

Система PascalABC.NET

PascalABC.NET – это **система программирования и язык Pascal нового поколения** для платформы Microsoft .NET. Язык PascalABC.NET содержит все основные элементы современных языков программирования: [модули](#), [классы](#), [перегрузку операций](#), [интерфейсы](#), [исключения](#), [обобщенные классы](#), [сборку мусора](#), [лямбда-выражения](#), а также некоторые средства параллельности, в том числе [директивы OpenMP](#). Система PascalABC.NET включает в себя также простую интегрированную среду, ориентированную на эффективное обучение современному программированию.

- [Описание языка PascalABC.NET](#).
- [Преимущества PascalABC.NET](#) для разработки программ и для обучения.
- [Отличия PascalABC.NET от Delphi \(Object Pascal\)](#).
- Примеры, иллюстрирующие основные особенности PascalABC.NET, находятся в меню "Помощь/Коротко о главном".

Язык Паскаль был разработан швейцарским ученым Никлаусом Виртом в 1970 г. как язык со строгой типизацией и интуитивно понятным синтаксисом. В 80-е годы наиболее известной реализацией стал компилятор Turbo Pascal фирмы Borland, в 90-е ему на смену пришла среда программирования Delphi, которая стала одной из лучших сред для быстрого создания приложений под Windows. Delphi ввела в язык Паскаль ряд удачных объектно-ориентированных расширений, обновленный язык получил название Object Pascal. С версии Delphi 7 язык Delphi Object Pascal стал называться просто Delphi. Из альтернативных реализаций Object Pascal следует отметить многоплатформенный open source компилятор Free Pascal.

Создание PascalABC.NET диктовалось двумя основными причинами: устаревание стандартного языка Pascal и систем, построенных на его основе (Free Pascal), а также необходимость в современной простой, бесплатной и мощной интегрированной среде программирования.

PascalABC.NET опирается на передовую платформу программирования Microsoft.NET, которая обеспечивает язык **PascalABC.NET** огромным количеством стандартных библиотек и позволяет легко сочетать его с другими .NET-языками: C#, Visual Basic.NET, управляемый C++, Охугене и др. Платформа .NET предоставляет также такие языковые средства как единый механизм обработки исключений, единый механизм управления памятью в виде *сборки мусора*, а также возможность свободного использования классов, наследования, полиморфизма и интерфейсов между модулями, написанными на разных .NET-языках. О том, что такое платформа Microsoft.NET, о ее преимуществах для программирования и для обучения можно прочитать [здесь](#).

Язык **PascalABC.NET** близок к реализации Delphi (Object Pascal). В нем отсутствует ряд [специфических языковых конструкций Delphi](#), некоторые конструкции изменены. Кроме этого, добавлен ряд возможностей: имеется автоопределение типа при описании, можно описывать переменные внутри блока, имеются операции +=, -=, *=, /=, методы можно описывать непосредственно в теле класса или записи, можно пользоваться встроенными в стандартные типы методами и свойствами, память под объекты управляется [сборщиком мусора](#) и не требует явного освобождения, множества **set** могут быть созданы на основе произвольных типов, введен операторы **foreach**, переменные циклов **for** и **foreach** можно описывать непосредственно в заголовке цикла, имеются обобщенные классы и подпрограммы (generics), последовательности, кортежи, срезы, лямбда-выражения и др.

Близким по идеологии к **PascalABC.NET** является язык RemObjects Охугене (Object Pascal 21 века). Однако он сильно изменен в сторону .NET: нет глобальных описаний, все описания помещаются в класс, содержащий статический метод Main, отсутствует ряд стандартных подпрограмм языка Паскаль. Кроме того, система RemObjects Охугене - платная и не содержит собственной оболочки (встраивается в Visual Studio и другие IDE), что практически делает невозможным ее использование в сфере образования.

Интегрированная среда **PascalABC.NET** обеспечивает подсветку синтаксиса, подсказку по коду (подсказка по точке, подсказка

параметров подпрограмм, всплывающая подсказка по коду), форматирование текста программы по запросу, переход к определению и реализации имени, элементы рефакторинга.

Все права на систему программирования **PascalABC.NET** принадлежат PascalABCCompiler Team (web-сайт <http://pascalabc.net>).

Коротко о главном

Данный текст содержит краткий **обзор особенностей** PascalABC.NET.

- PascalABC.NET – легковесная и мощная среда разработки программ с подробной справочной системой, средствами подсказки по коду, автоформатированием, встроенным отладчиком и встроенным дизайнером форм. Интегрированная среда разработки PascalABC.NET ориентирована на создание проектов малой и средней сложности, а также на обучение современному программированию.
- PascalABC.NET – мощный и современный язык программирования. По предоставляемым возможностям он превосходит язык Delphi и содержит практически все возможности языка C#.
- PascalABC.NET опирается на платформу Microsoft .NET - её языковые возможности и библиотеки, что делает его гибким, эффективным, постоянно развивающимся. Кроме того, можно легко сочетать библиотеки, разработанные на PascalABC.NET и других .NET-языках.
- Компилятор PascalABC.NET генерирует код, выполняющийся так же быстро, как и код на C#, и несколько медленнее, чем код на C++ и Delphi.
- PascalABC.NET является представителем линейки современных языков Паскаль вместе с Delphi XE и Oxygene.
- Мнение, что язык Паскаль устарел и утрачивает свои позиции, основано на представлении о старом Паскале и старых средах программирования (например, Free Pascal с его несовременной консольной оболочкой и языком Delphi образца 2002 года). К сожалению, масса отечественной учебной литературы с упорством, достойным лучшего применения, ориентируется на отживший Turbo Pascal с древней консольной оболочкой, бедной графической библиотекой и устаревшими средствами объектно-ориентированного программирования, развивая у обучающихся стойкое отвращение к языку Паскаль вообще.
- PascalABC.NET расширен современными языковыми

возможностями для легкого, компактного и понятного программирования.

- PascalABC.NET – достаточно зрелая среда. Ее прототип – учебная система Pascal ABC – появилась в 2002 году. PascalABC.NET – развивающаяся среда. Ведутся разработки новых языковых возможностей, новых библиотек.

Далее приводится **ряд программ с короткими комментариями**, раскрывающих возможности и особенности языка PascalABC.NET.

Тексты программ располагаются в рабочей папке (по умолчанию C:\PABCWork.NET) в подпапке Samples\!MainFeatures.

Для запуска программ данное окно должно быть открыто через пункт меню "Помощь/Коротко о главном" так, чтобы оно не полностью закрывало окно оболочки **PascalABC.NET**.

Основное

1. [AssignExt.pas](#). Данный пример иллюстрирует использование расширенных операторов присваивания += -= *= /= для целых и вещественных. Оператор /= для целых, разумеется, запрещен.
2. [BlockVar.pas](#). Переменные могут описываться внутри блока begin-end и инициализироваться при описании. Это крайне удобно для промежуточных переменных, а в PascalABC.NET в силу особенностей реализации еще и ускоряет доступ к переменным процентов на 30.
3. [AutoVars.pas](#). Если переменная инициализируется при описании, то ее тип можно не указывать: он определяется по типу правой части (автоопределение типа). Переменную - параметр цикла for можно описывать прямо в заголовке цикла, сочетая это с автоопределением типа.
4. [SimpleNewFeatures.pas](#). Пример, объединяющий возможности из предыдущих трех примеров.
5. [WriteAll.pas](#). Процедура write выводит любой тип. В частности, она выводит все элементы множества. Если тип ей неизвестен, то она выводит имя типа.
6. [WriteFormat.pas](#). Стандартная процедура WriteFormat позволяет осуществлять форматированный вывод. Вид форматной строки заимствуется из .NET.
7. [StandardTypes.pas](#). В этой программе приведены все

стандартные целые и вещественные типы. Программа выводит их размеры.

8. [RandomDiap.pas](#). К функциям генерации случайных чисел добавилась `Random(a,b)`, возвращающая случайное целое в диапазоне `[a,b]`. Процедуру `Randomize` в начале программы вызывать не надо.
9. [RealExtNums.pas](#). Действия с вещественными значениями не могут в `.NET` привести к переполнению. При некорректных операциях (деление на 0, переполнение или взятие логарифма отрицательного числа) мы получим либо значение "бесконечность", либо значение "NaN" (не число).
10. [Foreach.pas](#). Оператор `foreach` предназначен для цикла по контейнерам, таким как массивы, множества и контейнеры стандартной библиотеки (например, `List<T>`). Элементы контейнера доступны только на чтение.
11. [Amp.pas](#). Ключевые слова могут использоваться в качестве имен, в этом случае перед ними следует ставить значок `&` снятия атрибута ключевого слова. Кроме того, ключевые слова могут использоваться в качестве полей. Например, `&Type` или `System.Type`.

Типы

12. [CharFunc.pas](#). Символы `Char` хранят `Unicode` и поэтому занимают 2 байта. Для функций `Ord` и `Chr` оставлено, тем не менее, прежнее действие (предполагается, что символы находятся в `Windows`-кодировке). Для работы с кодами `Unicode` следует использовать `OrdUnicode` и `ChrUnicode`.
13. [StringTypes.pas](#). Строки `string` занимают память переменной длины и проецируются на `.NET`-тип `System.String`. Однако, в отличие от `NET`-строк они изменяемы и индексируются с 1. Для работы со строками фиксированной длины следует использовать тип `string[n]` или `shortstring=string[255]`. В частности, типизированные файлы допустимы только для коротких строк.
14. [StringMethods.pas](#). Строки `string` имеют ряд методов как `.NET`-классы. В этих методах предполагается, что строки индексируются с нуля.
15. [StringInteger.pas](#). Все типы - классы. Простые типы тоже.

Поэтому преобразование строки в целое и вещественное проще выполнять с помощью статических методов Parse соответствующего класса (например, integer.Parse(s)). Преобразование целого или вещественного в строку удобнее выполнять с помощью экземплярного метода ToString (например, r.ToString).

16. [Enum.pas](#). Перечислимый тип позволяет обращаться к его константам не только непосредственно, но и используя запись вида ИмяТипа.ИмяКонстанты. Нелишне отметить, что все перечислимые типы - производные от System.Enum.
17. [Sets.pas](#). Множества могут иметь произвольный базовый тип. Внутри множество хранится как хеш-таблица, однако при выводе множества в процедуре write его элементы упорядочиваются.
18. [DynArray.pas](#). Динамические массивы array of T представляют собой ссылки. Память под них должна выделяться либо вызовом стандартной процедуры SetLength, либо использованием инициализатора вида new T[n]. Процедура SetLength сохраняет старое содержимое массива. Динамические массивы являются классом, производным от класса System.Array, имеющего достаточно богатый интерфейс. Следует упомянуть прежде всего статические методы &Array.Sort и &Array.Resize.
19. [InitRecords.pas](#). В записях допустимы инициализаторы полей. Поля записи инициализируются при создании переменной-записи.
20. [UntypedFile.pas](#). Бестиповые файлы file изменены по сравнению с Delphi. Отсутствуют процедуры BlockRead и BlockWrite, но в бестиповой файл можно непосредственно записывать данные разных типов. Лишь бы считывание производилось в том же порядке.
21. [PointerToRef.pas](#). Имеют место некоторые ограничения для указателей на управляемую память. Так, указатель не может прямо или косвенно указывать на объект класса, память для которого выделена вызовом конструктора.
22. [Pointers.pas](#) и [References.pas](#). Указатели утрачивают свои позиции. Вместо них мы рекомендуем активно использовать

ССЫЛКИ.

23. [StructTypeEquiv.pas](#). В отличие от Delphi, для некоторых типов имеет место структурная, а не именная эквивалентность типов. Так, структурная эквивалентность имеет место для динамических массивов, указателей, множеств и процедурных типов.

Подпрограммы

24. [FuncParams.pas](#). Подпрограммы с переменным числом параметров делаются легко добавлением ключевого слова `params` перед параметром - динамическим массивом. Такой параметр должен быть последним в списке.
25. [Overload.pas](#). Перегрузка имен подпрограмм осуществляется без ключевого слова `overload`.
26. [ProcVars.pas](#). Процедурные переменные могут "накапливать" действия при помощи оператора `+=`. Эти действия можно отключать при помощи оператора `-=`. Процедурные переменные могут инициализироваться не только обычными подпрограммами, но и статическими и экземплярными методами класса.
27. [SwapT.pas](#). Обобщенные подпрограммы имеют простой синтаксис и используются сразу наряду с обычными: `procedure Swap<T>(var x,y: T);`

Модули

28. [SystemUnitTest.pas](#). Системный модуль имеет название `PABCSystem`, а не `System`, как в Delphi, и подключается неявно первым в списке `uses`. Причина такого именования состоит в том, что важнейшее пространство имен `.NET` имеет имя `System`. Системный модуль объединяет многие подпрограммы модулей `System`, `Math` и `Utils` языка Delphi. Данная программа иллюстрирует пересечение имен во модуле `PABCSystem` и пространстве имен `System`.
29. [MainProgram.pas](#) и [MyUnit.pas](#). Модуль может иметь упрощенный синтаксис (без деления на раздел интерфейса и раздел реализации), что удобно для начального обучения. В этом случае все описанные имена попадают в раздел интерфейса модуля.

30. [SystemUnitTest.pas](#). Для использования пространств имен .NET применяется тот же синтаксис, что и при подключении модулей: пространства имен .NET указываются в списке uses. Порядок поиска имен такой же, как и в Delphi - справа налево в списке uses, модуль PABCSystem просматривается последним.
31. [Main.pas](#) и [MyDll.pas](#). В PascalABC.NET легко создать и использовать dll. Библиотека dll по-существу представляет собой модуль, где вместо ключевого слова unit используется слово library. Для подключения dll к другой программе используется директива компилятора reference.
32. [CallCS.pas](#). PascalABC.NET - полноценный .NET-язык, легко совмещаемый с другими .NET-языками. В данном примере показывается, как в программе на PascalABC.NET вызвать функцию из dll, созданной на C#.
33. [CallNative.pas](#). PascalABC.NET позволяет легко вызывать функции из обычных dll.

Стандартные графические библиотеки

34. [GraphABCTest.pas](#). Графическая библиотека GraphABC заточена под легкое обучение программированию графики. Она скрывает большинство сложностей программирования графики: сама осуществляет перерисовку графического окна в нужный момент и заботится о синхронизации рисования в нескольких обработчиках. Кроме того, графические примитивы - процедурные, а значит, не надо создавать многочисленные классы, как в NET. И еще можно писать графические команды сразу после begin основной программы, то есть использовать графику в несобытийных приложениях.
35. [MouseEvents.pas](#). Для графических приложений можно использовать простейшие события мыши и клавиатуры, реализованные как глобальные процедурные переменные.
36. [ABC.pas](#). Библиотека векторных графических объектов ABCObjects используется нами для раннего обучения школьников основам объектно-ориентированного программирования. Однако, ее вполне можно использовать для написания несложных графических обучающе-игровых приложений.

Классы

37. [AllFromObject.pas](#). Все классы - наследники Object, все типы - классы. У каждой переменной можно узнать тип, вызвав метод GetType. Операция typeof для типа возвращает System.Type.
38. [WriteRecord.pas](#). Переопределив метод ToString в классе или записи, мы получаем возможность выводить их значения в процедуре writeln
39. [ClassConstructor.pas](#). Для статических методов и полей используется ключевое слово class. Статические конструкторы используются для нетривиальной инициализации статических полей.
40. [PersonInternal.pas](#). Новый синтаксис конструкторов использует ключевое слово new и является предпочтительным. По этой причине все конструкторы, определенные в старом стиле, должны иметь имя Create. Описание методов может производиться непосредственно внутри классов и записей (как в C++, C# и Java)
41. [Records.pas](#). Методы и конструкторы в записях можно использовать так же, как и в классах. От записей нельзя наследовать и записи нельзя наследовать.
42. [Boxing.pas](#). При присваивании размерного типа объекту типа Object происходит упаковка. Для распаковки следует использовать явное приведение типа.
43. [GarbageCollection.pas](#). Деструкторы отсутствуют. Автоматическая сборка мусора для возврата памяти, распределенной объектной переменной, требует, чтобы на эту память никто более не ссылался, прямо или косвенно. Поэтому для освобождения памяти обычно достаточно присвоить объектной переменной nil.
44. [OperatorOverloading.pas](#). Как и в C++ и C#, в PascalABC.NET можно перегружать знаки операций для записей и классов.
45. [Interf.pas](#). Интерфейсы семантически совпадают с интерфейсами в C# и Java. Сложная реализация интерфейсов Delphi на основе COM отвергнута.
46. [Stack.pas](#). Обобщенные классы (generics) позволяют создавать классы, параметризованные одним или несколькими типами.
47. [Where.pas](#). Можно задавать ограничения на типы параметров

обобщенных классов. Ограничения бывают трех сортов: наличие у типа-параметра конструктора по умолчанию, наследование его от конкретного класса или реализация интерфейса.

Стандартная библиотека .NET

48. [**DateTime.pas**](#). Данный пример иллюстрирует применение класса DateTime из стандартной библиотеки .NET.
49. [**LinkedList.pas**](#). Данный пример иллюстрирует использование контейнерных классов из стандартной библиотеки .NET.
50. [**WinFormWithButton.pas**](#). Данный пример иллюстрирует создание оконного приложения.

Что такое .NET

Платформа **Microsoft .NET** - это комплекс программ, устанавливаемый поверх операционной системы и обеспечивающий выполнение программ, написанных специально для .NET. .NET-программы компактны, пользуются единым набором типов данных и библиотек. Компания Microsoft активно развивает платформу .NET, выпуская новые версии с расширенными возможностями. На момент начала 2015 г. последней версией является .NET 4.5.

В результате компиляции .NET-программы генерируется не машинный код, а так называемый байт-код, содержащий команды *виртуальной машины* (в .NET он называется **IL-кодом** от англ. Intermediate Language - промежуточный язык). Команды байт-кода не зависят от процессора и используемой операционной системы. При запуске программа, содержащая IL-код, подается на вход виртуальной машины, которая и производит выполнение программы. Часть виртуальной машины, называемая **JIT-компилятором** (Just In Time - непосредственно в данный момент), сразу после запуска .NET-программы переводит ее промежуточный код в машинный (проводя при этом его оптимизацию), после чего запускает программу на исполнение. Если быть точными, то промежуточный код переводится в машинный частями по мере выполнения программы.

Такой способ двойной компиляции сложнее обычного, но имеет ряд преимуществ. Во-первых, JIT-компилятор может определить тип процессора, установленного на данном компьютере, поэтому генерирует максимально эффективный машинный код. Тесты показывают, что за счет этого некоторые программы выполняются даже быстрее обычных. Во-вторых, IL-код - гораздо более высокоуровневый, чем машинный, и содержит ряд объектно-ориентированных команд. В их числе - команда `newobj` вызова конструктора объекта, команда `callvirt` вызова виртуального метода объекта и команда `throw` генерации исключения.

Программа или библиотека для .NET называется *сборкой* и имеет традиционное расширение - `exe` или `dll`. Поскольку в сборках содержится IL-код, они значительно компактнее обычных программ и

библиотек. Так, приложение с главным окном, меню и элементами управления занимает на диске всего несколько десятков килобайт. Наиболее "чистым" .NET-языком является **C#**: он создавался специально для платформы .NET и включает практически все ее возможности. .NET-языки легко взаимодействуют друг с другом не только за счет высокоуровневого промежуточного кода, но и за счет общей системы типов (**CTS** - Common Type System - общая система типов). Все стандартные типы (строковые, символьные, числовые и логический) имеют одинаковое представление в памяти во всех .NET-языках. Это позволяет, например, создать библиотеку dll на C#, поместить в нее описание класса, а затем воспользоваться этой библиотекой из программы на PascalABC.NET, сконструировав объект данного класса. Можно также разработать библиотеку на PascalABC.NET, а потом подключить ее к проекту на Visual Basic.NET. Отметим, что традиционные библиотеки dll не позволяют хранить классы, доступные извне, и обладают рядом других ограничений.

Важнейшими средствами, предоставляемыми платформой .NET, являются единый способ обработки ошибок - генерация и перехват исключений, - а также автоматическое управление освобождением динамической памяти, называемое *сборкой мусора*. Последнее, в частности, означает, что отсутствует необходимость в деструкторах классов.

Имеются программы, которые могут восстанавливать текст программы по IL-коду (например, программа ILSpy).

Помимо JIT-компилятора, важной частью платформы .NET является набор стандартных библиотек (FCL - Foundation Class Library - общая библиотека классов). Среди них - библиотеки работы с графикой, сетью, базами данных, XML, контейнерами, потоками, содержащие тысячи классов. Каждый .NET-язык может пользоваться всеми возможностями этих библиотек.

Имеется открытая кроссплатформенная реализация среды Microsoft.NET - [среда Mono](#), позволяющая в частности разрабатывать и запускать .NET-программы под Linux.

Кратко отметим достоинства и недостатки платформы .NET.

Достоинства платформы .NET

1. Платформа .NET поддерживает множество .NET-языков. В их числе C#, Visual Basic.NET, F#, управляемый C++, Delphi Prism, Oberon, Zonnon, Iron Python, Iron Ruby, PascalABC.NET.
2. Любой .NET-язык содержит самые современные языковые возможности: классы, свойства, полиморфизм, исключения, перегрузка операций, легкое создание библиотек.
3. .NET-языки легко сочетаются друг с другом, похожи друг на друга по синтаксическим конструкциям и системе типов.
4. Имеется обширная библиотека стандартных классов FCL.
5. .NET-приложения компактны.
6. Платформа .NET активно развивается фирмой Microsoft, добавляются как новые языковые возможности, так и новые библиотеки.
7. Компилятор .NET-языка создать значительно проще, чем компилятор обычного языка.

Недостатки платформы .NET

1. Запуск .NET-приложения выполняется в несколько раз медленнее запуска обычного приложения, поскольку требует загрузки в оперативную память компонентов виртуальной машины и внешних библиотек.
2. .NET-код в некоторых ситуациях работает медленнее обычного (однако, в большинстве задач это отставание незначительно, а в некоторых - приложения .NET могут опережать обычные программы).
3. Сборщик мусора начинает работу в момент исчерпания динамической памяти, его работа занимает несколько миллисекунд. Для приложений реального времени это непозволительно.
4. Запуск .NET-приложения обязательно требует установки на компьютере платформы .NET. Без нее приложение работать не будет (Отметим, что в Windows Vista и в Windows 7 платформа .NET встроена).

Отметим, что достоинства платформы .NET многократно перекрывают ее недостатки.

Преимущества PascalABC.NET

Современный язык программирования Object Pascal

Язык **PascalABC.NET** включает в себя практически весь стандартный язык Паскаль, а также большинство языковых расширений языка Delphi. Однако, этих средств недостаточно для современного программирования. Именно поэтому **PascalABC.NET** расширен рядом конструкций, а его стандартный модуль - рядом подпрограмм, типов и классов, что позволяет создавать легко читающиеся приложения средней сложности.

Кроме этого, язык **PascalABC.NET** использует большинство средств, предоставляемых платформой .NET: единая система типов, классы, интерфейсы, исключения, делегаты, перегрузка операций, обобщенные типы (generics), методы расширения, лямбда-выражения.

Стандартный модуль PABCSystem, автоматически подключаемый к любой программе, содержит огромное количество стандартных типов и подпрограмм, позволяющих писать ясные и компактные программы.

В распоряжении **PascalABC.NET** находятся все средства .NET-библиотек классов, постоянно расширяющихся самыми современными возможностями. Это позволяет легко писать на **PascalABC.NET** приложения для работы с сетью, Web, XML-документами, использовать регулярные выражения и многое другое.

Язык **PascalABC.NET** позволяет программировать в классическом *процедурном стиле*, в *объектно-ориентированном стиле* и содержит множество элементов для программирования в *функциональном стиле*. Выбор стиля или комбинации этих стилей - дело вкуса программиста, а при использовании в обучении - методический подход преподавателя.

Сочетание богатых и современных языковых средств, возможностей выбора разных траекторий обучения позволяет рекомендовать **PascalABC.NET** с одной стороны как язык для обучения программированию (от школьников до студентов младших и средних курсов), с другой - как язык для создания проектов и библиотек

средней сложности.

Простая и мощная среда разработки

Интегрированная среда разработки **PascalABC.NET** ориентирована на создание проектов малой и средней сложности. Она достаточно легковесна и в то же время обеспечивает разработчика всеми необходимыми средствами, такими как встроенный отладчик, средства Intellisense (подсказка по точке, подсказка по параметрам, всплывающая подсказка по имени), переход к определению и реализации подпрограммы, шаблоны кода, автоформатирование кода.

В среде **PascalABC.NET** встроен также *дизайнер форм*, позволяющий создавать полноценные оконные приложения в стиле RAD (Rapid Application Development - быстрое создание приложений).

В отличие от многих профессиональных сред, среда разработки **PascalABC.NET** не имеет громоздкого интерфейса и не создает множество дополнительных вспомогательных файлов на диске при компиляции программы. Для небольших программ это позволяет соблюсти принцип "Одна программа - один файл на диске".

В среде **PascalABC.NET** большое внимание уделено связи запущенной программы с оболочкой: консольная программа, запущенная из-под оболочки, осуществляет ввод-вывод в специальное окно, встроенное в оболочку. Можно также запустить несколько программ одновременно - все они будут контролироваться оболочкой.

Интегрированная среда **PascalABC.NET** позволяет переключать в настройках русский и английский язык, при этом локализованы не только элементы интерфейса, но и сообщения об ошибках.

Кроме этого, внутренние представления **PascalABC.NET** позволяют создавать компиляторы других языков программирования и встраивать их в среду разработки с помощью специальных [плагинов](#).

