

DOI: 10.32517/2221-1993-2022-21-1-67-80

В. О. Дженжер, Л. В. Денисова*Оренбургский государственный педагогический университет, Оренбург, Россия*

ДИНАМИЧЕСКИЕ МАССИВЫ И СПИСКИ В PASCALABC.NET

Аннотация

В статье описывается методика преподавания школьникам и студентам темы «Одномерные динамические массивы» на языке PascalABC.NET. Показывается, что переход от изучения статических массивов к динамическим хотя и потребует от преподавателя времени и некоторых усилий, но все затраты окупаются приобретением нового мощного инструмента. Помимо динамических массивов излагаются основы работы со списками `List<T>`, которые во многих случаях оказываются удобнее массивов. Списки являются одной из основных структур данных во многих современных языках программирования. В статье описываются область применимости списков и отличие их от массивов. Приводятся примеры написания функций, входными и/или выходными параметрами которых являются массивы или списки. Рассматривается техника использования срезов для работы с массивами и списками. Срезы предоставляют лаконичный синтаксис для обращения к части массива. Это позволяет существенно сократить код, который в традиционном стиле содержит много циклических действий. В заключение делается вывод о том, что изучение динамических массивов и списков позволяет легче перейти к программированию на других современных языках.

Ключевые слова: программирование, динамический массив, список, PascalABC.NET.

1. Введение

Изучение линии «Алгоритмизация и программирование» в средней школе предполагает освоение темы «Массивы». В разных языках программирования, изучаемых в школе, используются разные виды массивов. Так, в традиционном школьном Pascal и в алгоритмическом языке рассматриваются только статические массивы. В C++ (STL), C# все с самого начала знакомятся с динамическими массивами (см., например, [2]). В PascalABC.NET принят такой же подход, правда, статические массивы в языке остаются для желающих. В Python привычных всем массивов нет, но их можно с некоторой степенью удобства заменить списками, ближайшими «родственниками» динамических массивов [7].

В школьных учебниках, опирающихся при изучении программирования на Pascal (например, [1, 3, 6, 8]), для решения задач применяются статические массивы. По всей видимости, это связано с устоявшейся традицией, а также с тем, что в ЕГЭ по информатике и ИКТ используются примеры кода в таком же стиле. По нашему мнению, сегодня при изучении Pascal в школе следует переходить к использованию динамических массивов, и тому есть несколько причин.

Во-первых, школьники познакомятся с понятием динамической памяти и ее отличием от статической памяти, что позволит им впоследствии легко осваивать современные структуры данных, являющиеся по преимуществу динамическими. Кроме того, изучение динамических переменных подготовит ученика к изуче-

Контактная информация

Дженжер Вадим Олегович, канд. физ.-мат. наук, доцент, зав. кафедрой информатики, физики и методики преподавания информатики и физики, Оренбургский государственный педагогический университет, Оренбург, Россия; *адрес:* 460014, Россия, г. Оренбург, ул. Советская, д. 19; *e-mail:* vdjenjer@yandex.ru

Денисова Людмила Викторовна, канд. пед. наук, доцент кафедры информатики, физики и методики преподавания информатики и физики, Оренбургский государственный педагогический университет, Оренбург, Россия; *адрес:* 460014, Россия, г. Оренбург, ул. Советская, д. 19; *e-mail:* lv-denisova@yandex.ru

V. O. Dzhenzher, L. V. Denisova

Orenburg State Pedagogical University, Orenburg, Russia

DYNAMIC ARRAYS AND LISTS IN PASCALABC.NET

Abstract

The article describes the methodology for teaching to schoolchildren and students the topic "One-dimensional dynamic arrays" in the PascalABC.NET programming language. It is shown that the transition from studying static arrays to dynamic arrays, although it will require time and some effort from the teacher, but all costs are paid off by the acquisition of a new powerful tool. In addition to dynamic arrays, the basics of working with `List<T>` lists are outlined, which in many cases turn out to be more convenient than arrays. Lists are one of the basic data structures in many modern programming languages. The article describes the scope of lists and their difference from arrays. Examples of writing functions whose input and/or output parameters are arrays or lists are given. The technique of using slices for working with arrays and lists is considered. Slices provide a concise syntax for accessing a portion of an array. This allows you to significantly reduce the code, which in the traditional style contains a lot of cyclic actions. In closing, conclusions are drawn that the study of dynamic arrays and lists makes it easier to switch to programming in other modern languages.

Keywords: programming, dynamic array, list, PascalABC.NET.

нию объектно-ориентированного программирования, в котором экземпляры классов (объекты) также являются динамическими переменными.

Во-вторых, динамические массивы имеют огромное количество полезных методов, которые избавляют программиста от рутины и позволяют записывать краткий и понятный код. Заметим, что сами эти методы не могут и не должны быть предметом изучения на уроке. Но во внеурочной работе более продвинутые ученики, как показывает наша практика, с удовольствием их используют.

В-третьих, динамические массивы много лет назад введены в язык PascalABC.NET [4, 5, 9], который является одним из самых распространенных в школах нашей страны. Для того чтобы включить их изучение в школьную программу, не нужны сверхусилия, а требуется лишь добрая воля учителя.

Ну и, наконец, в-четвертых, переходя к изучению других языков, таких как C#, C++, Python и др., учащийся встречается именно с динамическими структурами данных. Поэтому знакомство с такими структурами сделает переход к изучению промышленных языков программирования более легким.

Далее мы попытаемся развернуть и обосновать некоторые из прозвучавших тезисов, а также изложить сведения, необходимые для получения первичных навыков работы с динамическими массивами. Поскольку «нельзя объять необъятное», в этой работе мы рассмотрим только одномерные динамические массивы и списки.

2. Статическая и динамическая память

Чтобы объяснить разницу между статическими и динамическими переменными, следует немного рассказать о логическом устройстве памяти компьютера. При этом не будем углубляться в детали организации памяти в Microsoft.NET, в тонкости управления памятью и сборки мусора. Приведем сведения, достаточные для понимания предмета нашего разговора.

С точки зрения программы (процесса), которую мы запускаем, доступная память делится на две части: на статическую и динамическую. Схематически это представлено на рисунке 1. Здесь **виртуальная память** — это оперативная память плюс часть дисковой памяти, которую операционная система использует в случае нехватки оперативной памяти. Для простоты будем считать, что **процесс** представляет собой выполняемую в данный момент программу, за которой закреплены некоторые ресурсы компьютерной системы.

Статическая память выделяется программе при ее запуске и служит для хранения кода самой программы, значений статических переменных и для оперативных целей. Все переменные, которые мы описываем в программе при помощи служебного слова *var*, являются статическими, следовательно, хранятся в статической памяти. Объем такой памяти ограничен, так что мы не можем разместить в ней больше определенного количества данных. Еще одна особенность статических переменных состоит в том, что их невозможно удалить из памяти, если они больше не нужны. Также нельзя изменить размер статической переменной (например, количество элементов статического массива), если она однажды была создана. Очень неудобно разрабатывать

Виртуальная память



Рис. 1. Три программы, размещенные в памяти компьютера

подпрограммы, использующие статические массивы, потому что размер такого массива придется жестко прописать в заголовке подпрограммы.

Динамическая память (ее еще называют *кучей*, англ. heap) — это часть виртуальной памяти, не занятая процессами. Она может быть использована для размещения переменных, принадлежащих процессам. Если процессу нужна дополнительная память, он просит операционную систему выделить ему некоторый участок кучи. Операционная система выделяет память и закрепляет ее за соответствующим процессом. Чтобы процесс мог обращаться к этой памяти, ему передается ссылка на начало выделенного участка (рис. 2).

