ТУБ №29:
Массивы в Delphi

Массивы

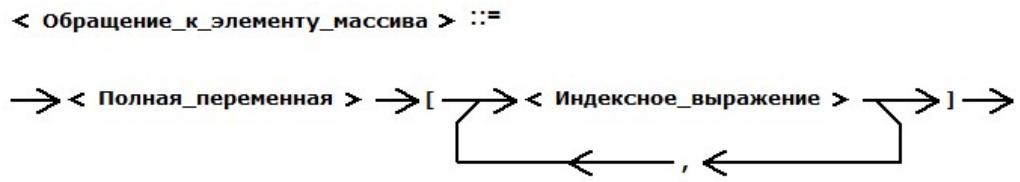
В любой хоть сколько-нибудь сложной программе массивы обязательно встречаются, причём не единожды. Каждый раз, когда Вы видите в задаче несколько чего-нибудь — сотрудников, товаров, игроков, — где-то там Вас поджидают массивы.

> *Массив* — упорядоченная совокупность однотипных элементов, имеющих общее имя.

Массив является примером типов данных, значения которых представляют собой совокупность нескольких значений. Такие типы данных собирательно называют *структурными типами данных*. В противовес им существуют *скалярные типы данных*, значениями которых являются отдельные величины — числа, символы и т.п.

Список студентов группы, список покупок, список товаров на складе — всё, что хоть как-то поддаётся нумерации, может стать в Вашей программе массивом. Необязательно это будет самым эффективным способом, но уж точно одним из возможных.

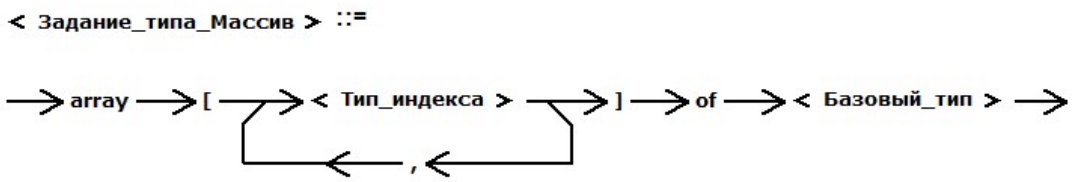
Переменная, которая используется для обозначения всего массива целиком (это её идентификатор будет тем самым общим именем из определения), называется *полной переменной*. При этом обращение к отдельным элементам будет иметь такой синтаксис:



Выбор одного из элементов массива осуществляется с использованием *индексов* — величин, задающих правило вычисления номера нужного элемента массива. Чаще всего индексы — целочисленные значения, однако в общем случае индексное выражение может быть любого скалярного перенумерованного типа. Сами элементы массива могут быть любого типа, причём этот тип ещё называется *базовым типом массива*.

Следует отличать индекс элемента массива от его значения. Рассмотрим эти отличия на примере камер хранения в магазине. Номера, написанные на ячейках, — это и есть индексы, и они действительно обычно представляют собой целые числа. Значением же «элемента массива» будет содержимое ячейки — все те вещи, которые туда сложены. Нетрудно заметить, что тип данных для элементов массива может быть практически произвольным: в камере хранения можно попытаться оставить хоть слона — лишь бы размер ячейки позволял. В то же время «индексы» должны обладать свойством перенумерованности, иначе возникнут серьёзные затруднения: в какую сторону нужно двигаться от ячейки, на которой нарисован домовёнок Кузя, чтобы найти ячейку с котом Леопольдом?

Для того, чтобы задать тип-массив, используется следующий синтаксис:



Количество элементов в массиве определяется типами индексов. Например, такое объявление

```
type
  TExample = array [Byte] of Boolean;
```

Задаёт массив из 256 элементов типа Boolean, причём индексы элементов будут принимать значения от 0 до 255 включительно. А, например, такое объявление

```
type
  TAnotherExample = array [Boolean] of Byte;
```

соответствует массиву с двумя элементами типа Integer, которые будут индексироваться либо значением False, либо значением True.

Намного чаще встречаются в литературе и на практике массивы, у которых типы индексов — типы-диапазоны. Например:

```
type
  TStudentMarks = array [1..30] of Integer;
```

Такое объявление соответствует массиву из 30 элементов типа Integer с индексами от 1 до 30 включительно.

Элементы массивов также называют *индексированными переменными*.

Следует заметить, что возможность нумерации элементов массива произвольным диапазоном — почти уникальная возможность Pascal/Delphi. В других языках программирования, как правило, либо нумерация элементов начинается строго с нуля (потому что так проще организовать работу с ними на уровне машинного кода), либо сами массивы не являются в действительности массивами (а представляют собой разновидность хеш-таблицы, что бы это ни означало). Здесь же язык позволяет выбрать удобные для конкретной решаемой задачи способы индексации массивов и берёт на себя задачу правильного обращения к соответствующим элементам.