Специализированные модули для обучения

Платформа Microsoft.NET обеспечивает **PascalABC.NET** стандартной библиотекой, состоящей из огромного количества классов для решения практически любых задач: от алгоритмических до прикладных. Именно поэтому в **PascalABC.NET** отсутствует необходимость в разработке большого числа собственных модулей.

Собственные модули, которые имеются в **PascalABC.NET**, ориентированы именно на начальное обучение программированию.

Для обучения программированию школьников реализованы модули классических школьных исполнителей Робот и Чертежник, содержащие около двухсот автоматически проверяемых заданий на основные конструкции языка программирования.

Кроме этого, среда **PascalABC.NET** содержит модуль электронного задачника [Programming Taskbook](#) (автор Абрамян М.Э.), позволяющий осуществлять автоматическую постановку и проверку заданий. Имеются также модули для преподавателя, позволяющие создавать задания для исполнителей Робот, Чертежник и электронного задачника.

Модуль растровой графики GraphABC и модуль векторных графических объектов ABCObjects могут быть использованы для создания простейших графических, а также интерактивных анимационных приложений, управляемых событиями.

Следует также отметить "студенческие" модули: модуль Collections упрощенных коллекций, модуль Arrays для простейших операций с динамическими массивами и модуль Forms для ручного создания простых приложений с оконным пользовательским интерфейсом.

Отличия языка PascalABC.NET от Delphi

Добавлено

1. Операции `+=` `-=` для событий .NET и для процедурных переменных.
2. Операции `+=` `-=` `*=` для целых и `+=` `-=` `*=` `/=` для вещественных.
3. Операция `+=` для строк.
4. Подпрограммы с переменным числом параметров.
5. Операция **new** для вызова конструктора (`ident := new type_name(params);`).
6. Операция **new** для создания динамического массива.
7. Операция **typeof**.
8. Использование **uses** для подключения пространств имен .NET (реализовано в Delphi Prism).
9. Вид доступа **internal** (наряду с **public**, **private**, **protected**).
10. Инициализация переменных: `var a: integer := 1;`
11. Инициализация переменных: `var a := 1;`
12. Объявление локальных переменных в блоке.
13. Объявление параметра цикла в заголовке цикла: `for var i := 1 to 10 do, foreach var x in a do.`
14. Оператор **lock**, обеспечивающий синхронизацию потоков.
15. Методы в записях.
16. Инициализаторы полей в классах и записях.
17. Обобщенные классы (generics).
18. Реализованы типизированные файлы (в отличие от Delphi Prism, где они убраны).
19. Упрощенный синтаксис модулей.
20. Описание методов внутри интерфейса класса или записи.
21. Реализация записью интерфейса.
22. Методы расширения.
23. Лямбда-выражения.

Изменено

1. Только сокращенное вычисление логических выражений.
2. Другой синтаксис **foreach**.
3. Интерфейсы **interface** в стиле .NET.

4. Другой синтаксис перегрузки операций.
5. Статические методы классов вместо классовых методов. Отсутствие типа `TClass`.
6. Деструкторы оставлены лишь для совместимости и не выполняют никаких действий.
7. Тип `object` - синоним `System.Object`.
8. Тип `exception` - синоним `System.Exception`.
9. Индексация `string` с 1, директива переключения на индексацию с 0.
10. Процедура `write` выводит любые типы.
11. Структурная эквивалентность типов для процедурных переменных, динамических массивов, типизированных указателей и множеств (в Delphi Object Pascal - именная эквивалентность типов за исключением открытых массивов).
12. Множества на базе произвольных типов (`set of string`).
13. Запрет использования указателей на управляемую память.
14. Процедурные переменные (делегаты) вместо `procedure of object`.
15. С бестиповыми файлами `file` можно работать с помощью процедур `read`, `write`.
16. Массивы массивов отличаются по типу от двумерных массивов (в частности, записи `a[i][j]` и `a[i, j]` неэквивалентны).
17. Перегрузка выполняется без ключевого слова `overload`.
18. Все конструкторы имеют имя `Create`.
19. Автоматическое управление памятью с помощью сборщика мусора (за исключением указателей на неуправляемую память).

Отсутствует

1. Ключевые слова и директивы `packed threadvar inline asm exports library unsafe resourcestring dispinterface in out absolute dynamic local platform requires abstract export message resident assembler safecall automated far near stdcall cdecl published stored contains implements varargs default deprecated package register dispid pascal writeonly` и связанные с ними возможности.
2. Приведение типов для переменных: `Char(b) := 'd'`.
3. Возможность присвоить адрес подпрограммы указателю

`pointer`.

4. Записи с вариантами.
5. Строки `PChar`.
6. Возможность использовать операцию `@` для процедурных переменных.
7. Вариантные типы.
8. Бестиповые параметры (`var a; const b`).
9. Открытые массивы (не путать с динамическими!).
10. Методы, связанные с сообщениями (`message`).
11. Классовые свойства.
12. Вложенные определения классов.
13. Константы-поля классов.

Описание языка PascalABC.NET

Язык программирования **PascalABC.NET** - это язык Pascal нового поколения, включающий в себя все возможности стандартного языка Pascal, расширения языка Delphi Object Pascal, ряд собственных расширений, а также ряд возможностей, обеспечивающих его совместимость с другими .NET-языками. PascalABC.NET является мультипарадигменным языком - на нем можно программировать в различных стилях: структурное программирование, объектно-ориентированное программирование, функциональное программирование.

Кроме того, наличие большого количества стандартных .NET-библиотек классов формирует стиль, ощутимо отличающийся от стиля стандартного Pascal.

Данный раздел содержит **описание языка PascalABC.NET**.

ОСНОВЫ

- [Структура программы](#)
- [Выражения и операции](#)

Типы данных

- [Обзор типов](#)
- [Целые типы](#)
- [Вещественные типы](#)
- [Логический тип](#)
- [Символьный тип](#)
- [Строковый тип](#)
- [Перечислимый и диапазонный типы](#)
- [Статические массивы](#)
- [Динамические массивы](#)
- [Записи](#)
- [Кортежи](#)
- [Множества](#)
- [Файлы](#)
- [Последовательности](#)
- [Указатели](#)
- [Процедурный тип](#)
- [Классы](#)
- [Размерные и ссылочные типы](#)
- [Управление памятью и сборка мусора](#)

Операторы

- [Операторы присваивания](#)
- [Составной оператор](#)
- [Оператор описания переменной](#)
- [Оператор цикла loop](#)
- [Оператор цикла for](#)
- [Оператор цикла foreach](#)
- [Операторы цикла while и repeat](#)
- [Условный оператор if](#)
- [Оператор выбора варианта case](#)
- [Оператор вызова процедуры](#)
- [Оператор try except](#)
- [Оператор try finally](#)
- [Оператор raise](#)
- [Операторы break, continue и exit](#)
- [Оператор yield](#)
- [Оператор yield sequence](#)
- [Оператор goto](#)
- [Оператор lock](#)
- [Оператор with](#)
- [Пустой оператор](#)

Структурное программирование

- [Процедуры и функции](#)
- [Модули](#)
- [Библиотеки dll](#)
- [Документирующие комментарии](#)

Объектно-ориентированное программирование

- [Обзор классов и объектов](#)
- [Наследование](#)
- [Полиморфизм](#)
- [Обобщенные типы](#)
- [Анонимные классы](#)
- [Автоклассы](#)
- [Обработка исключений](#)
- [Методы расширения](#)
- [Интерфейсы](#)
- [Атрибуты](#) (в разработке)

Функциональное программирование

- [Лямбда-выражения](#)
- [Захват переменных](#)
- [Последовательности](#)
- [Методы последовательностей](#)

Стандартные модули

- [Системный модуль PABCSystem](#)

Дополнительные вопросы

- [Область действия идентификатора](#)
- [Директивы компилятора](#)
- [Open MP](#)

Структура программы: обзор

Программа содержит [ключевые слова](#), [идентификаторы](#), [комментарии](#). Ключевые слова используются для выделения синтаксических конструкций и подсвечиваются жирным шрифтом в редакторе. Идентификаторы являются именами объектов программы и не могут совпадать с ключевыми словами.

Программа на языке **PascalABC.NET** имеет следующий вид:

```
program имя программы;  
раздел uses  
раздел описаний  
begin  
  операторы  
end.
```

Первая строка называется **заголовком программы** и не является обязательной.

[Раздел uses](#) состоит из нескольких подряд идущих секций **uses**, каждая из которых начинается с ключевого слова **uses**, за которым следует список имен модулей и пространств имен .NET, перечисляемых через запятую.

Раздел описаний может включать следующие подразделы:

- [раздел описания переменных](#)
- [раздел описания констант](#)
- [раздел описания типов](#)
- [раздел описания меток](#)
- [раздел описания процедур и функций](#)

Данные подразделы следуют друг за другом в произвольном порядке.

Далее следует [блок begin/end](#), внутри которого находятся [операторы](#), отделяемые один от другого символом "точка с запятой". Среди операторов может присутствовать [оператор описания переменной](#), который позволяет описывать переменные внутри блока.

Раздел **uses** и раздел описаний могут отсутствовать.

Например:

```
program MyProgram;  
var  
  a,b: integer;  
  x: real;  
begin  
  readln(a,b);  
  x := a/b;  
  writeln(x);  
end.
```

ИЛИ

```
uses GraphABC;  
begin  
  var x := 100;  
  var y := 100;  
  var r := 50;  
  Circle(x,y,r);  
end.
```

Идентификаторы и ключевые слова

Идентификаторы служат в качестве имен программ, модулей, процедур, функций, типов, переменных и констант. Идентификатором считается любая последовательность латинских букв или цифр, начинающаяся с буквы. Буквой считается также символ подчеркивания "_".

Например, `a1`, `_h`, `b123` - идентификаторы, а `1a` - нет.

С каждым идентификатором связана [область действия идентификатора](#).

Следующие слова являются **ключевыми**, служат для оформления конструкций языка и не могут использоваться как идентификаторы:

`and array as auto begin case class const constructor
destructor div do downto else end event except
extensionmethod file finalization finally for foreach
function goto if implementation in inherited initialization
interface is label lock loop mod nil not of operator or
procedure program property raise record repeat sealed set
sequence shl shr sizeof template then to try type typeof
until uses using var where while with xor`

Ряд слов является **контекстно ключевыми** (они являются ключевыми только в некотором контексте):

`abstract default external forward internal on overload
override params private protected public read reintroduce
unit virtual write`

Контекстно ключевые слова могут использоваться в качестве имен.

Некоторые ключевые слова совпадают с важнейшими именами платформы .NET. Поэтому в **PascalABC.NET** предусмотрена возможность использовать эти имена без конфликтов с ключевыми словами.

Первый способ состоит в использовании квалифицированного имени. Например:

```
var a: System.Array;
```

В этом контексте слово `Array` является именем внутри пространства имен `System`, и конфликта с ключевым словом `array` нет.

Второй способ состоит в использовании специального символа & перед именем. В этом случае имя может совпадать с ключевым словом. Например:

```
uses System;  
var a: &Array;
```

Комментарии

Комментарии - это участки кода, игнорируемые компилятором и используемые программистом для пояснения текста программы.

В **PascalABC.NET** имеется несколько типов комментариев.

Последовательность символов между фигурными скобками { } или символами (* и *) считается комментарием:

```
{ Это  
комментарий }  
  
(* Это  
тоже комментарий *)
```

Комментарием также считается любая последовательность символов после символов // и до конца строки:

```
var Version: integer; // Версия продукта
```

Комментарии разных типов могут быть вложенными:

```
{ Это еще один  
(* комментарий *)}
```

Описание переменных

Переменные могут быть описаны в разделе описаний, а также непосредственно внутри любого блока **begin/end**.

Раздел описания переменных начинается с ключевого слова **var**, после которого следуют элементы описания вида

список имен: тип;

или

имя: тип := выражение;

или

имя: тип = выражение; // для совместимости с Delphi

или

имя := выражение;

Имена в списке перечисляются через запятую. Например:

```
var
  a,b,c: integer;
  d: real := 3.7;
  s := 'PascalABC forever';
  al := new List<integer>;
  p1 := 1;
```

В последних трех случаях тип переменной автоматически определяется по типу правой части.

Переменные могут описываться непосредственно внутри [блока](#). Такие описания называются внутриблочными и представляют собой [оператор описания переменной](#).

Кроме того, переменные-параметры цикла могут описываться в заголовке операторов **for** и **foreach**.

Глобальные переменные инициализируются нулевыми значениями. Для локальных переменных это не гарантируется - их надо инициализировать явно.

Совмещение описания переменных и кортежного присваивания

Кортежное присваивание (распаковку кортежа в переменные) можно совмещать с описанием переменных:

```
var t := (1,2);  
(var a, var b) := (1,2);
```

или

```
var (a,b) := (1,2);
```

Распаковка кортежа в переменные часто используется при возвращении функцией кортежа:

```
function SP(a,b: real) := (a*b,2*(a+b));  
...  
var (S,P) := SP(2,3);
```

Описание констант

Раздел описания именованных констант начинается со служебного слова **const**, после которого следуют элементы описания вида

имя константы = значение;

или

имя константы : тип = значение;

Например:

```
const
  Pi = 3.14;
  Count = 10;
  Name = 'Mike';
  DigitsSet = ['0'..'9'];
  Arr: array [1..5] of integer = (1,3,5,7,9);
  Rec: record name: string; age: integer end = (name:
'Иванов'; age: 23);
  Arr2: array [1..2,1..2] of real = ((1,2),(3,4));
```

Описание меток

Раздел описания меток начинается с зарезервированного слова **label**, после которого следует список меток, перечисляемых через запятую. В качестве меток могут быть использованы идентификаторы и положительные целые числа:

```
label a1,12,777777;
```

Метки используются для перехода в [операторе goto](#).

Описание типов

Раздел описания типов начинается со служебного слова **type**, после которого следуют строки вида

```
имя типа = тип;
```

Например,

```
type arr10 = array [1..10] of integer;  
myint = integer;  
pinteger = ^integer;  
IntFunc = function(x: integer): integer;
```

Обычно описание используется для составных типов (статические массивы, процедурные переменные, записи, классы) чтобы дать имя сложному типу. Если для типа определена [именная эквивалентность типов](#), это единственный способ передать переменные этого типа в подпрограмму.

Описание типов для классов использовать обязательно:

```
type  
A = class  
  i: integer;  
  constructor Create(ii: integer);  
  begin  
    i:=ii;  
  end;  
end;
```

Если описание типа используется просто для того чтобы заменить одно имя на другое, то такие типы называются *синонимами типов*:

```
type  
int = integer;  
double = real;
```

Описания типов могут быть обобщёнными, т.е. включать параметры-типы в угловых скобках после имени типа.

```
type  
Dict<K,V> = Dictionary<K,V>;  
Arr<T> = array of T;
```

Использование такого типа с конкретным параметром-типом называется *инстанцированием* типа:

```
var
  a: Arr<integer>;
  d: Dict<string, integer>;
```

При описании рекурсивных структур данных указатель на тип может фигурировать раньше описания самого типа в определении другого типа:

```
type
  PNode = ^TNode;
  TNode = record
    data: integer;
    next: PNode;
  end;
```

При этом важно, чтобы определения обоих типов находились в одном разделе **type**.

В отличие от Delphi Object Pascal следующее рекурсивное описание верно:

```
type
  TNode = record
    data: integer;
    next: ^TNode;
  end;
```

Отметим, что для ссылочных типов (классов) разрешается описание поля с типом, совпадающим с типом текущего класса:

```
type
  Node = class
    data: integer;
    next: Node;
  end;
```

Область действия идентификатора

Любой используемый в программе идентификатор должен быть предварительно описан. Идентификаторы описываются в разделе описаний. Идентификаторы для переменных могут также описываться внутри блока.

Основная программа, подпрограмма, [блок](#), [модуль](#), [класс](#) образуют так называемое **пространство имен** - область в программе, в которой имя должно иметь единственное описание. Таким образом, в одном пространстве имен не может быть описано двух одинаковых имен (исключение составляют [перегруженные имена подпрограмм](#)). Кроме того, в сборках .NET имеются явные определения пространств имен.

Область действия идентификатора (т.е. место, где он может быть использован) простирается от момента описания до конца блока, в котором он описан. Область действия глобального идентификатора, описанного в модуле, простирается на весь модуль, а также на основную программу, к которой данный модуль подключен в разделе **uses**.

Кроме этого, имеются переменные, определенные в блоке и связанные с некоторыми конструкциями (**for**, **foreach**). В этом случае действие переменной **i** простирается до конца соответствующей конструкции. Так, следующий код корректен:

```
var a: array of integer := (3,5,7); for i: integer := 1
to 9 do
    write(a[i]);
foreach i: integer in a do
    write(i);
```

Идентификатор с тем же именем, определенный во вложенном пространстве имен, *скрывает* идентификатор, определенный во внешнем пространстве имен. Например, в коде

```
var i: integer;
procedure p;
var i: integer;
begin
```

```
    i := 5;  
end;
```

значение 5 будет присвоено переменной *i*, описанной в процедуре *p*; внутри же процедуры *p* сослаться на глобальную переменную *i* невозможно.

Переменные, описанные внутри блока, не могут иметь те же имена, что и переменные из раздела описаний этого блока. Например, следующая программа ошибочна:

```
var i: integer;  
begin  
    var i: integer; // ошибка  
end.
```

В производных классах, напротив, можно определять члены с теми же именами, что и в базовых классах, при этом их имена скрывают соответствующие имена в базовых классах. Для обращения к одноименному члену базового класса из метода производного класса используется ключевое слово [inherited](#):

```
type  
  A=class  
    i: integer;  
    procedure p;  
    begin  
      i := 5;  
    end;  
  end;  
  B=class(A)  
    i: integer;  
    procedure p;  
    begin  
      i := 5;  
      inherited p;  
    end;  
  end;
```

Алгоритм поиска имени в классе следующий: вначале имя ищется в текущем классе, затем в его базовых классах, а если не найдено, то в глобальной области видимости.

Алгоритм поиска имени в глобальной области видимости

при наличии нескольких подключенных модулей следующий:
вначале имя ищется в текущем модуле, затем, если не найдено, по цепочке подключенных модулей в порядке справа налево.

Например, в программе

```
uses unit1,unit2;  
begin  
  id := 2;  
end.
```

описание переменной `id` будет искааться вначале в основной программе, затем в модуле `unit2`, затем в модуле `unit1`. При этом в разных модулях могут быть описаны разные переменные `id`. Данная ситуация означает, что `unit1` образует внешнее пространство имен, пространство имен `unit2` в него непосредственно вложено, а пространство имен основной программы вложено в `unit2`.

Если в последнем примере оба модуля - `unit1` и `unit2` - определяют переменные `id`, то рекомендуется уточнять имя переменной именем модуля, используя конструкцию *ИмяМодуля.Имя*:

```
uses unit1,unit2;  
begin  
  unit1.id := 2;  
end.
```

Обзор типов

Типы в **PascalABC.NET** подразделяются на простые, структурированные, типы [указателей](#), [процедурные](#) типы, [последовательности](#) и [классы](#).

К **простым** относятся [целые](#) и [вещественные](#) типы, [логический](#), [символьный](#), [перечислимый](#) и [диапазонный](#) тип.

Тип данных называется **структурированным**, если в одной переменной этого типа может содержаться множество значений.

К структурированным типам относятся [массивы](#), [строки](#), [записи](#), [кортежи](#), [множества](#), [файлы](#) и [классы](#).

Особым типом данных является [последовательность](#), которая хранит по-существу алгоритм получения данных последовательности один за другим.

Все простые типы, кроме вещественного, называются **порядковыми**. Только значения этих типов могут быть индексами статических массивов и параметрами цикла **for**. Кроме того, для порядковых типов используются функции **Ord**, **Pred** и **Succ**, а также процедуры **Inc** и **Dec**.

Все типы, кроме типов указателей, являются производными от типа **Object**. Каждый тип в **PascalABC.NET** имеет [отображение на тип .NET](#). Тип указателя принадлежит к неуправляемому коду и моделируется типом **void***.

Все типы в **PascalABC.NET** подразделяются на две большие группы: [размерные](#) и [ссылочные](#).

Размерные и ссылочные типы

Все типы в **PascalABC.NET** подразделяются на две большие группы: **размерные** и **ссылочные**. К размерным относятся все простые типы, указатели, записи, статические массивы, множества и строки. К ссылочным типам относятся классы, динамические массивы, файлы и процедурный тип.

Размерные типы более эффективны при вычислениях: они занимают меньше памяти и операции, выполняемые над небольшими размерными типами, максимально эффективны. Ссылочные типы обладают большей гибкостью: память под них выделяется динамически в процессе работы программы и освобождается автоматически, когда объект ссылочного типа перестаёт использоваться.

Выделение памяти

Память под переменную размерного типа распределяется на программном стеке в момент её описания. При этом переменная размерного типа хранит значение этого типа.

```
var i: integer; // здесь под i выделяется память  
i := 5;
```

Переменная ссылочного типа представляет собой ссылку на объект некоторого класса в динамической памяти. Если она не инициализирована, то хранит специальное значение `nil` (нулевая ссылка). Для инициализации ссылочных переменных используется вызов конструктора соответствующего класса:

```
type Person = auto class  
    name: string;  
    age: integer;  
end;  
var p: Person; // p хранит значение nil, память под  
объект не выделена  
p := new Person('Иванов',20); // конструктор выделяет  
память под объект
```

Присваивание

При присваивании переменных размерного типа копируются значения этого типа. Если размерный тип имеет большой размер, эта операция может выполняться долго. Например:

```
var a, a1: array [1..1000000] of integer;  
a1 := a; // копируются все 1000000 элементов
```

При присваивании переменных ссылочного типа осуществляется присваивание ссылок, в итоге после присваивания обе ссылки ссылаются на один объект в динамической памяти:

```
var p1: Person;  
p1 := p; // копируется ссылка
```

Сравнение на равенство

Сравнение на равенство объектов размерного типа сравнивает их значения. В частности, две переменные типа запись равны если равны все поля этих записей.

```
type PersonRec = record
  name: string;
  age: integer;
end;
var p, p1: PersonRec;
p.name := 'Иванов'; p.age := 20;
p1.name := 'Иванов'; p1.age := 20;
writeln(p1=p); // True
```

При сравнении на равенство переменных ссылочного типа проверяется, что они ссылаются на один и тот же объект.

```
var p := new Person('Иванов', 20);
var p1 := new Person('Иванов', 20);
writeln(p1=p); // False
```

Управление памятью

Размерные типы распределяются на программном стеке, поэтому не нуждаются в специальном управлении памятью. Под глобальные размерные переменные память распределена всё время работы программы. Под локальные размерные переменные память выделяется в момент вызова подпрограммы, а освобождается в момент завершения работы этой подпрограммы.

Управление памятью для ссылочных типов осуществляется автоматически сборщиком мусора. Сборщик мусора запускается в неопределенный момент времени когда управляемой памяти перестаёт хватать. Он возвращает в пул неиспользуемой памяти те объекты, на которые больше никто не ссылается, после чего дефрагментирует оставшуюся память, в результате чего динамическая память всегда дефрагментирована и ее выделение при вызове конструктора происходит практически мгновенно.

Передача в подпрограммы

При передаче размерных типов по значению происходит копирование значения фактического параметра в переменную-формальный параметр. Если размерный тип имеет большой размер, это может занимать продолжительное время, поэтому размерный тип в этом случае передаётся по ссылке на константу:

```
type Arr = array [1..100] of integer;  
  
procedure PrintArray(const a: Arr; n: integer);  
begin  
  for var i:=1 to n do  
    Print(a[i])  
end;
```

Ссылочные типы передаются в подпрограмму, как правило, по значению. При передаче таких параметров происходит копирование ссылки, в результате формальный и фактический параметр будут ссылаться на один объект.

```
procedure Change666(a: array of integer);  
begin  
  a[0] := 666;  
end;
```

При этом в результате изменения формального параметра внутри подпрограммы меняется и содержимое соответствующего фактического параметра при вызове подпрограммы.

Целые типы

Ниже приводится таблица целых типов, содержащая также их размер и диапазон допустимых значений.

Тип	Размер, байт	Диапазон значений
<code>shortint</code>	1	-128..127
<code>smallint</code>	2	-32768..32767
<code>integer</code> , <code>longint</code>	4	-2147483648..2147483647
<code>int64</code>	8	-9223372036854775808..9223372036854775807
<code>byte</code>	1	0..255
<code>word</code>	2	0..65535
<code>longword</code> , <code>cardinal</code>	4	0..4294967295
<code>uint64</code>	8	0..18446744073709551615
<code>BigInteger</code>	переменный	неограниченный

Типы `integer` и `longint`, а также `longword` и `cardinal` являются синонимами.

Максимальные значения для каждого целого типа определены как внешние [стандартные константы](#): `MaxInt64`, `MaxInt`, `MaxSmallInt`, `MaxShortInt`, `MaxUInt64`, `MaxLongWord`, `MaxWord`, `MaxByte`.

Для каждого целого типа `T` кроме `BigInteger` определены следующие константы как статические члены:

`T.MinValue` - константа, представляющая минимальное значение типа `T`;

`T.MaxValue` - константа, представляющая максимальное значение типа `T`;

Для каждого целого типа `T` определены статические функции:

`T.Parse(s)` - функция, конвертирующая строковое представление числа в значение типа `T`. Если преобразование

невозможно, то генерируется исключение;

`T.TryParse(s, res)` - функция, конвертирующая строковое представление числа в значение типа `T` и записывающая его в переменную `res`. Если преобразование возможно, то возвращается значение `True`, в противном случае - `False`.

