# ТУБ №18:
# Работа с вещественными типами данных в Delphi

## *Операции над вещественными типами данных*

Что можно делать с вещественными числами? Ну очевидно же: много чего! Если для их представления в памяти потребовалось такую сложную конструкцию придумывать, то уж и операций-то, определённых над вещественными типами данных, должно быть как гуталина у дяди кота Матроскина — «ну просто завались!»

Так-то оно так, да не так. На деле нашлось всего две группы операций, которые можно выполнять над вещественными числами.

Первая группа операций — арифметические:

| Операция | Вид | Описание | Тип результата |
|---|---|---|---|
| + | Одноместная | Сохранение знака | Вещественный |
| – | Одноместная | Изменение знака | Вещественный |
| + | Двухместная | Сложение | Вещественный |
| – | Двухместная | Вычитание | Вещественный |
| * | Двухместная | Умножение | Вещественный |
| / | Двухместная | Деление | Вещественный |

И этих операций, как видим, даже меньше, чем у целочисленных типов данных. И если с операцией div (целочисленного деления) всё более-менее понятно, то куда делся mod? Это, конечно, взятие остатка от целочисленного (тоже целочисленного) деления, но ведь было бы неплохо иметь возможность взять 0.5 mod 0.2 и получить 0.1, правда?

Давайте рассуждать. Вещественные числа в памяти компьютера могут представляться с некоторой погрешностью. Скажем, число 0.1 в двоичной системе счисления является бесконечной периодической дробью, а при записи в переменную вещественного типа (например, Single) количество значащих цифр оказывается ограниченным. В результате в уже упомянутом Single вместо 0.1 на самом деле будет записано слегка другое число: 0.100000001490116119384765625.

Возьмём теперь 0.5 и 0.2. Число 0.5 в двоичной системе счисления — это просто $0.1_{(2)}$, т.к. 0.5 — это $2^{-1}$. А вот 0.2 — снова бесконечная дробь, поэтому на самом деле в переменной будет храниться не 0.2, а 0.200000002980232238769531 25. Попробуем теперь найти остаток от деления одного на другое, вычитая делитесь из делимого:

```
  0.5
- 0.200000002980232238769531 25
------------------------------
  0.299999997019767761230468 75
- 0.200000002980232238769531 25
------------------------------
  0.099999994039535522460937 5
```

И вроде близко к искомому 0.1, но не равно. И уж совсем не равно тому, во что превратится 0.1 при попытке его записи в переменную вещественного типа. Ну и скажите, кому нужна была бы операция, у которой 0.5 mod 0.2 = 0.09999999? Вот её и нет.

Остальные операции по сравнению с такими же для целочисленных типов просто поменяли тип результата. Оно и понятно: если два нецелых числа сложить или перемножить, может получиться (и скорее всего так и будет!) нецелое число. А если даже и целое получится, всё равно: тип результата должен быть таким, чтобы на все случаи жизни. Ведь в памяти-то что Single, что какой-нибудь Integer — 32 бита. И вот видим мы, например, такие 32 бита в памяти:

```
01000000 01001001 00001111 11011011
```

И что это? Кто-то скажет, что целое число 1’078’530’011. А кто-то присмотрится повнимательнее, мантиссу там выделит, порядок учтёт — и скажет: «Да это ж число π, записанное в тип Single!» А как программа должна разбираться? Она ведь и такой, и такой результат может получить. Поэтому договорились, что все арифметические операции над вещественными типами данных дают результат именно вещественного типа.

Вторая группа операций — операции сравнения:

| Операция | Вид | Описание | Тип результата |
|---|---|---|---|
| = | Двухместная | Равно | Логический |
| <> | Двухместная | Не равно | Логический |
| < | Двухместная | Меньше | Логический |
| > | Двухместная | Больше | Логический |
| <= | Двухместная | Меньше или равно | Логический |
| >= | Двухместная | Больше или равно | Логический |

Здесь вообще всё, как у целочисленных типов. Единственное отличие, о котором нужно помнить:

**Сравнения на точное равенство (= и <>) работать будут, но из-за точности представления результаты могут не соответствовать ожиданиям.**

## *Встроенные процедуры и функции над вещественными типами данных*

У тех, кто в школе не прогуливал математику, может возникнуть вопрос: «Подождите-подождите, а как же мне теперь синус посчитать? И косинус?»

Для этих целей умные математики придумали раскладывать функции в бесконечные ряды:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^{n-1}}{(2n-1)!} \cdot x^{2n-1} + \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + \frac{(-1)^n}{(2n)!} \cdot x^{2n} + \ldots$$

Ну а как вычислять такие бесконечные суммы, Вы уже наверняка знаете: итерационные циклы, какими бы сложными они ни казались, никогда не подведут.