Предположим, что нам нужно, например, сохранить сведения о количестве граждан призывного возраста. На момент написания этого текста это возраст от 18 до 26 лет включительно. В Pascal/Delphi это можно сделать, например, так:

```
var
  AgeStats: array [18..26] of Integer;
```

Если предположить, что массив уже заполнен правильными значениями, то для вывода количества призывников в возрасте 21 года можно будет написать что-то наподобие

```
WriteLn(AgeState[21]);
```

При этом в массиве будет 9 элементов — ровно столько, сколько минимально необходимо для хранения требуемой статистики.

В языках с принудительной нумерацией с нуля обычно используется один из двух подходов. Первый заключается в том, чтобы создавать массив с индексами от 0 до 26 и не использовать элементы до 18-го. Это, разумеется, ужасно неэффективно по использованию памяти: 27 элементов вместо 9, т.е. в три раза больше необходимого. Второй подход заключается в том, чтобы прямо в тексте программы прописывать преобразование интересующего нас возраста в индексы массива:

```
AgeStats[Age - 18]
```

Этот подход решает проблему расхода памяти, но захламляет исходный код однотипными вычислениями, которые, к тому же, могут иногда и изменяться — каждый раз, когда будут изменяться границы призывного возраста. Это не добавляет программе ни читаемости, ни сопровождаемости, ни надёжности. На уровне машинного кода компиляторы Pascal/Delphi используют аналогичный подход, но при этом исходный код не захламляется и адаптируется к изменениям в индексации массива автоматически. Это соответствует популярному инженерному принципу

Тривиальные вычисления, которые не оказывают значимого влияния на быстродействие программы и могут быть автоматизированы, должны быть автоматизированы

вытекающему из более общего принципа DRY — Don’t Repeat Yourself («Не повторяйся»):

Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы

В нашем примере такой частью знания является информация о том, как нумеруются элементы массива и как найти элемент, соответствующий определённому возрасту. Распространение выражения `Age - 18` по всей программе как раз и было бы тем самым повторением.

Многомерные массивы

Из синтаксических диаграмм задания массивов и обращения к их элементам видно, что индексов у массива может быть несколько. В зависимости от количества индексов выделяют *одномерные массивы* — те, у которых один индекс, — и *многомерные* — те, у которых два и более индекса. Многомерные массивы также ещё называют *матрицами*.

Попробуем объявить типы данных для представления в программе состояния игры «Крестики-нолики»:

```
type
  TCellState = (csEmpty, csX, csO);
  TGameDesk = array [1..3, 1..3] of TCellState;
```

Тип `TGameDesk` в примере представляет собой двухмерный массив, у которого тип обоих индексов — тип-диапазон от 1 до 3 включительно, а базовый тип — перечислимый тип `TCellState`.

Приведённая форма объявления двухмерного массива является *сокращённой*. Для того, чтобы получить полную форму объявления, необходимо обратить внимание на то, что двухмерный массив можно рассматривать как одномерный массив, элементы которого — строки — также являются массивами:

```
type
  TGameDesk = array [1..3] of array [1..3] of TCellState;
```

Аналогично при обращении к элементам двухмерного массива можно использовать полную и сокращённую формы:

```
WriteLn(Desk[1, 3]);
WriteLn(Desk[1][3]);
```

Полная и сокращённая формы абсолютно взаимозаменяемы, причём как при объявлении, так и при обращении к элементам матрицы. Тем не менее, хорошим стилем считается использование для работы с массивом той же формы записи, которая была использована при его объявлении. Рассмотрим это на примере трёхмерных массивов.

В компьютерной графике (особенно в 3D-графике) широко применяются квадратные матрицы размера 4×4, причём таких матриц обычно используется несколько: проекционная, видовая, мировая, причём мировая матрица обычно формируется путём перемножения матриц поворота, масштабирования и сдвига. Таким образом, в каждый момент времени в программе может потребоваться использовать целый набор двухмерных матриц:

```
type
  TMatrixData = array [1..3] of array [1..4, 1..4] of Real;

var
  M: TMatrixData;

M[2][3, 4] := 42;
```

Обратите внимание на то, как в приведённом примере сочетаются полная и сокращённая формы объявления и последующего использования матриц (условно можно назвать такую форму объявления *смешанной*): объявление подчёркивает, что перед нами массив двухмерных массивов (array of 2D arrays). Использование любых других способов объявления и обращения к элементам будет абсолютно эквивалентным

```
type
  TExample1 = array [1..3] of array [1..4] of array [1..4] of Real;
  TExample2 = array [1..3, 1..4, 1..4] of Real;
  TExample3 = array [1..3, 1..4] of array [1..4] of Real;

...

M[2][3][4] := 42;
M[2, 3, 4] := 42;
M[2, 3][4] := 42;
```

однако для человека, читающего текст программы, они будут содержать намёк на несколько иную интерпретацию (группировку) хранящихся в таком массиве данных.