Кроме того, для `T` определена экземплярная функция `ToString`, возвращающая строковое представление переменной данного типа.

Константы целого типа могут представляться как в десятичной, так и в шестнадцатеричной форме, перед шестнадцатеричной константой ставится знак `$`:

`25` `3456` `$FFFF`

Вещественные типы

Ниже приводится таблица вещественных типов, содержащая их размер, количество значащих цифр и диапазон допустимых значений:

Тип	Размер, байт	Количество значащих цифр	Диапазон значений
<code>real</code>	8	15-16	$-1.8 \cdot 10^{308} .. 1.8 \cdot 10^{308}$
<code>double</code>	8	15-16	$-1.8 \cdot 10^{308} .. 1.8 \cdot 10^{308}$
<code>single</code>	4	7-8	$-3.4 \cdot 10^{38} .. 3.4 \cdot 10^{38}$
<code>decimal</code>	16	28-29	$-79228162514264337593543950335 .. 79228162514264337593543950335$

Типы `real` и `double` являются синонимами. Самое маленькое положительное число типа `real` приблизительно равно $5.0 \cdot 10^{-324}$, для типа `single` оно составляет приблизительно $1.4 \cdot 10^{-45}$.

Максимальные значения для каждого вещественного типа определены как внешние [стандартные константы](#): `MaxReal`, `MaxDouble` и `MaxSingle`.

Для каждого вещественного типа `R` кроме `decimal` определены также следующие константы как статические члены класса:

`R.MinValue` - константа, представляющая минимальное значение типа `R`;

`R.MaxValue` - константа, представляющая максимальное значение типа `R`;

`R.Epsilon` - константа, представляющая самое маленькое положительное число типа `R`;

`R.NaN` - константа, представляющая не число (возникает, например, при делении $0/0$);

`R.NegativeInfinity` - константа, представляющая отрицательную бесконечность (возникает, например, при

деления $-2/0$);

`R.PositiveInfinity` - константа, представляющая положительную бесконечность (возникает, например, при делении $2/0$).

Для каждого вещественного типа `R` кроме `decimal` определены следующие статические функции:

`R.IsNaN(r)` - возвращает `True`, если в `r` хранится значение `R.NaN`, и `False` в противном случае;

`R.IsInfinity(r)` - возвращает `True`, если в `r` хранится значение `R.PositiveInfinity` или `R.NegativeInfinity`, и `False` в противном случае;

`R.IsPositiveInfinity(r)` - возвращает `True`, если в `r` хранится значение `R.PositiveInfinity`, и `False` в противном случае;

`R.IsNegativeInfinity(r)` - возвращает `True`, если в `r` хранится значение `R.NegativeInfinity`, и `False` в противном случае;

Для каждого вещественного типа `R` определены следующие статические функции:

`R.Parse(s)` - функция, конвертирующая строковое представление числа в значение типа `R`. Если преобразование невозможно, то генерируется исключение;

`R.TryParse(s, res)` функция, конвертирующая строковое представление числа в значение типа `R` и записывающая его в переменную `res`. Если преобразование возможно, то возвращается значение `True`, в противном случае - `False`.

Кроме того, определена экземплярная функция `ToString`, возвращающая строковое представление переменной типа `R`.

Вещественные константы можно записывать как в форме с плавающей точкой, так и в экспоненциальной форме:

`1.7` `0.013` `2.5e3` (2500) `1.4e-1` (0.14)

Логический тип

Значения логического типа `boolean` занимают 1 байт и принимают одно из двух значений, задаваемых предопределенными константами `True` (истина) и `False` (ложь).

Для логического типа определены статические методы:

`boolean.Parse(s)` - функция, конвертирующая строковое представление числа в значение типа `boolean`. Если преобразование невозможно, то генерируется исключение;

`boolean.TryParse(s, res)` - функция, конвертирующая строковое представление числа в значение типа `boolean` и записывающая его в переменную `res`. Если преобразование возможно, то возвращается значение `True`, в противном случае - `False`.

Кроме этого, определена экземплярная функция `ToString`, возвращающая строковое представление переменной типа `boolean`.

Логический тип является порядковым. В частности, `False < True`, `Ord(False)=0`, `Ord(True)=1`.

Символьный тип

Символьный тип `char` занимает 2 байта и хранит Unicode-символ. Символы реализуются типом `System.Char` платформы .NET.

Операция `+` для символов означает конкатенацию (слияние) строк. Например: `'a'+'b' = 'ab'`. Как и для строк, если к символу прибавить число, то число предварительно преобразуется к строковому представлению:

```
var s: string := ' '+15; // s = ' 15'  
var s1: string := 15+' '; // s = '15 '
```

Над символами определены операции сравнения `<` `>` `<=` `>=` `=` `<>`, которые сравнивают коды символов:

```
'a'<'b' // True  
'2'<'3' // True
```

Для преобразования между символами и их кодами в кодировке Windows (CP1251) используются стандартные функции `Chr` и `Ord`:

`Chr(n)` - функция, возвращающая символ с кодом `n` в кодировке Windows;
`Ord(c)` - функция, возвращающая значение типа `byte`, представляющее собой код символа `c` в кодировке Windows.

Для преобразования между символами и их кодами в кодировке Unicode используются стандартные функции `ChrUnicode` и `OrdUnicode`:

`ChrUnicode(w)` - возвращает символ с кодом `w` в кодировке Unicode;
`OrdUnicode(c)` - возвращает значение типа `word`, представляющее собой код символа `c` в кодировке Unicode.

Кроме того, выражение `#число` возвращает Unicode-символ с кодом `число` (число должно находиться в диапазоне от 0 до 65535).

Аналогичную роль играют явные преобразования типов:

`char(w)` возвращает символ с кодом `w` в кодировке Unicode;
`word(c)` возвращает код символа `c` в кодировке Unicode.

Стандартные подпрограммы работы с символами.

Статические методы типа char.

Перечислимый и диапазонный типы

Перечислимый тип определяется упорядоченным набором идентификаторов.

```
type typeName = (value1, value2, ..., valuen);
```

Значения перечислимого типа занимают 4 байта. Каждое значение `value` представляет собой константу типа `typeName`, попадающую в текущее пространство имен.

Например:

```
type
  Season = (Winter, Spring, Summer, Autumn);
  DayOfWeek = (Mon, Tue, Wed, Thi, Thr, Sat, Sun);
```

К константе перечислимого типа можно обращаться непосредственно по имени, а можно использовать запись `typeName.value`, в которой имя константы уточняется именем перечислимого типа, к которому она принадлежит:

```
var a: DayOfWeek;
a := Mon;
a := DayOfWeek.Wed;
```

Значения перечислимого типа можно сравнивать на `<`:

```
DayOfWeek.Wed < DayOfWeek.Sat
```

Для значений перечислимого типа можно использовать функции `Ord`, `Pred` и `Succ`, а также процедуры `Inc` и `Dec`. Функция `Ord` возвращает порядковый номер значения в списке констант соответствующего перечислимого типа, нумерация при этом начинается с нуля.

Для перечислимого типа определена экземплярная функция `ToString`, возвращающая строковое представление переменной перечислимого типа. При выводе значения перечислимого типа с помощью процедуры `write` также выводится строковое представление значения перечислимого типа.

Например:

```
type Season = (Winter, Spring, Summer, Autumn);
var s: Season;
begin
```

```
s := Summer;
writeln(s.ToString); // Summer
writeln(s); // Summer
end.
```

Диапазонный тип представляет собой подмножество значений целого, символьного или перечислимого типа и описывается в виде $a..b$, где a - нижняя, b - верхняя граница интервального типа, $a < b$:

```
var
  intI: 0..10;
  intC: 'a'..'z';
  intE: Mon..Thr;
```

Тип, на основе которого строится диапазонный тип, называется *базовым* для этого диапазонного типа. Значения диапазонного типа занимают в памяти столько же, сколько и значения соответствующего базового типа.

Строковый тип

Строки имеют тип `string`, состоят из набора последовательно расположенных символов `char` и используются для представления текста.

Строки могут иметь произвольную длину. К символам в строке можно обращаться, используя индекс: `s[i]` обозначает *i*-тый символ в строке, нумерация начинается с единицы. Если индекс *i* выходит за пределы длины строки, то генерируется исключение.

Над строками определены операции сравнения: `<` `>` `<=` `>=` `=` `<>`.

Сравнение строк на неравенство осуществляется

лексикографически: `s1 < s2` если для первого несовпадающего символа с номером *i* `s1[i]<s2[i]` или все символы строк совпадают, но `s1` короче `s2`.

Операция `+` для строк означает конкатенацию (слияние) строк.

Например: `'Петя'+ 'Маша' = 'ПетяМаша'`.

Расширенный оператор присваивания `+=` для строк добавляет в конец строки - левого операнда строку - правый операнд. Например:

```
var s: string := 'Петя';  
s += 'Маша'; // s = 'ПетяМаша'
```

Строка может складываться с числом, при этом число предварительно преобразуется к строковому представлению:

```
s := 'Ширина: '+15; // s = 'Ширина: 15'  
s := 20.5+''; // s = '20.5'  
s += 1; // s = '20.51'
```

Над строками и целыми определена операция `*`: `s*n` и `n*s` означает строку, образованную из строки `s`, повторенной *n* раз:

```
s := '*'*10; // s = '*****'  
s := 5*'ab' // s = 'ababababab'  
s := 'd'; s *= 3; // s = 'ddd'
```

Строки реализуются типом `System.String` платформы .NET и представляют собой ссылочный тип. Таким образом, все операции над строками унаследованы от типа `System.String`. Однако, в отличие от .NET - строк, строки в **PascalABC.NET** изменяемы.

Например, можно изменить `s[i]` (в .NET нельзя). Более того, строки `string` в **PascalABC.NET** ведут себя как размерные: после

```
var s2 := 'Hello';  
var s1 := s2;  
s1[2] := 'a';
```

строка `s2` не изменится. Аналогично при передаче строки по значению в подпрограмму создается копия строки, т.е. обеспечивается поведение, характерное для Delphi Object Pascal, а не для .NET.

Однако, строке можно присвоить `nil`, что необходимо для работы с NET-кодом.

Кроме того, в **PascalABC.NET** реализованы *размерные строки*. Для их описания используется тип `string[n]`, где `n` - константа целого типа, указывающая длину строки. Размерные строки, в отличие от обычных, можно использовать как компоненты [типизированных файлов](#). Для совместимости с Delphi Object Pascal в стандартном модуле описан тип `shortstring=string[255]`.

[Стандартные подпрограммы работы со строками.](#)

[Члены класса string.](#)

Методы типа `string`

Тип `string` в PascalABC.NET является классом и содержит ряд свойств, статических и экземплярных методов, а также методов расширения.

В методах класса `string` считается, что строки индексируются с нуля. Кроме того, ни один метод не меняет строку, т.к. строки в .NET являются неизменяемыми.

Свойства класса String

Свойство	Описание
<code>s[i]</code>	Индексное свойство. Возвращает или позволяет изменить <i>i</i> -тый символ строки <i>s</i> . Строки в PascalABC.NET индексируются от 1.
<code>Length: integer</code>	Возвращает длину строки

Статические методы класса String

Метод	Описание
<code>String.Compare(s1,s2: string): integer</code>	Сравнивает строки s1 и s2. Возвращает число <0 если s1<s2, =0 если s1=s2 и >0 если s1>s2
<code>String.Compare(s1,s2: string; ignorecase: boolean): integer</code>	То же. Если ignorecase=True, то строки сравниваются без учета регистра букв
<code>String.Format(fmtstr: string, params arr: array of object): string;</code>	Форматирует параметры arr согласно форматной строке fmtstr
<code>String.Join(ss: array of string; delim: string): string</code>	Возвращает строку, полученную слиянием строк ss с использованием delim в качестве разделителя

Экземплярные методы класса String

Отметим, что все экземплярные методы не меняют строку, как это может показаться на первый взгляд, а при необходимости возвращают измененную строку. Кроме того, считается, что символы в строке индексируются с нуля.

Метод	Описание
<code>Contains(s: string): boolean</code>	Возвращает True, если текущая строка содержит s, и False в противном случае
<code>EndsWith(s: string): boolean</code>	Возвращает True, если текущая строка заканчивается на s, и False в противном случае
<code>IndexOf(s: string): integer</code>	Возвращает индекс первого вхождения подстроки s в текущую строку или -1 если подстрока не найдена
<code>IndexOf(s: string; start, count: integer): integer</code>	Возвращает индекс первого вхождения подстроки s в текущую строку или -1 если подстрока не найдена. Поиск начинается с символа с номером start и распространяется на следующие count символов
<code>IndexOfAny(cc: array of char): integer</code>	Возвращает индекс первого вхождения любого символа из массива cc
<code>Insert(from: integer; s: string): string</code>	Возвращает строку, полученную из исходной строки вставкой

	подстроки s в позицию from
<code>LastIndexOf(s: string): integer</code>	Возвращает индекс последнего вхождения подстроки s в текущую строку
<code>LastIndexOf(s: string; start, count: integer): integer</code>	Возвращает индекс последнего вхождения подстроки s в текущую строку или -1 если подстрока не найдена. Поиск начинается с символа с номером start и распространяется на следующие count символов
<code>LastIndexOfAny(a: array of char): integer</code>	Возвращает индекс последнего вхождения любого символа из массива ss
<code>PadLeft(n: integer): string</code>	Возвращает строку, полученную из исходной строки выравниванием по правому краю с заполнением пробелами слева до длины n
<code>PadRight(n: integer): string</code>	Возвращает строку, полученную из исходной строки выравниванием по левому краю с заполнением пробелами справа до длины n
<code>Remove(from, len: integer): string</code>	Возвращает строку, полученную из исходной строки удалением len символов с позиции from

<code>Replace(s1,s2: string): string</code>	Возвращает строку, полученную из исходной строки заменой всех вхождений подстроки s1 на строку s2
<code>Split(params delim: array of char): array of string</code>	Возвращает массив строк, полученный расщеплением исходной строки на слова, при этом в качестве разделителей используется любой из символов delim (по умолчанию - пробел)
<code>StartsWith(s: string): boolean</code>	Возвращает True, если текущая строка начинается на s, и False в противном случае
<code>Substring(from,len: integer): string</code>	Возвращает подстроку исходной строки с позиции from длины len
<code>ToCharArray: array of char</code>	Возвращает динамический массив символов исходной строки
<code>ToLower: string</code>	Возвращает строку, приведенную к нижнему регистру
<code>ToUpper: string</code>	Возвращает строку, приведенную к верхнему регистру
<code>Trim: string</code>	Возвращает строку, полученную из исходной удалением лидирующих и завершающих пробелов
<code>TrimEnd(params cc: array of char):</code>	Возвращает строку, полученную из исходной

string	удалением завершающих символов из массива сс
TrimStart(params сс: array of char): string	Возвращает строку, полученную из исходной удалением лидирующих символов из массива сс

Методы расширения класса String

Некоторые методы расширения - стандартные для .NET, некоторые реализованы только в PascalABC.NET.

Метод	Описание
<code>Inverse: string</code>	Возвращает инверсию строки
<code>Print</code>	Выводит буквы строки, разделенные пробелом
<code>Println</code>	Выводит буквы строки, разделенные пробелом, и осуществляет переход на новую строку
<code>ReadInteger(var from: integer): integer</code>	Считывает из строки целое число с позиции <code>from</code> и возвращает его. Позиция <code>from</code> при этом увеличивается на считанный элемент
<code>ReadReal(var from: integer): real</code>	Считывает из строки вещественное число с позиции <code>from</code> и возвращает его. Позиция <code>from</code> при этом увеличивается на считанный элемент
<code>ReadWord(var from: integer): string</code>	Считывает из строки слово до пробела или до конца строки с позиции <code>from</code> и возвращает его. Позиция <code>from</code> при этом увеличивается на считанный элемент
<code>ToInteger: integer</code>	Преобразует строку к целому и возвращает его. Если это невозможно, генерируется исключение

<p>ToIntegers: array of integer</p>	<p>В строке должны храниться целые, разделенные пробелами. Возвращается массив целых. Если это невозможно, генерируется исключение</p>
<p>ToReal: real</p>	<p>Преобразует строку к вещественному и возвращает его. Если это невозможно, генерируется исключение</p>
<p>ToReals: array of real</p>	<p>В строке должны храниться вещественные, разделенные пробелами. Возвращается массив вещественных. Если это невозможно, генерируется исключение</p>
<p>ToWords(params delim: array of char): array of string</p>	<p>Возвращает массив строк, полученный расщеплением исходной строки на слова, при этом в качестве разделителей используется любой из символов delim (по умолчанию - пробел). В отличие от s.Split не включает в итоговый массив пустые строки. В частности, это означает, что слова могут быть разделены несколькими разделителями delim</p>

Массивы

Массив представляет собой набор элементов одного типа, каждый из которых имеет свой номер, называемый *индексом* (индексов может быть несколько, тогда массив называется *многомерным*).

Массивы в **PascalABC.NET** делятся на [статические](#) и [динамические](#).

При выходе за границы изменения индекса в **PascalABC.NET** всегда генерируется исключение.

Статические массивы

Описание статического массива

Статические массивы в отличие от [динамических](#) задают свой размер непосредственно в типе. Память под такие массивы выделяется сразу при описании.

Тип статического массива конструируется следующим образом:

array [*тип индекса1*, ..., *тип индексаN*] **of** *базовый тип*

Тип индекса должен быть [порядковым](#). Обычно тип индекса является [диапазонным](#) и представляется в виде *a*..*b*, где *a* и *b* - константные выражения целого, символьного или перечислимого типа. Например:

```
type   MyEnum = (w1,w2,w3,w4,w5);
        Arr = array [1..10] of integer;
var
    a1,a2: Arr;
    b: array ['a'..'z',w2..w4] of string;
    c: array [1..3] of array [1..4] of real;
```

Инициализация статического массива

При описании можно также задавать инициализацию массива значениями:

```
var
  a: Arr := (1,2,3,4,5,6,7,8,9,0);
  cc: array [1..3,1..4] of real := ((1,2,3,4),
(5,6,7,8), (9,0,1,2));
```

Присваивание статического массива

Статические массивы одного типа можно присваивать друг другу, при этом будет производиться копирование содержимого одного массива в другой:

```
a1 := a2;
```

Вывод статического массива

Процедура write выводит статический массив, заключая элементы в квадратные скобки и разделяя их запятыми:

```
var a: Arr := (1,2,3,4,5,6,7,8,9,0);  
var m := array [1..3,1..3] of integer := ((1,2,3),  
(4,5,6),(7,8,9));  
writeln(a); // [1,2,3,4,5]  
writeln(m); // [[1,2,3],[4,5,6],[7,8,9]]
```

Передача статического массива в подпрограмму

При передаче статического массива в подпрограмму по значению также производится копирование содержимого массива - фактического параметра в массив - формальный параметр:

```
procedure p(a: Arr); // передавать статический массив
по значению - плохо!
...
p(a1);
```

Это крайне расточительно, поэтому статические массивы рекомендуется передавать [по ссылке](#). Если массив не меняется внутри подпрограммы, то его следует передавать как ссылку на константу, если меняется - как ссылку на переменную:

```
type Arr = array [2..10] of integer;

procedure Squares(var a: Arr);
begin
  for var i:= Low(a) to High(a) do
    a[i] := Sqr(a[i]);
end;

procedure PrintArray(const a: Arr);
begin
  for var i:= Low(a) to High(a) do
    Print(a[i])
end;

var a: Arr := (1,3,5,7,9,2,4,6,8);

begin
  Squares(a);
  PrintArray(a);
end.
```

Для доступа к нижней и верхней границам размерности одномерного массива используются функции Low и High.

Динамические массивы

Описание динамического массива

Тип динамического массива конструируется следующим образом:

array of *тип элементов* (одномерный массив)

array [,] of *тип элементов* (двумерный массив)

и т.д.

Переменная типа динамический массив представляет собой ссылку. Поэтому динамический массив нуждается в инициализации (выделении памяти под элементы).

Выделение памяти под динамический массив

Для выделения памяти под динамический массив используется два способа. Первый способ использует операцию `new` в стиле вызова конструктора класса:

```
var
  a: array of integer;
  b: array [,] of real;
begin
  a := new integer[5];
  b := new real[4,3];
end.
```

Данный способ хорош тем, что позволяет совместить описание массива и выделение под него памяти:

```
var
  a: array of integer := new integer[5];
  b: array [,] of real := new real[4,3];
```

Описание типа можно при этом опускать - тип автовыводится:

```
var
  a := new integer[5];
  b := new real[4,3];
```

Второй способ выделения памяти под динамический массив использует стандартную процедуру `SetLength`:

```
SetLength(a, 10);
SetLength(b, 5, 3);
```

Элементы массива при этом заполняются значениями по умолчанию.

Процедура `SetLength` обладает тем преимуществом, что при ее повторном вызове старое содержимое массива сохраняется.

Инициализация динамического массива

Можно инициализировать динамический массив при выделении под него память операцией `new`:

```
a := new integer[3](1,2,3);  
b := new real[4,3] ((1,2,3),(4,5,6),(7,8,9),(0,1,2));
```

Инициализацию динамического массива в момент описания можно проводить в сокращенной форме:

```
var  
  a: array of integer := (1,2,3);  
  b: array [,] of real := ((1,2,3),(4,5,6),(7,8,9),  
  (0,1,2));  
  c: array of array of integer := ((1,2,3),(4,5),  
  (6,7,8));
```

При этом происходит выделение памяти под указанное справа количество элементов.

Инициализация одномерного массива проще всего осуществляется стандартными функциями `Seq...`, которые выделяют память нужного размера и заполняют массив указанными значениями:

```
var a := Arr(1,3,5,7,8); // array of  
integer  
var s := Arr('Иванов','Петров','Сидоров'); // array of  
string  
var b := ArrFill(777,5); // b =  
[777,777,777,777,777]  
var r := ArrRandom(10); //  
заполнение 10 случайными целыми в диапазоне от 0 до 99
```

В таком же стиле можно инициализировать массивы массивов:

```
var a := Arr(Arr(1,3,5),Arr(7,8),Arr(5,6)); // array of  
array of integer
```

Длина динамического массива

Динамический массив помнит свою длину (n-мерный динамический массив помнит длину по каждой размерности). Длина массива (количество элементов в нем) возвращается стандартной функцией `Length` или свойством `Length`:

```
l := Length(a);  
l := a.Length;
```

Для многомерных массивов длина по каждой размерности возвращается стандартной функцией `Length` с двумя параметрами или методом `GetLength(i)`:

```
l := Length(a, 0);  
l := a.GetLength(0);
```

Ввод динамического массива

После выделения памяти ввод динамического массива можно осуществлять традиционно в цикле:

```
for var i:=0 to a.Length-1 do  
    read(a[i]);
```

Ввод динамического массива можно осуществлять с помощью стандартной функции ReadSeqInteger:

```
var a := ReadSeqInteger(10);
```

При этом под динамический массив выделяется память нужного размера.

Вывод динамического массива

Процедура `write` выводит динамический массив, заключая элементы в квадратные скобки и разделяя их запятыми:

```
var a := Arr(1,3,5,7,9);  
writeln(a); // [1,3,5,7,9]
```

n-мерный динамический массив выводится так, что каждая размерность заключается в квадратные скобки:

```
var m := new integer[3,3] ((1,2,3),(4,5,6),(7,8,9));  
writeln(m); // [[1,2,3],[4,5,6],[7,8,9]]
```

Динамический массив можно выводить также методом расширения `Print` или `Println`:

```
a.Println;
```

При этом элементы по умолчанию разделяются пробелами, но можно это изменить, задав параметр `Print`, являющийся разделителем элементов. Например:

```
a.Print(NewLine);
```

выводит каждый элемент на отдельной строке.

Массивы массивов

Если объявлен массив массивов

```
var c: array of array of integer;
```

то его инициализацию можно провести только с помощью `SetLength`:

```
SetLength(c,5);  
for i := 0 to 4 do  
    SetLength(c[i],3);
```

Для инициализации такого массива с помощью `new` следует ввести имя типа для `array of integer`:

```
type IntArray = array of integer;  
var c: array of IntArray;  
...  
c := new IntArray[5];  
for i := 0 to 4 do  
    c[i] := new integer[3];
```

Инициализацию массива массивов можно также проводить в сокращенной форме:

```
var  
    c: array of array of integer := ((1,2,3),(4,5),  
    (6,7,8));
```

Присваивание динамических массивов

Динамические массивы одного типа можно присваивать друг другу, при этом обе переменные-ссылки будут указывать на одну память:

```
var a1: array of integer;  
var a2: array of integer;  
a1 := a2;
```

Следует обратить внимание, что для динамических массивов принята [структурная эквивалентность типов](#): можно присваивать друг другу и передавать в качестве параметров подпрограмм динамические массивы, совпадающие по структуре.

Чтобы одному динамическому массиву присвоить копию другого массива, следует воспользоваться стандартной функцией `Copy`:

```
a1 := Copy(a2);
```

Передача динамического массива в подпрограмму

Динамический массив обычно передается в подпрограмму по значению, т.к. сама переменная уже является ссылкой:

```
procedure Squares(a: array of integer);  
begin  
    for var i:=0 to a.Length-1 do  
        a[i] := Sqr(a[i]);  
end;  
  
begin  
    var a := Arr(1,3,5,7,9);  
    Squares(a);  
end.
```

Динамический массив передается по ссылке только в одном случае: если он создается или пересоздается внутри подпрограммы. В частности, это необходимо делать если для динамического массива внутри подпрограммы вызывается SetLength:

```
procedure Add(var a: array of integer; x: integer);  
begin  
    SetLength(a, a.Length+1);  
    a[a.Length-1] := x;  
end;  
  
begin  
    var a := Arr(1,3,5,7,9);  
    Add(a, 666);  
    writeln(a);  
end.
```

[Подпрограммы для работы с динамическими массивами](#)

[Подпрограммы для генерации динамических массивов](#)

[Методы расширения для последовательностей](#)

[Методы расширения для динамических массивов](#)

Записи

Запись представляет собой набор элементов разных типов, каждый из которых имеет свое имя и называется полем записи.