Критически настроенный читатель поморщится и скажет: «Стоп, это что же, мне теперь каждый раз, когда синус посчитать надо, придётся цикл строить? А если у меня сложная формула и там этих синусов штук пять да ещё и логарифмы?» И здесь-то на помощь приходят встроенные процедуры и функции.

| Имя | Вид | Описание | Тип результата |
|---|---|---|---|
| Round(x) | Функция | Возвращает результат округления числа x к ближайшему целому | Целочисленный |
| Trunc(x) | Функция | Возвращает целую часть числа x | Целочисленный |
| Int(x) | Функция | Возвращает целую часть числа x | Вещественный |
| Frac(x) | Функция | Возвращает дробную часть числа x | Вещественный |
| Abs(x) | Функция | Возвращает модуль числа x | Вещественный |
| ArcTan(x) | Функция | Возвращает арктангенс x | Вещественный |
| Cos(x) | Функция | Возвращает косинус x | Вещественный |
| Sin(x) | Функция | Возвращает синус x | Вещественный |
| Exp(x) | Функция | Возвращает $e^x$ | Вещественный |
| Ln(x) | Функция | Возвращает натуральный логарифм x | Вещественный |
| Sqr(x) | Функция | Возвращает квадрат x | Вещественный |
| Sqrt(x) | Функция | Возвращает корень квадратный числа x | Вещественный |
| Определённые для всех типов | | | |
| SizeOf(x) | Функция | Возвращает размер типа в байтах | Integer |

Начнём с функции Round(x). Как объясняют округление к целому числу в школе? Примерно так: берём первую цифру дробной части и пристально смотрим ей в глаза, если она от 0 до 4, то отбрасываем, если от 5 до 9 — добавляем единицу к целой части.

Реальная жизнь намного разнообразнее. Настолько, что Intel придумали целых 4 способа округления вещественных чисел к целым:

- к ближайшему целому (чётному);
- в сторону $-\infty$;
- в сторону $+\infty$;
- в сторону 0.

В последнем случае результат получается отбрасыванием дробной части, способы округления к бесконечностям — по большей части на любителя, а вот округление к ближайшему целому (чётному) используется по умолчанию.

Что означает слово «чётный» в скобках? Это касается ситуации, когда округляемое число находится в точности между двумя целыми: например, 0.5 находится ровно между 0 и 1, 27.5 — ровно между 27 и 28 и т.д. В школе обычно учат, что половина округляется в большую сторону, и это правильно: стакан наполовину полон. Но что будет, если всегда округлять половины в большую сторону?

По всей видимости, первыми возможные проблемы узрели хитрюги-банкиры: если всё время округлять половины в одну и ту же сторону, то в какой-то момент получится, что погрешность от округления накопилась и полученные числа превышают те величины, которые они описывают. Та же проблема и с округлением половин в меньшую сторону.

Подумали банкиры немного, да и придумали так называемое «банковское округление». По его правилам половины округляются всегда так, чтобы предыдущая (более старшая) цифра была чётной. Например, 0.5 округляется к 0, а 1.5 — уже к 2. В результате примерно половина таких чисел округляется в большую сторону и примерно половина — в меньшую. В итоге на большом количестве операций погрешности округления начинают компенсировать друг друга.

Но вернёмся к функции Round. Конечно же, разработчики Delphi не стали изобретать велосипедов: ведь сопроцессор уже умеет выполнять эту операцию, да ещё и четырьмя разными способами! Поэтому функция Round тоже по умолчанию использует банковское округление к ближайшему целому, но при желании с помощью процедуры SetRoundMode можно выбрать и другой способ.

Следующие две функции — Trunc и Int. Они делают одно и то же — выделяют целую часть числа, — отличаясь только типом результата. Зачем это нужно? Дело в том, что преобразование из вещественных типов в целочисленные имеет свою цену в плане производительности. Поэтому, если необходимо просто отбросить дробную часть, но продолжить вычисления с вещественными данными, предпочтительнее будет использовать Int и сэкономить на преобразовании. Если же дальше работа будет вестись уже в целых числах, то выгоднее (и точнее для результата) использовать функцию Trunc.

Назначение остальных функций достаточно очевидно из описания. Следует лишь отметить, что тригонометрические функции — ArcTan, Cos и Sin — работают с углами, выраженными в радианах, а не градусах.

## *Свойства значений вещественного типа*

Обратите внимание, что среди встроенных функций отсутствуют Ord, Pred и Succ. Как Вам уже должно быть известно, эти функции определены для всех перенумерованных типов. Следовательно, можно предположить, что вещественные типы не являются перенумерованными, т.е. их значения нельзя выстроить одно за другим по возрастанию. Но почему?