Возможно указание меньшего количества индексов, чем использовано при объявлении. В этом случае подразумевается обращение к *подмассиву* — массиву меньшей размерности,

являющемуся частью многомерного массива. Например, в приведённом выше примере выражение

```
M[3]
```

будет иметь тип «двухмерный массив размера 4×4» и задавать третью по счёту матрицу, а выражение

```
M[3][2]
```

будет иметь тип «одномерный массив из 4 элементов» и задавать вторую строку третьей по счёту матрицы.

Типы индексов многомерного массива не обязаны быть одинаковыми. Например, возможно и такое объявление:

```
type
  TState = (stStart, stPlay, stPause, stAbout);
  TMatrix = array [TState, 1..10, Char] of Integer;
```

Тогда обращение к элементам такой матрицы будет выглядеть, например, так:

```
M[stPlay, 7, '!'] := 293;
```

Представление массивов в памяти

В памяти компьютера массив представляет собой последовательно расположенные значения каждого из его элементов. Например, при объявлениях

```
type
  TArray = array [1..20] of Word;

var
  MyArr: TArray;
```

Переменная `MyArr` займёт в памяти 40 байт, причём первые 2 байта будут отведены под хранения значения элемента `MyArr[1]`, следующие 2 байта — под `MyArr[2]` и т.д.

Для того, чтобы найти в памяти элемент с заданным индексом, необходимо определить, каким по счёту этот элемент является в массиве, и отступить от начала массива на соответствующее количество элементов. Так, например, для массива

```
type
  TAgeStats = array [18..26] of Integer;

var
  AgeStats: TAgeStats;
```

обращение к элементу `AgeStats[21]` потребует некоторых вычислений. Для начала, элементу с индексом `21` предшествует `21 - 18 = 3` других элемента. Размер каждого элемента `SizeOf(Integer) = 4` байта. Таким образом, элемент `AgeStats[21]` находится через `3 × 4 = 12` байт от начала массива. Машинный код для выполнения приведённых вычислений компилятор генерирует автоматически.

Аналогично устроены и многомерные массивы. Как мы уже обсудили ранее, многомерный массив представляет собой массив из массивов меньшей размерности, поэтому, например, элементы двухмерной матрицы

```
type
  T2DMatrix = array [1..3, 1..4] of Char;

var
  M: T2DMatrix;
```

будут размещаться в памяти в следующем порядке:

M [1, 1]	M [1, 2]	M [1, 3]	M [1, 4]	M [2, 1]	M [2, 2]	M [2, 3]	M [2, 4]	M [3, 1]	M [3, 2]	M [3, 3]	M [3, 4]

Действия над массивами

Над полной переменной массива не определено никаких операций, однако полные переменные могут появляться в операторе присваивания. Например, следующее присваивание

```
type
  TArray = array [1..15] of Boolean;

var
  A, B: TArray;

...
A := B;
...
```

скопирует значения элементов массива `B` в массив `A`. Аналогичного результата можно было бы добиться следующим фрагментом кода:

```
for I := 1 to 15 do
  A[I] := B[I];
```

однако в ситуации, когда требуется именно копирование массива, такой подход будет хуже, чем присваивание полных переменных: во-первых, требуется объявление большего числа переменных, во-вторых, он сопряжён с написанием большего количества кода (с замечательной возможностью ошибиться в организации цикла, например, в начальном и конечном значениях параметра цикла). Наконец главный аргумент в пользу оператора присваивания заключается в том, что в современных процессорах зачастую поддерживаются способы быстрого копирования больших блоков данных. При копировании с помощью оператора цикла компилятору нужно будет постараться, чтобы распознать именно поэлементное копирование и доказать, что его можно заменить на быстрое копирование блока памяти. При использовании оператора присваивания программист явно указывает компилятору на это, а значит, даже более простой компилятор сможет сгенерировать более эффективный машинный код.