Описание записей

Тип записи в классическом языке Паскаль описывается следующим образом:

```
record   описания полей  
end
```

где описания полей имеет такой же вид, что и [раздел описания переменных](#) без ключевого слова **var**.

Например:

```
type  
  Person = record  
    Name: string;  
    Age: integer;  
  end;
```

Переменные типа запись

Переменные типа запись хранят в непрерывном блоке памяти значения всех полей записи.

Для доступа к полям записей используется точечная нотация:

```
var p: Person;  
begin  
  p.Name := 'Иванов';  
  p.Age := 20;  
  writeln(p); // (Иванов,20)  
end.
```

По умолчанию процедура `write` выводит содержимое всех полей записи в круглых скобках через запятую.

Методы и модификаторы доступа для записей

В **PascalABC.NET** внутри записей допустимо определять [методы](#) и [свойства](#), а также использовать [модификаторы доступа](#). Таким образом, описание записи в **PascalABC.NET** имеет вид:

```
record  
    секция1  
    секция2  
    ...  
end
```

Каждая секция имеет вид:

```
модификатор доступа  
описания полей  
объявления или описания методов и описания свойств
```

[Модификатор доступа](#) в первой секции может отсутствовать, в этом случае подразумевается модификатор **public** (все члены открыты).

Например:

```
type  
    Person = record  
        private  
            Name: string;  
            Age: integer;  
        public  
            constructor Create(Name: string; Age: integer);  
            begin  
                Self.Name := Name;  
                Self.Age := Age;  
            end;  
            procedure Print;  
        end;  
  
    procedure Person.Print;  
    begin  
        writelnFormat('Имя: {0} Возраст: {1}', Name, Age);  
    end;
```

Как и в [классах](#), методы могут описываться как внутри, так и вне

тела записи. В примере выше конструктор описывается внутри записи, а метод Print объявляется внутри, а описывается вне тела записи. Метод-конструктор всегда имеет имя Create и предназначен для инициализации полей записи.

Инициализация записей

При описании переменной или константы типа запись можно использовать инициализатор записи (как и в Delphi Object Pascal):

```
const p: Person = (Name: 'Петрова'; Age: 18);  
var p: Person := (Name: 'Иванов'; Age: 20);
```

Конструкторы для записей имеют тот же синтаксис, что и для классов. Однако, в отличие от классов, вызов конструктора записи не создает новый объект в динамической памяти, а только инициализирует поля записи:

```
var p: Person := new Person('Иванов',20);
```

Более традиционно в записи определяется обычный метод-процедура, традиционно с именем **Init**, инициализирующая поля записи:

```
type  
  Person = record  
    ...  
  public  
    procedure Init(Name: string; Age: integer);  
  begin  
    Self.Name := Name;  
    Self.Age := Age;  
  end;  
  ...  
end;  
...  
var p: Person;  
p.Init('Иванов',20);
```

В системном модуле определена также функция **Rec**, которая создает переменную типа запись "на лету":

```
var p := Rec('Иванов',20);  
Println(p); // (Иванов,20)
```

Тип этой записи - безымянный. Поля данной записи автоматически именуются **Item1**, **Item2** и т.д.:

```
Println(p.Item1, p.Item2); // Иванов 20
```

Отличие записей от классов

Список отличий между записями и классами приводятся ниже:

1. Запись представляет собой размерный тип (переменные типа запись располагаются на стеке).
2. Записи нельзя наследовать; от записей также нельзя наследовать (отметим, что записи, тем не менее, могут реализовывать интерфейсы). В .NET тип записи неявно предполагается наследником типа `System.ValueType` и реализуется struct-типом.
3. Если в записи не указан модификатор доступа, то по умолчанию подразумевается модификатор **public** (все члены открыты), а в классе - **internal**.

Вывод переменной типа запись

По умолчанию процедура `write` для переменной типа запись выводит содержимое всех её публичных свойств и полей в круглых скобках через запятую. Чтобы изменить это поведение, в записи следует [переопределить](#) виртуальный метод `ToString` класса `Object` - в этом случае именно он будет вызываться при выводе объекта.

Например:

```
type
  Person = record
    ...
    function ToString: string; override;
  begin
    Result := string.Format('Имя: {0}  Возраст: {1}',
Name, Age);
  end;
end;
  ...
var p: Person := new Person('Иванов',20);
writeln(p); // Имя: Иванов  Возраст: 20
```

Присваивание и передача в качестве параметров подпрограмм

Поскольку запись, в отличие от класса, представляет собой размерный тип, то присваивание записей копирует содержимое полей одной переменной-записи в другую:

```
d2 := d1;
```

Для записей принята [именная эквивалентность типов](#): можно присваивать друг другу и передавать в качестве параметров подпрограмм записи, совпадающие только по имени.

Во избежание копирования те записи, которые содержат несколько полей, передаются в подпрограммы по ссылке. Если запись не меняется внутри подпрограммы, то используют ссылку на константу, если меняется - то ссылку на переменную:

```
procedure PrintPerson(const p: Person);  
begin  
    Print(p.Name, p.Age);  
end;
```

```
procedure ChangeName(var p: Person; NewName: string);  
begin  
    p.Name := Name;  
end;
```

Сравнение на равенство

Записи одного типа можно сравнивать на равенство, при этом записи считаются равными если значения всех полей совпадают:

```
type Person = record
  name: string;
  age: integer;
end;

var p1,p2: Person;

begin
  p1.age := 20;
  p2.age := 20;
  p1.name := 'Ivanov';
  p2.name := 'Ivanov';
  writeln(p1=p2); // True
end.
```

Замечание

В отличие от Delphi Object Pascal, в **PascalABC.NET** отсутствуют записи с вариантами.

Кортежи

Кортеж - это тип данных, являющийся разновидностью [записи](#). Как и запись, кортеж представляет собой набор элементов разных типов. В отличие от записей, тип кортежей описывается очень просто. Кроме того, поля имеют predetermined имена и являются неизменными: поменять поля кортежа после его создания невозможно.

Тип кортежа

Кортежи представляются типом `System.Tuple` платформы .NET:

```
var t: System.Tuple<string,integer>;
```

Для типа кортежа может быть использована сокращенная запись:

```
var t: (string,integer);
```

Данная запись похожа на объявление перечислимого типа. Если в круглых скобках - новые имена, то это перечислимый тип, а если имена типов, то это - тип кортежа.

Конструирование значений типа кортеж

Значения типа кортеж могут быть сконструированы в виде заключенного в круглые скобки перечисления составляющих кортеж значений через запятую. Например:

```
t := ('Иванов', 23);
```

Для построенных таким образом значений работает автовыведение типа:

```
var t1 := ('Иванов', (5, 3, 4)); // кортеж, вторым  
элементом которого является кортеж
```

Вывод кортежей

Как и при выводе записей, при выводе кортежей значения их элементов заключаются в круглые скобки и перечисляются через запятую:

```
writeln(t); // (Иванов,23)
writeln(t1); // (Иванов,(5,3,4))
```

Доступ к элементам кортежа

Элементы (поля) кортежа имеют имена Item1, Item2 и т.д.:

```
Print(t.Item1, t.Item2);
```

К элементам кортежа можно обращаться также по индексу:

```
Print(t[0], t[1]);
```

Индексы должны быть константными выражениями.

После создания кортеж неизменен: его поля нельзя менять:

```
t[1] := 20; // ошибка
```

Кортежное присваивание (распаковка кортежа в переменные)

Значения типа кортеж можно распаковать в переменные соответствующих типов, используя кортежное присваивание:

```
var t := ('Иванов', 23);  
var name: string;  
var age: integer;  
(name, age) := t;
```

Последнее присваивание, где в левой части оператора присваивания используется заключенное в круглые скобки перечисление переменных, называется **кортежным**. Кортежное присваивание компилятор заменяет на несколько подряд идущих одиночных присваиваний:

```
name := t[0];  
age := t[1];
```

С использованием кортежного присваивания меняется стиль программирования. Например, чтобы поменять местами значения двух переменных *a* и *b*, достаточно написать следующее кортежное присваивание:

```
(a, b) := (b, a);
```

В кортежном присваивании справа допускается количество элементов большее, чем количество переменных слева:

```
(a, b) := (1, 2, 3);
```

Кортежное присваивание можно совмещать с описанием переменных:

```
(var a, var b) := (1, 2);
```

или

```
var (a, b) := (1, 2);
```

Использование кортежей в функциях

Кортежи позволяют упаковывать несколько значений в одно. Это оказывается удобным если надо передавать несколько взаимосвязанных значений в качестве параметра или возвращать из функции несколько значений.

Например, функция, вычисляющая площадь и периметр прямоугольника, может быть записана в виде:

```
function SP(a,b: real) := (a*b,2*(a+b));
```

Чтобы воспользоваться результатом такой функции, удобно использовать распаковку кортежа в переменные с описанием:

```
var (S,P) := SP(2,3);
```

Множества

Множество представляет собой набор элементов одного типа. Элементы множества считаются неупорядоченными; каждый элемент может входить во множество не более одного раза. Тип множества описывается следующим образом:

set of базовый тип

В качестве базового может быть **любой** тип, в том числе строковый и классовый.

Например:

```
type   ByteSet = set of byte;
       StringSet = set of string;
       Digits = set of '0'..'9';
       SeasonSet = set of (Winter, Spring, Summer, Autumn);
       PersonSet = set of Person;
```

Элементы базового типа сравниваются на равенство следующим образом: у простых типов, строк и указателей сравниваются значения, у структурированных и у классов - значения всех элементов или полей. Однако, если поля относятся к ссылочному типу, то сравниваются только их адреса (неглубокое сравнение).

Переменная типа множество может содержать несколько значений базового типа. Чтобы сконструировать значение типа множество, используется конструкция вида

[*список значений*]

где в списке могут перечисляться через запятую либо выражения базового типа, либо (для порядковых типов) их диапазоны в виде *a..b*, где *a* и *b* - выражения базового типа. Например:

```
var
  bs: ByteSet := [1, 3, 5, 20..25];
  fios: StringSet := ['Иванов', 'Петров', 'Сидорова'];
```

Значения в списке могут отсутствовать, тогда множество является пустым:

```
bs:=[];
```

Пустое множество совместимо по присваиванию с множеством любого типа.

Для множеств имеет место структурная эквивалентность типов.

Множества целых и множества на базе типа и его диапазонного подтипа или на базе двух диапазонных типов одного базового типа неявно преобразуются друг к другу. Если при присваивании `s := s1` во множестве `s1` содержатся элементы, которые не входят в диапазон значений базового типа для множества `s`, то они отсекаются.

Например:

```
var st: set of 3..9;
...
st := [1..5,8,10,12]; // в st попадут значения [3..5,8]
```

Операция `in` проверяет принадлежность элемента множеству:

```
if Wed in bestdays then ...
```

Для множеств определены операции `+` (объединение), `-` (разность), `*` (пересечение), `=` (равенство), `<>` (неравенство), `<=` (нестрогое вложение), `<` (строгое вложение), `>=` (нестрого содержит) и `>` (строгое содержит).

Процедура `write` при выводе множества выводит все его элементы. Например,

```
write(['Иванов', 'Петров', 'Сидорова']);
```

выведет `['Иванов', 'Петров', 'Сидорова']`, при этом данные, если это возможно, будут отсортированы по возрастанию.

Для перебора всех элементов множества можно использовать цикл `foreach`, данные перебираются в некотором внутреннем порядке:

```
foreach var s in fios do
  write(s, ' ');
```

Для добавления элемента `x` к множеству `s` используется конструкция `s += [x]` или стандартная процедура `Include: Include(s, x)`. Для удаления элемента `x` из множества `s` используется конструкция `s -= [x]` или стандартная процедура `Exclude: Exclude(s, x)`.

Файловые типы

Файл представляет собой последовательность элементов одного типа, хранящихся на диске. В **PascalABC.NET** имеется два типа файлов - *двоичные* и *текстовые*. Текстовые файлы хранят символы, разделенные на строки символами #13#10 (Windows) и символом #10 (Linux). Последовательность символов для перехода на новую строку хранится в константе `NewLine`. Двоичные файлы в свою очередь делятся на типизированные и бестиповые.

Для описания текстового файла используется стандартное имя типа `text`, бестиповые файлы имеют тип `file`, а для описания типизированного файла используется конструкция `file of mun элементов`:

```
var f1: file of real;  
    f2: text;  
    f3: file;
```

В качестве типа элементов в типизированном файле не могут фигурировать указатели, ссылочные типы, а также тип записи, содержащий ссылочные поля или указатели.

Стандартные файловые процедуры и функции описываются в пунктах

- [Подпрограммы ввода](#)
- [Подпрограммы вывода](#)
- [Общие подпрограммы для работы с файлами](#)
- [Подпрограммы для работы с текстовыми файлами](#)
- [Подпрограммы для работы с двоичными файлами](#)
- [Подпрограммы для работы с именами файлов](#)
- [Общие методы файлов](#)
- [Методы текстовых файлов](#)
- [Методы типизированных файлов](#)
- [Методы двоичных файлов](#)
- [Методы расширения типизированных файлов](#)

Кроме того, в .NET имеется ряд классов, связанных с работой с файлами. Они находятся в пространствах имен `System.Text` и `System.IO`.

Последовательности

Последовательность - это набор данных, которые можно перебрать один за другим в некотором порядке. К разновидностям последовательностей относятся одномерные динамические массивы `array of T`, списки `List<T>`, двусвязные списки `LinkedList<T>`, множества `HashSet<T>` и `SortedSet<T>`.

Тип последовательности конструируется следующим образом:

sequence of *тип элементов*

Последовательности доступны **только на чтение**. Если требуется изменить последовательность, то генерируется и возвращается новая последовательность.

Тип **sequence of T** является синонимом типа .NET `System.Collections.Generic.IEnumerable<T>`, а последовательность - синонимом объекта типа, поддерживающего интерфейс `System.Collections.Generic.IEnumerable<T>`.

Инициализация последовательности

Последовательность инициализируется с помощью [стандартных функций](#) Seq, SeqGen, SeqFill, SeqWhile, SeqRandom, SeqRandomReal, ReadSeqInteger, ReadSeqReal, ReadSeqString. Например:

```
var s: sequence of integer;  
s := Seq(1,3,5);  
s.Println;  
s := SeqGen(1,x->x*2,10);  
writeln(s);
```

Хранение последовательности

Последовательность **не хранится целиком в памяти**. Элементы последовательности генерируются алгоритмически и возвращаются по одному при обходе.

Таким образом, в коде

```
var s := SeqFill(1, 10000000);  
writeln(s.Sum());
```

основное время выполнения будет занимать вторая строка, а выполнение первой строки будет сводиться лишь к запоминанию алгоритма генерации последовательности в переменной s.

Соединение последовательностей

Две последовательности одного типа могут быть соединены операцией +, при этом вторая последовательность дописывается в конец первой. Например:

$$\text{Seq}(1, 2, 3) + \text{Seq}(5, 6, 7)$$
$$\text{Seq}(1, 2, 3) + \text{Arr}(5, 6, 7)$$

Кроме того, к последовательности некоторого типа можно присоединить операцией + значение этого типа как первый или последний элемент последовательности, например:

$$\text{Seq}(1, 2, 3) + 5$$
$$3 + \text{Seq}(5, 6, 7)$$
$$3 + \text{Seq}(5, 6, 7) + 9$$

Операция + является сокращённым вариантом операции [Concat](#).

Для последовательностей доступна также операция умножения на число:

$$\text{Seq}(1, 2, 3) * 3$$

означает повторение последовательности 1 2 3 три раза: 1 2 3 1 2 3
1 2 3

Цикл по последовательности

Элементы последовательности можно обойти с помощью цикла **foreach**:

```
foreach var x in s do  
  if x>2 then  
    Print(x);
```

Совместимость по присваиванию

Переменной типа "последовательность" с элементами типа **T** можно присвоить одномерный массив **array of T**, список **List<T>**, двусвязный список **LinkedList<T>**, множество **HashSet<T>** или **SortedSet<T>**, а также объект любого класса, поддерживающего интерфейс **System.Collections.Generic.IEnumerable<T>**.

Стандартные подпрограммы и методы

Для последовательностей доступны:

- [Методы обработки последовательностей](#)
- [Подпрограммы для генерации последовательностей](#)
- [Подпрограммы для генерации бесконечных последовательностей](#)
- [Методы расширения для последовательностей](#)

Указатели

Указатель - это ячейка памяти, хранящая адрес. В PascalABC.NET указатели делятся на **типизированные** (содержат адрес ячейки памяти данного типа) и **бестиповые** (содержат адрес оперативной памяти, не связанный с данными какого-либо определенного типа).

Тип указателя на тип T имеет форму $\wedge T$, например:

```
type pinteger = ^integer;
var p: ^record r,i: real end;
```

Бестиповой указатель описывается с помощью слова `pointer`.

Для доступа к ячейке памяти, адрес которой хранит типизированный указатель, используется **операция разыменования** \wedge :

```
var
  i: integer;
  pi: ^integer;
  ...
  pi := @i; // указателю присвоили адрес переменной i
  pi^ := 5; // переменной i присвоили 5
```

Операция разыменования не может быть применена к бестиповому указателю.

Типизированный указатель может быть неявно преобразован к бестиповому:

```
var
  p: pointer;
  pr: ^real;
  ...
  p := pr;
```

Обратное преобразование также может быть выполнено неявно:

```
pr := p;
pr^ := 3.14;
```

Указатели можно сравнивать на равенство (=) и неравенство (<>). Для того чтобы отметить тот факт, что указатель никуда не указывает, используется стандартная константа `nil` (нулевой указатель): `p := nil`.

Внимание! Ввиду особенностей платформы .NET тип `T` типизированного указателя не должен быть ссылочным или содержать ссылочные типы на каком-то уровне (например, запрещены указатели на записи, у которых одно из полей имеет ссылочный тип). Причина такого ограничения проста: указатели реализуются неуправляемым кодом, который не управляется сборщиком мусора. Если в памяти, на которую указывает указатель, содержатся ссылки на управляемые переменные, то они становятся недействительными после очередной сборки мусора. Исключение составляют динамические массивы и строки, обрабатываемые особым образом. То есть, можно делать указатели на записи, содержащие в качестве полей строки и динамические массивы.

Процедурный тип

Тип, предназначенный для хранения ссылок на процедуры или функции, называется процедурным, а переменная такого типа - процедурной переменной. Основное назначение процедурных переменных - хранение и косвенный вызов действий (функций) в ходе выполнения программы и передача их в качестве параметров.

Описание процедурного типа

Описание процедурного типа совпадает с заголовком соответствующей процедуры или функции без имени. Например:

```
type ProcI = procedure (i: integer);  
      FunI = function (x,y: integer): integer;
```

Процедурной переменной можно присвоить процедуру или функцию с совместимым типом, например:

```
function Mult(x,y: integer): integer;  
begin  
    Result := x*y;  
end;  
var f: FunI := Mult;
```

Процедурной переменной можно также присвоить [лямбда-выражение](#) с соответствующим количеством параметров и типом возвращаемого значения:

```
var f2: FunI := (x,y) -> x+2*y;
```

После этого можно вызвать процедуру или функцию через эту процедурную переменную, пользуясь обычным синтаксисом вызова:

```
write(f(2)); // 8  
write(f2(3)); // 10
```

Синонимы для процедурных типов

Для [наиболее распространенных процедурных типов](#) в системном модуле определен ряд синонимов. Приведем примеры с их использованием:

```
var f3: IntFunc := x -> 2*x-1;  
var f4: Func<integer,real> := x -> 2.5*x;  
var f3: Action<real> := x -> write(x, ' ');  
var pr: Predicate<string> := s -> s.Length>0;
```

Сокращенные конструкции для процедурных типов

Для процедурных типов определены также сокращенные конструкции:

```
() -> T;           // функция без параметров,
возвращающая T
T1 -> T;           // функция с параметром T1,
возвращающая T
(T1, T2) -> T      // функция с параметрами T1 и T2,
возвращающая T
(T1, T2, T3) -> T  // функция с параметрами T1, T2 и T3,
возвращающая T
и т.д.
() -> ();          // процедура без параметров
T1 -> ();          // процедура с параметром T1
(T1, T2) -> ()     // процедура с параметрами T1 и T2
(T1, T2, T3) -> () // процедура с параметрами T1, T2 и T3
и т.д.
```

Сокращенные конструкции не могут описывать процедурные переменные с параметрами, передаваемыми по ссылке.

Для процедурных переменных принята [структурная эквивалентность типов](#): можно присваивать друг другу и передавать в качестве параметров процедурные переменные, совпадающие по структуре (типы и количество параметров, тип возвращаемого значения).

Процедурные переменные в качестве параметров

Обычно процедурные переменные передаются как параметры для реализации *обратного вызова* - вызова подпрограммы через процедурную переменную, переданную в качестве параметра в другую подпрограмму:

```
procedure forall(a: array of real; f: real->real);  
begin  
    for var i := 0 to a.Length-1 do  
        a[i] := f(a[i]);  
end;  
  
...  
forall(a, x->x*2); // умножение элементов массива на 2  
forall(a, x->x+3); // увеличение элементов массива на 3
```

Процедурная переменная может хранить нулевое значение, которое задается константой `nil`. Вызов подпрограммы через нулевую процедурную переменную приводит к ошибке.

Операции += и -= для процедурных переменных

Процедурные переменные реализуются через делегаты .NET. Это означает, что они могут хранить несколько подпрограмм. Для добавления/отсоединения подпрограмм используются операторы += и -=:

```
p1 += mult2;  
p1 += add3;  
forall(a, p1);
```

Подпрограммы в этом случае вызываются в порядке прикрепления: вначале умножение, потом сложение.

Отсоединение неприкрепленных подпрограмм не выполняет никаких действий:

```
p1 -= print;
```

Кроме того, к процедурной переменной можно прикреплять/откреплять [классовые и экземплярные методы классов](#). В последнем случае процедурная переменная в полях объекта запоминает некоторое состояние, которое меняется между вызовами метода, связанного с этой процедурной переменной.

Пример

```
type
  A = class
    x0: integer := 1;
    h: integer := 2;
    procedure PrintNext;
    begin
      Print(x0);
      x0 *= h;
    end;
  end;
begin
  var p: procedure;
  var a1 := new A();
  p := a1.PrintNext;
  for var i:=1 to 10 do
    p;
  // 1 2 4 8 16 32 64 128 256 512
end.
```

Подобное поведение гораздо проще реализовать с помощью [захвата переменной](#) лямбда-выражением:

```
begin
  var x0 := 1;
  var p: Action0 := procedure -> begin Print(x0); x0 *=
2 end;
  for var i:=1 to 10 do
    p;
end.
```

Эквивалентность и совместимость типов

Совпадение типов

Говорят, что типы T1 и T2 совпадают, если они имеют одно имя либо же определены в секции **type** в виде T1 = T2. Таким образом, в описаниях

```
type   IntArray = array [1..10] of integer;  
       IntArrayCopy = IntArray;  
var  
   a1: IntArray;  
   a2: IntArrayCopy;  
   b1,c1: array [1..15] of integer;  
   b2: array [1..15] of integer;
```

переменные a1 и a2 и переменные b1 и c1 имеют один и тот же тип, а переменные b1 и b2 - разные типы.

Эквивалентность типов

Говорят, что типы **T1** и **T2** эквивалентны, если выполняется одно из следующих условий:

1. **T1** и **T2** совпадают
2. **T1** и **T2** - динамические массивы с совпадающими типами элементов
3. **T1** и **T2** - указатели с совпадающими базовыми типами
4. **T1** и **T2** - множества с совпадающими базовыми типами
5. **T1** и **T2** - процедурные типы с совпадающим списком формальных параметров (и типом возвращаемого значения - для функций)

Если типы эквивалентны только если их имена совпадают, то говорят, что имеет место **именная эквивалентность типов**. Если типы эквивалентны если они совпадают по структуре, то говорят, что имеет место **структурная эквивалентность типов**. Таким образом, в **PascalABC.NET** имеет место именная эквивалентность для всех типов, кроме динамических массивов, множеств, типизированных указателей и процедурных типов, для которых имеет место структурная эквивалентность типов.

Только если типы **T1** и **T2** эквивалентны, фактический параметр типа **T1** может быть подставлен вместо формального параметра-переменной типа **T2**.