Виртуальная память

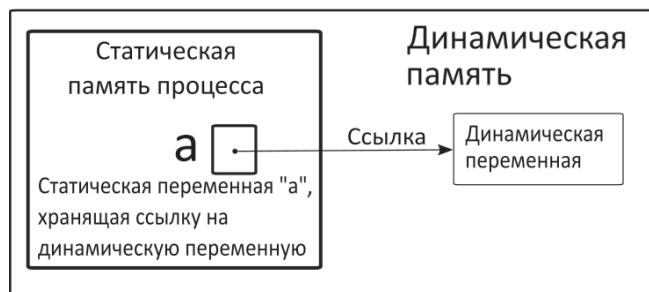


Рис. 2. Программа, в которой определена статическая переменная *a*, ссылающаяся на динамическую переменную. Динамическая переменная не имеет своего имени, доступ к ней может быть осуществлен только по ссылке

Ссылка похожа на указатель, т. е. переменную, хранящую адрес памяти. Но ссылка приспособлена к жизни в среде Microsoft.NET, где часто происходит перемещение динамических переменных с места на место при

очистке и уплотнении данных. При этом указатель не изменится, даже если переменная, на которую он указывал, будет перенесена на другое место. А вот ссылка будет обновлена, так что, где бы ни оказалась динамическая переменная, мы по-прежнему будем иметь доступ к ней.

Далее мы в основном будем говорить про динамические массивы, и может сложиться неверное впечатление, что только массивы могут быть размещены в динамической памяти. На самом деле это не так и существует множество других типов данных, которые размещаются в динамической памяти: списки, множества, стеки, очереди, деревья и др.

3. Статические и динамические массивы

Как мы уже выяснили, статические массивы размещаются в статической памяти, а динамические — в динамической. Рассмотрим код для создания обычного **статического массива**. Желательно описывать именованный тип массива, поскольку это позволяет использовать такой тип для создания подпрограмм с соответствующими параметрами (рис. 3).

Как мы видим, при работе со статическими массивами динамическая память не используется.

Динамические массивы, напротив, располагаются в динамической памяти. Рассмотрим программный код и фрагмент памяти для него (рис. 4).

Обратим внимание на несколько моментов.

1. *Описание динамического массива (в отличие от статического) не приводит к выделению памяти для него.* Статическая переменная, содержащая ссылку, создается, но содержит значение *nil*, т. е. ведет «в никуда»:

```
var a: array of integer;
```

Если проводить аналогию с приобретением земельного участка, то мы уже завели папку для документов (которую назвали *a*), но она пока пуста. Создание массива, т. е. выделение памяти для него и получение ссылки, производится при помощи команды **new**:

```
a := new integer[5];
```

Эта команда используется для создания любых динамических переменных и выполняет две функции, описанные выше: 1) выделяет в динамической памяти

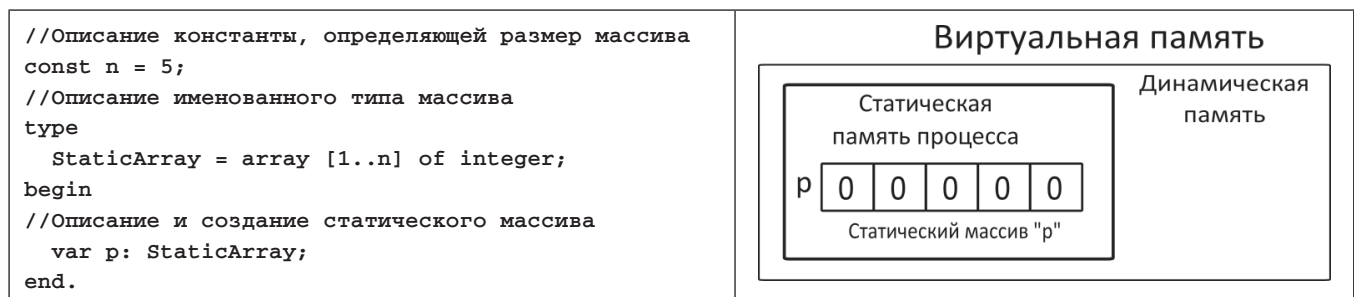


Рис. 3. Описание статического массива. Массив создается в памяти немедленно после описания переменной *p*. Каждая ячейка массива занимает 4 байта (размер переменной типа *integer*). Таким образом, полный размер массива = 4 байта × 5 = 20 байт

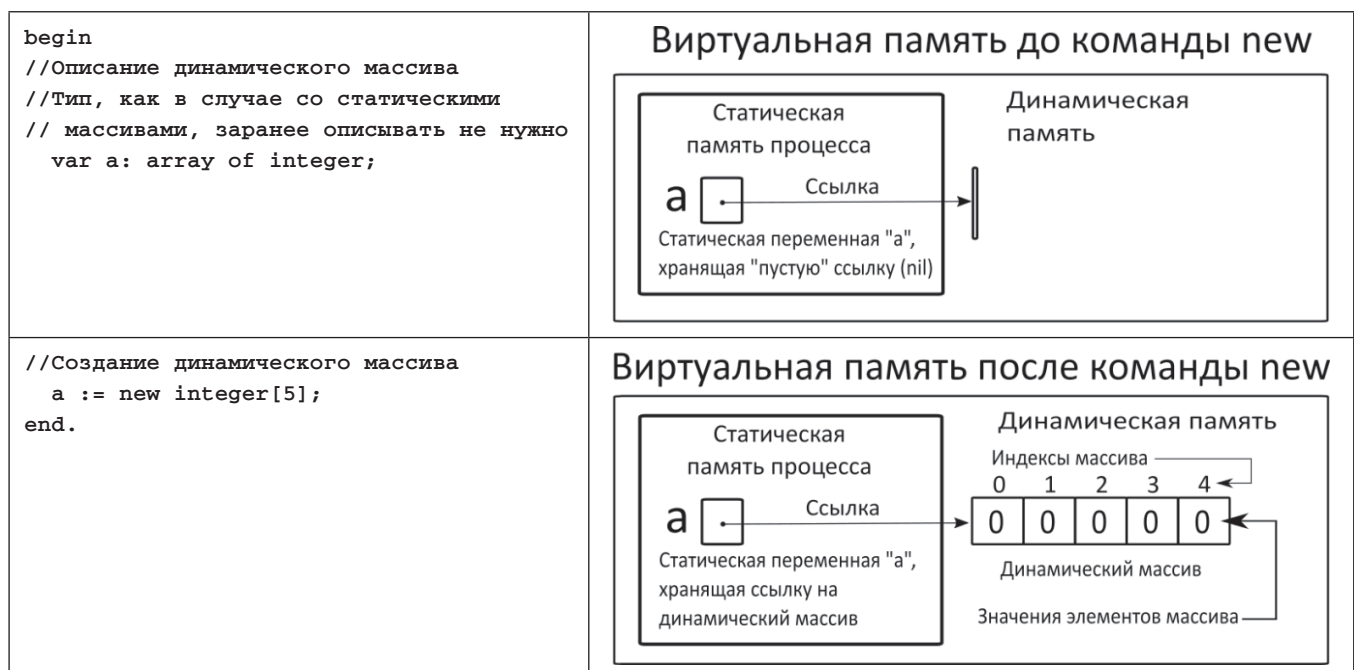


Рис. 4. Описание и создание динамического массива

участок необходимого размера и 2) возвращает ссылку на него. Ссылка посредством оператора присваивания записывается в статическую переменную *a*. Иногда в разговорной речи переменную *a* называют «динамической», что, в общем, неверно: динамическая переменная расположена в другом месте и не имеет имени. Но мы тоже будем говорить «динамический массив *a*», заменяя этой фразой более длинную: «статическая переменная *a*, ссылающаяся на динамический массив». Продолжая нашу аналогию, заметим, что после приведенной выше строки земельный участок нам выделен. Мы также получили географические координаты своего участка (ссылку на него), чтобы легко его отыскать, а сами координаты распечатали, и сложили листы с координатами в ранее заведенную папку *a*.