Особую ценность при работе с массивами имеют следующие встроенные функции:

Имя	Вид	Описание	Тип результата
Low(x)	Функция	Возвращает значение индекса для первого элемента массива (минимальное значение индекса)	Тип индекса
High(x)	Функция	Возвращает значение индекса для последнего элемента массива (максимальное значение индекса)	Тип индекса
Length(x)	Функция	Возвращает количество элементов в массиве	Integer
SizeOf(x)	Функция	Возвращает размер массива в байтах	Integer

Первые две функции позволяют писать максимально гибкий и сопровождаемый код. Начнём с традиционного примера из учебников, выводящего значения элементов одномерного массива:

```
var
  A: array [1..20] of Integer;
  I: Integer;

...

for I := 1 to 20 do
  WriteLn(A[I]);
```

Обратите внимание на то, что код, осуществляющий проход по массиву, жёстко привязан к индексации элементов от 1 до 20. Как правило, учебники ограничиваются замечанием о том, что количество элементов может измениться в будущем, и предлагают задавать размер массива константой:

```
const
  N = 20;

var
  A: array [1..N] of Integer;
  I: Integer;

...

for I := 1 to N do
  WriteLn(A[I]);
```

Такая реализация, безусловно, несколько гибче, однако по-прежнему создаст проблемы, если понадобится изменить минимальное значение индекса (например, начать нумерацию с 0). Кроме того, если имеется много различных типов-массивов, то окажется, что для каждого из них требуется своя константа, а значит, придётся придумывать идентификаторы для каждой из них и они явно будут не самыми короткими.

А вот как можно обойти массив с использованием встроенных функций Low и High:

```
var
  A: array [1..20] of Integer;
  I: Integer;

...

for I := Low(A) to High(A) do
  WriteLn(A[I]);
```

Над индексированными переменными (элементами массива) определены те же самые операции, процедуры и функции, что и для базового типа массива. Другими словами, элемент массива ведёт себя так же, как и обычная переменная того же самого типа.

Выход за границу массива

При работе с массивами важно помнить:

Выход за границу массива — грубая ошибка

Выходом за границу массива называют ситуацию, когда программа обращается к элементу с индексом, которого в массиве нет. К сожалению, отловить подобные ошибки во время компиляции программы технически невозможно, т.к. индекс может быть указан как сложное выражение, значение которого зависит от величин, получаемых уже во время работы программы.

Существует настройка компилятора, при включении которой каждое обращение программы к массиву дополняется проверкой вычисленного значения индекса и, если индекс выходит за границы допустимого диапазона, генерируется ошибка времени выполнения. Тем не менее, следует понимать, что это дополнительные действия, выполняемые программой, и они негативно сказываются на её быстродействии. К тому же, если подобная ситуация возникла во время работы программы, исправить её никак не удастся, т.к. это проблема в реализованном программистом алгоритме. Поэтому на практике эту настройку компилятора включают на время разработки программы, однако отключают перед передачей программы пользователям.

Грубой подобная ошибка считается потому, что массив — это не единственные данные в оперативной памяти: остальные переменные, как правило, размещаются по соседству. Если проверки на выход за границы массива отключены, программа вычисляет место в памяти, занимаемое элементом, по уже рассмотренной нами схеме. В результате, например, в такой ситуации:

```
var
  A: array [1..10] of Integer;

...

X := A[11];
```

обращение произойдёт к первым 4 байтам сразу за последним элементом массива. Этот участок памяти может быть отведён для хранения значения любой другой переменной в той же программе, причём необязательно целочисленного типа. В результате вместо значения одного из элементов массива в переменную `X` будут скопированы совершенно не связанные с ним данные, просто по стечению обстоятельств оказавшиеся поблизости.

При попытке записи в несуществующий элемент массива

```
A[11] := X;
```

ситуация окажется ещё печальнее, т.к. теперь величина, предназначавшаяся для записи в массив, может оказаться записанной в ничего не подозревающую переменную (или даже несколько!), отвечающую за хранение совсем других данных. Если дальнейшие вычисления используют значение такой переменной, то результаты явно будут отличаться от ожидаемых — и ракета вместо околоземной орбиты отправляется прямиком на Пентагон, после чего Третья мировая война неизбежна. Поначалу, конечно, всем будет не до разработчика чудо-программы, но потом, на пепелище цивилизации, разбор полётов и пожизненное заключение покажутся ему куда лучшим наказанием, чем голодная смерть.

Вышесказанное справедливо для Delphi и некоторых других языков. Следует, однако, иметь в виду, что в языках наподобие C и C++ выход за границу массива будет иметь намного более серьёзные последствия, причём куда менее предсказуемые.

Дополнительные вопросы

1. Что делает следующий фрагмент программы?

```
for I := Low(A) to High(A) do
begin
  Temp := A[I];
  A[I] := A[High(A) - I + Low(A)];
  A[High(A) - I + Low(A)] := Temp;
end;
```

2. Какую задачу пытался решить разработчик, написавший фрагмент программы из предыдущего вопроса? В чём заключается его ошибка?

3. Как нужно исправить приведённый фрагмент программы, чтобы задача решалась правильно?