Совместимость типов

Говорят, что типы T_1 и T_2 совместимы, если выполняется одно из следующих условий:

1. T_1 и T_2 эквивалентны
2. T_1 и T_2 принадлежат к целым типам
3. T_1 и T_2 принадлежат к вещественным типам
4. Один из типов - поддиапазон другого или оба - поддиапазоны некоторого типа
5. T_1 и T_2 - множества с совместимыми базовыми типами

Совместимость типов по присваиванию

Говорят, что значение типа **T2** можно присвоить переменной типа **T1** или тип **T2** совместим по присваиванию с типом **T1**, если выполняется одно из следующих условий:

1. **T1** и **T2** совместимы
2. **T1** - вещественного типа, **T2** - целого
3. **T1** - строкового типа, **T2** - символьного
4. **T1** - **pointer**, **T2** - типизированный указатель
5. **T1** - указатель или процедурная переменная, **T2=nil**
6. **T1** - процедурная переменная, **T2** - имя процедуры или функции с соответствующим списком параметров
7. **T1**, **T2** - классовые типы, один из них - наследник другого.
Поскольку в **PascalABC.NET** все типы кроме указателей являются потомками типа **Object**, то значение любого типа (кроме указателей) можно присвоить переменной типа **Object**
8. **T1** - тип интерфейса, **T2** - тип класса, реализующего этот интерфейс

Если тип **T2** совместим по присваиванию с типом **T1**, то говорят также, что тип **T2** **неявно приводится** к типу **T1**.

Отображение на типы .NET

Стандартные типы **PascalABC.NET** реализуются типами библиотеки классов .NET. Далее приводится таблица соответствий стандартных типов **PascalABC.NET** и типов .NET.

Тип PascalABC.NET	Тип .NET
int64	System.Int64
uint64	System.UInt64
integer, longint	System.Int32
longword, cardinal	System.UInt32
BigInteger	System.BigInteger
smallint	System.Int16
word	System.UInt16
shortint	System.SByte
byte	System.Byte
boolean	System.Boolean
real	System.Double
double	System.Double
char	System.Char
string	System.String
object	System.Object
array of T	T[]
record	struct

Выражения и операции: обзор

Выражение - это конструкция, возвращающая значение некоторого типа. Простыми выражениями являются переменные и константы. Более сложные выражения строятся из простых с помощью операций, [ВЫЗОВОВ](#) [функций](#) и скобок. Данные, к которым применяются операции, называются *операндами*.

В **PascalABC.NET** имеются следующие операции: **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **+**, **-**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **as**, **is**, **in**, а также операция **new** и операция приведения типа. Операции **@**, **-**, **+**, **^**, **not**, операция приведения типа и операция **new** являются унарными (имеют один операнд), остальные являются бинарными (имеют два операнда), операции **+** и **-** являются и бинарными и унарными.

Справка по операциям PascalABC.NET

- [Арифметические операции](#)
- [Логические операции](#)
- [Операции сравнения](#)
- [Строковые операции](#)
- [Побитовые операции](#)
- [Операции с множествами](#)
- [Операция явного приведения типов](#)
- [Операции is и as](#)
- [Операция new](#)
- [Операция @ получения адреса](#)
- [Операции с указателями](#)
- [Операции typeof и sizeof](#)
- [Срезы](#)
- [Условная операция](#)

Порядок выполнения операций определяется их приоритетом. В языке PascalABC.NET имеется четыре уровня приоритетов операций, задаваемых таблицей приоритетов.

- [Таблица приоритетов операций](#)

Ряд операций для определяемых пользователем типов можно перегружать. Можно также перегружать операции для .NET-типов если они не были перегружены.

- [Перегрузка операций](#)

Арифметические операции

К *арифметическим* относятся бинарные операции $+$, $-$, $*$, $/$ для вещественных и целых чисел, бинарные операции **div** и **mod** для целых чисел и унарные операции $+$ и $-$ для вещественных и целых чисел. Тип выражения $x \text{ op } y$, где *op* - знак бинарной операции $+$, $-$ или $*$, определяется из следующей таблицы:

	shortint	byte	smallint	word	integer
shortint	integer	integer	integer	integer	integer
byte	integer	integer	integer	integer	integer
smallint	integer	integer	integer	integer	integer
word	integer	integer	integer	integer	integer
integer	integer	integer	integer	integer	integer
longword	int64	longword	int64	longword	int64
int64	int64	int64	int64	int64	int64
uint64	uint64	uint64	uint64	uint64	uint64
BigInteger	BigInteger	BigInteger	BigInteger	BigInteger	BigInte
single	single	single	single	single	single
real	real	real	real	real	real
decimal	decimal	decimal	decimal	decimal	decimal

То есть, если операнды - целые, то результатом является самый короткий целый тип, требуемый для представления всех получаемых значений.

При выполнении бинарной операции с **uint64** и знаковым целым результирующим типом будет **uint64**, при этом может произойти переполнение, не вызывающее исключения.

Для операции $/$ данная таблица исправляется следующим образом: результат деления любого целого на целое имеет тип **real**.

Для операций **div** и **mod** выполняются эти же правила, но операнды могут быть только целыми. Правила вычисления операций **div** и **mod** - следующие:

$x \text{ div } y$ - результат целочисленного деления x на y . Точнее, $x \text{ div } y = x / y$, округленное до ближайшего целого по направлению к 0; $x \text{ mod } y$ - остаток от целочисленного деления x на y . Точнее, $x \text{ mod } y = x - (x \text{ div } y) * y$.

Унарная арифметическая операция $+$ для любого целого типа возвращает этот тип. Унарная арифметическая операция $-$ возвращает для целых типов, меньших или равных `integer`, значение типа `integer`, для `longword` и `int64` - значение типа `int64`, к `uint64` унарная операция $-$ не применима, для типов `single` и `real` - соответственно типы `single` и `real`. То есть так же результатом является самый короткий тип, требуемый для представления всех получаемых значений.

Логические операции

К *логическим* относятся бинарные операции **and**, **or** и **xor**, а также унарная операция **not**, имеющие операнды типа `boolean` и возвращающие значение типа `boolean`. Эти операции подчиняются стандартным правилам логики: $a \text{ and } b$ истинно только тогда, когда истинны a и b , $a \text{ or } b$ истинно только тогда, когда истинно либо a , либо b , $a \text{ xor } b$ истинно только тогда, когда только одно из a и b истинно, **not** a истинно только тогда, когда a ложно.

Выражения с **and** и **or** вычисляются по *короткой схеме*:

в выражении $x \text{ and } y$ если x ложно, то все выражение ложно, и y не вычисляется;

в выражении $x \text{ or } y$ если x истинно, то все выражение истинно, и y не вычисляется.

Побитовые операции

К *побитовым* относятся бинарные операции **and**, **or**, **not**, **xor**, **shl**, **shr**. Они производят побитовые манипуляции с операндами целого типа. Результирующий тип для **and**, **or**, **xor** будет наименьшим целым, включающим все возможные значения обоих типов операндов. Для **shl**, **shr** результирующий тип совпадает с типом левого операнда, для **not** - с типом операнда.

Побитовые операции осуществляются следующим образом: с каждым битом (0 принимается за False, 1 - за True) производится соответствующая логическая операция. Например:

```
00010101 and 00011001 = 00010001
00010101 or  00011001 = 00011101
00010101 xor 00011001 = 00001100
not 00010101 = 11101010
```

(операнды и результат представлены в двоичной форме).

Операции **shl** и **shr** имеют вид:

```
a shl n
a shr n
```

где n - целое положительное число, a - целое число.

a **shl** n представляет собой целое положительное, полученное из двоичного представления числа a сдвигом влево на n позиций. Добавляемые справа позиции заполняются нулями.

a **shr** n представляет собой целое положительное, полученное из двоичного представления числа a сдвигом вправо на n позиций.

Например:

```
3 shl 2 = 12
12 shr 2 = 3
```

поскольку $3=11_2$, после сдвига влево на 2 позиции 11_2 преобразуется в $1100_2=12$, а $12=1100_2$ после сдвига вправо на 2 позиции преобразуется в $11_2=3$.

Операции сравнения

Операции сравнения `<`, `>`, `<=`, `>=`, `=`, `<>` возвращают значение типа `boolean` и применяются к операндам [простого типа](#) и к строкам.

Операции `=` и `<>` также применяются ко всем типам. Для размерных типов по умолчанию сравниваются значения, для ссылочных типов - ссылки. Можно переопределить это поведение, [перегрузив](#) операции `=` и `<>`. Аналогично можно перегрузить все операции сравнения для типов записей и классов, вводимых пользователем.

Строковые операции

К строкам применимы все операции сравнения `<`, `>`, `<=`, `>=`, `=`, `<>`.

Сравнение строк на неравенство осуществляется

лексикографически: `s1 < s2` если для первого несовпадающего символа с номером `i` `s1[i]<s2[i]` или все символы строк совпадают, но `s1` короче `s2`.

Кроме этого, к строкам и символам применима операция конкатенации (слияния) `+`, ее результат имеет строковый тип.

Например, `'a'+'b'='ab'`.

К строкам также применима операция `+=`:

```
s += s1; // s := s + s1;
```

Строка может складываться с числом, при этом число предварительно преобразуется к строковому представлению:

```
s := 'Ширина: '+15; // s = 'Ширина: 15'  
s := 20.5+''; // s = '20.5'  
s += 1; // s = '20.51'
```

Над строками и целыми определена операция `*`: `s*n` и `n*s` означает строку, образованную из строки `s`, повторенной `n` раз:

```
s := '*'*10; // s = '*****'  
s := 5*'ab' // s = 'ababababab'  
s := 'd'; s *= 3; // s = 'ddd'
```

Операции с указателями

Ко всем указателям применимы операции сравнения = и <>.

К типизированным указателям применима **операция разыменования** \wedge : если p является указателем на тип T , то p^\wedge - элемент типа T , на который указывает p . Указатели `pointer` разыменовывать нельзя.

Операции с множествами

К множествам с базовыми элементами одного типа применимы операции + (объединение), - (разность) и * (пересечение), а также операторы +=, -= и *:=:

```
var s1,s2,s: set of byte;
begin
  s1 := [1..4];
  s2 := [2..5];
  s := s1 + s2; // s = [1..5]
  s := s1 - s2; // s = [1]
  s := s1 * s2; // s = [2..4]
  // s += s1 эквивалентно s := s + s1
  // s -= s1 эквивалентно s := s - s1
  // s *= s1 эквивалентно s := s * s1
  s += [3..6]; // s = [2..6]
  s -= [3]; // s = [2,4..6]
  s *= [1..5]; // s = [2,4..5]
end.
```

К множествам с базовыми элементами одного типа применимы также операции сравнения = (равенство), <> (неравенство), <= (нестрого вложено), < (строго вложено), >= (нестрого содержит) и > (строго содержит):

```
[1..3] = [1,2,3]
['a'..'z'] <> ['0'..'9']
[2..4] < [1..5]
[1..5] <= [1..5]
[1..5] > [2..4]
[1..5] >= [1..5]
```

Но неверно, что `[1..5] < [1..5]`.

Наконец, операция **in** определяет, принадлежит ли элемент множеству: `3 in [2..5]` вернет `True`, `1 in [2..5]` вернет `False`.

Операция @

Операция @ применяется к переменной и возвращает ее адрес. Тип результата представляет собой типизированный указатель на тип переменной. Например:

```
var   r: real;  
      pr: ^real := @r;
```

Операции `is` и `as`

Операция `is` предназначена для проверки того, имеет ли классовая переменная указанный динамический тип. Операция `as` позволяет безопасно преобразовать переменную одного классового типа к другому классовому типу (в отличие от [явного приведения классового типа](#)).

Операция `is` имеет вид:

```
a is ClassType
```

и возвращает `True` если `a` принадлежит к классу `ClassType` или одному из его потомков.

Например, если `Base` и `Derived` - классы, причем, `Derived` - потомок `Base`, переменные `b` и `d` имеют соответственно типы `Base` и `Derived`, то выражения `b is Base` и `d is Base` возвращают `True`, а `b is Derived` - `False`.

Операция `as` имеет вид:

```
a as ClassType
```

и возвращает ссылку на объект типа `ClassType` если преобразование возможно, в противном случае возвращает `nil`.

Например, в программе

```
type Base = class
end;
Derived = class(Base)
  procedure p;
  begin
  end;
end;
var b: Base;
begin
  b := new Base;
  writeln(b is Derived);
  b := new Derived;
  writeln(b is Derived);
end.
```

первый раз выводится `False`, второй - `True`.

Операции `is` и `as` используются для работы с переменной базового класса, содержащей объект производного класса.

1 способ.

```
if b is Derived then
    Derived(b).p;
```

2 способ.

```
var d: Derived := b as Derived;
d.p;
```

Операция new

Операция **new** имеет вид:

```
new ИмяКласса(ПараметрыКонструктора)
```

Она вызывает конструктор класса *ИмяКласса* и возвращает созданный объект.

Например:

```
type   My = class  
    constructor Create(i: integer);  
    begin  
    end;  
end;  
  
var m: My := new My(5);
```

Эквивалентным способом создания объекта является вызов конструктора в стиле Object Pascal:

```
var m: My := My.Create(5);
```

Создание объекта класса при инициализации переменной проще проводить, используя **автоопределение типа**:

```
var m := new My(5);
```

В записи также могут быть определены конструкторы, которые вызываются аналогично. Но в отличие от класса вызов конструктора записи не выделяет память (она уже выделена) и только заполняет значения полей.

Операции `typeof` и `sizeof`

Операция `sizeof`(*имя типа*) возвращает для этого типа его размер в байтах.

Операция `typeof`(*имя типа*) возвращает для этого типа объект класса `System.Type`. Приведем пример использования `typeof`:

```
type Base = class ... end;  
    Derived = class(Base) ... end;  
var b: Base := new Derived;  
begin  
    writeln(b.GetType = typeof(Derived));  
end.
```

Операция явного приведения типов

Операция явного приведения типов имеет вид

ИмяТипа(*выражение*)

и позволяет преобразовать выражение к типу *ИмяТипа*. Тип выражения и тип с именем *ИмяТипа* должны оба принадлежать либо к порядковому типу, либо к типу указателя, либо один тип должен быть наследником другого, либо тип выражения должен поддерживать интерфейс с именем *ИмяТипа*. В случае указателей запрещено преобразовывать типизированный указатель к типу указателя на другой тип.

Пример.

```
type   pinteger = ^integer;
      Season = (Winter, Spring, Summer, Autumn);
var i: integer;
    b: byte;
    p: pointer := @i;
    s: Season;
begin
  i := integer('z');
  b := byte(i);
  i := pinteger(p);
  s := Season(1);
end.
```

При приведении размерных типов к типу Object происходит упаковка.

Пример.

```
var i: integer := 5;
begin
  var o: Object := Object(i);
end.
```

Условная операция

Условная операция имеет следующий вид:

условие ? выражение1 : выражение2

Если условие выполняется, то результатом является значение выражения1, иначе значение выражения2.

Например:

```
var min := a < b ? a : b;
```

Нетрудно видеть, что условная операция позволяет в простых случаях не использовать условный оператор. Например, предыдущий код для вещественных *a* и *b* эквивалентен следующему:

```
var min: real;  
if a<b then  
    min := a  
else min := b;
```

Срезы

Срез - это набор элементов динамического массива, списка `List<T>` или строки, расположенных последовательно либо с некоторым шагом.

Срез имеет вид:

```
a[from:to]
```

или

```
a[from:to:step]
```

или

```
a?[from:to]
```

или

```
a?[from:to:step]
```

и содержит копию элементов исходного контейнера в диапазоне `[from, to)` с шагом `step`. Запись `[from, to)` означает, что элемент с индексом `from` включается в диапазон, а элемент с индексом `to` - не включается. Значения `from`, `to`, `step` должны быть целыми.

Если `step` не указывается, то предполагается, что `step=1`, т.е. элементы расположены непрерывно.

Тип среза совпадает с типом контейнера: срез элементов массива представляет собой массив, срез строки - строку и срез для `List<T>` также принадлежит к типу `List<T>`.

Для обычных срезов исключение генерируется в следующих случаях. Если индекс `from` выходит за границы, то во время выполнения программы возникает исключение. Индекс `to` может выходить за границы на 1, т.е. находиться в диапазоне `[-1..a.Count]` для динамических массивов и списков `List<T>` и в диапазоне `[0..s.Length+1]` для строк, в противном случае также возникает исключение.

Если `step=0`, то для всех видов срезов генерируется исключение.

Срезы вида `a?` называются *безопасными срезами* и во всех остальных случаях не генерируют исключений. Они работают

следующим образом. Вначале контейнер считается бесконечным в обе стороны и срез выбирает в нём определённые элементы. После этого в срезе оставляются только те элементы, которые принадлежат исходному контейнеру.

Приведём ряд примеров:

```
var a := Arr(0,1,2,3,4,5,6);
Println(a[2:5]); // [2,3,4]
Println(a[2:a.Length]); // [2,3,4,5,6]
Println(a[2:1000]); // Исключение!
Println(a?[2:1000]); // [2,3,4,5,6]
Println(a[2:7:2]); // [2,4,6]
Println(a[2:0]); // пустой массив
var s := '0123456';
Println(s[2:5]); // 123
Println(s[0:2]); // Исключение!
Println(s?[0:2]); // 12
Println(s?[-2::2]); // 135
Println(s[2:s.Length]); // 12345
```

Шаг `step` может быть отрицательным. Например:

```
var a := Arr(0,1,2,3,4,5,6);
Println(a[5:2:-1]); // [5,4,3]
Println(a[2:5:-1]); // пустой массив
Println(a[a.Length-1:-1:-2]); // [6,4,2,0]
```

В записи среза выражение `from` или выражение `to` могут быть пропущены, в этом случае в срез попадают все элементы с соответствующей стороны. Если `step > 0`, то пропущенный `from` полагается равным индексу первого элемента, пропущенный `to` полагается равным индексу последнего элемента + 1. Если `step < 0`, то пропущенный `from` полагается равным индексу последнего элемента, а пропущенный `to` - индексу первого элемента - 1.

Например:

```
var s := '0123456';
Println(s[2:]); // 23456
Println(s[:4]); // 012
Println(s[:: -1]); // 6543210
```

Родственным к срезам является метод `Slice`, определённый для динамических массивов, списков `List<T>`, строк и

последовательностей. Однако, смысл параметров у него - другой:
a.Slice(from,step,count).

Приоритет операций

Приоритет определяет порядок выполнения операций в выражении. Первыми выполняются операции, имеющие высший приоритет. Операции, имеющие одинаковый приоритет, выполняются слева направо.

Таблица приоритетов операций

@, not , ^, +, - (унарные), new	1 (наивысший)
* , /, div , mod , and , shl , shr , as , is	2
+ , - (бинарные), or , xor	3
= , <> , < , > , <= , >= , in	4
?:	5 (низший)

Для изменения порядка выполнения операций в выражениях используются скобки.

Операторы: обзор

В PascalABC.NET определены следующие операторы.

- [Операторы присваивания](#)
- [Составной оператор](#)
- [Оператор описания переменной](#)
- [Оператор цикла for](#)
- [Оператор цикла foreach](#)
- [Операторы цикла while и repeat](#)
- [Условный оператор if](#)
- [Оператор выбора варианта case](#)
- [Оператор вызова процедуры](#)
- [Оператор try except](#)
- [Оператор try finally](#)
- [Оператор raise](#)
- [Операторы break, continue и exit](#)
- [Оператор goto](#)
- [Оператор lock](#)
- [Оператор with](#)
- [Пустой оператор](#)

Оператор присваивания

Оператор присваивания имеет вид:

переменная := выражение

В качестве переменной может быть простая переменная, разыменованный указатель, переменная с индексами или компонент переменной типа запись. Символ := называется значком присваивания. Выражение должно быть [совместимо по присваиванию](#) с переменной.

Оператор присваивания заменяет текущее значение переменной значением выражения.

Например:

```
i := i + 1; // увеличивает значение переменной i на 1
```

В **PascalABC.NET** определены также операторы присваивания со значками +=, -=, *=, /=. Для числовых типов действие данных операторов описано [здесь](#). Кроме того, использование операторов += и *= для строк описано [здесь](#) и операторов +=, -= и *= для множеств - [здесь](#). Их действие для процедурных переменных описано [здесь](#).

Операторы +=, -=, *=, /= имеют следующий смысл: $a \# = b$ означает $a := a \# b$, где # - знак операции +, -, *, /.

Например:

```
a += 3; // увеличить a на 3  
b *= 2; // увеличить b в 2 раза
```

Оператор /= неприменим, если выражение слева - целое.

Операторы +=, -=, *=, /= могут также использоваться со [свойствами классов](#) соответствующих типов в левой части.

Составной оператор (блок)

Составной оператор предназначен для объединения нескольких операторов в один. Он имеет вид:

```
begin  
    операторы  
end
```

В **PascalABC.NET** составной оператор также называется **блоком**. (традиционно в Паскале блоком называется раздел описаний, после которого идет составной оператор; в **PascalABC.NET** принято другое решение, поскольку можно описывать переменные непосредственно внутри составного оператора).

Операторы отделяются один от другого символом ";". Ключевые слова **begin** и **end**, окаймляющие операторы, называются **операторными скобками**.

Например:

```
s := 0;  
p := 1;  
for var i:=1 to 10 do  
begin  
    p := p * i;  
    s := s + p  
end
```

Перед **end** также может ставиться ";". В этом случае считается, что последним оператором перед **end** является пустой оператор, не выполняющий никаких действий.

Помимо операторов, в блоке могут быть внутриблочные описания переменных:

```
begin  
    var a,b: integer;  
    var r: real;  
    readln(a,b);  
    x := a/b;  
    writeln(x);  
end.
```

Пустой оператор

Пустой оператор не включает никаких символов, не выполняет никаких действий и используется в двух случаях:

1. Для использования символа ";" после последнего оператора в блоке:

```
begin  
  a := 1;  
  b := a;  
end
```

Поскольку в языке Паскаль символ ";" разделяет операторы, то в приведенном выше коде считается, что после последней ";" находится пустой оператор. Таким образом, ";" перед end в блоке можно либо ставить, либо нет.

2. Для пометки места, следующего за последним оператором в блоке:

```
label a;  
begin  
  goto a;  
  x := 1;  
a:  
end
```

Условный оператор

Условный оператор имеет *полную* и *краткую* формы.

Полная форма условного оператора выглядит следующим образом:

```
if условие then оператор1 else оператор2
```

В качестве условия указывается некоторое логическое выражение. Если условие оказывается истинным, то выполняется *оператор1*, в противном случае выполняется *оператор2*.

Краткая форма условного оператора имеет вид:

```
if условие then оператор
```

Если условие оказывается истинным, то выполняется *оператор*, в противном случае происходит переход к следующему оператору программы.

В случае конструкции вида

```
if условие1 then  
    if условие2 then оператор1  
    else оператор2
```

else всегда относится к ближайшему предыдущему оператору **if**, для которого ветка **else** еще не указана. Если в предыдущем примере требуется, чтобы **else** относилась к первому оператору **if**, то необходимо использовать составной оператор:

```
if условие1 then  
    begin  
        if условие2 then оператор1  
    end  
else оператор2
```

Например:

```
if a<b then  
    min := a  
else min := b;
```

Оператор описания переменной

В PascalABC.NET можно описывать переменные внутри [составного оператора](#) begin-end в специальном операторе описания переменной. Такие описания называются внутриблочными.

Внутриблочное описание имеет одну из форм:

```
var список имен: тип;
```

или

```
var имя: тип := выражение;
```

или

```
var имя: тип = выражение; // Для совместимости с Delphi
```

или

```
var имя := выражение;
```

Имена в списке перечисляются через запятую. Например:

```
begin  
  var a1, a2, a3: integer;  
  var n: real := 5;  
  var s := ' '  
  ...  
end.
```

В последнем случае тип переменной автовыводится по типу выражения в правой части. Автовыведение типа активно используется при инициализации переменной вызовом конструктора или функции, возвращающей объект:

```
begin  
  var l := new List<integer>;  
  var a := Seq(1,3,5); // тип a выводится по типу  
  возвращаемого значения Seq: array of integer  
end.
```

Кортежное присваивание с описанием переменных

Кортежное присваивание (распаковку кортежа в переменные) можно совмещать с описанием переменных:

```
var t := (1,2);  
(var a, var b) := (1,2);
```

или

```
var (a,b) := (1,2);
```

Распаковка кортежа в переменные часто используется при возвращении функцией кортежа:

```
function SP(a,b: real) := (a*b,2*(a+b));  
...  
var (S,P) := SP(2,3);
```

Инициализация лямбда-выражением

Автовыведение типа при описании невозможно при инициализации переменной лямбда-выражением:

```
// var f := x -> x*x; // так нельзя!  
var f : Func<integer, integer> := x -> x*x;
```

Внутриблочные описания используются чтобы не захламлять раздел описаний описанием вспомогательных переменных. Кроме этого, внутриблочные описания позволяют вводить переменные именно в тот момент когда они впервые потребовались. Оба этих фактора существенно повышают читаемость программы.

Оператор выбора

Оператор выбора выполняет одно действие из нескольких в зависимости от значения некоторого выражения, называемого переключателем. Он имеет следующий вид:

```
case переключатель of  
    список выбора 1: оператор1;  
    ...  
    список выбора N: операторN;  
else список операторов  
end;
```

Переключатель представляет собой выражение [порядкового типа](#) или строкового типа, а списки выбора содержат константы совместимого по присваиванию типа. Как и в операторе **if**, ветка **else** может отсутствовать.

Оператор **case** работает следующим образом. Если в одном из списков выбора найдено текущее значение переключателя, то выполняется оператор, соответствующий данному списку. Если же значение переключателя не найдено ни в одном списке, то выполняется список операторов по ветке **else** или, если ветка **else** отсутствует, оператор **case** не выполняет никаких действий.

Список выбора состоит либо из одной константы, либо для перечислимого типа из диапазона значений вида **a..b** (константа **a** должна быть меньше константы **b**); можно также перечислить несколько констант или диапазонов через запятую. Например:

```
case Country of  
    'Россия': Capital := 'Москва';  
    'Франция': Capital := 'Париж';  
    'Италия': Capital := 'Рим';  
    else Capital := 'Страна отсутствует в базе данных';  
end;
```

```
case DayOfWeek of  
    1..5: writeln('Будний день');  
    6,7: writeln('Выходной день');  
end;
```

Списки выбора не должны пересекаться. Например, следующий фрагмент

```
case i of
  2,5: write(1);
  4..6: write(2);
end;
```

приведет к ошибке компиляции.