2. В команде **new** имеются указания на размер памяти, которую необходимо выделить. Действительно, запись «new integer[5]» означает, что нам нужен новый (new) динамический массив (на что указывают квадратные скобки), содержащий 5 (число в квадратных скобках) элементов целого (integer) типа. Выше мы уже отмечали, что на каждый такой элемент отводится по 4 байта, значит, общий размер массива составит 20 байт. Разумеется, вместо целого типа может быть указан любой другой тип. Где же именно расположен массив? А вот этого нам знать не нужно. Важно, что мы легко можем добраться до него при помощи ссылки. Более того, в процессе выполнения программы массив может быть перемещен в другую область памяти, о чем мы даже не узнаем. Но при этом ссылка в переменной *a* будет автоматически обновлена, и все это произойдет незаметно для нас.

3. При создании динамический массив инициализируется значениями по умолчанию. Статические массивы тоже как-то инициализируются, но как именно — может зависеть от языка и компилятора. Динамические массивы в PascalABC.NET обязательно будут проинициализированы значениями по умолчанию для

данного типа: числовые — нулями, строковые — пустыми строками и т. д. Поэтому после создания не обязательно производить инициализацию динамического массива, если нас устраивают значения по умолчанию.

4. Индексация элементов динамического массива всегда ведется целыми числами от нуля. И это невозможно изменить. Статические массивы могут иметь и символьные индексы, а нумерация элементов может идти, например, с -10 до $+10$. В некоторых задачах это удобно, но при работе с динамическими массивами такая возможность отсутствует.

5. Доступ к элементам динамического массива осуществляется точно так же, как доступ к элементам статического массива. Мы привыкли, что обращаться к элементу массива можно, указав имя массива и номер элемента в квадратных скобках. Так же мы обращаемся и к элементам динамического массива: *a*[3]. Надо только помнить, что если массив имеет пять элементов, то элемента с номером 5 у него нет (см. рис. 4).

6. Не требуется предварительное описание типа динамического массива. Мы можем описывать параметры-массивы процедур и функций без этого (см. ниже).

7. Все, что сказано о принципах работы с динамическими массивами, справедливо не только для PascalABC.NET, но и для других современных языков. Например, в C#, Python и многих других языках все происходит практически так же, с точностью до синтаксиса.

4. Инициализация, ввод и вывод динамического массива

Выше был приведен код, создающий динамический массив из пяти элементов и инициализирующий его нулями. Есть более удобные и короткие способы выполнить такое действие. Рассмотрим ряд примеров (табл. 1).

Ввод массива с клавиатуры выполняется при помощи функций (табл. 2).

Таблица 1

Код	Пояснение
<code>var a := new integer[5];</code>	// Описание динамического массива можно совмещать с его созданием
<code>var a: array of integer;</code>	// Описание динамического массива
<code>SetLength(a, 10);</code>	// Создание динамического массива длины 10
<code>var b := Arr(5, 7, 9);</code>	// Динамический массив из 3 элементов, заполненный значениями 5, 7 и 9. Размер массива определяется автоматически по количеству параметров функции Arr
<code>var c := Arr(-10..10);</code>	// Динамический массив из 21 элемента, заполненный целыми значениями от -10 до 10
<code>var c1 := Arr('a'..'z');</code>	// Динамический массив из символов (char) от 'a' до 'z'
<code>var d := 5, 7, 9 ;</code>	// Аналогично Arr(5, 7, 9)
<code>var d1 := 7 * 12;</code>	// Массив из 12 семерок. «Питоновский» стиль
<code>var e := ArrFill(7, 3);</code>	// Динамический массив из 7 элементов, каждый элемент равен 3
<code>var f := ArrRandom;</code>	// Динамический массив из 10 случайных целых элементов в диапазоне [0; 100]. Значения параметров берутся по умолчанию
<code>f := ArrRandom(5);</code>	// Динамический массив из 5 случайных целых элементов в диапазоне [0; 100]. Можно повторно использовать ссылку; при этом старый массив уничтожится

Окончание табл. 1

Код	Пояснение
<code>f := ArrRandom(5, -2, 2);</code>	// Динамический массив из 5 случайных целых элементов в диапазоне $[-2; 2]$
<code>var g := ArrRandomReal;</code>	// Динамический массив из 10 случайных действительных элементов в диапазоне $[0; 10]$
<code>g := ArrRandomReal(5);</code>	// Динамический массив из 5 случайных действительных элементов в диапазоне $[0; 100]$
<code>g := ArrRandomReal(5, -2, 2);</code>	// Динамический массив из 5 случайных действительных элементов в диапазоне $[-2; 2]$

Таблица 2

Код	Пояснение
<code>var a := ReadArrInteger(10);</code>	// Вводим с клавиатуры в массив 10 целых чисел
<code>var b := ReadArrReal(5);</code>	// Вводим с клавиатуры в массив 5 действительных чисел
<code>var c := ReadArrString(12);</code>	// Вводим с клавиатуры в массив из 12 строк
<code>var d := ReadArrInt64(7);</code>	// Вводим с клавиатуры в массив из 7 целых чисел типа Int64
<code>var e := ReadArrInteger('Введите десять целых элементов массива:', 10);</code>	// Перед размером массива можно указывать приглашение

Печать массива можно выполнять процедурами **Write (Writeln)** и **Print (Println)**, а также методом **Print (Println)**. Первые две процедуры для динамических массивов работают одинаково. Рассмотрим код и пример вывода (табл. 3).

Таблица 3

Код	Результат вывода
<code>var a := ArrRandom(7); Writeln(a);</code>	[11, 24, 81, 41, 72, 2, 92]
<code>Println(a);</code>	[11, 24, 81, 41, 72, 2, 92]
<code>a.Println;</code>	11 24 81 41 72 2 92

Последний вариант наиболее удобен, так как не вводит лишних символов (скобок и запятых). Так следует выводить массивы при сдаче решений задач в системы автоматического тестирования.

5. Особенности ссылочной модели

Работая только в статической памяти, мы привыкаем, что переменная — это именованный программный объект, хранящий некоторое значение. При использовании ссылочной модели, как уже говорилось, хранимое в переменной значение на самом деле является не просто значением, а *ссылкой* на участок памяти, в котором, в свою очередь, размещены наши данные. Рассмотрим пример кода, результат работы и данные в памяти (рис. 5).

