ТУБ №34:
Динамические массивы

Динамические массивы

Обычные массивы, доставшиеся Delphi по наследству от языка Pascal, хороши для многих задач, но имеют свои недостатки. Главный из них заключается в следующем…

Предположим, Вы открыли свою компанию и хотите вести учёт клиентов, чтобы своевременно предугадывать их потребности, предлагая свои услуги, и наконец просто для того, чтобы отвечать на их входящие звонки, обращаясь не иначе как «Милостивый сударь Иван Иваныч». Ваша компания инновационна от вывески на входе до жидкого мыла в туалете, поэтому существующие CRM — программы для управления данными о клиентах, если по-простому, — для решения Ваших задач не подходят.

Вы садитесь писать свою программу и, конечно же, для хранения данных о клиентах у Вас будут использоваться массивы. Есть только одна проблемка:

```
type
  TClient = ...   // Что будет написано здесь, зависит от Ваших задач
  TClientArray = array [1..1000] of TClient;
```

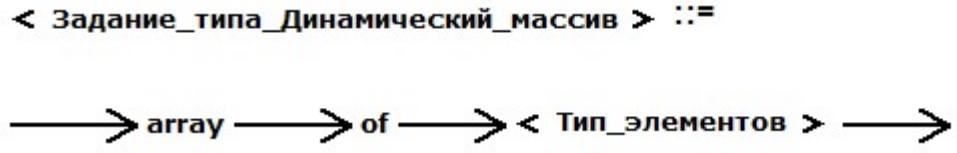
Сколько элементов на самом деле нужно отвести для такого массива? Достаточно ли будет тысячи? Что делать, если бизнес пойдёт в гору и клиентов станет больше? Отвести сразу миллион записей? Но что, если Ваш бизнес — это обширная сеть продуктовых магазинов? Тогда миллиона тоже может оказаться маловато.

Или наоборот: что, если предлагаемая Вами услуга уникальна и интересна только 3 потенциальным клиентам во всём мире (но они, разумеется, готовы платить за неё любые деньги)? Не будет ли в этом случае глупо и расточительно отводить сразу 1000 или 1’000’000 элементов?

Обычно эта проблема неопределённости решается путём использования динамической памяти. Но для этого хорошо бы знать и понимать указатели, а этим даже среди опытных программистов, к сожалению, могут смело похвастаться немногие. К счастью, в Delphi для этих целей есть альтернативное решение — динамические массивы.

> *Динамический массив* — это массив, количество элементов в котором может быть изменено во время работы программы.

Поскольку заранее выбранного фиксированного количества элементов у таких массивов нет, соответствующая часть объявления — перечисление типов индексов в квадратных скобках — просто исключается:



В тексте программы это выглядит примерно так:

```
type
  TIntArray = array of Integer;

var
  Numbers: TIntArray;
```

Изначально такой массив содержит 0 элементов. Разумеется, пользы от такого массива немного, поэтому, когда становится понятно, сколько элементов нужно для работы, следует воспользоваться процедурой `SetLength`, например, так:

```
SetLength(Numbers, 10);
```

После такого вызова массив `Numbers` будет содержать 10 элементов. Разумеется, вместо литерала здесь можно указывать произвольное выражение, значение которого будет определять новую длину массива.

Процедура SetLength может как увеличивать, так и уменьшать количество элементов в массиве. Вновь добавленные элементы считаются неинициализированными, т.е. их значения могут оказаться совершенно произвольными. На практике чаще всего в этих элементах оказываются нулевые значения, но это гарантируется только для некоторых типов элементов. Если же количество элементов уменьшается, пропадают (т.е. становятся недоступными) элементы в конце массива.

Теперь можно работать с элементами. Но при объявлении обычных массивов (их иногда ещё называют *статическими*) мы сами определяли, какими будут индексы. Например:

```
type
  TSomeArray = array [5..20] of Word;
  TOtherArray = array [Char] of Real;
```

В первом случае индексы были бы целыми числами от 5 до 20 включительно, во втором — символами, которые входят во множество значений типа Char. Но как быть в случае динамического массива? Ведь при его объявлении информация об индексах вообще не указывается!

Решение очень простое: договориться о каком-либо общем правиле. В случае динамических массивов в Delphi оно звучит так:

Элементы динамического массива имеют целочисленные индексы, начинающиеся с 0.

Это означает, что для динамического массива с 10 элементами допустимы значения индексов от 0 до 9 включительно. Все прочие значения индекса будут означать выход за границу массива, а, как Вы должны помнить…

Выход за границу массива — грубая ошибка

Вооружившись этими знаниями, попробуем написать фрагмент кода, который позволяет пользователю создать и заполнить целочисленный массив произвольного размера:

```
var
  MyArray: array of Integer;
  N, I: Integer;

...

Write('Введите количество элементов: ');
ReadLn(N);

SetLength(MyArray, N);
for I := 0 to N - 1 do
begin
  Write('Введите элемент MyArray[', I, ']: ');
  ReadLn(MyArray[I]);
end;
```

Приведённый код решает поставленную нами задачу, однако может быть улучшен. Дело в том, что не всегда бывает удобно иметь в программе отдельную переменную для хранения текущей длины массива: всегда есть риск ошибиться и не обновить её значение при вызове `SetLength`.

