по которому к данной подпрограмме можно будет обратиться, например, из внешнего объектного файла — см. главу 3, разделы 3.6 и 3.4.

"**`assembler`**" — ассемблерная подпрограмма, в которой *блок исполняемых операторов* ("`begin ... end`") заменен на *ассемблерный блок* ("`asm ... end`") — см. главу 3, раздел 3.6.

"**`cdecl`**" — подпрограмма использует соглашения о вызове, принятые в языке C: параметры помещаются в стек справа налево, очистка стека производится вызывающей программой. Кроме того, внутреннее имя подпрограммы с данным модификатором тождественно ее идентификатору, тогда как обычно Free Pascal формирует составное внутреннее имя по определенным правилам.

"**`export`**" — помечает подпрограмму как доступную для экспорта, распознается для совместимости с Borland Pascal, реально ничего не делает. *???*

"**`external`**" — обозначает внешнюю подпрограмму. См. главу 3, раздел 3.4.

"**forward**" — означает предварительное объявление подпрограммы — только заголовок, а полное объявление находится далее в том же исходном файле, аналогично тому, как в интерфейсной части модуля описывается заголовок, а сама подпрограма — в части реализации.

"**inline**" — объявляет inline-подпрограмму. Код такой подпрограммы *не вызывается*, а *вставляется* непосредственно в место, где она использована. Inline-подпрограммы подчинены следующим ограничениям:

1. Непосредственная вставка доступна только в рамках одного модуля (исходного файла). При использовании в другом исходном файле такая подпрограмма будет вызвана обычным образом.
2. Для inline-подпрограмм запрещена рекурсия.
3. Inline-подпрограмма не может быть ассемблерной[18].

Использование inline-подпрограмм по умолчанию недоступно, включить его можно директивой `{$INLINE ON}` или через ключ командной строки.

"**interrupt**" — используется при написании обработчиков прерываний для тех платформ, где это возможно.

"**overload**" — требуется при *перегрузке* подпрограмм (см. 1.4.5) в режиме Delphi. В режиме ObjFPC игнорируется.

"**pascal**" — подпрограмма использует соглашения о вызове, принятые в Turbo Pascal: параметры помещаются в стек слева направо, очистку стека производит вызываемая подпрограмма.

"**popstack**" — соглашения о вызове полностью аналогичны "`cdecl`", однако внутреннее имя подпрограммы формируется по умолчанию.

"**public**" — делает подпрограмму публикуемой в объектном файле. См. главу 3, разделы 3.6 и 3.4.

"**register**" — параметры передаются через регистры общего назначения вместо стека. Точнее, таким образом передаются первые три параметра.

"**saveregisters**" — указывает компилятору сохранять перед вызовом подпрограммы регистры в стеке и восстанавливать их при выходе.

"**safecall**" — соответствует модификатору "`stdcall`" вкупе с сохранением всех регистров.

"**stdcall**" — соглашения о вызове соответствуют принятым в стандартных библиотеках Microsoft Windows: параметры помещаются в стек справа налево, очистку стека производит вызываемая подпрограмма.

"**varargs**" — используется совместно с модификатором "`cdecl`" для *внешних* подпрограмм. Означает, что после задекларированных параметров может располагаться произвольное количество других без какой-либо проверки типа (подобно "`array of const`"), при этом квадратные скобки не требуются.

[18] Это ограничение относится к описываемой версии компилятора (2.0.0) и более старым. Не исключено, что в новых версиях оно будет снято или смягчено.

1.4.4. Определение операций

Free Pascal в режимах FPC и ObjFPC позволяет программисту определить собственные операции для своих типов. Проиллюстрируем на примере:

```
{$MODE OBJFPC}

type
  Complex = record
            Re, Im : Double
            end;

operator + (X, Y : Complex) : Complex;
 begin
 result.Re := X.Re + Y.Re;
 result.Im := X.Im + Y.Im
 end;

operator := (X : Double) : Complex;
 begin
 result.Re := X;
 result.Im := 0
 end;

var
  X, Y, Z : Complex;

begin
 X := 10.2;
 Y.Re := 0.0;
 Y.Im := 1.0;
 Z := X + Y
end.
```

Как видим, определение операций похоже на определение функций, за исключением того, что начинается ключевым словом “operator”, а вместо идентификатора используется символ операции. Кроме того, список параметров определяется характером операции: для бинарных операций параметров два, для унарных — один.

Особняком стоит определение оператора присваивания “:=”. Впрочем, можно сказать, что определяется не оператор присваивания, а операция *приведения типа*. В дальнейшем данная операция может использоваться и неявно.