В этом примере мы два раза печатаем массив *a*, не изменяя его явно после создания. Тем не менее второй элемент этого массива становится равен 7. Все дело в том, что в строке `var b := a;` мы создаем вторую ссылочную переменную (*b*), которая равна *a*. Это значит, что ссылка, хранящаяся в *a*, скопирована в *b*, и теперь обе переменные ссылаются на одну и ту же динамическую переменную (массив). Другими словами, у созданного динамического массива теперь *два* равноправных имени:

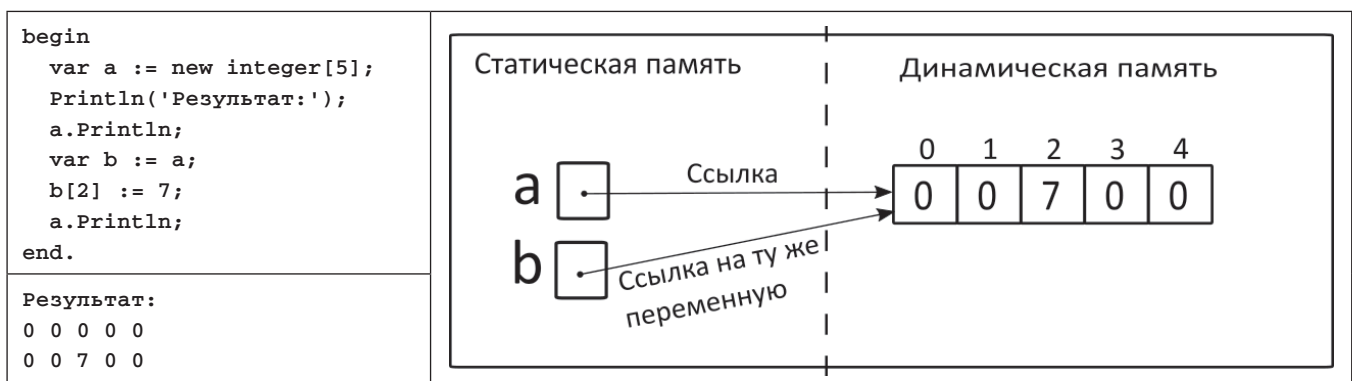


Рис. 5. Две ссылочные переменные указывают на одну и ту же область памяти

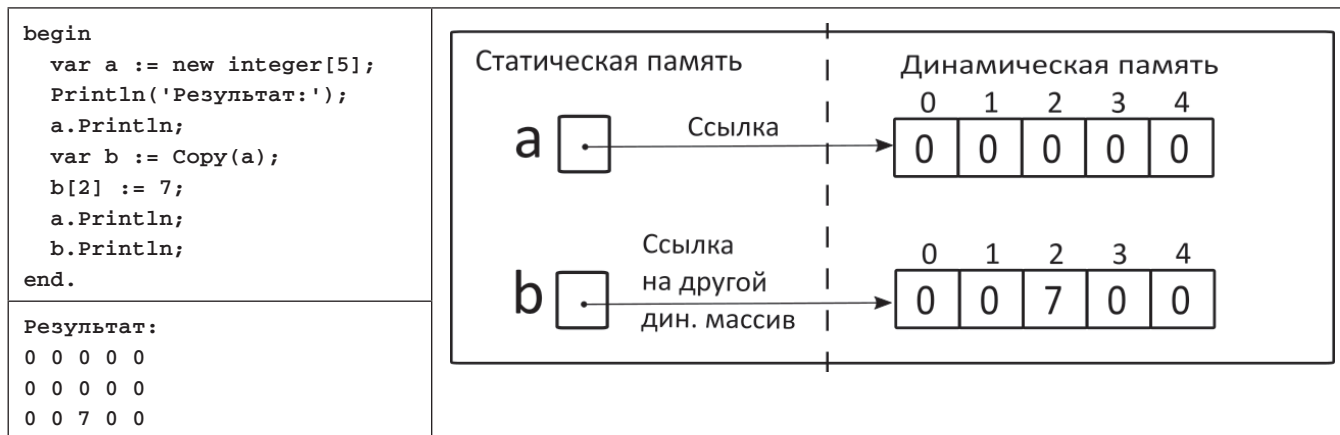


Рис. 6. Две копии одного и того же массива

a и *b*. Именно поэтому, когда меняется значение элемента массива *b*, эти изменения происходят и с массивом *a*. Разумеется, мы можем сделать и больше ссылок на один и тот же массив, как можем создать в папке на диске несколько ярлычков, ссылающихся на один и тот же файл.

Если нам нужны не две ссылки на один и тот же массив, а именно две копии одного и того же массива, то следует применить функцию **Copy** (рис. 6).

Оператор **var b := Copy(a)**; создает копию массива *a* и сохраняет ссылку на эту копию в переменной *b*.

6. Удаление динамических массивов. Сборщик мусора

В Microsoft.NET динамическая память находится под управлением **сборщика мусора** (garbage collector). Эта системная программа становится активной в некоторые, заранее нам неизвестные, моменты времени и выполняет работу по очистке и дефрагментации памяти. **Дефрагментация памяти** — это группировка динамических переменных в одной непрерывной области памяти. Она производится, если в памяти разбросано множество динамических переменных, так что не остается больших участков свободной памяти для новых переменных. Таким образом, сборщик мусора может перемещать динамические переменные по своему желанию, но при этом каждый раз будет автоматически обновлена ссылка на эту переменную.

Прибегать к очистке памяти следует, когда ее не хватает и при этом есть множество ненужных динамических переменных, лишь занимающих место. Сборщик мусора узнает, что переменная не нужна, если на нее нет ни одной ссылки (см. рис. 4, сверху). Таким образом, программист не должен вручную освобождать память, как в языках типа C++. Можно уничтожить динамическую переменную, просто уничтожив ссылку на нее:

```

var a := ArrFill(10, 3); // Новый массив
                        // из 10 троек
a := nil;               // Уничтожаем ссылку
                        // на массив a

```

Заметим, что такая распространенная ошибка, как утечка памяти, невозможна на платформе NET, именно благодаря наличию сборщика мусора.

7. Обработка массивов в цикле

Удобство массивов состоит в том, что мы можем работать с большим количеством значений как с единым целым. Одним из классических способов работы с массивами является обработка их элементов в цикле.

Рассмотрим несколько примеров с комментариями.

7.1. Удвоение каждого элемента в цикле For.

```

// Вместо Begin..End. можно использовать
// последовательность ## в начале программы
## var a := ArrRandom;
a.Println;
// a.High - индекс последнего элемента
// массива
// a.Length - длина (количество элементов)
// массива
// Т.е. a.High = a.Length - 1
for var i := 0 to a.High do
  a[i] *= 2; // То же, что a[i] := a[i] * 2
a.Println;

```

Это классический цикл For, который мы привыкли использовать для работы с массивами. Следует заметить, что динамический массив сам «знает» свой размер. Поэтому нет нужды в заведении дополнительных констант, хранящих эти сведения.

7.2. Нахождение суммы элементов массива в цикле Foreach.

Будем считать, что мы продолжаем писать предыдущую программу.

```

var sum := 0;
foreach var x in a do
  sum += x;
println(sum);

```

Цикл Foreach меньше знаком программистам на Pascal, чем цикл For. Между тем он бывает удобнее в тех случаях, когда не требуется изменения элементов массива, а нужно только перебрать их. В нашем случае мы перебираем все элементы и добавляем их к переменной *sum*. Здесь переменная *x* пробегает не по индексам, а по значениям элементов массива.

7.3. Удвоение каждого элемента в цикле Foreach с использованием свойства Indices.

Как упоминалось выше, в цикле Foreach нельзя изменять элементы массива. Однако у массива есть полезное свойство **Indices** (индексы), которое формально позволяет обойти указанное ограничение.

```
foreach var i in a.Indices do
    a[i] *= 2;
```

Здесь переменная *i* пробегает по всем индексам массива. То есть это просто другой способ записи примера 7.1. На самом деле свойство **Indices** гораздо интереснее, поскольку позволяет указывать в качестве своего параметра лямбда-функцию для отбора нужных элементов. Эта тема выходит за рамки статьи. Для интересующихся читателей приведем следующие примеры (подробнее см. [11]).