Дело в том, что форматы представления вещественных чисел в памяти, описанные в стандарте IEEE 754, позволяют записывать не только обычные вещественные числа, но и такие значения, как –0, $+\infty$, $-\infty$ и самая обширная группа значений — не-числа (Not a Numbers, NaN).

Многие операции и функции над вещественными типами данных не определены для некоторых значений: например, корень квадратный из отрицательного числа, перемножение нуля и бесконечности, деление бесконечности на бесконечность или нуля на ноль, синус и косинус бесконечности и т.п. Но что делать, если возникает подобная ситуация? Можно, конечно, аварийно завершать работу программы, но тогда программисту придётся очень тщательно анализировать используемые формулы и те исходные данные, которые в них подставляются. И это было бы очень непросто. Например, попробуйте определить, при каких значениях переменных нельзя вычислять такую формулу:

$$y = \frac{x + \dfrac{a}{\log_x a + \log_{a+1} x - 1}}{arctg\left(\sqrt{a} + \sqrt{\ln x + \lg a} - a^x\right)} \cdot b$$

А теперь вспомните, что вещественные числа плохо поддаются сравнению на точное равенство. Получается, возлагать на программиста задачу проверки всех возможных случаев было бы по меньшей мере жестоко.

Разработчики Intel поступили хитрее: если какое-либо действие над вещественными данными не может быть выполнено, его результатом становится одно из значений NaN. Кроме того, любая операция над NaN имеет своим результатом NaN. Таким образом, в большинстве случаев программист может просто смело запускать вычисление огромной формулы, не задумываясь об исходных данных, а затем просто проверить, не получился ли в результате NaN. Если получился NaN — где-то по пути что-то пошло не так, формула не может быть вычислена для этих исходных данных, если обычное число — так вот он результат!

Но как проверить вещественное число… э-э-э… на не-число? Дело в том, что NaN не равен ни одному числу, в том числе самому себе. Если Вам нужно проверить, не NaN ли случайно в вещественной переменной `x`, достаточно записать сравнение `x = x`. Результат такого сравнения будет ложным, если в переменной `x` записан NaN. Если же Вы предпочитаете более очевидный способ проверки, на помощь придёт функция IsNan.

Собственно, из-за этих самых NaN и получается, что вещественные типы не обладают свойством перенумерованности: куда поместить NaN? До бесконечности? После? Между? Перед нулями или после? Или между –0 и +0? А два разных NaN-значения в каком порядке разместить?

Но даже если бы NaN не существовало, перенумерованность вещественных чисел не имела бы смысла. Какая была бы практическая польза от знания о том, что порядковый номер 0 имеет число +0, порядковый номер 1 — число $1.4 \cdot 10^{-45}$, порядковый номер 2 — число $2.8 \cdot 10^{-45}$, а число –0 вообще имеет порядковый номер –2’147’483’648. И самое главное: такая нумерация была бы справедлива только для Single. Для Double, например, числом с порядковым номером 1 было бы число $4.94 \cdot 10^{-324}$. Нет смысла — нет и свойства перенумерованности.

Вещественные константы

Чтобы использовать все эти возможности работы с вещественными данными, необходимо для начала эти самые данные где-то добыть. Часто необходимо задать значение вещественного типа прямо в тексте программы. В Delphi для этого есть две формы записи:

- запись в форме с фиксированной точкой;
- запись в форме с плавающей точкой.

Запись в форме с фиксированной точкой выглядит точно так же, как и на бумаге:

```
0.25     -2.73     +73.8
```

Запись в форме с плавающей точкой содержит указание мантиссы и порядка, разделённых латинской буквой E (регистр не имеет значения, но, как правило, используют заглавную букву):

```
14.3E5     681e-2     -5.16E+3
```

Часть записи до буквы E — мантисса, после буквы E — порядок. Приведённым числам соответствуют следующие математические записи:

$14.3 \cdot 10^{5}$ $681 \cdot 10^{-2}$ $-5.16 \cdot 10^{3}$

Кроме того, в языке Delphi имеется стандартная константа вещественного типа `Pi`, равная, как нетрудно догадаться, значению числа π (разумеется, приближённому).

## *Дополнительные вопросы*

1. Сформулируйте правила банковского округления к целому числу для двоичной системы счисления.

2. Как возвести произвольное число в произвольную степень с использованием стандартных операций и функций?

3. Какие проблемы могут возникнуть с применением способа из предыдущего вопроса? Почему следует использовать функцию Power из модуля Math и аналогичные ей с особой осторожностью?