В режиме FPC нет псевдопеременой “result”, и в то же время операции, в отличие от функций, не имеют идентификатора, которому можно присвоить значение. Поэтому должна использоваться следующая форма заголовка:

operator *<символ>* **(***<операнды>***)** *<результат>* **:** *<тип>*;

Здесь «результат» — идентификатор, котрому будет присвоено результирующее значение, своего рода out-параметр. Операция “+” из приведенного выше примера будет иметь следующий вид:

```
operator + (X, Y : Complex) R : Complex;
 begin
 R.Re := X.Re + Y.Re;
 R.Im := X.Im + Y.Im
 end;
```

Могут быть определены следующие операции:

- Присваивание / приведение типа: “:=”;
- Арифметические: “+”, “-”, “*”, “/”, “**”, “div”, “mod”, унарный “-”;
- Логические[19]: “and”, “or”, “xor”, “not”, “shl”, “shr”;
- Сравнения: “=”, “<”, “>”, “<=”, “>=”.

Операция сравнения «не равно» — “<>” не переопределяется, а считается определенной как отрицание операции «равно» — “=”.

1.4.5. Расширения языка

Перегрузка подпрограмм

Перегрузкой (overloading) называется определение нескольких подпрограмм с одним идентификатором, отличающихся по составу параметров. При этом состав параметров должен быть таков, чтобы полностью исключить возможные неоднозначности, т.е. если количество параметров, например, одинаково, а различаются только типы, то эти типы дожны быть не просто разными, но и несовместимыми.

Перегружаемые подпрограммы должны быть объявлены в одной области видимости, например, в одном модуле. В противном случае их идентификаторы будут замещаться в соответствии с приоритетом.

Перегрузка подпрограмм возможна в режимах FPC, ObjFPC и Delphi, причем в последнем случае требуется ключевое слово “overload” в заголовках подпрограмм.

Расширенный синтаксис

Расширенный синтаксис позволяет вызывать *функции* как *процедуры*, игнорируя возвращаемое ими значение. Это может быть полезно для написания, использования функций, содержательная задача которых — выполнение некоторых действий, а возвращаемое значение лишь содержит некую информацию состояния.

По умолчанию расширенный синтаксис включен, однако его можно запретить директивой компилятора {$EXTENDEDSYNTAX OFF} или {$X-}.

[19] В документации об определении операций данной группы ничего не сказано, но на практике — работает.

1.5. Типы

Тип в языке Pascal определяет формат представления данных, а также набор допустимых действий над ними. Как правило, определяя тип, программист определяет и ряд подпрограмм, которые с данными такого типа работают. В особенности это относится к объектным типам данных, подпрограммы-методы для которых являются неотъемлемой частью самого типа.

Типы определяются в секции описаний, начинающейся с ключевого слова "`type`", за которым следуют конструкции вида:

<Идетификатор> **=** *<определение>;*

Определение может быть конструктором, зависящим от того, какой именно тип определяется; идентификатором ранее определенного типа — в этом случае говорится о создании типа-синонима — в дальнейшем использование любого из идентификаторов равнозначно; или конструкцией вида "**type** *<идентификатор типа>*" — в этом случае определяемый тип будет *совместимым*, но не тождественным ранее определенному типу.

Основные типы, такие как `Integer`, `Boolean`, `Pointer` и т.д. являются предопределенными. Переопределение их идентификаторов возможно, но крайне нежелательно.

1.5.1. Простые типы

Понятие простого типа несколько расплывчато, мы будем исходить из функциональной посылки и считать простыми те типы, для которых возможны *константные выражения*.

Порядковые типы. К порядковым типам относятся *целые* числа (кроме типов `QWord` и `Int64`), *булевы* типы, *перечисления*, тип `Char`, а также *типы-диапазоны*, построенные на них.

Все порядковые типы приводятся к целому типу функцией `Ord()`, кроме того, для любого из них можно получить наименьшее и наибольшее значения функциями `Low()` и `High()`, а также значение, предшествующее данному, функцией `Prev()` и значение, следующее за данным, функцией `Succ()`.

Значения порядковых типов, как и следует ожидать, считаются *упорядоченными*, таким образом, для них определены операции «больше» и «меньше».

Символьные типы. Еще одна функциональная группа различных по структуре типов. Сюда входят типы `Char`, `WideChar` и строковые типы. Общее у них то, что они совместимы друг с другом и допускают представление в виде *строковых констант*.

Целые числа

Предопределенные целые типы представлены в таблице 1.3.