Оператор цикла for

Оператор цикла **for** имеет одну из двух форм:

for *переменная* := начальное значение **to** конечное значение **do** оператор

или

for *переменная* := начальное значение **downto** конечное значение **do**
оператор

Кроме того, переменную можно описать непосредственно в заголовке цикла:

for *переменная*: *тип* := начальное значение **to** или **downto** конечное значение **do**
оператор

или

for var *переменная* := начальное значение **to** или **downto** конечное значение **do**
оператор

В последнем случае используется **автоопределение типа** переменной по типу начального значения. В двух последних случаях область действия объявленной переменной распространяется до конца тела цикла, которое в данном случае образует неявный блок. Вне тела цикла такая переменная недоступна, поэтому следующий цикл может использовать переменную с тем же именем:

```
for var i := 1 to 10 do  
  Print(i);  
for var i := 1 to 5 do  
  Print(i*i);
```

Текст от слова **for** до слова **do** включительно называется **заголовком цикла**, а оператор после **do** - **телом цикла**.

Переменная после слова **for** называется **параметром цикла**. Для первой формы цикла с ключевым словом **to** параметр цикла

меняется от начального значения до конечного значения, увеличиваясь всякий раз на единицу, а для второй формы ключевым словом **downto** - уменьшаясь на единицу. Для каждого значения переменной-параметра выполняется тело цикла. Однократное повторение тела цикла называется **итерацией цикла**. Значение параметра цикла после завершения цикла считается неопределенным.

Переменная-параметр цикла может иметь любой [порядковый тип](#). При этом начальное и конечное значения должны быть [совместимы по присваиванию](#) с переменной-параметром цикла.

Например:

```
var en: (red, green, blue, white);  
...  
for en := red to blue do  
  write(Ord(en):2);  
for var c := 'a' to 'z' do  
  write(c);
```

Если для цикла **for ... to** начальное значение переменной цикла больше конечного значения или для цикла **for ... downto** начальное значение переменной цикла меньше конечного значения, то тело цикла не выполнится ни разу.

Если цикл используется в подпрограмме, то переменная-параметр цикла должна быть описана как локальная. Наилучшим решением в PascalABC.NET является описание переменной в заголовке цикла.

Изменение переменной-параметра цикла внутри цикла является логической ошибкой. Например, следующий фрагмент со вложенным оператором **for** является ошибочным:

```
for i := 1 to 10 do  
  i -= 1;
```

Оператор цикла loop

Оператор цикла **loop** имеет форму:

loop *выражение* **do** *оператор*

Выражение должно быть целого типа и указывает количество повторений тела цикла. Если значение выражения ≤ 0 , то тело цикла не выполняется ни разу.

Цикл **loop** используется в простых ситуациях, когда тело цикла не зависит от номера итерации цикла:

```
loop 5 do  
    Print(1);  
var x := 1;  
loop 5 do  
begin  
    Print(x);  
    x += 2;  
end;
```

Оператор цикла `foreach`

Оператор цикла `foreach` имеет одну из следующих форм:

```
foreach переменная in контейнер do оператор
```

или

```
foreach переменная: тип in контейнер do  
оператор
```

или

```
foreach var переменная in контейнер do  
оператор
```

В качестве контейнера может фигурировать динамический массив, строка, множество, а также любой контейнер, удовлетворяющий интерфейсу `IEnumerable` или `IEnumerable<T>` (например, `List<T>`, `Dictionary<Key, Value>` и т.д.). Переменная цикла должна иметь тип, **совпадающий** с типом элементов контейнера (если контейнер удовлетворяет интерфейсу `IEnumerable`, то это тип `object`). В последней форме `foreach` тип переменной цикла автовыводится по типу элементов контейнера.

Переменная цикла пробегает все значения элементов контейнера и для каждого значения переменной цикла выполняется тело цикла. Изменение переменной цикла внутри тела цикла не меняет элементы контейнера, т.е. они доступны только на чтение.

Например:

```
var  
  ss: set of string := ['Иванов', 'Петров', 'Сидоров'];  
  a: array of integer := (3,4,5);  
  b: array [1..5] of integer := (1,3,5,7,9);  
  l := new List<real>;  
begin  
  foreach s: string in ss do  
    write(s, ' ');  
  writeln;  
  foreach x: integer in a do  
    write(x, ' ');
```

```
writeln;  
  foreach var x in b do  
    write(x, ' ');  
  writeln;  
  foreach var r in l do  
    write(r, ' ');  
end.
```

Операторы цикла **while** и **repeat**

Оператор цикла **while** имеет следующую форму:

```
while условие do   оператор
```

Условие представляет собой выражение логического типа, а оператор после **do** называется *телом цикла*. Перед каждой итерацией цикла условие вычисляется, и если оно истинно, то выполняется тело цикла, в противном случае происходит выход из цикла.

Если *условие* всегда оказывается истинным, то может произойти *зацикливание*:

```
while 2>1 do  
  write(1);
```

Оператор цикла **repeat** имеет следующую форму:

```
repeat  
  операторы  
until условие
```

В отличие от цикла **while**, условие вычисляется после очередной итерации цикла, и если оно истинно, то происходит выход из цикла. Таким образом, операторы, образующие тело цикла оператора **repeat**, выполняются по крайней мере один раз.

Обычно оператор **repeat** используют в ситуациях, где условие нельзя проверить, не выполнив тело цикла. Например:

```
repeat  
  read(x);  
until x=0;
```

Если *условие* всегда оказывается ложным, то может произойти *зацикливание*:

```
repeat  
  write(1);  
until 2=1;
```

Оператор with

Оператор **with** позволяет сократить обращение к полям записи, а также к полям, методам и свойствам объекта. Он имеет вид:

with *имя записи или объекта* **do** *оператор*

или

with *список имен* **do** *оператор*

Всюду внутри оператора можно опускать имя записи при обращении к полю указанной записи или имя объекта при обращении к полю, методу или свойству указанного объекта. Например, пусть описана переменная

```
var    DateOfBirthday = record
        Day: Integer;
        Month: Integer;
        Year: Integer;
end;
```

Тогда присваивание значений ее полям без использования оператора **with** имеет вид:

```
DateOfBirthday.Day := 23;
DateOfBirthday.Month := 2;
DateOfBirthday.Year := 1965;
```

Использование оператора **with** позволяет сократить предыдущую запись:

```
with DateOfBirthday do
begin
    Day := 23;
    Month := 2;
    Year := 1965;
end;
```

Если внешняя переменная имеет то же имя, что и поле (метод, свойство), то предпочтение отдается полю (методу, свойству). При наличии вложенных операторов **with** вначале предпринимается попытка рассматривать переменную как поле записи или объекта самого внутреннего оператора **with**, затем непосредственно объемлющего его оператора **with** и т.д. Если оператор **with**

содержит список объектов, то они рассматриваются справа налево.
Например, если имеются описания

```
var
  x,y,z: integer;
  a: record
    x,y: integer;
  end;
  b: record
    x: integer;
  end;
```

то фрагмент программы

```
with a,b do
begin
  x := 1;
  y := 2;
  z := 3;
end;
```

эквивалентен фрагменту

```
with a do
  with b do
  begin
    x := 1;
    y := 2;
    z := 3;
  end;
```

а также фрагменту

```
b.x:=1;
a.y:=2;
z:=3;
```

Оператор with устарел и сейчас практически не используется.

Оператор безусловного перехода **goto**

Оператор безусловного перехода **goto** имеет следующую форму:

goto *метка*

Он переносит выполнение программы к оператору, помеченному меткой *метка*.

Метка представляет собой идентификатор или целое без знака. Чтобы пометить оператор меткой, необходимо перед оператором указать метку с последующим двоеточием:

label1: оператор

Метки должны быть описаны в разделе меток с использованием служебного слова **label**:

label 1, 2, 3;

Например, в результате выполнения программы

```
label 1, 2;  
begin  
  var i := 5;  
2: if i < 0 then goto 1;  
  write(i);  
  Dec(i);  
  goto 2;  
1:  
end.
```

будет выведено 543210.

Метка должна помечать оператор в том же блоке, в котором описана. Метка не может помечать несколько операторов.

Переход на метку может осуществляться либо на оператор в том же блоке, либо на оператор в объемлющей конструкции. Так, запрещается извне цикла переходить на метку внутри цикла.

Использование оператора безусловного перехода в программе считается признаком плохого стиля программирования. Для основных вариантов использования **goto** в язык Паскаль введены специальные процедуры: **break** - переход на оператор, следующий

за циклом, **exit** - переход за последний оператор процедуры, **continue** - переход за последний оператор в теле цикла.

Один из немногих примеров уместного использования оператора **goto** в программе - выход из нескольких вложенных циклов одновременно. Например, при поиске элемента **k** в двумерном массиве:

```
var a: array [1..10,1..10] of integer;
...
var found := False;
for var i:=1 to 10 do
for var j:=1 to 10 do
    if a[i,j]=k then
        begin
            found := True;
            goto c1;
        end;
c1: writeln(found);
```

Операторы **break**, **continue** и **exit**

Операторы **break** и **continue** используются только внутри циклов.

Оператор **break** предназначен для досрочного завершения цикла. При его выполнении происходит немедленный выход из текущего цикла и переход к выполнению оператора, следующего за циклом. Оператор **continue** завершает текущую итерацию цикла, осуществляя переход к концу тела цикла. Например:

```
flag := False;
for var i:=1 to 10 do
begin
  read(x);
  if x<0 then continue; // пропуск текущей итерации
цикла
  if x=5 then
  begin
    flag := True;
    break; // выход из цикла
  end;
end;
```

Использование операторов **break** и **continue** вне тела цикла ошибочно.

Оператор **exit** предназначен для досрочного завершения процедуры или функции. Например

```
function Analyze(x: integer): boolean;
begin
  if x<0 then
  begin
    Result := False;
    exit
  end;
  ...
end;
```

Вызов **exit** в разделе операторов основной программы приводит к ее немедленному завершению.

Следует отметить, что в **PascalABC.NET** (в отличие от Borland Pascal и Borland Delphi) **break**, **continue** и **exit** являются не

процедурами, а именно операторами.

Оператор yield

Оператор `yield` используется в функциях, генерирующих последовательности, и имеет вид:

yield *выражение*

Функция, содержащая `yield`ы, называется *итератором*. При каждом вызове такая функция выполняет код до следующего `yield`, после чего заканчивает свою работу, возвращая значение выражения, указанного в `yield`, и сохраняет своё состояние до следующего вызова.

Например:

```
function Squares(n: integer): sequence of integer;
begin
  for var i:=1 to n do
    yield i*i
  end;

begin
  var q := Squares(5);

  foreach var x in q do
    Print(x);
    Println;

  q.Println;
end.
```

В данном примере переменная `q` хранит [последовательность](#), т.е. алгоритм вычисления квадратов первых `n` чисел, который будет запущен либо в цикле `foreach` по последовательности `q`, либо вызовом метода расширения `Println` для последовательности `q`. Данные последовательности поочередно возвращаются вызовами оператора `yield` в теле функции `Squares`.

После каждого вызова оператора `yield` функция возвращает следующее значение `i*i`, заканчивает свою работу и сохраняет значения всех своих переменных во внутреннем контексте. При следующем вызове этой функции её тело как бы начинает

выполняться с той точки, в которой мы оказались в конце предыдущего вызова.

В выражении `yield` могут фигурировать внешние относительно функции переменные. Говорят, что оператор `yield` осуществляет захват таких переменных. Например:

```
var a := 2;

function Squares(n: integer): sequence of integer;
begin
  for var i:=1 to n do
    yield i*a
  end;

begin
  var q := Squares(5);

  q.Println;
  a := 3;
  q.Println;
end.
```

В данном коде оператор `yield` захватывает переменную `a` из внешнего контекста. Захват производится по ссылке: если изменить переменную `a` в основной программе и повторно вызвать функцию-итератор, то при генерации последовательности будет использовано измененное значение переменной `a`. В итоге вывод данной программы будет иметь вид:

```
2 4 6 8 10
3 6 9 12 15
```

Для функций, в теле которых присутствуют `yield`, имеется ряд ограничений:

1. Функции, содержащие `yield`, могут возвращать только последовательности.
2. Среди параметров функций-итераторов не может быть `const`, `var`, `params`-параметров и параметров по умолчанию.
3. Если функция использует `yield`, в ней запрещено использовать переменную `Result`, и наоборот.
4. Функции с `yield` не могут содержать операторы `lock`, `try...except`,

try..finally.

5. Yield не может быть вложена в оператор with.
6. Yield не может быть использована внутри лямбда-выражений.
7. Функции с yield не могут содержать вложенные подпрограммы и сами быть вложенными подпрограммами.
8. Функции с yield не могут содержать локальные определения типов
9. Методы расширения с yield не могут быть рекурсивными.

Оператор `yield sequence`

Оператор `yield sequence` используется в функциях, генерирующих последовательности, вместе с [оператором `yield`](#), и имеет вид:

```
yield sequence выражение
```

В отличие от [оператора `yield`](#), оператор `yield sequence` перебирает элементы последовательности, указанной в выражении и возвращает эти элементы в качестве значений основной функции-итератора. Например, следующий код:

```
function f: sequence of real;  
begin  
  yield sequence Seq(1,2,3);  
  yield 4;  
end;  
  
begin  
  f.Println;  
end.
```

выведет последовательность

```
1 2 3 4
```

Следующий пример иллюстрирует формирование последовательности элементов при обходе бинарного дерева в инфиксном порядке:

```
function InfixPrintTree<T>(root: Node<T>): sequence of  
T;  
begin  
  if root = nil then exit;  
  yield sequence InfixPrintTree(root.left);  
  yield root.data;  
  yield sequence InfixPrintTree(root.right);  
end;
```

Для функций, в теле которых присутствуют `yield sequence`, действуют те же ограничения, что и для функций с `yield`.

Оператор `try ... except`

Оператор `try ... except` имеет вид:

```
try операторы
except
    блок обработки исключений
end;
```

Блок `try` называется *защищаемым блоком*. Если при выполнении программы в нем происходит ошибка, то он завершается и выполнение передается блоку `except`. Если исключение обрабатывается в блоке `except`, то после его обработки программа продолжает выполняться с оператора, следующего за `try ... except ... end`. Если исключение остается необработанным и имеется объемлющий блок `try`, то выполнение передается его блоку `except`. Если объемлющего блока `try` нет, то программа завершается с ошибкой. Наконец, если в блоке `try` ошибки не произошло, то блок `except` игнорируется и выполнение программы продолжается дальше.

Если в процессе обработки исключения (в блоке `except`) произошло другое исключение, то текущий блок `except` завершается, первое исключение считается необработанным и обработка нового исключения передается объемлющему блоку `try`. Таким образом, в каждый момент времени существует максимум одно необработанное исключение.

Блок обработки исключений представляет собой либо последовательность операторов, разделенных точкой с запятой, либо последовательность обработчиков исключений вида

```
on имя: тип do оператор
```

Обработчики разделяются символом `;`, после последнего обработчика также может следовать символ `;`. Здесь *тип* - тип исключения (должен быть производным от стандартного типа `Exception`), *имя* - имя переменной исключения (имя с последующим двоеточием может быть опущено). В первом случае при обработке исключения выполняются все операторы из блока `except`. Во втором

случае среди обработчиков осуществляется поиск типа текущего исключения (обработчики перебираются последовательно от первого до последнего), и если обработчик найден, то выполняется соответствующий оператор обработки исключения, в противном случае исключение считается необработанным и передается объемлющему блоку **try**. В последнем случае после всех обработчиков **on** может идти ветвь **else**, которая обязательно обработает исключение, если ни один из обработчиков не выполнен.

Следует обратить внимание, что имя переменной исключения в разных обработчиках может быть одинаковым, т.е. оно локально по отношению к обработчику.

Поиск типа исключения в обработчиках производится с учетом наследования: исключение будет обработано, если оно принадлежит к указанному в обработчике типу или производному от него. Поэтому принято записывать вначале обработчики производных классов, а затем - обработчики базовых (в противном случае обработчик исключения производного класса никогда не сработает). Обработчик исключения `Exception` обрабатывает все возможные исключения и поэтому должен быть записан последним.

Пример.

```
var a: array [1..10] of integer;
try
  var i: integer;
  readln(i);
  writeln(a[i] div i);
  ...
except
  on System.DivideByZeroException do
    writeln('Деление на 0');
  on e: System.IndexOutOfRangeException do
    writeln(e.Message);
  on System.FormatException do
    writeln('Неверный формат ввода');
  else writeln('Какое-то другое исключение');
end;
```

Оператор `try ... finally`

Оператор `try ... finally` имеет вид:

```
try операторы
finally
    операторы
end;
```

Операторы в блоке `finally` выполняются безотносительно к тому, возникло или нет исключение в блоке `try`. При этом само исключение не обрабатывается.

Блок `finally` используется для возвращения ранее выделенных ресурсов.

Пример 1. Закрытие открытого файла.

```
reset(f);
try
    ...
finally
    close(f);
end;
```

Файл будет закрыт независимо от того, произошло ли исключение в блоке `try`.

Пример 2. Возвращение выделенной динамической памяти.

```
New(p);
try
    ...
finally
    Dispose(p);
end;
```

Динамическая память, контролируемая указателем `p`, будет возвращена независимо от того, произошло ли исключение в блоке `try`.

Оператор raise

Оператор **raise** предназначен для возбуждения исключения и имеет вид:

```
raise объект
```

Здесь *объект* - объект класса, производного от `Exception`.

Например:

```
raise new Exception('Ошибка');
```

При возбуждении специфического исключения желательно определить свой тип исключения.

Для повторной генерации исключения внутри секции **except** используется также вызов **raise** без параметров:

```
raise;
```

Операторы += и -= для процедурных переменных

Оператор присваивания += предназначен для присоединения к процедурной переменной процедуры, оператор присваивания -= - для отсоединения. Подпрограммы вызываются в порядке присоединения. Например:

```
procedure mult2(var r: real);
begin
  r := 2 * r;
end;

procedure add3(var r: real);
begin
  r := r + 3;
end;

var
  p: procedure (var x: real);
  r: real;

begin
  r := 1;
  p := mult2;
  p += add3;
  p(r); // r := 2 * r; r := r + 3;
  p -= mult2;
  p(r); // r := r + 3;
end.
```

Отсоединение не присоединенных подпрограмм не выполняет никаких действий.

Кроме того, к процедурной переменной можно прикреплять/откреплять статические и экземплярные методы классов. Пример см. в теме [процедурные переменные](#).

Операторы += и -= используются также для добавления/удаления обработчиков для событий .NET. Например:

```
procedure OnTimer1(sender: object; e:
System.Timers.ElapsedEventArgs);
begin
  write(1);
end;
```

```
begin
  var Timer1 := new System.Timers.Timer(1000);
  Timer1.Elapsed += OnTimer1;
  Timer1.Start;
  while True do
    Sleep(1000);
end.
```

Оператор lock

Оператор lock имеет вид:

lock *объект* **do** *оператор*

Объект обязательно принадлежит к ссылочному типу.

Оператор **lock** гарантирует, что *оператор* будет выполняться только одним потоком. *Объект* здесь хранит блокировку, а *оператор*, представляющий собой тело оператора **lock**, называется блоком синхронизации. После того как первый поток заходит в блок синхронизации, он блокирует *объект*, при выходе из блока синхронизации - разблокирует. Если *объект* заблокирован, то никакой другой поток не может зайти в блок синхронизации и приостанавливается до разблокировки *объекта*.

Оператор

```
lock obj do  
    oper;
```

полностью эквивалентен следующему участку кода:

```
Monitor.Enter(obj);  
try  
    oper;  
finally  
    Monitor.Exit(obj);  
end;
```

Процедуры и функции: обзор

Что такое процедуры и функции

Процедура или функция представляет собой последовательность операторов, которая имеет имя, список параметров и может быть вызвана из различных частей программы. Функции, в отличие от процедур, в результате своего выполнения возвращают значение, которое может быть использовано в выражении. Для единообразия функции и процедуры называются *подпрограммами*.

Описание процедур и функций

Любая используемая в программе процедура или функция должна быть предварительно описана в разделе описаний.

Описание процедуры имеет вид:

```
procedure имя(список формальных параметров);  
раздел описаний  
begin  
    операторы  
end;
```

Описание функции имеет вид:

```
function имя(список формальных параметров): тип  
возвращаемого значения;  
раздел описаний  
begin  
    операторы  
end;
```

Операторы подпрограммы, окаймленные операторными скобками **begin/end**, называются *телом* этой подпрограммы.

Список формальных параметров вместе с окружающими скобками может отсутствовать. Он состоит из одной или нескольких секций, разделенных символом ";". Каждая секция состоит из списка переменных, перечисляемых через запятую, после которого следуют двоеточие и тип. Каждая секция может предваряться ключевым словом **var** или **const**, что указывает на то, что параметры передаются по ссылке (см.п. [Параметры процедур и функций](#)). Тип формального параметра должен быть либо именем, либо динамическим массивом, либо множеством, либо процедурной переменной (для последних трех типов имеет место [структурная эквивалентность типов](#)).

Раздел описаний процедуры или функции устроен так же, как и [раздел описаний](#) основной программы. Здесь описываются так называемые **локальные** переменные и константы, типы (за исключением классов - классы можно описывать только глобально) а

также вложенные процедуры и функции. Все такие локальные объекты доступны лишь внутри данной подпрограммы и не видны извне.

В разделе описаний подпрограммы можно описывать другие подпрограммы. Исключение составляют методы класса, описываемые непосредственно в теле класса: в них нельзя описывать вложенные подпрограммы ввиду синтаксической неоднозначности.

Например:

```
procedure DoAdd(a,b: real; var res: real);  
begin  
    res := a + b;  
end;
```

Вызов подпрограммы

Подпрограмма один раз описывается и может быть многократно вызвана. Для вызова процедуры используется [оператор вызова процедуры](#):

```
begin  
  var x := ReadInteger;  
  var y := ReadInteger;  
  var res: integer;  
  DoAdd(x, y, res);  
  Print(res);  
  DoAdd(2*x, y, res);  
  Print(res);  
end;
```

Для вызова функции используется [выражение вызова функции](#).

Переменная Result

Внутри тела любой функции определена специальная переменная с именем `Result`, которая хранит результат вычисления функции. Ее тип совпадает с типом возвращаемого значения функции. Например:

```
function Sum(a,b: real): real;
begin
  Result := a + b;
end;

function MinElement(a: array of real): real;
begin
  Result := real.MaxValue;
  foreach var x in a do
    if x < Result then
      Result := x;
  end;
begin
  var a := Seq(1,5,3);
  writeln(MinElement(a) + Sum(2,3));
end.
```

Если внутри функции не присвоить переменной `Result` некоторое значение, то функция вернет в результате своего вызова непредсказуемое значение.

Упрощенный синтаксис описания подпрограмм

В PascalABC.NET имеется упрощенный синтаксис описания однооператорных процедур:

```
procedure WriteStar := write('*');
```

Аналогичный синтаксис имеется для функций, вычисляющих одно выражение:

```
function Add(a,b: real): real := a + b;
```

При этом в ряде случаев для возвращаемого значения функции возможен автовывод типов:

```
function Add(a,b: real) := a + b;
```

Параметры процедур и функций

Параметры, указываемые при описании подпрограммы, называются **формальными**. Параметры, указываемые при вызове подпрограммы, называются **фактическими**.

Если формальный параметр описан с предваряющим ключевым словом **var** или **const**, то его называют **параметром-переменной** и говорят, что он передается **по ссылке**. Если же параметр описан без слов **var** или **const**, то его называют **параметром-значением** и говорят, что он передается **по значению**. Слово ссылка используется в PascalABC.NET также в другом значении - для [ССЫЛОЧНЫХ ТИПОВ](#).

Если параметр передается по значению, то при вызове подпрограммы значения фактических параметров присваиваются соответствующим формальным параметрам. Типы фактических параметров-значений должны быть [совместимы по присваиванию](#) с типами соответствующих формальных параметров.

Например, пусть имеется следующее описание процедуры:

```
procedure PrintSquare(i: integer);  
begin  
    writeln(i*i);  
end;
```

Тогда при вызове `PrintSquare(5*a-b)` значение $5*a-b$ будет вычислено и присвоено переменной `i`, после чего выполнится тело процедуры.

Если параметр передается по ссылке, то при вызове подпрограммы фактический параметр заменяет собой в теле процедуры соответствующий ему формальный параметр. В итоге любые изменения формального параметра-переменной внутри процедуры приводят к соответствующим изменениям фактического параметра. Фактические параметры-переменные должны быть переменными, а их типы должны быть [эквивалентны](#) типам соответствующих формальных параметров.

Например, если описана процедура

```
procedure Mult2(var a: integer);  
begin
```

```
    a := a*2;  
end;
```

то после вызова `Mult2(d)` значение `d` увеличится в 2 раза.

В качестве фактического параметра-значения можно указывать любое выражение, тип которого совпадает с типом формального параметра или неявно к нему приводится. В качестве фактического параметра-переменной можно указывать только переменную, тип которой в точности совпадает с типом формального параметра.