7.4. Обработка некоторых элементов с использованием свойства Indices.

```
foreach var i in a.Indices(p -> p in 3..7) do
    a[i] += 1;
```

В этом примере переменная *i* пробегает не по всем индексам массива, а только по индексам 3, 4, 5, 6 и 7. К каждому элементу на этих позициях прибавляется единица. Имя переменной в лямбда-функции (*p*) может быть любым. Читается это так: для всех индексов массива *a*, которые лежат на промежутке 3..7, увеличить соответствующий элемент на единицу.

```
foreach var i in a.Indices((x, p)
    -> p.IsEven and (x > 0)) do
    a[i] += 1;
```

Здесь в условии отбора в лямбда-функции мы обращаемся как к позиции (*p*) элемента, так и к его значению (*x*). (По соглашению первый параметр лямбды здесь элемент, а второй — его индекс.) Прочитаем код: для всех *положительных* элементов массива *a* (*x* > 0), стоящих на *четных* (*IsEven*) позициях, увеличить такой элемент на единицу.

8. Динамические массивы как параметры и результат работы подпрограмм

Работая со статическими массивами, мы привыкаем писать подпрограммы для массивов определенного и неизменного размера. Динамические массивы более гибкие в этом плане. Поскольку они знают свой размер, отпадает необходимость в заведении констант, хранящих эти данные. А значит, имеется возможность написания функций и процедур, работающих с массивами произвольного размера.

8.1. Передача динамического массива в функцию.

Рассмотрим пример, в котором разработаем функцию для нахождения суммы элементов целочисленного массива.

Пример.

```
function ArrSum(a: array of integer): integer;
begin
    result := 0;
    foreach var x in a do
        result += x;
    end;
begin
    var t := ArrRandom;
    t.Println;
    ArrSum(t).Print;
end.
```

В функцию передается динамический массив из целых чисел. Размер массива произволен, значение имеет только тип элементов. Далее обнуляем переменную *result*, через которую происходит возврат значений из функции. В принципе, этого можно и не делать, так как компилятор сам присвоит этой переменной значение по умолчанию (в данном случае ноль) в начале выполнения кода функции, но правила хорошего кода требуют от нас этого действия. Затем в цикле Foreach суммируются все элементы массива, а результат суммирования накапливается в переменной *result*.

В главной программе создается массив *t* из 10 случайных целых чисел. Массив выводится на экран. После этого вызывается функция, которой передается созданный массив, и сразу печатается результат (сумма элементов).

Обратите внимание на отсутствие раздела описания констант, типов и переменных, привычного всем по старому стилю программирования.

8.2. Возвращение динамического массива из функции.

Пример.

Рассмотрим две функции. В первой возьмем натуральное число *n* и создадим массив из чисел от 1 до *n*. Во второй прибавим к каждому элементу массива целое *k*.

```
function ArrMake(n: integer): array of integer;
begin
    result := new integer[n];
    // Или: SetLength(result, n)
    for var i := 0 to result.High do
        result[i] := i + 1;
    end;
function ArrPlusK(a: array of integer;
    k: integer): array of integer;
begin
    result := Copy(a);
    for var i := 0 to a.High do
        result[i] += k;
    end;
begin
    ArrPlusK(ArrMake(12), 3).Print;
end.
```

Функция *ArrMake* принимает целое *n* и возвращает массив из целых. Вначале создается новый динамический массив из целых и связывается с переменной *result*. Размер массива устанавливается равным параметру *n* функции. Затем в цикле For каждому *i*-му элементу при-

сваивается значение $(i + 1)$, так как нумерация элементов начинается с нуля, а в массиве по условию должны быть натуральные числа.

Функция *ArrPlusK* принимает два параметра: массив из целых и целое число k . Возвращает эта функция массив из целых. Поскольку нужно прибавлять k к каждому элементу массива a , в первой строке тела функции создается копия данного массива, которая связывается с переменной *result*. Затем в цикле *For* к каждому элементу массива *result* добавляется k .

В главной программе вызывается функция *ArrPlusK*, которой в качестве параметров передается то, что возвращает *ArrMake*(12) (т. е. массив из натуральных чисел 1..12), и число 3. Результат выводится на экран. В данном случае это натуральные числа 4..15.

Обратим особое внимание на первые строки тела функций. Если функция возвращает массив, то первым делом мы должны *создать* его.

Рассмотрим распространенные ошибки новичков. Первая ошибка легко находится компилятором, но для тех, кто не усвоил принципов работы с динамическими переменными, она может оказаться не вполне понятной.

```
function ArrMakeError(n: integer):
  array of integer;
begin
  for var i := 0 to n - 1 do
    result[i] := i + 1;
  end;
```

Здесь не создан массив *result*. Он объявлен, но не создан. Поэтому при выполнении тела цикла получаем сообщение об ошибке:

Ошибка времени выполнения: Ссылка на объект не указывает на экземпляр объекта.

Аналогичная ошибка случится, если убрать первую строку в теле функции *ArrPlusK*:

```
function ArrPlusKError(a: array of integer;
  k: integer): array of integer;
begin
  //result := Copy(a);
  for var i := 0 to a.High do
    result[i] += k;
  end;
```

Вторую ошибку можно продемонстрировать следующим кодом:

```
function ArrPlusKNotPure(a: array of integer;
  k: integer): array of integer;
begin
  for var i := 0 to a.High do
    a[i] += k;
  result := a;
end;
```

Эта функция будет работать практически так, как ожидалось. По крайней мере, она будет выдавать верный результат. Ошибка, однако, состоит в том, что в процессе своего выполнения эта функция изменит массив, который ей передали в качестве параметра, т. е. массив a . Если вы привыкли к тому, что параметр, переданный по значению (как в функции *ArrPlusKNotPure*), не может измениться вне функции, даже если поменять его

внутри, то с динамическими переменными это не так. Когда в подпрограмму передается динамический массив как параметр-значение, на самом деле передается ссылка на этот массив, что следует из самого устройства динамических массивов. Так что, если в подпрограмме размер динамического массива не меняется, ссылка останется прежней и при выходе из подпрограммы мы заметим, что массив поменялся, если его элементы менялись внутри подпрограммы. Если же нужно изменить размер массива в подпрограмме, то его передача должна происходить по ссылке, т. е. с указанием служебного слова *var*. При этом изменится и сама ссылка на массив.

Таким образом, в последнем случае описана функция, которая не является *чистой*, так как она меняет глобальные переменные. Это плохая практика программирования, и ее следует избегать.

8.3. Динамические массивы в процедурах.

Работа с массивами в процедурах мало отличается от таковой в функциях. Мы требуем от своих учеников выполнения стандартного правила: если параметр является входным, то передаем его по значению, если выходным или входным и выходным — то по ссылке. Это сокращает количество ошибок, о которых мы говорили выше.

Пример.

Написать процедуру, которая добавляет в конец массива сумму его элементов.

Из условия ясно, что размер массива изменится, поэтому передадим его по ссылке. У процедуры один параметр, который будет одновременно и входным, и выходным. Напишем два варианта процедуры: более традиционный и использующий конкатенацию массивов.

```
// Имена в PascalABC.NET могут быть русскими
procedure ДобавитьСумму1(var a: array of real);
begin
  SetLength(a, a.Length + 1);
  a[a.High] := a.Sum;
end;
procedure ДобавитьСумму2(var a: array of real);
begin
  a := a + |a.Sum|;
end;
begin
  var a := |1.0, 2, 3|;
  ДобавитьСумму1(a);
  a.Println;
  ДобавитьСумму2(a);
  a.Print;
end.
```

Процедура *ДобавьСумму1* принимает на вход массив из действительных чисел. Вначале мы увеличиваем длину массива на 1 при помощи процедуры *SetLength*. А затем в последнюю (только что добавленную) ячейку записываем сумму элементов массива. Заметим, что использовать вместо *SetLength* команду *new* неверно, так как она создаст массив со значениями по умолчанию (нулями), и мы потеряем содержимое массива.