Кроме перечисленных в таблице, предопределены следующие целые типы, диапазон и размер которых зависит от режима компиляции и/или целевой платформы: `Integer` — в режимах компиляции, отличных от ObjFPC и Delphi соответствует

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

Таблица 1.3.: Целые типы

типу SmallInt, а в указанных режимах — SmallInt, LongInt или Int64 в зависимости от платформы; тип Cardinal обычно соответствует LongWord, а в режимах ObjFPC и Delphi — Word, LongWord или QWord, аналогично Integer; наконец, тип PtrInt — целое со знаком, по размерности соответствующее указателю.

Вещественные числа

В данную группу принято относить числовые типы, для работы с которыми привлекается математический сопроцессор. Они не являются ни целыми, ни порядковыми, хотя значения данных типов и образуют упорядоченное множество. Для вещественных типов определены математические операции и операции сравнения.

Вещественные типы перечислены в таблице 1.4. Все они, кроме типов Comp и Currency являются числами *с плавающей точкой*, т.е. представлены в виде пары «мантисса – порядок».

Тип	Диапазон	Значащих цифр	Размер
Single	$1.5*10^{-45}$ .. $3.4*10^{38}$	7–8	4B
Double	$5.0*10^{-324}$ .. $1.7*10^{308}$	15–16	8B
Extended	$1.9*10^{-4951}$ .. $1.1*10^{4932}$	19–20	10B
Comp	-2^{63} .. $2^{63}-1$	19–20	8B

Таблица 1.4.: Вещественные типы

Тип Comp по сути преставляет собой целое число, соответствующее типу Int64, однако трактуется в выражениях как число вещественное. Помимо прочего, сама возможность использования типов Comp и Extended зависит от возможностей целевой платформы, тогда как типы Single и Double определены стандартом IEEE-754.

Помимо перечисленных в таблице 1.4 типов Free Pascal поддерживает следующие:

- Real — данный тип (аналогично целым Integer и Cardinal) зависит от платформы и трактуется как Double или Single.
- Real48 — 48ми-битное вещественное число. Данный тип соответствует типу Real в Turbo Pascal и введен для совместимости. Использовать его не рекомендуется, поскольку такой формат не поддерживается никаким сопроцес-

сором и преобразуется компилятором в один из других вещественных типов, что влечет потери в быстродействии.

- `Currency` — число с фиксированной точкой. Занимает 8 байт и соответствует приблизительно типу `Comp` «поделенному» на 10000. Иными словами — 64х-битное число с фиксированной точкой с четырьмя десятичными знаками после запятой.
- `TDateTime` — тип для представления значений даты/времени. Данный тип тождественен типу `Double`, где целая часть содержит дату, а дробная — время. Хотя сам тип определен в модуле `System`, процедуры и функции для работы с ним размещены в других модулях: `SysUtuls` и `DateUtils`. Следует заметить, что данный тип — не единственный и не лучший способ представления даты и времени, поддерживаемый FPC. Его главное достоинство — совместимость с Delphi и Turbo Pascal.

Булевы типы

Булевы типы отличаются от прочих тем, что их значения непосредственно используются в разного рода условиях. В целях совместимости предопределено несколько булевых типов, для всех "`Ord(false) = 0`". См. таблицу 1.5.

Тип	**Размер**	**`Ord(true)`**
`Boolean`	1B	1
`ByteBool`	1B	Любое ненулевое значение
`WordBool`	2B	Любое ненулевое значение
`LongBool`	4B	Любое ненулевое значение

Таблица 1.5.: Булевы типы

Перечисления

Тип-перечисление задается набором значений, каждому из которых соответствует порядковый номер. В отличие от использования целочисленных констант, такой тип позволяет контролировать допустимые значения на этапе компиляции. Конструктор типа представляет собой список идентификаторов через запятую, заключенный в круглые скобки. Кроме того, для элемента списка может быть явно указано его целочисленное представление. На примере:

```
type
  TTest = (ttZero, ttOne, ttTwo, ttTen := 10, ttEleven);
```

Здесь `Ord(ttZero) = 0`, `Ord(ttOne) = 1`, ..., `Ord(rrTen) = 10`, `Ord(ttEleven) = 11`, то есть нумерация после явного указания номера продолжается с него, а не последовательно от предшествующих. Нельзя указывать номера в обратном порядке, то есть элемент с большим номером не может предшествовать элементу с меньшим.

Размер типа, то есть объем памяти, отводимый под соответствующую переменную, зависит от директивы компилятора `{$PACKENUM n}`, где `n` может принять одно из следующих значений: 1, 2, 4, `NORMAL` или `DEFAULT`. Два послед-

них варианта в текущей реализации обозначают 4 байта. Значение по умолчанию зависит от режима компиляции. Синонимами данной директивы выступают {$MINENUMSIZE *n*} и {$Z*n*}.