При передаче параметра по ссылке в подпрограмму передается адрес фактического параметра. Поэтому если параметр занимает много памяти (массив, запись, строка), то обычно он также передается по ссылке. В результате в процедуру передается не сам параметр, а его адрес, что экономит память и время работы. При этом если параметр меняется внутри подпрограммы, то он передается с ключевым словом `var`, если не меняется - с ключевым словом `const`:

Например:

```
type  
    Person = record  
        name: string;  
        age,height,weight: integer;  
    end;  
  
procedure Print(const p: Person);  
begin  
    write(p.name, ' ', p.age, ' ', p.height, ' ', p.weight);  
end;  
  
procedure IncAge(var p: Person);  
begin  
    Inc(p.age);  
end;
```

Отметим особенности передачи динамических массивов в качестве параметров подпрограмм.

Поскольку динамический массив является ссылкой, то при изменении формального параметра-динамического массива внутри подпрограммы меняется соответствующий фактический параметр. Например, в результате работы программы

```
procedure p(a: array of integer);  
begin  
  a[1] := 2;  
end;  
var b: array of integer := (1,1);  
begin  
  p(b);  
  writeln(b[1]);  
end.
```

будет выведено 2. Передавать динамические массивы по ссылке имеет смысл только в случае если память под динамический массив перераспределяется внутри подпрограммы:

```
procedure q(var a: array of integer);  
begin  
  SetLength(a,10);  
end;
```

Переменное число параметров

Для указания того, что подпрограмма должна иметь переменное число параметров, используется ключевое слово **params**, за которым следует описание динамического массива. Например:

```
function Sum(params a: array of integer): integer;  
begin  
    Result := 0;  
    for i: integer := 0 to a.Length do  
        Inc(Result, a[i]);  
end;
```

При вызове подпрограммы на месте формального параметра **params** может быть любое ненулевое количество фактических параметров совместимого типа, перечисляемых через запятую:

```
var s: integer := Sum(1, 2, 3, 4, 5);  
s := s + Sum(6, 7);
```

В списке параметров ключевое слово **params** может указываться только для последнего параметра, причем, этот параметр не должен быть параметром по умолчанию. Параметры **params** всегда передаются только по значению.

Параметры по умолчанию

В заголовке подпрограммы можно использовать параметры по умолчанию. Для этого достаточно после параметра поставить знак присваивания и значение. Если при вызове не указать значение параметра по умолчанию, то будет использовано то значение, которое указано в описании подпрограммы. Параметры по умолчанию должны передаваться по значению и идти последними в списке параметров.

Например:

```
procedure PrintTwo(a,b: integer; delim: char := ' ');  
begin  
    write(a,delim,b);  
end;  
  
...  
PrintTwo(3,5);  
PrintTwo(4,2,';');
```

Все варианты вызова подпрограммы с параметрами по умолчанию могут участвовать в разрешении [перегрузки](#).

Опережающее объявление

В некоторых ситуациях возникает необходимость вызвать подпрограмму, описанную далее по тексту программы. Например, такая необходимость возникает при косвенной рекурсии (подпрограмма *A* вызывает подпрограмму *B*, а та в свою очередь вызывает подпрограмму *A*). В этом случае используется опережающее объявление подпрограммы, состоящее из ее заголовка, за которым следует ключевое слово **forward**. Например:

```
procedure B(i: integer); forward;
procedure A(i: integer);
begin
    ...
    B(i-1);
end;
procedure B(i: integer);
begin
    ...
    A(i div 2);
end;
```

Запрещено делать опережающее объявление для уже описанной подпрограммы.

Для методов ключевое слово **forward** запрещено. В нем нет необходимости, потому что можно вызывать методы, определенные в теле класса позднее.

Перегрузка имен подпрограмм

В одном [пространстве имен](#) может быть определено несколько процедур или функций с одним именем, но разным количеством или типами параметров. Имена таких процедур и функций называются перегруженными, а их создание - **перегрузкой имен**.

Разновидностью перегрузки имен является [перегрузка операций](#).

При вызове перегруженной процедуры или функции выбирается та версия, у которой типы формальных параметров совпадают с типами фактических или наиболее близки к ним. Например, если имеются описания

```
procedure p(b: byte); begin
end;
procedure p(r: real);
begin
end;
```

то при вызове `p(1.0)` будет выбрана перегруженная версия с параметром типа `real` (точное соответствие), а при вызове `p(1)` будет выбрана перегруженная версия с параметром типа `byte` (при этом произойдет преобразование фактического параметра типа `integer` к типу `byte`).

Заметим, что, в отличие от Object Pascal, использовать при перегрузке служебное слово **overload** не нужно.

Если ни одна версия в текущем пространстве имен не подходит к данному вызову, то возникает ошибка компиляции. Если две и более версии одинаково хорошо подходят к данному вызову, то также возникает ошибка компиляции, заключающаяся в неоднозначности выбора подпрограммы. Например, если имеются описания

```
procedure p(i: integer; r: real);
begin
end;
procedure p(r: real; i: integer);
begin
end;
```

то при вызове `p(1, 2)` оба они одинаково подходят, что приводит к

неоднозначности.

Запрещено перегружать подпрограмму другой подпрограммой с тем же количеством и типами параметров, отличающихся лишь тем, передается ли параметр по значению или по ссылке. Например, описания

```
procedure p(i: integer);
```

и

```
procedure p(var i: integer);
```

считаются одинаковыми.

Возвращаемое значение функции не участвует в разрешении перегрузки, т.е. перегружаемые функции не могут различаться только типами возвращаемых значений.

Алгоритм перегрузки имен при наличии нескольких подключенных модулей, а также алгоритм перегрузки имен методов имеют особенности. Основная особенность этих алгоритмов состоит в том, что они работают через границы пространств имен.

Поиск перегруженного имени глобальной подпрограммы при наличии нескольких подключенных модулей происходит во всех модулях. При этом вначале осуществляется просмотр текущего модуля, а потом всех модулей, подключенных в секции **uses**, в порядке *справа налево*. Если при этом поиске находится объект, который не может перегружать предыдущие (например, перегружается процедура, а найдено имя переменной), то цепочка перегрузки заканчивается, и поиск наилучшей перегруженной подпрограммы идет среди найденных до этого момента. Если в модуле, откомпилированном позже, имеется подпрограмма с точно такими же параметрами, то она скрывает версию из модуля, откомпилированного раньше.

Например, пусть основная программа подключает два модуля - **un1** и **un2**:

main.pas

```
uses un2, un1;  
procedure p(i: integer);  
begin
```

```
    write(1);  
end;  
begin  
    p(2.2);  
    p(2);  
end.
```

un2.pas

```
unit un2;  
procedure p(r: real);  
begin  
    write(3);  
end;  
end.
```

un1.pas

```
unit un1;  
procedure p(r: real);  
begin  
    write(2);  
end;  
end.
```

В результате будет выведено 21, что означает, что первой была вызвана процедура `p` из модуля `un1`.

Поиск перегруженного имени метода осуществляется аналогично: вначале осуществляется просмотр текущего класса, затем его базового класса и т.д. до класса `Object`, либо до того момента, как будет встречен объект, который не может перегружать предыдущие (имя поля или свойства). Из всех найденных таким образом одноименных методов выбирается наилучший. При этом в разных классах могут быть методы с идентичными параметрами; в этом случае вызывается первый встреченный метод от данного класса к классу `Object`.

Подпрограммы с переменным числом параметров также участвуют в перегрузке, однако, обычные подпрограммы имеют над ними приоритет. Например, в ситуации

```
procedure p(i: integer);
```

```
begin  
  write(1);  
end;  
procedure p(params a: array of integer);  
begin  
  write(2);  
end;  
begin  
  p(1)  
end.
```

будет вызвана первая процедура.

Вызов подпрограмм из неуправляемой dll

Для вызова подпрограммы из неуправляемой dll (содержащей обычный, а не .NET-код) используется конструкция вида:

заголовок функции **external** '*имя dll*' **name** '*имя функции в dll*';

Например:

```
function MessageBox(h: integer; m,c: string; t: integer): integer;  
    external 'User32.dll' name 'MessageBox';  
...  
MessageBox(0, 'Hello!', 'Сообщение', 0);
```

Структура модуля

Модули предназначены для разбиения текста программы на несколько файлов. В модулях описываются переменные, константы, типы, классы, процедуры и функции. Для того чтобы эти объекты можно было использовать в вызывающем модуле (которым может быть и основная программа), следует указать имя файла модуля (без расширения .pas) в разделе **uses** вызывающего модуля. Файл модуля (.pas) или откомпилированный файл модуля (.psu) должен находиться либо в том же каталоге, что и основная программа, либо в подкаталоге **Lib** системного каталога программы **PascalABC.NET**.

Модуль имеет следующую структуру:

```
unit имя модуля;  
interface  
раздел интерфейса  
implementation  
раздел реализации  
initialization  
раздел инициализации  
finalization  
раздел финализации  
end.
```

Имеется также [упрощенный синтаксис модулей](#) без разделов интерфейса и реализации.

Первая строка обязательна и называется **заголовком модуля**. Имя модуля должно совпадать с именем файла.

Раздел интерфейса и раздел реализации модуля могут начинаться с [раздела uses](#) подключения внешних модулей и пространств имен .NET. Имена в двух разделах **uses** не должны пересекаться.

Раздел интерфейса включает объявление всех имен, которые экспортируются данным модулем в другие модули (при подключении

его в разделе **uses**). Это могут быть константы, переменные, процедуры, функции, классы, интерфейсы. Реализация методов классов может быть дана прямо в разделе интерфейса, но это не рекомендуется.

Раздел реализации содержит реализацию всех процедур, функций и методов, объявленных в разделе интерфейса. Кроме этого, в разделе реализации могут быть описания внутренних имен, которые не видны вне модуля и используются лишь как вспомогательные.

Раздел инициализации и *раздел финализации* представляют собой последовательность операторов, разделяемых символом **;**. Операторы из раздела инициализации модуля выполняются до начала основной программы, операторы из раздела финализации модуля - после окончания основной программы. Порядок выполнения разделов инициализации и разделов финализации подключенных модулей непредсказуем. Как раздел инициализации, так и раздел финализации могут отсутствовать.

Вместо разделов инициализации и финализации может присутствовать только раздел инициализации в виде

```
begin последовательность операторов  
end.
```

Например:

```
unit Lib;  
interface  
uses GraphABC;  
const Dim = 5;  
var Colors: array [1..Dim] of integer;  
function RandomColor: integer;  
procedure FillByRandomColor;  
implementation  
function RandomColor: integer;  
begin  
    Result := RGB(Random(255), Random(255), Random(255));  
end;  
procedure FillByRandomColor;  
begin
```

```
    for i: integer := 1 to Dim do
        Colors[i] := RandomColor;
    end;
initialization
    FillByRandomColor;
end.
```

Циклические ссылки между модулями возможны при определенных ограничениях.

Раздел `uses`

Раздел `uses` состоит из нескольких подряд идущих секций `uses`, каждая из которых имеет вид:

```
uses список имен;
```

Имена в списке перечисляются через запятую и могут быть либо именами подключаемых внешних модулей **PascalABC.NET**, либо пространствами имен .NET. Например:

```
uses System, System.Collections.Generic, MyUnit;
```

Здесь `MyUnit` - модуль **PascalABC.NET**, представленный в виде исходного текста или откомпилированного .рси-модуля, `System` и `System.Collections.Generic` - пространства имен .NET.

В модуле или основной программе, которая содержит раздел `uses`, можно использовать все имена из подключаемых модулей **PascalABC.NET** и пространств имен .NET. Основное отличие между модулями и пространствами имен .NET состоит в том, что модуль содержит код, а пространства имен .NET содержат лишь имена - для использования кода его необходимо подключить с помощью [директивы компилятора](#) `{$reference ИмяСборки}`, где *ИмяСборки* - имя dll-файла, содержащего .NET-код. Другое не менее важное отличие состоит в том, что в модуле или основной программе нельзя использовать имена, определенные в другом модуле, без подключения этого модуля в разделе `uses`. Напротив, если сборка .NET подключена директивой `$reference`, то можно использовать ее имена, явно уточняя их пространством имен, не подключая это пространство имен в разделе `uses`. Например:

```
begin    System.Console.WriteLine('PascalABC.NET');
end.
```

По умолчанию в первой секции `uses` неявно первым подключается системный модуль `PABCSystem`, содержащий стандартные константы, типы, процедуры и функции. Даже если раздел `uses` отсутствует, модуль `PABCSystem` подключается неявно. Кроме того, по умолчанию с помощью неявной директивы `$reference` подключаются сборки `System.dll`, `System.Core.dll` и `mscorlib.dll`,

содержащие основные .NET-типы.

Поиск глобальных имен осуществляется вначале в текущем модуле или основной программе, затем во всех подключенных модулях и пространствах имен, начиная с самого правого в секции **uses** и заканчивая самым левым. При этом считается, что пространство имен более правого модуля вложено в пространство имен более левого. Таким образом, конфликта имен не происходит. Если необходимо использовать имя из конкретного модуля или пространства имен, следует использовать запись

ИмяМодуля.Имя

или

ИмяПространстваИменNET.Имя

В качестве имени модуля может выступать также имя основной программы если у нее присутствует заголовок **program**.

Упрощенный синтаксис модуля

Упрощенный синтаксис модулей без разделов интерфейса и реализации имеет вид:

```
unit имя модуля;  
    раздел описаний  
end.
```

или

```
unit имя модуля;  
    раздел описаний  
begin  
    раздел инициализации  
end.
```

В разделе описаний описываются константы, переменные, процедуры, функции, классы, интерфейсы. Все имена экспортируются. Упрощенный синтаксис модулей удобно использовать при начальном обучении - модуль отличается от программы только заголовком и, возможно, отсутствием раздела операторов.

Циклические ссылки между модулями

Циклические ссылки модулей в интерфейсных частях запрещены. Например, следующая ситуация ошибочна:

```
unit A; interface
uses B;
implementation
end.
```

```
unit B;
interface
uses A;
implementation
end.
```

Таким образом, невозможно определить два общедоступных класса в разных модулях с объектными полями, ссылающимися друг на друга.

Однако, если одна ссылка находится в интерфейсной части, а вторая - в части реализации, или обе - в частях реализации, то циклические ссылки в этом случае разрешены:

```
unit A;
interface
implementation
uses B;
end.
```

```
unit B;
interface
uses A;
implementation
end.
```

Библиотеки dll

Библиотеки dll (dynamically linked libraries):

- содержат группу взаимосвязанных подпрограмм
- находятся в откомпилированном файле
- предназначены для обращения к ним из различных программ

Они находятся в файле с расширением .dll либо в текущем каталоге приложения (*локальные*), либо в системном каталоге (*глобальные библиотеки*). Глобальными библиотеками могут пользоваться одновременно несколько приложений.

По своему назначению библиотеки очень похожи на [модули](#), однако, имеют ряд важных отличий.

Отличия библиотек от модулей

1. При создании из модулей исполняемого файла .exe программа-л типы и константы, которые используются (вызываются) в основном все подпрограммы, потому что неизвестно, какие подпрограммы
2. Библиотеки .dll при выполнении программы полностью загружаются в оперативную память.
3. Библиотеки .dll часто используются одновременно несколькими программами.
4. Библиотека .dll может быть написана и откомпилирована на одном языке, а обращаться к ней можно из программ, написанных на других языках. Например, программа на PascalABC.NET может вызывать функцию из библиотеки, созданной на языке C# и наоборот. Таким образом, библиотеки обеспечивают **межъязыковое взаимодействие**.

Структура библиотеки

Библиотека имеет практически ту же структуру, что и [МОДУЛЬ](#):

library *имя библиотеки*;

interface

раздел интерфейса

implementation

раздел реализации

end.

Имя библиотеки должно совпадать с именем рас-файла, в котором библиотека находится.

Имеется также упрощенный синтаксис библиотек - без разделов интерфейса и реализации, совпадающий с [упрощенным синтаксисом модулей](#) (за исключением заголовка).

В результате компиляции библиотеки в текущем каталоге создается .dll-файл, содержащий откомпилированную библиотеку.

Подключение библиотеки к основной программе

Для подключения библиотеки к основной программе используется директива компилятора `{$reference ИмяБиблиотеки}`. Например:

```
{$reference ABC.dll}  
{$reference ABC1.dll}
```

```
begin  
    writeln(a.GetType);  
end.
```

Подключение библиотеки может проводиться в любом месте исходного файла.

Библиотеки ABC и ABC1 имеют соответственно вид:

```
library ABC;  
var a: integer;  
end.
```

и

```
library ABC1;  
var a: real;  
end.
```

Алгоритм поиска имен в библиотеках

В первую очередь имя ищется в исходном модуле, затем в модулях, подключенных в разделе `uses` в порядке справа налево, и только потом - в подключенных библиотеках в порядке подключения.

Согласно этому правилу в примере из предыдущего пункта переменная `a` будет иметь тип `integer`.

В случае коллизии имен используемое имя можно предварять именем библиотеки с последующей точкой:

```
{$reference ABC.dll}  
{$reference ABC1.dll}
```

```
begin  
    writeln(ABC1.a.GetType);  
end .
```

Документирующие комментарии

Можно помечать заголовки процедур, функций, методов, имена классов, типов, констант и переменных так называемыми документирующими комментариями. Документирующие комментарии всплывают в подсказках редактора при наведении курсора мыши на слово, при открытии скобки после имени подпрограммы и при выборе поля из списка полей, выпадающих при нажатии точки после имени. Система всплывающих подсказок в редакторе получила название Intellisense.

Документирующий комментарий располагается на строке, предшествующей помечаемому объекту, и начинается с символов `///`. Например:

```
const    /// Константа Pi
    Pi = 3.14;

type
    /// TTT - синоним целого типа
    TTT = integer;
    /// Документирующий комментарий класса XXX
    XXX = class
    end;

    /// Документирующий комментарий процедуры p
procedure p(a : integer);
begin
end;

var
    /// Документирующий комментарий переменной t1
    t1: TTT;
```

Документирующие комментарии могут занимать несколько строк, каждая из которых должна начинаться с `///`. Для комментирования подпрограмм можно использовать в первой строке документирующий комментарий `///-`, тогда его содержимое меняет заголовок подпрограммы в подсказке при наведении курсора мыши. Например:

```
///- Exclude(var s : set of T; e1 : T)
```

```
///Удаляет элемент e1 из множества s  
procedure Exclude(var s: TypedSet; e1: object);
```

Если первая строка документирующего комментария имеет вид `///-`, то подсказка не всплывает. Это делается для элементов, которые хочется скрыть от системы всплывающих подсказок.

Обзор классов и объектов

Описание классов

Класс представляет собой составной тип, состоящий из полей (переменных), [методов](#) (процедур и функций) и [свойств](#). Описание класса имеет вид:

```
type   имя класса = class  
    секция1  
    секция2  
    ...  
end;
```

Каждая секция имеет вид:

```
модификатор доступа  
    описания полей  
    объявления или описания методов и описания свойств
```

Модификатор доступа в первой секции может отсутствовать, при этом подразумевается модификатор **internal** (видимость всюду внутри сборки).

Методы могут описываться как внутри, так и вне класса. При описании метода внутри класса его имя предваряется именем класса с последующей точкой. Например:

```
type  
    Person = class  
    private  
        fName: string;  
        fAge: integer;  
    public  
        constructor Create(Name: string; Age: integer);  
        begin  
            fName := Name;  
            fAge := Age;  
        end;  
        procedure Print;  
        property Name: string read fName;  
        property Age: integer read fAge;
```

```
end;  
  
procedure Person.Print;  
begin  
    writelnFormat('Имя: {0} Возраст: {1}', Name, Age);  
end;
```

После слова **class** в скобках может быть указано имя класса-предка (см. [Наследование](#)), а также через запятую список поддерживаемых [интерфейсов](#).

Перед словом **class** может быть указано ключевое слово **sealed** – в этом случае от класса запрещено [наследовать](#).

Все описания и объявления внутри класса образуют **тело класса**. Поля и методы образуют **интерфейс** класса. Инициализаторы полей описаны [здесь](#).

Классы могут описываться только на глобальном уровне. Локальные определения классов (т.е. определения в разделе описания подпрограмм) запрещены.

Переменные типа класс

В языке **PascalABC.NET** классы являются [ссылочными типами](#). Это значит, что переменная типа класс хранит в действительности ссылку на объект.

Переменные типа класс называются **объектами** или **экземплярами класса**. Они инициализируются вызовом [конструктора](#) класса - специального метода, выделяющего память под объект класса и инициализирующего его поля:

```
var p: Person := new Person('Иванов', 20);
```

После инициализации через переменную типа класс можно обращаться к публичным членам класса (полям, методам, свойствам), используя точечную нотацию:

```
Print(p.Name, p.Age);  
p.Print;
```

Вывод переменной типа класс

По умолчанию процедура `write` для переменной типа класс выводит содержимое её публичных полей и свойств в круглых скобках через запятую:

```
write(p); // Иванов 20
```

Чтобы изменить это поведение, в классе следует [переопределить](#) виртуальный метод `ToString` класса `Object` - в этом случае именно он будет вызываться при выводе объекта.

Например:

```
type
  Person = class
    ...
    function ToString: string; override;
  begin
    Result := string.Format('Имя: {0}  Возраст: {1}',
Name, Age);
  end;
end;
...
var p: Person := new Person('Иванов',20);
writeln(p); // Имя: Иванов  Возраст: 20
```

Присваивание и передача в качестве параметров подпрограмм

Переменная типа класс является ссылкой и хранит ссылку на объект, создаваемый вызовом конструктора.

Как ссылка переменная типа класс может хранить значение **nil**:

```
p := nil;  
...  
if p = nil then ...
```

При присваивании переменных типа класс копируется только ссылка. После присваивания обе переменные типа класс будут ссылаться на один объект и совместно модифицировать его:

```
var p1,p2: Person;  
...  
p1 := new Person('Петров',20);  
p2 := p1;  
p1.IncAge;  
p2.Print; // Имя: Петров  Возраст: 21
```

Сравнение на равенство

При сравнении переменных типа класс на равенство сравниваются ссылки, а не значения.

```
var p1 := new Person('Петров',20);  
var p2 := new Person('Петров',20);  
writeln(p1=p2); // False  
p2 := p1;  
writeln(p1=p2); // True
```

Это поведение можно изменить, [перегрузив](#) операцию = для класса.

Видимость членов класса и модификаторы доступа

Каждое поле, метод или свойство класса имеет модификатор (атрибут) доступа, задающий правила его видимости. В **PascalABC.NET** существуют четыре вида модификаторов доступа: **public** (открытый), **private** (закрытый), **protected** (защищенный) и **internal** (внутренний). К члену класса, имеющему атрибут **public**, можно обратиться из любого места программы, члены класса с атрибутом **private** доступны только внутри методов этого класса, члены класса с атрибутом **protected** доступны внутри методов этого класса и всех его подклассов, члены класса с атрибутом **internal** доступны внутри сборки (термин .NET, сборка в нашем понимании - это множество файлов, необходимых для генерации .exe или .dll-файла). Кроме того, **private** и **protected** члены видны отовсюду в пределах модуля, в котором определен класс.

Тело класса делится на секции. В начале каждой секции располагается модификатор доступа, после которого идут поля, а затем методы и свойства с доступом, определяемым этим модификатором. В первой секции модификатор доступа может отсутствовать, в этом случае подразумевается модификатор **internal**. В классе может быть произвольное количество секций, располагающихся в произвольном порядке.

Например, пусть данный код располагается в одном модуле:

```
type A = class
  private
    x: integer;
  protected
    a: integer;
  public
    constructor Create(xx: integer);
  begin
    x := xx; // верно, т.к. внутри метода класса
    можно обращаться к его закрытому полю x
    a := 0; // верно
  end;
end;
```

Следующий же код пусть располагается в другом модуле:

```
type
  B = class(A)
  public
    procedure print;
  begin
    writeln(a); // верно, т.к. a - защищенное поле
    writeln(x); // неверно, т.к. x - закрытое поле
  end;
end;

...
var b1: B := new B(5);
...
writeln(b1.x); // неверно, т.к. x - закрытое поле
writeln(b1.a); // неверно, т.к. a - защищенное поле
b1.print; // верно, т.к. print - открытый метод
```

Комментарии по тексту программы описывают верное и неверное в смысле доступа обращение к полям и методам.

Методы

Методы представляют собой процедуры и функции, объявленные внутри класса или записи. Особыми разновидностями методов являются конструкторы, деструкторы и перегруженные операции.

Определение методов можно давать как внутри класса (стиль Java, C#, C++), так и вне класса (стиль Delphi, C++). При определении метода вне интерфейса класса его имя предваряется именем класса с последующей точкой. Например:

```
type Rectangle = class
  x1,y1,x2,y2: integer;
  constructor Create(xx1,yy1,xx2,yy2: integer);
begin
  x1 := xx1; x2 := xx2;
  y1 := yy1; y2 := yy2;
end;
  function Square: integer;
end;

function Rectangle.Square: integer;
begin
  Result := abs(x2-x1) * abs(y2-y1);
end;
```

Обычно когда класс определяется в интерфейсной части модуля, то в интерфейсе класса производят лишь объявление методов, реализацию же методов класса дают в секции реализации модуля.

Методы делятся на *классовые* и *экземплярные*. Классовые методы в .NET называются *статическими*. Объявление классического метода начинается с ключевого слова **class**. Экземплярные методы можно вызывать только через переменную-объект класса. Классовые же методы не связаны с конкретным экземпляром класса; их следует вызывать в виде:

имя класса.имя метода(параметры)

Внутри классического метода не может быть обращения к полям класса, а может быть только обращение к другим классическим методам. Напротив, экземплярный метод может вызывать классический.