Вторая процедура использует возможность конкатенации (склеивания) массивов. Эта операция знакома

многим по работе со строками. В единственной строке к массиву *a* приклеивается массив из одного элемента — суммы элементов данного массива. В контексте обсуждаемой темы полезно рассмотреть, как работает оператор присваивания:

```
a := a + |a.Sum|;
```

Сначала при помощи метода *a.Sum* вычисляется сумма элементов массива и из этого числа создается динамический массив. (Здесь мы забегаем вперед, в раздел 10, чтобы не загромождать код поиском суммы элементов.) Затем массив *a* и только что созданный массив из одного элемента склеиваются (операция конкатенации), так что в динамической памяти появляется новый динамический массив, а именно тот, который нам нужен. Ссылка на этот массив и присваивается переменной *a*. Какое-то время в памяти будут находиться все три массива: исходный *a*, новый *a* и массив из одного элемента. Но, поскольку ссылка есть только на увеличенный массив, он и останется в памяти после работы сборщика мусора. Остальные два массива будут уничтожены.

Код главной программы очевиден. Заметим только, что при создании массива командой `var a := |1.0, 2, 3|` мы были вынуждены хотя бы для одного элемента явно указать действительный тип: 1.0. Если этого не сделать, то при автовыводе типа массива будет установлено, что массив целочисленный (все его элементы — целые). А такой массив нельзя передавать в процедуру, принимающую на вход массив из действительных чисел.

В нашем случае программа выведет:

```
1 2 3 6
1 2 3 6 12
```

Кроме операции конкатенации (склеивания) для массивов определены операции повторения (*) и проверки вхождения элементов (in):

```
Var b := a * 3; // массив b есть массив a,
                // повторенный трижды
print(2 in a); // если число 2 есть в массиве a,
                // возвращается true, иначе false
```

Заметим, что операция in, которая раньше могла использоваться только при работе с множествами, теперь может применяться к любым контейнерам: массивам, спискам, строкам и т. п.

9. Списки List<T>

Говоря про динамические массивы, многие объясняют прилагательное «динамический» тем, что размер такого массива можно изменять в процессе выполнения программы. Последний разобраный нами пример демонстрирует, что изменение размера массива не является хорошей практикой, поскольку в процессе такого изменения создается несколько новых динамических массивов. Конечно, мы не замечаем этого, поскольку сборщик мусора работает очень эффективно. Но если размер массива значителен, да еще его изменение заключено в цикл, то мы можем столкнуться с неприятным замедлением работы программы и системы в целом.

В случае, если необходимо многократно менять размер контейнера данных, лучше использовать не массив,

а какой-то другой тип. В этой статье мы остановимся на самом ближайшем родственнике динамических массивов — на списках (List). Эти списки не следует путать с одно- или двунаправленными списками, построенными из записей или объектов и соединенными при помощи указателей или ссылок.

Если говорить совсем просто, то список — это по способу использования тот же массив, но в его конец можно легко добавлять элементы, не теряя при этом эффективности кода.

Рассмотрим основы работы со списками.

9.1. Описание и создание списка.

```
var a: List<integer>; // Описание списка
                        // из целых
a := new List<integer>; // Создание пустого
                        // списка
a := new List<integer>(12); // Создание списка
                        // из 12 элементов
```

Как обычно, описание и создание можно объединить:

```
var a := new List<integer>;
```

При описании мы указываем вид контейнера данных (List) и тип его элементов в угловых скобках (<integer>). То есть List<integer> семантически эквивалентно List of integer, но второй вариант использовать нельзя. Отличие в синтаксисе обусловлено тем, что массивы являются типами данных Pascal, а списки (а также стеки, очереди, множества на хэш-таблицах и др.) — типами платформы Microsoft.NET, где принята именно такая форма записи. Также можно указать размер списка при его создании.

9.2. Добавление элементов в список.

Пополнение списка элементами осуществляется двумя методами.

```
a.Add(1); // Добавление одного
           // элемента в конец списка
a.Add(2); // Добавление еще одного
           // элемента в конец списка
a.AddRange(5..8); // Добавление чисел из
                  // диапазона в конец списка
a.AddRange(|10, 40, 70|); // Добавление содержимого
                           // жимого массива
```

Результатом работы этих команд будет список: 1 2 5 6 7 8 10 40 70. Обычно операция добавления используется в цикле. Кроме того, для быстрого создания списка можно (по аналогии с командой `Agg` у массивов) воспользоваться командой `Lst`:

```
var a := Lst(3, 7, 5)
```

Пример.

Написать функцию, которая принимает на вход массив целых и возвращает список положительных элементов массива.

Сначала необходимо понять, почему возвращать надо не массив, а именно список. Заранее неизвестно, сколько положительных элементов будет в массиве. Какой размер выходного массива можно было бы указать? Неизвестно.

Пришлось бы многократно увеличивать размер выходного массива при обнаружении очередного положительного числа. Как было показано ранее, это не очень эффективно. Посмотрим, как в этой ситуации поможет список.

```
function Положительные(a: array of integer):
  List<integer>;
begin
  result := new List<integer>;
  foreach var x in a do
    if (x > 0) then
      result.Add(x);
end;
begin
  var t := ArrRandom(10, -20, 20);
  t.Println;
  var p := Положительные(t);
  p.Println;
end.
```

Как следует из заголовка, функция принимает на вход массив целых и возвращает список из целых. В первой строке тела функции создается динамическая переменная-список, связанная с *result*. Пока этот список пустой. Далее в цикле *ForEach* совершается проход по массиву, и каждый элемент *x* проверяется на соответствие условию (*x* > 0). Подходящие элементы добавляются в *result*.

В главной программе строится массив *t* из 10 случайных целых, лежащих на отрезке [−20; 20], который передается в функцию. Результат работы функции записывается в переменную *p*. Тип этой переменной — *List<integer>*.

Как изменилась бы программа, если бы в функцию передавался не массив, а список? В заголовке функции надо было бы заменить *a: array of integer* на *a: List<integer>*. В главной программе нужно было бы сделать *t* не массивом, а списком. В языке наибольшее количество вспомогательных подпрограмм разработано для массивов, для списков их гораздо меньше. Например, нет функции создания случайного списка. В этом случае можно сделать массив, а затем преобразовать его в список. Взаимные преобразования из

массива в список и обратно осуществляются методами *ToList* и *ToArray* соответственно. То есть при создании списка надо написать так:

```
var t := ArrRandom(10, -20, 20).ToList;
```

Еще одним отличием списков от массивов является общепринятый для Microsoft.NET способ определения длины контейнера. Это не *Length*, как принято в Pascal, а *Count*. Таким образом, если нужен цикл *For* по списку *p*, пишут так:

```
for var i := 0 to p.Count - 1 do ...
```

В остальном все, что говорилось выше о массивах, справедливо и для списков.

10. Некоторые методы массивов и списков

Методом называют подпрограмму, описанную внутри класса и реализующую функциональность класса. Здесь, чтобы не уходить глубоко в теорию объектно-ориентированного программирования, мы будем называть методами подпрограммы, которые можно вызывать «через точку»: **ИмяПеременной.ИмяМетода**. Не будем делать различий между методами класса и методами расширения. Кроме того, рассмотрим некоторые методы последовательностей, которые широко используются при работе с массивами и списками. В этом изложении нам также придется опустить те методы, которые имеют параметры-лямбды, так как последние не рассматривались в статье. Далее запись: **a.IndexMax(start := 0)** означает, что метод *IndexMax* может быть вызван с параметром, либо параметр можно опустить. В последнем случае (т. е. по умолчанию) значение параметра *start* будет принято равным нулю.