Тип-диапазон

Тип-диапазон определяет подмножество некоторого порядкового типа, задавая минимальное и максимальное допустимые значения. Таким образом, проверка вхождения в заданный диапазон выполняется на уровне компиляции.

Конструктором такого типа является *диапазон*, см. определение на странице 11. Например:

```
type
  TFirst = 122..28282;
  TSecond = 'A'..'Z';
```

Типы Char и WideChar

Тип Char предназначен для представления единичного 8ми-битного символа и имеет размер, соответственно, 1 байт. Тип WideChar предназначен для представления 16ти-битного символа Unicode, и имеет размер 2 байта.

Будучи порядковыми типами, Char и WideChar могут использоваться в диапазонах; Char может служить базовым типом для множества и т.д. В то же время, они являются символьными типами, и их значения могут быть указаны простыми строковыми константами.

Строковые типы

К строковым типам принято относить «длинные» и «короткие» строки, работа с которыми, несмотря на различие внутреннего строения, осуществляется одинаково. Для строк определена операция конкатенации "+" — второй операнд дописывается в конец первого, длина определяется стандартной функцией Length(), а к составляющим строку символам можно обратиться по индексу через квадратные скобки, как к элеменам массива — индексация символов в строке начинается с единицы.

```
var
  S : AnsiString;
  C : Char;

begin
 S := 'This is a string!';
 C := S[2] // C = 'h'
end;
```

Также для строковых типов определен ряд стандартных функций, реализующих такие действия как: поиск подстроки — Pos(), выделение подстроки — Copy(), и т.д.

«Короткие» строки — данный строковый тип (точнее, семейство типов) унаследован из ранних версий языка Pascal. Это статические типы, то есть память, занимаемая переменной-строкой не зависит от ее текущего значения.

По структуре это массивы символов, где первый (с нулевым индексом) символ трактуется как байт, содержащий текущую длину строки.

- **`string[`***`<длина>`***`]`** — позволяет объявить строку максимальной длины «длина».
- `ShortString` — короткая строка максимальной длины, то есть — эквивалент конструкции "**`string`**`[255]`".
- **`string`** (без указания длины) — если не включена опция «длинных строк», то синоним `ShortString`; в противном случае — синоним `AnsiString` (см. далее).

  Режим длинных строк включается директивой компилятора `{$LONGSTRINGS ON}`, или `{$H+}`.

«Длинные» строки — тип `AnsiString` — строки с динамически изменяющимся размером. Длина такой строки ограничена только возможностями динамической памяти (которые зависят от платформы и других факторов). Кроме того, в конце каждой строки находится нулевой символ для совместимости с принятым в различных языках, и в первую очередь — C/C++, представлением строковых данных.

Управление выделением и освобождением памяти для «длинных» строк производится полностью автоматически, что приводит к некоторым проблемам при их использовании в разделяемых библиотеках. Данный вопрос будет подробно рассмотрен в главе 3, раздел 3.4.

Строки Unicode — тип `WideString` («широкие» строки) — также динамический тип. От `AnsiString` отличаются только тем, что содержат 16ти-битные символы Unicode.

Работа с Unicode требует отдельного рассмотрения, что и будет слелано в главе 3, раздел 3.7.

Типы `PChar` и `PWideChar`

Данные типы представляют собой указатели на `Char` и `WideChar` соответственно. Тем не менее, компилятор трактует их не только как указатели, но и как символьные типы: им может присваиваться значение строковой константой, они совместимы по присваиванию со строковыми типами и т.д. Когда данные типы используются как строки, предполагается, что они завершаются нулевым символом.

При использовании данных типов следует помнить, что, во-первых, для них отсутствует автоматическое управление выделением и освобождением памяти[20], а во-вторых, индексация символов в них начинается с нуля, а не с единицы, как в строках.

[20] Что не всегда недостаток, поскольку означает контроль программиста за данным процессом.

Указатели

Указатели используются для косвенного (по ссылке) обращения к данным. В языке Pascal их основное назначение — работа с динамической памятью.

Указатели могут быть:

- *Нетипизированные* — тип `Pointer` — как правило, используется, когда тип данных не определен заранее.
- *Типизированные* — определяются конструкцией "`^`*<тип>*", соответственно указывают на данные определенного типа.