Определить текущий размер динамического массива можно с помощью *функции Length*:

```
for I := 0 to Length(MyArray) - 1 do
begin
  Write('Введите элемент MyArray[', I, ']: ');
  ReadLn(MyArray[I]);
end;
```

Обратите внимание на то, что самое большое значение индекса всегда на единицу меньше, чем длина динамического массива. Часто встречающаяся ошибка — пропуск этой поправки и, как результат, обращение к элементу за пределами массива.

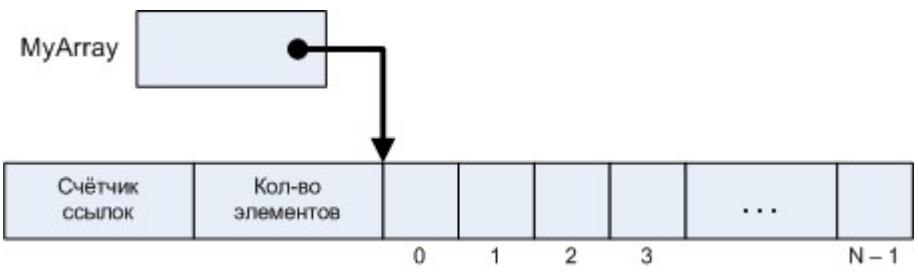
Удачное совпадение заключается в том, что функции `Low` и `High`, которые были доступны для использования с обычными статическими массивами, можно применять и к динамическим:

```
for I := Low(MyArray) to High(MyArray) do
begin
  Write('Введите элемент MyArray[', I, ']: ');
  ReadLn(MyArray[I]);
end;
```

Преимущество такой реализации заключается в том, что она в равной мере подходит как для статических, так и для динамических массивов.

Представление в памяти динамических массивов

Для того, чтобы грамотно использовать динамические массивы, необходимо понимать, как они устроены. А устроены они достаточно просто:



Как можно видеть на рисунке, по внутреннему устройству динамические массивы похожи на динамические строки. Как и в случае с динамическими строками, переменная типа динамический массив содержит в себе не сам массив, а лишь информацию о том, где в памяти он находится. Это позволяет нескольким переменным совместно использовать один и тот же экземпляр массива. При этом внутри этого экземпляра отводится место для хранения количества элементов и счётчика ссылок, который в каждый момент времени содержит ответ на вопрос о том, сколько переменных на него ссылается, и позволяет организовать автоматическое освобождение памяти в тот момент, когда массив становится ненужным.

Тем не менее, есть и несколько отличий. Одно из них заключается в том, что в динамическом массиве отсутствует последний элемент с нулевым значением. Как Вы помните, в динамических строках он был нужен только для совместимости с С-строками. У динамических массивов нет необходимости быть совместимыми с чем-либо, поэтому надобность в таком элементе отсутствует.

Ещё один нюанс связан с присваиванием полной переменной. Динамические строки создавались так, чтобы для работы с ними можно было без изменений использовать код, написанный для коротких Pascal-строк. По этой причине после выполнения следующих действий

```
var
  S1, S2: String;

...

// Пусть в S1 записано, например, 'ABCDEF'
S2 := S1;
S1[1] := 'B';
```

переменная `S1` будет содержать строку `'BBCDEF'`, а `S2` — `'ABCDEF'`. Правило можно сформулировать так:

При попытке изменения строки, счётчик ссылок которой не равен 1, происходит создание её отдельной копии и все изменения производятся над этой копией

Для динамических массивов, опять же, необходимости быть совместимыми со статическими массивами не было, т.к. синтаксис их объявлений отличается. При этом создание копии массива — операция, которая в общем случае требует копирования большого объёма данных, поэтому в этом случае ничто не мешало применить один из базовых принципов проектирования:

Сложные операции должны выглядеть сложно

Применительно к динамическим массивам это означает, что сложная операция копирования должна записываться сложнее, чем просто присваивание полных переменных. На самом деле в результате выполнения оператора присваивания

```
type
  TDynArray = array of Integer;

var
  A, B: TDynArray;

...

// Заполнение массива A
B := A;
```

обе переменные, `A` и `B`, будут ссылаться на один и тот же экземпляр динамического массива. В результате попытка изменения какого-либо элемента через одну из этих переменных будет сразу же «видна» через другую переменную.

Для того, чтобы создать копию динамического массива, можно использовать *функцию Copy*:

```
B := Copy(A);
```

В результате использования этой функции переменная `B` будет связана с отдельным экземпляром динамического массива — копией массива, на который ссылается переменная `A`.