В таблице 4 представлено несколько примеров. Пусть *a* — динамический массив или список из чисел.

Нам нередко приходилось вступать в дискуссию с преподавателями, утверждавшими, что наличие такого огромного арсенала методов отучает учащихся думать,

Таблица 4

Код	Пояснение
a.Sum	Возвращает сумму элементов
a.Product	Возвращает произведение элементов
a.CountOf(x)	Возвращает количество элементов <i>x</i> в последовательности
a.PrintLines	Печатает все элементы, причем каждый с новой строки. Удобно, когда элемент имеет структурный тип (строки, записи и т. п.)
a.High	Возвращает индекс последнего элемента. Только для массивов
a.Max	Возвращает максимальный элемент
a.Min	Возвращает минимальный элемент
a.IndexMax(start := 0)	Возвращает индекс первого максимального элемента. Поиск ведется с позиции <i>start</i> . Заметим, что максимум может встречаться в массиве несколько раз
a.IndexMin(start := 0)	Возвращает индекс первого минимального элемента. Поиск ведется с позиции <i>start</i>
a.LastIndexMax()	Возвращает индекс последнего максимального элемента.

Окончание табл. 4

Код	Пояснение
<code>a.LastIndexMin()</code>	Возвращает индекс последнего минимального элемента
<code>a.LastIndexMax(stop)</code>	Возвращает индекс последнего максимального элемента в диапазоне $[0; stop-1]$
<code>a.LastIndexMin(stop)</code>	Возвращает индекс последнего минимального элемента в диапазоне $[0; stop-1]$
<code>a.IndexOf(x)</code>	Возвращает индекс первого вхождения элемента x в массив, или (-1) , если элемент не найден
<code>a.IndexOf(x, start)</code>	Возвращает индекс первого вхождения элемента x в массив, или (-1) , если элемент не найден. Поиск ведется с позиции <i>start</i>
<code>a.LastIndexOf(x)</code>	Возвращает индекс последнего вхождения элемента x в массив, или (-1) , если элемент не найден
<code>a.LastIndexOf(x, stop)</code>	Возвращает индекс последнего вхождения элемента x в массив, или (-1) , если элемент не найден. Поиск ведется до позиции <i>stop</i>
<code>a.RandomElement</code>	Возвращает один случайно выбранный элемент
<code>a.Shuffle</code>	Возвращает массив, в котором все элементы перемешаны случайным образом
<code>a.Sort</code>	Сортирует массив по возрастанию значения элементов. Сортировка по ключу возможна с использованием лямбда-функций
<code>a.SortDescending</code>	Сортирует массив по убыванию значения элементов
<code>a.Indices</code>	Возвращает последовательность индексов массива (см. раздел 7)
<code>a.Insert(p, x)</code>	<i>Для списков:</i> вставляет элемент x в список на позицию p . Позиция должна лежать в диапазоне $[0; a.Length]$
<code>a.Remove(x)</code>	<i>Для списков:</i> удаляет первое вхождение элемента x в список. Возвращает <code>True</code> , если удаление произошло успешно, и <code>False</code> в противном случае
<code>a.RemoveAt(pos)</code>	<i>Для списков:</i> удаляет элемент на позиции pos . Если позиция выходит за границы списка, возникает ошибка

готовит программистов, которые хотят иметь отдельную кнопку для решения любой задачи. Мы считаем, что это глубоко ошибочное утверждение. Если ученик напишет 10 раз алгоритм нахождения максимума в массиве, он не станет лучше думать, он просто выучит алгоритм. А когда он выучил, не нужно писать этот алгоритм в 11-й раз, можно воспользоваться готовыми методами. Мы опираемся в обучении на следующий принцип: вначале изучение алгоритма и его реализация, потом — использование готовой подпрограммы. Это позволяет работать на более высоком уровне абстракции, думать более «крупными блоками» и не загромождать код тривиальными алгоритмами, что особенно ценно при решении масштабных задач, например, в проектной деятельности.

11. Срезы

Срезом (slice) называется некоторая (не обязательно непрерывная) подпоследовательность исходной последовательности. Использование срезов в программировании не является чем-то новомодным. Впервые срезы появились в языке Fortran 66 (1966 год) [10]. С тех пор срезы являются традиционным способом работы с массивами, строками и другими последовательностями. Синтаксис срезов в различных языках может существенно отличаться, но семантика в целом одинакова: возьми элементы с такого-то по такой-то с таким-то шагом. Синтаксис

срезов PascalABC.NET больше всего похож на таковой в Python. Общий формат следующий:

`a[from:to:step]`

что означает: выбрать из массива a все элементы с позиции *from* (включая) до позиции *to* (не включая) с шагом *step*. Обратим еще раз внимание на то, что *правая граница не включается*. Заметим также, что шаг среза можно не указывать: по умолчанию он равен единице. После имени переменной (перед открывающейся квадратной скобкой) может стоять символ вопроса:

`a?[from:to:step]`

Это будет означать *безопасный срез*, при котором выход за границы диапазона индексов не вызывает ошибку, а в срезе останутся только допустимые значения индексов. Рассмотрим ряд примеров.

11.1. Срезы в направлении увеличения индексов (табл. 5).

11.2. Срезы в направлении уменьшения индексов (табл. 6).

Обратите внимание, что теперь правая граница среза меньше левой. Поэтому именно меньшая (она теперь стала правой) граница исключается из диапазона. Последний пример очень показателен: так можно развернуть массив от конца к началу.

Таблица 5

Код	Результат работы и пояснение
<code>## var a := 10, 1, 2, 3, 4, 5, 6, 7, 8, 9;</code>	Создаем массив из 10 элементов. Номер элемента совпадает с его значением
<code>a[2:7].Println;</code>	2 3 4 5 6
<code>a[5:].Println;</code>	5 6 7 8 9 Можно не указывать правую границу. При этом будет считаться, что мы берем элементы до последнего <i>включительно</i>
<code>a[:5].Println;</code>	0 1 2 3 4 Можно не указывать левую границу. При этом будет считаться, что мы берем все элементы с нулевого
<code>a[2:70].Println;</code>	Ошибка времени выполнения: Параметр <i>to</i> за пределами диапазона
<code>a[20:7].Println;</code>	Ошибка времени выполнения: Параметр <i>from</i> за пределами диапазона
<code>Println(a?[2:70]);</code>	[2,3,4,5,6,7,8,9] Ошибки нет. Справа ограничиваемся последним индексом
<code>Println(a?[20:7]);</code>	[] Ошибки нет, но возвращен пустой массив
<code>a[2:8:2].Println;</code>	2 4 6
<code>a[:].Println;</code>	0 1 2 3 4 5 6 7 8 9 Весь массив. То же, что просто <i>a</i>

Таблица 6

Код	Результат работы
<code>Println(a[7:2]);</code>	[]
<code>a[7:2:-1].Println;</code>	7 6 5 4 3
<code>a[:2:-2].Println;</code>	9 7 5 3
<code>a[7::-2].Println;</code>	7 5 3 1
<code>a[::-1].Println;</code>	9 8 7 6 5 4 3 2 1 0

11.3. Срезы с нумерацией от конца массива.

Мы привыкли нумеровать элементы массива слева направо, от нуля и далее. В PascalABC.NET можно нумеровать элементы справа налево. В этом случае *нумерация будет вестись с единицы, а не с нуля*. Для этого перед индексом нужно поставить знак «^» (крышечка).