Ко всем указателям применима операция разыменования (см. стр. 12): для типизированного указателя результат эквивалентен переменной базового типа; а в случае нетипизированного — тип результата не определен и может использоваться только в качестве параметра неопределенного типа или в сочетании с явным приведением.

Работа с динамической памятью производится посредством стандартных подпрограмм `New()` и `Dispose()` или `GetMem()` и `FreeMem()`. Данные подпрограммы выделения/освобождения памяти должны использоваться попарно, т.е. нельзя освободить память, выделенную посредством `New()`, процедурой `FreeMem()` и наоборот.

Подробно работа с динамической памятью описывается в главе 3, раздел 3.2, здесь укажем лишь, что пара `GetMem() / FreeMem()` использует явное указание объема выделяемой памяти, а `New() / Dispose()` ориентируется на тип указателей. Кроме того, в отличие от Delphi и Turbo Pascal, `new` и `dispose` являются *ключевыми словами*.

Обращение к данным, на которые ссылается указатель, производится посредством операции разыменования. Однако Free Pascal допускает исключения из этого правила:

- Как описывалось выше, обращение к отдельным символам строки типа `PChar` и `PWideChar` производится так же, как к элементам массива. Такой синтаксис действует во всех режимах компиляции.
- В режиме Delphi действуют исключения, принятые в Object Pascal от Borland: при обращении к элементу массива или полю записи, знак разыменования может опускаться.
- В режимах FPC и ObjFPC любой типизированный указатель может рассматриваться как массив, аналогично `PChar` и `PWideChar`.

Данные исключения лучше всего продемонстрировать на примерах:

```
type
  TTest = record F : Integer end;
  PTest = ^TTest;
  TTestArray = array [0..9] of TTest;
  PTestArray = ^TTestArray;

var
  PChr : PChar;
```

```
  PTst : PTest;
  PArr : PTestArray;

begin
GetMem (PChr, 10);
GetMem (PTst, SizeOf(TTest) * 10);
GetMem (PArr, SizeOf(TTestArray));

// Работает всегда
PChr[2] := 'X';
PTst^.F := 7;
PArr^[2].F := 8;

// Только в режиме Delphi
PTst.F := 7;
PArr[2].F := 8;

// Только в режимах FPC и ObjFPC
PTst[2].F := 8;

FreeMem (PChr);
FreeMem (PTst);
FreeMem (PArr)
end.
```

Адресная арифметика. Free Pascal поддерживает для указателей следующие операции:

- *Изменение указателя на целое число* — "`P + I`" или "`P - I`". При этом, значение указателя, если он типизированный, изменяется не на `I`, а на `I * SizeOf(<базовый тип P>)`. То есть число `I` следует рассматривать не как смещение в байтах, а как индекс массива. Для нетипизированных указателей `I` — это смещение в байтах.
- *Разность указателей* — "`P1 - P2`". Если оба указателя типизированные и имеют один тип, то результат — разность в байтах, деленная на размер базового типа. В противном случае оба операнда рассматриваются как нетипизированные указатели, и результат — разность в байтах.

Хотя использование адресной арифметики может быть в каких-то случаях удобным, отсутствие какого-либо контроля и неоднозначность индексации делают ее уязвимой для появления трудноуловимых ошибок. Рекомендуется очень осторожно использовать подобные средства языка, тем более что Free Pascal поддерживает динамические массивы, работа с которыми существенно безопаснее и проще в отладке.

Предварительное объявление. В языке Pascal существует одно исключение из общего правила, запрещающего использование какого-либо идентификатора до его объявления. Тип-указатель можно объявить *перед* типом, на который он ссылается, однако это обязательно должно быть сделано в той же секции "`type`". Такое исключение сдулано для удобства программирования таких структур, как, например, связанные списки.

```
type
  PItem = ^TItem;
  TItem = record
          Data : Integer;
          Next : PItem
          end;
```

Процедурные типы

Процедурные типы — по сути те же указатели, однако они указывают не на данные, а на подпрограммы. Определение процедурного типа похоже на определение процедуры или функции без идентификатора. Например:

```
type
  TRealFunc = function (X : Double) : Double; stdcall;
```

Переменной такого типа присваивается адрес некоторой функции, объявленной таким же образом. Синтаксис взятия адреса и обращения к переменной различается в зависимости от режима компиляции. Пусть `V` — переменная типа «функция», а `F` — функция, соответствующая по заголовку. Тогда:

- *В режиме TP*
  - Присвоение значения переменной — “`@V := @F`”;
  - `V` — вызов функции;
  - `@V` — адрес функции — значение переменной;
  - `@@V` — адрес переменной;
- *В режиме Delphi*
  - Присвоение значения переменной — “`@V := @F`” или “`V := @F`”;
  - `V` — вызов функции;
  - `@V` — адрес функции — значение переменной;
  - `@@V` — адрес переменной;
- *В режимах FPC и ObjFPC*
  - Присвоение значения переменной — “`V := @F`”;
  - `V` — адрес функции — значение переменной, для вызова функции требуются круглые скобки, даже если нет параметров — “`V()`”;
  - `@V` — адрес переменной;
- *В режимах GPC и MacPas*
  - Присвоение значения переменной — “`@V := @F`” или “`V := F`”;
  - `V` — вызов функции;
  - `@V` — адрес функции — значение переменной;
  - `@@V` — адрес переменной;

Из-за стольких различий рекомендуется при использовании процедурных типов явно указывать режим компиляции директивой `{$MODE xxx}` непосредственно в исходниках, не полагаясь на настройки компилятора.

1.5.2. Структурированные типы

Массивы

Массив представляет собой одно- или многомерную таблицу, элементами которой являются значения некоторого базового типа. Массивы во Free Pascal могут быть *статическими* — количество элементов задано на этапе компиляции, или *динамическими* — количество элементов может меняться во время выполнения программы.

Обращение к элементу массива производится посредством индексов в квадратных скобках:

```
A[23]
F[i,j]
```

Для любого массива можно получить минимальное и максимальное значение его первого индекса функциями `Low()` и `High()`.

Статические массивы. Конструктор статического массива состоит из ключевого слова "`array`" за которым следует список измерений в квадратных скобках, после чего ключевое слово "`of`" и идентификатор или конструктор базового типа.

```
type
  <идентификатор> = array [<изм-1>, ..., <изм-N>] of <базовый тип>;
```

Здесь «изм-1» ... «изм-N» — измерения, задаваемые произвольным порядковым типом (идентификатором или конструктором), как правило используется конструктор типа-диапазона. Количество и размер измерений может быть любым, существует лишь ограничение на общий размер любого типа во Free Pascal — 2GB минус 1B (для 32х-разрядных процессоров)[21].

Если базовым типом является другой массив, то тип в результате соответствует тому, как если б измерения базового записать в конец списка измерений текущего массива. Например, следующие два объявления задают тип одной и той же структуры:

```
type
  TFirst = array [0..12] of array [4..8] of Integer;
  TSecond = array [0..12, 4..8] of Integer;
```

К элементам массива любого из этих типов можно обращаться как "`A[i,j]`" и как "`A[i][j]`".

Динамические массивы. Объявление динамического массива аналогично статическому объявлению, за исключением того, что квадратные скобки с измерениями отсутствуют. Динамический массив считается одномерным, индексы — целые числа, индексация начинается с нуля. Тем не менее, объявление вида "`array of array`" позволяет создавать и многомерные динамические массивы. Примеры:

[21] В документации [2, Chapter 8, p. 77] сказано, что размер не ограничен, т.е. ограничен только возможностями машины. Это *почти* так, но не совсем.

```
type
  TArray = array of Integer;
  TMatrix = array of array of Double;
```

Размер динамического массива задается стандартной процедурой `SetLength()`, при этом память выделяется и освобождается автоматически.

Записи

Тип запись представляет собой структуру из нескольких полей произвольного типа. Таким образом записи объединяют *неоднородные* данные, в отличие от массивов.

Поле записи идентифицируется посредством квалификатора.

Определение типа записи состоит из ключевого слова "`record`", за которым следует список полей, определяемых подобно переменным, и завершает все ключевое слово "`end`".

Например:

```
type
  Complex = record
            Re, Im : Double
            end;
```

Перед ключевым словом "`record`" может быть указано ключевое слово "`packed`", которое предписывает компилятору не выравнивать поля по границе слова или двойного слова. Обычно такое выравнивание производится для ускорения обращений к данным. Выравнивание регулируется директивой компилятора `{$PACKRECORDS n}`, где `n` может принимать значения: 1, 2, 4, 8, 16, `C` и `DEFAULT`. Значение по умолчанию — 2.

Тип-запись может содержать *вариантную часть* — поля, накладывающиеся друг на друга. Вариантная часть может располагаться только после всех обязательных полей и начинается с ключевого слова "`case`", за которым следует *селектор* вариантной части — это может быть идентификатор порядкового типа, или поле порядкового типа (само это поле будет обязательным), затем ключевое слово "`of`" и список вариантов, где список полей каждого варианта заключен в круглые скобки. Например:

```
type
  TData = packed record
          case Byte of
               1 : (L : LongWord);
               2 : (W1, W2 : Word);
               3 : (B1, B2, B3, B4 : Byte)
          end;
```

Такая запись позволяет рассматривать одну и ту же область памяти размером 4В как одно двойное слово, два слова или четыре отдельных байта. Другой пример записи с вариантной частью — описание типа `TVarRec` на стр. 26.