Кроме того, функцию Copy можно применять к динамическим массивам не только с одним, но и с тремя параметрами, как к строкам:

```
B := Copy(A, 2, 3);
```

В результате будет создан новый динамический массив из 3 элементов — копий элементов `A[2]`, `A[3]` и `A[4]`. Все правила работы с функцией Copy для строк аналогичным образом применимы и к динамическим массивам.

Многомерные динамические массивы

Разумеется, многомерные динамические массивы тоже возможны. Единственное затруднение заключается в том, что запись будет чуть более многословной, чем для статических:

```
var
  A: array of array of Real;
```

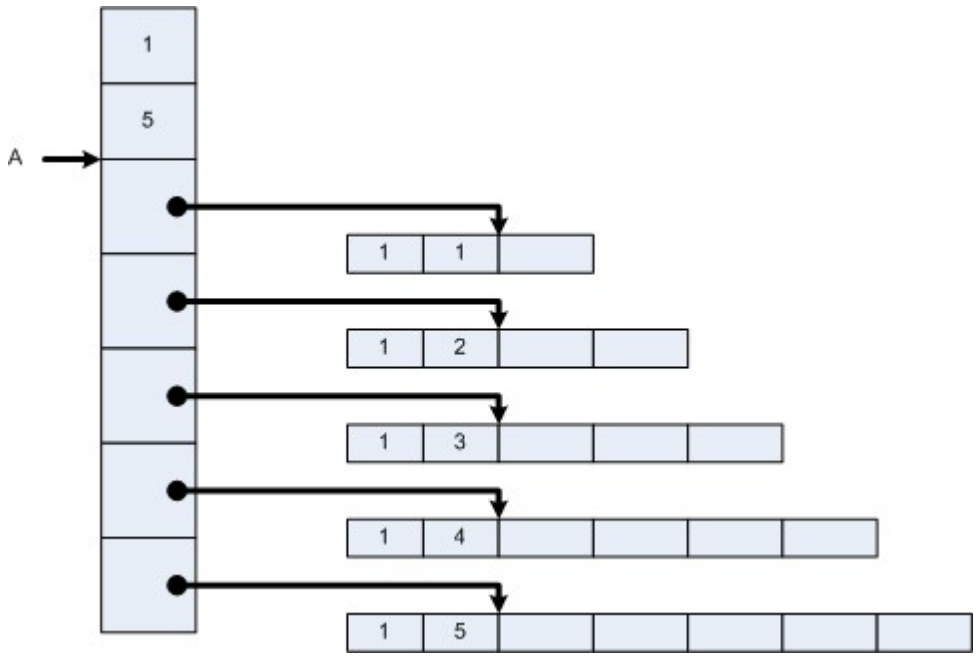
Приведённый пример соответствует объявлению двухмерного динамического массива. Изменить его размеры можно, например, так:

```
SetLength(A, 5, 10);
```

В результате `A` станет матрицей размером 5×10. Однако при необходимости можно построить и более сложную конструкцию:

```
SetLength(A, 5);
for I := Low(A) to High(A) do
  SetLength(A[I], I + 1);
```

Чтобы понять, как это работает, следует рассматривать `A` как динамический массив, каждый элемент которого также является динамическим массивом. Результат выполнения приведённого фрагмента кода будет следующим:



Элемент массива `A[0]` содержит ссылку на экземпляр динамического массива с одним элементом, `A[1]` — с двумя и т.д.

Следует иметь в виду, что массивы (особенно многомерные) могут занимать много памяти. Использование динамических массивов позволяет ограничиться использованием только необходимого её количества, однако иногда имеет смысл освободить занятую ненужным массивом память принудительно, не дожидаясь, пока это будет сделано автоматически.

Для того, чтобы освободить занятую экземпляром динамического массива память, необходимо удалить все ссылки на него. Удаление ссылки можно произвести присваиванием специального значения `nil` полной переменной. В этом случае переменная считается никуда не ссылающейся и, если она была единственной переменной, использовавшей экземпляр динамического массива, его счётчик ссылок станет равным 0 и в результате память будет освобождена.

Полная переменная типа «динамический массив», которой присвоено значение `nil`, считается эквивалентной динамическому массиву нулевой длины. Именно это значение имеют все такие переменные до начала работы с ними.

Дополнительные вопросы

1. Найдите ошибку в приведённом фрагменте программы:

```
var
  A: array of Integer;

...

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

2. Что выведет следующий фрагмент программы?

```
var
  A, B: array of Integer;

...

SetLength(A, 10);
for I := 0 to Length(A) - 1 do
  A[I] := I;

B := A;
B[3] := 42;
A[4] := 8;
WriteLn(A[3]:8, B[4]:8);
```

3. Почему можно не обнулять длину динамического массива (или не присваивать его полной переменной значение `nil`), если программа уже завершает работу? В каких случаях это можно не сработать? Почему даже в тех случаях, когда это не является ошибкой, делать так нежелательно?