Примеры приведены в таблице 7.

11.4. Использование срезов при решении задач.

В [12] предлагается интересный способ объяснения того, почему при определении среза мы берем интервалы, открытые справа. Нужно представить себе, что индексы в срезе нумеруют не элементы массива, а «перегородки» между элементами (позиции среза). Двоеточия тут обозначают крайние положения. На примере строки

Таблица 7

Код	Результат работы и пояснение
<code>a[^2].Println;</code>	8 Обращаемся ко второму элементу с конца
<code>a[a.Length - 2].Println;</code>	8 То же, но более длинная запись
<code>a[^7:~3].Println;</code>	3 4 5 6
<code>a[:^3].Println;</code>	0 1 2 3 4 5 6
<code>a[^4:].Println;</code>	6 7 8 9
<code>a[^1:1:-2].Println;</code>	9 7 5 3

Индексы:	0	1	2	3	4	5	6	7	8	
Позиция среза:	:	1	2	3	4	5	6	7	8	:
	+-----+									
	P a s c a l A B C									
	+-----+									
Индексы с конца:		^9	^8	^7	^6	^5	^4	^3	^2	^1
Позиция среза:	:	^8	^7	^6	^5	^4	^3	^2	^1	:

Рис. 7. Срезы

`s := "PascalABC"` это выглядит так, как показано на рисунке 7.

Таким образом, срез `s[2:7]` выдаст подстроку «scalA», т. е. с позиции среза 2 до позиции среза 7.

Использование срезов может значительно сократить код программы, но при этом потребуются дополнительная память для хранения среза и время для его создания в памяти. Рассмотрим пример.

Пример.

Написать функцию, которая принимает на вход массив, число и позицию внутри массива и возвращает новый массив, в котором данное число вставлено на данную позицию.

То есть нам нужно вставить элемент в массив на заданную позицию. Конечно, решить эту задачу можно и без помощи срезов. Придется увеличить размер массива на единицу, сдвинуть часть элементов вправо, чтобы освободить место для данного элемента и записать его на освободившееся место. Но мы напишем функцию с использованием срезов:

```
function Вставка(t: array of integer; x, p: integer): array of integer;
```

```
begin
    result := t[:p] + |x| + t[p:];
end;

begin
    var a := |1, 2, 3, 4, 5|;
    a := Вставка(a, 100, 2);
    a.Print; // 1 2 100 3 4 5
end.
```

К срезу от начала массива до позиции p (не включая) прибавили (подклеили) массив из данного элемента, а справа еще подклеили массив с позиции p (включая) до конца. Это же можно записать и в виде короткой функции:

```
function Вставка1(t: array of integer;
    x, p: integer) := t[:p] + |x| + t[p:];
```

Здесь важно не только заметное сокращение кода, но и то, что выполняемые действия ясны и доступны даже начинающему. Получить тот же результат с использованием цикла сложнее.

Правда, эта функция работает неверно для одного из граничных случаев, а именно, когда мы пытаемся вставить элемент в самый конец массива:

```
a := Вставка1(a, 100, 5); // Ошибка времени
// выполнения: Параметр from
// за пределами диапазона
```

Действительно, посмотрим, какие массивы будут склеиваться в этом случае:

```
t[:5] + |100| + t[5:]
```

Первый срез вполне нормальный: он задает весь массив. А вот второй срез пытается взять элементы с 5-го и далее. Но элемента с индексом 5 в массиве нет, отсюда и ошибка. Чтобы ее исправить, нужно воспользоваться безопасным срезом (добавляем знак вопроса после имени массива). Окончательно функция примет такой вид:

```
function Вставка2(t: array of integer;
    x, p: integer) := t[:p] + |x| + t[p?];
```

На этом же примере становится ясно, почему при срезах используют полуоткрытые интервалы: не нужно задумываться, где надо писать $p + 1$ или $p - 1$. Мы пишем только p , и в одном случае соответствующий элемент будет взят, а в другом — нет.

Надо, однако, понимать, что в некоторый момент времени при работе этой функции в памяти будут существовать одновременно пять массивов: исходный массив, два среза, массив из одного вставляемого элемента и склеенный массив-результат. То, что на них будет потрачена память, — это полбеды. Самое неприятное, что эти массивы надо *создавать* в памяти, а это требует времени, особенно если исходный массив велик. Об этом следует помнить, применяя срезы, и стараться не выполнять подобные операции над большими массивами в цикле.

12. Заключение

История использования обычных статических массивов в школе уходит корнями в тридцатилетнее прошлое. Наблюдая, как быстро меняются информационные технологии в последние десятилетия, мы с удивлением видим в школьных программах нечто действительно незбылемое, поистине соответствующее своему названию, — статические массивы. В этой статье мы попытались показать, что переход к изучению динамических массивов не сильно усложнит задачу учителя. Зато ученики получают в руки гибкий и удобный инструмент со множеством встроенных методов, а также смогут приобрести навыки современного программирования. Тем, кто впоследствии решит освоить Python, C#, C++ (STL), навыки работы с динамическими массивами будут очень полезны, а изучение второго языка пройдет гораздо легче. Потому что действительно важно не то, какой язык выучил школьник, а какие концепции программирования он усвоил и насколько быстро сможет перейти на другой язык, если ему это понадобится.

Список источников

1. Босова Л. И., Босова А. Ю. Информатика. 9 класс: учебник. 6-е изд. М.: БИНОМ. Лаборатория знаний, 2016. 208 с.
2. Галовиц Я. С++17. Стандартная библиотека шаблонов. СПб.: Питер, 2018. 432 с.
3. Калинин И. А., Самылкина Н. Н. Информатика. 10 класс. Углубленный уровень: учебник. М.: БИНОМ. Лаборатория знаний, 2019. 256 с.
4. Михалкович С. С., Дженджер В. О. Новые примитивы в школьном программировании // Преподавание информационных технологий в Российской Федерации: сборник научных трудов. Материалы Восемнадцатой открытой всероссийской конференции (Москва, онлайн, 14–15 мая 2020 г.) / отв. ред. А. В. Альминдеров. М., 2020. С. 333–334. https://it-education.ru/conf2020/thesis/IT-education_in_Russia_Thesis_2020_preview.pdf
5. Осипов А. PascalABC.NET. Введение в современное программирование. Ростов-на-Дону, 2019. 572 с. <http://pascalabc.net/downloads/OsipovBook/ModernProgr.pdf>
6. Поляков К. Ю., Еремин Е. А. Информатика. 10 класс. Базовый и углубленный уровни: учебник: в 2 ч. Ч. 2. М.: БИНОМ. Лаборатория знаний, 2020. 352 с.
7. Седжвик Р., Уэйн К., Дондеро Р. Программирование на языке Python: учебный курс / пер. с англ. СПб.: Альфа-книга, 2017. 736 с.
8. Семакин И. Г., Шеина Т. Ю., Шестакова Л. В. Информатика. Углубленный уровень: учебник для 10 класса: в 2 ч. Ч. 1. М.: БИНОМ. Лаборатория знаний, 2014. 184 с.
9. Система программирования PascalABC.NET. <http://pascalabc.net/>
10. Array slicing // Wikipedia. https://en.wikipedia.org/wiki/Array_slicing
11. PascalABC.NET: статьи, примеры. Программы и алгоритмы с использованием массивов. https://pascalabcnet.github.io/progr_arrays.html
12. The Python Tutorial. An Informal Introduction to Python. <https://docs.python.org/3/tutorial/introduction.html#strings>