Заметим, что даже если селектор является полем записи, значение этого поля при обращении к полям вариантной части не проверяется. Вариантные части

могут быть вложенными. Вариантная часть не может содержать полей, которые предусматривают автоматическое управление памятью: «длинных» и «широких» строк, динамических массивов и интерфейсов.

Множества

Типы-множества моделируют математическое понятие множества. Данные такого типа хранят информацию о наличии/отсутствии элемента во множестве. Конструктор типа выглядит так:

set of *<базовый тип>*

Базовым типом может быть любой порядковый тип, содержащий не более 256 возможных значений. Если базовый тип содержит не более 32 значений, размер переменной-множества 4 байта (32 бита), в противном случае — 32 байта (256 бит).

Для множеств определены операции объединения, пересечения и разности, проверка вхождения элемента в множество, а также стандартные процедуры включения и исключения элемента:

```
Include(<множество>, <элемент>);
Exclude(<множество>, <элемент>);
```

Каждому элементу в данных типа «множество» соответствует один бит. Таким образом мнодества можно использовать в качестве крайне упакованных массивов булева типа.

Конструктор значений типа-множества описан в подразделе 1.1.2 на странице 11.

1.5.3. Объектные типы

Объектно-ориентированному программированию посвящена глава 2, где и будут описаны объектные типы языка Pascal. Здесь скажем лишь, что Free Pascal поддерживает три разновидности таких типов: `object`-типы, впервые появившиеся еще в Turbo Pascal, классы Delphi и интерфейсные типы, появившиеся в Delphi 5. Классы и интерфейсы существуют только в режимах ObjFPC и Delphi.

1.5.4. Файловые типы

Стандарные, независимые от платформы, средства работы с файлами Free Pascal оперируют со следующими типами:

- *Текстовые файлы* — типы `Text` и `TextFile`. Ввод/вывод осуществляется последовательно в текстовом формате.
- *Типизированные файлы* — типы, определяемые как “**file of** *<базовый тип>*”, где базовым может быть любой тип, кроме динамических («длинные» и «широкие» строки, динамические массивы), записей с вариантной частью, вариантов, объектных и файловых типов. Такой файл представляется как последовательность элементов фиксированного размера. Ввод/вывод может производиться в произвольном порядке.

- *Нетипизированные файлы* — тип “`file`”. О содержимом такого файла компилятор не делает никаких предположений — вся обработка определяется программистом. Ввод/вывод происходит в произвольном порядке, блоками данных произвольной длины.

Помимо базовых средств, в стандартных модулях Free Pascal находится множество различных подпрограмм для работы с файлами. Подробное описание этой темы см. в главе 3, раздел 3.3.

1.5.5. Тип `Variant`

Тип `Variant` предназначен для хранения данных, тип которых при компиляции еще не определен. Значениями переменной типа `Variant` могут быть строки, числа, данные любого порядкового типа и интерфейсы. В выражениях варианты используются как и переменные любого из этих типов. Если текущий тип не подходит, генерируется ошибка времени выполнения.

Преобразования типов, управление памятью и т.д. для данного типа выполняется автоматически, независимо от программиста.

Поддержка вызова диспетчеризованных методов (OLE Automation) через варианты на данный момент (версия компилятора 2.0.0) не реализована.

1.5.6. Совместимость и преобразование типов

Совместимость

В языке Pascal можно выделить три вида совместимости:

- *Полная* совместимость — данные одного типа, или типов, объявленных как синонимы. Типы-синонимы могут использоваться один вместо другого всегда, в том числе при указании актуальных `var`- и `out`-параметров.
- Совместимость *в операциях сравнения.* Зависит от того, как определены сами операции. Если говорить о стандарных, предопределенных сравнениях, то между собой сравнимы: а) все числовые типы; б) все строковые типы; в) множества одного базового типа.
- Совместимость *по присваиванию.* Кроме оператора присваивания такая совместимость требуется при указании актуальных параметров-значений и параметров-констант. Типы могут быть сделаны совместимыми переопределением оператора присваивания (см. 1.4.4). Стандартная совместимость по присваиванию приведена в таблице 1.6. Следует отметить, что данный вид совместимости односторонний, т.е. допустимость оператора “`A := B`” еще не означает допустимости “`B := A`”.

Преобразования типов

Будем различать *преобразование* и *приведение* типов. При предобразовании для некоторого значения одного типа находится соответствующее значение второго. При приведении же исходное значение *расматривается как* значение второго

Приемник		**Источник**
Любой целый	<=	*Любой целый*
Любой целый	<=	`Variant`
Любой вещественный	<=	*Любой вещественный*
Любой вещественный	<=	*Любой целый*
Любой вещественный	<=	`Variant`
Любой булев	<=	*Любой булев*
Любой булев	<=	`Variant`
Диапазон	<=	*Базовый тип диапазона*
Базовый тип диапазона	<=	*Диапазон*
Любой строковый	<=	*Любой строковый*
Любой строковый	<=	`Char`, `WideChar`
Любой строковый	<=	`PChar`, `PWideChar`
Любой строковый	<=	`Variant`
`Pointer`	<=	*Любой указатель*
`Pointer`	<=	*Любой процедурный*
Любой указатель	<=	`Pointer`
Любой порядковый, `Int64`, `QWord`	<=	`Variant`
`Variant`	<=	*Любой порядковый*, `Int64`, `QWord`
`Variant`	<=	*Любой вещественный*
`Variant`	<=	*Любой строковый*
`Variant`	<=	*Любой интерфейсный*

Таблица 1.6.: Совместимость по присваиванию

типа — с точки зрения программиста, компилятор может и выполнять какие-то преобразования.

Для преобразования типов существуют стандартные функции и процедуры. В первую очередь это процедуры `Val()` и `Str()`, преобразующие строку в число и наоборот; кроме того следует упомянуть функцию `Trunc()`, которая выполняет округление вещественного числа и возвращает значение целого типа (другие функции округления возвращают значение вещественного типа). Кроме этих существует множество других функций преобразования, расположенных в различных стандартных модулях, в основном — модуль `SysUtils`.

Приведение типа может быть явное и неявное. В первом случае программист сам указывает к какому типу приводится значение, а во втором тип выбирается компилятором, исходя из контекста.

- *Неявное* приведение выполняется при указании в параметрах подпрограмм или выражениях, значений, *совместимых по присваиванию* с требуемыми. Иногда в выражениях возможности неявного приведения могут вызывать неоднозначность: компилятор объявит об ошибке. В таком случае следует использовать явное приведение.
- *Явное* приведение задается программистом как “*<тип>*`(`*<значение>*`)`”. Такое приведение возможно в следующих случаях:
  1. Типы совместимы по присваиванию;
  2. Оба типа порядковые;
  3. Оба типа — указатели, процедурные, «длинные» строки, динамические

массивы, классы или интерфейсы (т.е. указатели по внутреннему строению);

4. Типы имеют один и тот же размер;
5. «Значение» имеет неопределенный тип — параметр неопределенного типа или разыменование нетипизированного указателя.

Отдельно следует упомянуть приведение *классов* операцией “`as`” — в отличие от простого явного приведения, в этом случае выполняется некоторый контроль допустимости. Подробнее см. в главе 2.

2. Объектно-ориентированное программирование

3. Технологии программирования

3.1. Обработка исключений

3.2. Управление памятью

3.3. Работа с файлами

3.4. Внешние библиотеки

3.5. Многопоточность

3.6. Ассемблер

3.7. Поддержка Unicode

3.8. RTTI

4. Использование компилятора и утилит

4.1. Состав дистрибутива

4.2. Настройки компиляции

4.3. Отладка

4.4. Утилиты

5. Кросскомпиляция

Часть II.

Приложения

А. Перечень ключевых слов

Ссылки

[I] **Официальная документация Free Pascal**

Документация и руководства от разработчиков компилятора. Ссылки на страницы даны по PDF-версии документации. Вся документация — на английском языке.

Свежую версию документации можно найти по адресу:
`http://www.freepascal.org/docs.html`

[1] *Michaël Van Canneyt, «Free Pascal : Reference guide»*; Reference guide for Free Pascal, version 2.0.0; Document version 2.0; May 2005.

[2] *Michaël Van Canneyt, «Free Pascal : Programmers' manual»*; Programmers' manual for Free Pascal, version 2.0.0; Document version 2.0; May 2005.

[3] *Michaël Van Canneyt, Florian Klämpfl, «Free Pascal : Users' manual»*; Users' manual for Free Pascal, version 2.0.0; Document version 2.0; May 2005.

[4] *Michaël Van Canneyt, «Run-Time Library (RTL) : Reference guide»*; Reference guide for RTL units; Document version 2.0; May 2005.

[II] **Литература**

[III] **Сайты**