Например:

```
type
  Rectangle = class
    ...
    class procedure Move(var r: Rectangle; dx,dy:
integer);
    begin
      r.x1 += dx; r.x2 += dx;
      r.y1 += dy; r.y2 += dy;
    end;
  end;
  ...
var r := new Rectangle(10,10,100,100);
  Rectangle.Move(r,5,5);
```

По существу, классовые методы являются разновидностью глобальных подпрограмм, но находятся внутри класса, что подчеркивает, что они осуществляют действия, связанные именно с этим классом. Класс в этом случае выступает только в роли пространства имен.

Нередко создаются классы, целиком состоящие из классовых методов. Таков, например, класс `System.Math`, содержащий определения математических подпрограмм.

Инициализаторы полей

При создании объекта *его поля инициализируются автоматически нулевыми значениями* если они не инициализированы явно. Их инициализация может проводиться как в конструкторе, так и непосредственно при описании. Инициализация поля при описании приводит к тому, что код инициализации вставляется в начало ВСЕХ конструкторов.

Например:

```
type A = class
  private
    x: integer := 1;
    y: integer;
    l := new List<integer>;
  public
    constructor Create(xx,yy: integer);
    begin
      x := xx;
      y := yy;
    end;
    constructor Create;
    begin
    end;
end;
```

В данном примере код `x:=1; l := new List<integer>` вставляется в начало каждого конструктора.

Конструкторы

Объекты создаются с помощью специальных методов, называемых **конструкторами**.

Конструктор представляет собой функцию, создающую объект в динамической памяти, инициализирующую его поля и возвращающую указатель на созданный объект. Этот указатель обычно сразу присваивается переменной типа класс. При описании конструктора вместо служебного слова **function** используется служебное слово **constructor**. Кроме того, для конструктора не указывается тип возвращаемого значения.

Например:

```
type Person = class
  private
    nm: string;
    ag: integer;
  public
    constructor Create(name: string; age: integer);
end;
...
constructor Person.Create(name: string; age: integer);
begin
  nm := name;
  ag := age;
end;
```

В **PascalABC.NET** конструктор всегда должен иметь имя **Create**. При описании конструктора внутри класса можно опускать его имя:

```
type
  Person = class
    constructor (name: string; age: integer);
  begin
    nm := name;
    ag := age;
  end;
end;
```

В силу особенностей реализации вызовов конструкторов в **.NET** в **PascalABC.NET** всегда создается конструктор без параметров

(независимо от того, определен ли другой конструктор). Этот конструктор инициализирует все поля нулевыми значениями (строковые поля - пустыми строками, логические - значением False).

Для вызова конструктора можно использовать два способа.

1 способ. В стиле Object Pascal.

Для вызова конструктора следует указать имя класса, за которым следует точка-разделитель, имя конструктора и список параметров. Например:

```
var p: Person;  
p := Person.Create('Иванов', 20);
```

2 способ. С помощью операции `new` - в стиле C# (предпочтительный).

```
var p: Person;  
p := new Person('Иванов', 20);
```

Деструктор в Object Pascal - специальная процедура, уничтожающая объект и освобождающая динамическую память, которую этот объект занимал. При описании деструктора вместо служебного слова **procedure** используется служебное слово **destructor**.

Например:

```
destructor Destroy;  
begin  
    ...  
end;
```

Поскольку в **PascalABC.NET** память управляется сборщиком мусора, деструктор в **PascalABC.NET** не играет никакой роли и представляет собой обычную процедуру-метод.

Предварительное объявление классов

Два или более класса могут содержать в качестве полей объекты других классов, циклически ссылающиеся друг на друга.

Например:

```
type   AAA = class
    b: BBB;
end;
BBB = class
    a: AAA;
end;
```

Данный код вызовет ошибку компиляции, поскольку тип `BBB` в момент описания поля `b` еще не определен. В такой ситуации следует воспользоваться предварительным описанием класса в виде

```
ИмяКласса = class;
```

Предварительно описанный класс должен быть полностью описан в той же секции `type`:

```
type
    BBB = class;
    AAA = class
        b: BBB;
    end;
    BBB = class
        a: AAA;
    end;
```

Переменная Self

Внутри каждого нестатического метода неявно определяется переменная `Self`, ссылающаяся на объект, вызвавший этот метод.

Например:

```
type    A = class
  i: integer;
  constructor Create(i: integer);
  begin
    Self.i := i;
  end;
end;
```

В момент вызова конструктора `Create` объект будет уже создан. Конструкция `Self.i` ссылается на поле `i` этого объекта, а не на параметр `i` функции `Create`. Фактически в любом нестатическом методе перед именем любого поля и методу этого класса неявно присутствует `Self`.

Свойства

Свойство внешне выглядит как поле класса, однако, при доступе к нему на чтение или запись позволяет выполнять некоторые действия. Свойство описывается в классе следующим образом:

```
property Prop: mun read имя функции чтения write имя процедуры записи;
```

Как правило, каждое свойство связано с некоторым полем класса и возвращает значение этого поля с помощью *функции чтения*, а меняет - с помощью *процедуры записи*. Функция чтения и процедура записи должны быть методами этого класса и иметь следующий вид:

```
function getProp: mun; procedure setProp(v: mun);
```

Обычно функция чтения и процедура записи описываются в приватной секции класса. Они могут быть виртуальными, в этом случае их уместно описывать в защищенной секции класса.

Вместо *имени функции чтения* и *имени процедуры записи* может фигурировать имя поля, с которым данное свойство связано. Любая из секций **read** или **write** может быть опущена, в этом случае мы получаем свойство с доступом только на запись или только на чтение.

Например:

```
type  
Person = class  
  private  
    nm: string;  
    ag: integer;  
  procedure setAge(a: integer);  
  begin  
    if a >= 0 then  
      ag := a  
    else raise new Exception('Возраст не может быть отрицательным');  
  end;  
  function getId: string;  
  begin  
    Result := nm + ag.ToString;
```

```

    end;
public
...
property Age: integer read ag write setAge;
property Name: string read nm;
property Id: string read getId;
end;
var p: Person;
p := new Person('Иванов', 20);
p.Age := -3; // генерируется исключение
var i: integer := p.Age;
writeln(p.Id);

```

Всякий раз, когда мы присваиваем свойству `Age` новое значение, вызывается процедура `setAge` с соответствующим параметром. Всякий раз, когда мы считываем значение свойства `Age`, происходит обращение к полю `ag`. Поле `nm` доступно только на чтение. Наконец, свойство `Id` осуществляет доступ на чтение к информации, находящейся в двух полях.

Обычно для доступа к полю на чтение в секции **read** свойства указывается именно поле, так как обычно при чтении поля никаких дополнительных действий производить не требуется.

Свойства не могут передаваться по ссылке в процедуры и функции. Например, следующий код ошибочен:

```
Inc(p.Age); // ошибка!
```

Если требуется обработать значение свойства, передав его по ссылке, то надо воспользоваться вспомогательной переменной:

```
a := p.Age;
Inc(a);
p.Age := a;
```

Однако, свойства соответствующих типов можно использовать в левой части операций присваивания `+=` `-=` `*=` `/=`:

```
p.Age += 1;
```

Свойства очень удобны при работе с визуальными объектами, поскольку позволяют автоматически перерисовывать объект, если изменить какие-либо его визуальные характеристики. Например,

если создана кнопка `b1` типа `Button`, то для визуального изменения ее ширины достаточно присвоить значение ее свойству `Width`:

```
b1.Width := 100;
```

Процедура для записи этого свойства в приватное поле `fwidth` будет выглядеть примерно так:

```
procedure SetWidth(w: integer);  
begin  
  if (w>0) and (w<>fwidth) then  
  begin  
    fwidth := w;  
    код перерисовки кнопки  
  end  
end;
```

Следует обратить внимание на вторую часть условия в операторе `if`: `w<>fwidth`. Добавление этой проверки позволяет избежать лишней перерисовки кнопки в случае, если ее ширина не меняется.

Индексные свойства

Индексные свойства ведут себя аналогично полям-массивам и используются, как правило, для доступа к элементам контейнеров. Как и при использовании обычных свойств, при использовании индексных свойств могут попутно выполняться некоторые действия.

Индексное свойство описывается в классе следующим образом:

```
property Prop[описание индексов]: тип read имя функции  
чтения write имя процедуры записи;
```

В простейшем случае одного индекса описание индексного свойства выглядит так:

```
property Prop[ind: тип индекса]: тип read имя функции  
чтения write имя процедуры записи;
```

В этом случае функция чтения и процедура записи должны быть методами этого класса и иметь следующий вид:

```
function GetProp(ind: тип индекса): тип; procedure  
SetProp(ind: тип индекса; v: тип);
```

Всякий раз, когда мы для объекта *a*, содержащего свойство *Prop*, выполняем присваивание *a.Prop[ind] := value*, вызывается процедура *SetProp(ind, value)*, а когда считываем значение *a.Prop[ind]*, вызывается функция *GetProp(ind)*.

Индексное свойство, после которого добавлено ключевое слово **default** с последующей **;**, называется **индексным свойством по умолчанию** и позволяет пользоваться объектами класса как массивами, т.е. использовать запись *a[ind]* вместо *a.Prop[ind]*.

Принципиальное отличие индексных свойств от полей-массивов состоит в том, что тип индекса может быть произвольным (в частности, строковым). Это позволяет легко реализовать так называемые ассоциативные массивы, элементы которых индексируются строками.

В следующем примере индексное свойство используется для закрашивания/стирания клеток шахматной доски в графическом

режиме.

```
uses GraphABC;
const
  n = 8;
  sz = 50;
type ChessBoard = class
private
  a: array [1..n,1..n] of boolean;
  procedure setCell(x,y: integer; value: boolean);
  begin
    if value then
      Brush.Color := clWhite
    else Brush.Color := clBlack;
    Fillrect((x-1)*sz+1,(y-1)*sz+1,x*sz,y*sz);
    a[x,y] := value;
  end;
  function getCell(x,y: integer): boolean;
  begin
    Result := a[x,y];
  end;
public
  property Cells[x,y: integer]: boolean read getCell
  write setCell; default;
end;

var c: ChessBoard := new ChessBoard;
begin
  var x,y: integer;
  for x:=1 to n do
    for y:=1 to n do
      c[x,y] := Odd(x+y);
    end;
  end.
end.
```

Наследование

Класс может быть унаследован от другого класса. Класс, от которого наследуют, называют **базовым классом** (**надклассом**, **предком**), а класс, который наследуется, называется **производным классом** (**подклассом**, **потомком**). При наследовании все поля, методы и свойства базового класса переходят в производный класс, кроме этого, могут быть добавлены новые поля, методы и свойства и переопределены (замещены) старые методы. Конструкторы наследуются по особым правилам, которые рассматриваются [здесь](#).

При описании класса его базовый класс указывается в скобках после слова **class**.

Например:

```
type   BaseClass = class
    procedure p;
    procedure q(r: real);
end;
MyClass = class(BaseClass)
    procedure p;
    procedure r(i: integer);
end;
```

В данном примере процедура **p** переопределяется, а процедура **r** добавляется в класс **MyClass**.

Если не указать имя базового класса, то считается, что класс наследуется от класса **Object** - предка всех классов. Например, **BaseClass** наследуется от **Object**.

Переопределение методов при наследовании рассматривается [здесь](#).

Перед словом **class** может быть указано ключевое слово **sealed** – в этом случае от класса запрещено наследовать.

Переопределение методов

Метод базового класса может быть переопределен (замещен) в подклассах. Если при этом требуется вызвать метод базового класса, то используется служебное слово **inherited** (англ.- унаследованный). Например:

```
type Person = class
  private
    name: string;
    age: integer;
  public
    constructor Create(nm: string; ag: integer);
    begin
      name := nm;
      age := ag;
    end;
    procedure Print;
    begin
      writeln('Имя: ', name, '  Возраст: ', age);
    end;
end;

Student = class(Person)
  private
    course, group: integer;
  public
    constructor Create(nm: string; ag, c, gr: integer);
    begin
      inherited Create(nm, ag);
      course := c;
      group := gr;
    end;
    procedure Print;
    begin
      inherited Print;
      writeln('Курс: ', course, '  Группа: ', group);
    end;
end;
```

Здесь метод `Print` производного класса `Student` вызывает вначале метод `Print`, унаследованный от базового класса `Person`, с помощью

конструкции **inherited Print**. Аналогично конструктор **Create** класса **Student** вызывает вначале конструктор **Create** базового класса **Person**, также используя служебное слово **inherited**.

Правила наследования конструкторов рассматриваются [здесь](#).

Следует обратить внимание, что конструктор базового класса вызывается в этом случае как процедура, а не как функция, при этом создания нового объекта не происходит.

Если в методе вызывается метод базового класса с теми же параметрами, то можно использовать запись **inherited**, не указывая имя метода и параметры. Например, метод **Student.Print** можно записать таким образом:

```
procedure Print;  
begin  
    inherited;  
    writeln('Курс: ',course,'   Группа: ',group);  
end;
```

Наследование конструкторов

Правила наследования конструкторов - достаточно сложные. В разных языках программирования приняты разные решения на этот счет. В частности, в Delphi Object Pascal все конструкторы наследуются. В .NET, напротив, конструкторы не наследуются. Причина такого решения - каждый класс сам должен отвечать за инициализацию своих экземпляров. Единственное исключение в .NET - если класс вообще не определяет конструкторов, то автоматически генерируется конструктор без параметров, называемый конструктором по умолчанию.

В **PascalABC.NET** принято промежуточное решение. Если класс не определяет конструкторов, то все конструкторы предка автоматически генерируются в потомке, вызывая соответствующие конструкторы предка (можно также говорить, что они наследуются). Если в классе определяются конструкторы, то конструкторы предка не генерируются. Конструктор по умолчанию, если он явно не определен, генерируется автоматически в любом случае и является `protected`.

Кроме того, в .NET обязательно в конструкторе потомка первым оператором должен быть вызван конструктор предка; в Object Pascal это необязательно. Если в **PascalABC.NET** конструктор предка вызывается из конструктора потомка, то этот вызов должен быть первым оператором. Если конструктор предка явно не вызывается из конструктора потомка, то неявно первым оператором в конструкторе потомка вызывается конструктор предка по умолчанию (т.е. без параметров). Если такого конструктора у предка нет (это может быть класс, откомпилированный другим .NET-компилятором или входящий в стандартную библиотеку классов - все классы, откомпилированные **PascalABC.NET**, имеют конструктор по умолчанию), то возникает ошибка компиляции.

Например:

```
type   A = class
        i: integer;
        // конструктор по умолчанию не определен явно, поэтому
        генерируется автоматически
```

```
    constructor Create(i: integer);
begin
    Self.i := i;
end;
end;
B = class(A)
    j: integer;
    constructor Create;
    begin
        // конструктор по умолчанию базового класса вызывается
автоматически
        // конструктор по умолчанию определен явно, поэтому не
генерируется автоматически
        j := 1;
    end;
    constructor Create(i,j: integer);
    begin
        inherited Create(i);
        Self.j := j;
    end;
end;
C = class(B)
    // класс не определяет конструкторов, поэтому
    // конструктор по умолчанию и constructor Create(i,j:
integer)
    // генерируются автоматически, вызывая в своем теле
соответствующие конструкторы предка
    end;
```

Виртуальные методы и полиморфизм

Полиморфизм (от греч. "много форм") - это свойство классов, связанных наследованием, иметь различную реализацию входящих в них методов, и способность переменной базового класса вызывать методы того класса, объект которого содержится в этой переменной в момент вызова метода.

Полиморфизм используется в ситуации, когда для группы взаимосвязанных объектов требуется выполнить единое действие, но каждый из этих объектов должен выполнить указанное действие по-своему (т.е. у действия возникает много форм). Для этого определяется базовый для всех объектов класс с виртуальными методами, предусмотренными для меняющегося поведения, после чего эти методы переопределяются в потомках.

Для пояснения рассмотрим переопределение метода в подклассе:

```
type Base = class
  public
    procedure Print;
  begin
    writeln('Base');
  end;
end;
Derived = class(Base)
  public
    procedure Print;
  begin
    writeln('Derived');
  end;
end;
```

Присвоим переменной базового класса `Base` объект производного класса `Derived` и вызовем метод `Print`.

```
var b: Base := new Derived;
b.Print;
```

Какая версия метода `Print` вызывается - класса `Base` или класса `Derived`? В данном случае решение будет принято еще на этапе компиляции: вызовется метод `Print` класса `Base`, заявленного при

описании переменной `b`. Говорят, что имеет место **раннее связывание** имени метода с его телом. Если же решение о том, какой метод вызывать, принимается на этапе выполнения программы в зависимости от реального типа объекта, на который ссылается переменная `b`, то в данном случае вызывается метод `Derived.Print` (говорят также, что имеет место **позднее связывание**). Методы, для которых реализуется позднее связывание, называются **виртуальными**, а переменная базового класса, через которую осуществляется вызов виртуального метода, - **полиморфной переменной**. Таким образом, полиморфизм реализуется вызовом виртуальных функций через переменную базового класса. Тип класса, который хранится в данной переменной на этапе выполнения, называется **динамическим типом** этой переменной.

Для того чтобы сделать метод виртуальным, следует в объявлении этого метода после заголовка указать ключевое слово `virtual` с последующей `;`. Для переопределения виртуального метода следует использовать ключевое слово `override`:

```
type
  Base = class
  public
    procedure Print; virtual;
  begin
    writeln('Base');
  end;
end;
Derived = class(Base)
public
  procedure Print; override;
begin
  writeln('Derived');
end;
end;
```

Теперь в аналогичном участке кода.

```
var b: Base := new Derived;
b.Print;
```

вызывается метод `Print` класса `Derived` за счет того что решение о

вызове метода откладывается на этап выполнения программы.

Говорят, что методы `Print` завязаны в **цепочку виртуальности**. Чтобы разорвать ее (не вызывать методы в подклассах виртуально) используется ключевое слово **reintroduce**:

```
type
  DerivedTwice1 = class(Derived)
  public
    procedure Print; reintroduce;
  begin
    writeln('DerivedTwice1');
  end;
end;
```

Если мы хотим начать новую цепочку виртуальности, то следует использовать и **virtual** и **reintroduce**:

```
type
  DerivedTwice2 = class(Derived)
  public
    procedure Print; virtual; reintroduce;
  begin
    writeln('DerivedTwice2');
  end;
end;
```

Если переопределить виртуальную функцию не виртуальной без ключевого слова **reintroduce**, то ошибки не произойдет - будет выведено лишь предупреждение о том, что цепочка виртуальности нарушена. Таким образом, ключевое слово **reintroduce** в этой ситуации лишь подавляет вывод предупреждения.

При переопределении виртуального метода в подклассе его уровень доступа должен быть не ниже, чем в базовом классе. Например, **public** виртуальный метод не может быть переопределен в подклассе **private**-методом.

Абстрактные методы

Методы, предназначенные для переопределения в подклассах, объявляются с ключевым словом **abstract** и называются абстрактными. Данные методы являются [виртуальными](#), но ключевое слово **virtual** использовать не нужно. Например:

```
type Shape = class
  private
    x,y: integer;
  public
    constructor Create(xx,yy: integer);
  begin
    x := xx;
    y := yy;
  end;
  procedure Draw; abstract;
end;
```

Классы, содержащие абстрактные методы, также называются *абстрактными*. Экземпляры этих классов создавать нельзя.

Классы с абстрактными методами используются как "полуфабрикаты" для создания других классов. Например:

```
type
  Point = class(Shape)
  public
    procedure Draw; override;
  begin
    PitPixel(x,y,Color.Black);
  end;
end;
```

Использование **override** при переопределении абстрактных методов обязательно, поскольку абстрактные методы являются разновидностью виртуальных.

Класс можно явно объявить абстрактным, используя ключевое слово **abstract**. Обычно абстрактные классы содержат абстрактные методы, но необязательно:

```
type
  A = abstract class(Shape)
```

end;

Перегрузка операций

Перегрузка операций - это средство языка, позволяющее вводить операции над типами, определяемыми пользователем. В **PascalABC.NET** можно использовать только predefined значки операций. Перегрузка операций для типа **T**, являющегося классом или записью, осуществляется при помощи статической (классовой) функции-метода со специальным именем **operator** *ЗнакОперации*. Перегрузка специальных операций **+=**, **-=**, ***=**, **/=** осуществляется с помощью статической (классовой) процедуры-метода, первый параметр которой передается по ссылке.

Например:

```
type Complex = record
    re,im: real;
class function operator+(a,b: Complex): Complex;
begin
    Result.re := a.re + b.re;
    Result.im := a.im + b.im;
end;
class function operator=(a,b: Complex): boolean;
begin
    Result := (a.re = b.re) and (a.im = b.im);
end;
end;
```

Для перегрузки операций действуют следующие правила:

1. Перегружать можно все операции за исключением **@** (взятие адреса), **as**, **is**, **new**. Кроме того, можно перегружать специальные бинарные операции **+=**, **-=**, ***=**, **/=**, не возвращающие значений.
2. Перегружать можно только еще не перегруженные операции.
3. Тип по крайней мере одного операнда должен совпадать с типом класса или записи, внутри которого определена операция.
4. Перегрузка осуществляется с помощью статической функции-метода, количество параметров которой совпадает с количеством параметров соответствующей операции (2 - для бинарной, 1 - для унарной).
5. Перегрузка операций **+=**, **-=**, ***=**, **/=** для соответствующих

- операторов осуществляется с помощью статической процедуры-метода, первый параметр которой передается по ссылке и имеет тип записи или класса, в котором определяется данная операция, второй - передается по значению и совместим по присваиванию с первым. Перегрузка остальных операций осуществляется с помощью статических функций-методов.
6. Типы интерфейсов не могут быть типами параметров. Причина: типы параметров должны вычисляться на этапе компиляции.
 7. Операции приведения типа задаются статическими функциями, у которых вместо имени используется **operator implicit** (для неявного приведения типа) или **operator explicit** (для явного приведения типа).

Например:

```
type
  Complex = record
    ...
    class function operator implicit(d: real): Complex;
  begin
    Result.re := d;
    Result.im := 0;
  end;
  class function operator explicit(c: Complex):
string;
  begin
    Result := Format('{{0}},{{1}}',c.re,c.im);
  end;
  class procedure operator+=(var c: Complex; value:
Complex);
  begin
    c.re += value.re;
    c.im += value.im;
  end;
  class function operator+(c,c1: Complex): Complex;
  begin
    Result.re := c.re + c1.re;
    Result.im := c.im + c1.im;
  end;
end;
```

Можно перегружать операции с помощью методов расширения - в

этом случае при описании подпрограммы не следует писать слово `class`. Например, так в системном модуле реализовано добавление числа к строке:

```
function string.operator+(str: string; n: integer):  
string;  
begin  
    result := str + n.ToString;  
end;
```

Классовые поля, методы, свойства и конструкторы

В классе можно объявить так называемые *классовые (статические)* поля, свойства и методы. Они не принадлежат конкретному экземпляру класса, а связаны с классом. Для их вызова используется точечная нотация, причем, перед точкой используется не имя объекта, а имя класса. Чтобы поле или свойство или метод сделать классовым (статическим), перед его именем следует указать ключевое слово **class**. При описании статических свойств в секциях `read` и `write` можно указывать только статические поля или методы.

Например, определим для класса `Person` количество созданных объектов этого класса как статическое поле и организуем доступ к этому полю на чтение с помощью статической функции. После каждого вызова конструктора значение статического поля будет увеличиваться на 1:

```
type Person = class
  private
    name: string;
    age: integer;
    class cnt: integer := 0;
  public
    class property Coun: integer read cnt;
    constructor (n: string; a: integer);
    begin
      cnt += 1;
      name := n;
      age := a;
    end;
    class function Count: integer;
    begin
      Result := cnt;
    end;
  end;
begin
  var p: Person := new Person('Иванов',20);
  var p1: Person := new Person('Петров',18);
  writeln(Person.Count); // обращение к классовому методу
```

Count
end.

В отличие от классовых полей и методов, обычные поля и методы называются экземпляльными. Из обычных методов можно обращаться к экземплярным и классовым полям, но из классовых методов можно обращаться только к классовым полям.

Аналогично можно определить также классовой (статический) конструктор, предназначенный для автоматической инициализации классовых полей. Классовый конструктор описывается с ключевым словом **class** и гарантированно вызывается перед вызовом любого статического метода и созданием первого объекта этого класса.

Например, определим в классе **Person** классовое поле - массив объектов типа **Person** - и инициализируем его в классическом конструкторе. Потом указанный массив можно использовать в реализации классовой функции **RandomPerson**, возвращающей случайный объект типа **Person**:

```
type
  Person = class
  private
    class arr: array of Person;
    name: string;
    age: integer;
  public
    class constructor;
    begin
      SetLength(arr, 3);
      arr[0] := new Person('Иванов', 20);
      arr[1] := new Person('Петрова', 19);
      arr[2] := new Person('Попов', 35);
    end;
    //...
    class function RandomPerson: Person;
    begin
      Result := arr[Random(3)];
    end;
  end;
const cnt = 10;
begin
  var a := new Person[cnt];
```

```
for var i:=0 to a.Length-1 do
  a[i] := Person.RandomPerson;
end.
```

Методы расширения

Любой существующий тип, хранящийся во внешней dll, и все типы стандартной библиотеки .NET можно расширить новыми методами. Метод расширения определяется как процедура или функция с модификатором **extensionmethod**. Первый параметр метода расширения обязательно должен иметь имя Self и принадлежит к расширяемому типу. Сравним две процедуры, описанные ниже:

```
procedure MyPrint(Self: integer);  
begin  
    writeln(Self)  
end;  
  
procedure MyPrintEx(Self: integer); extensionmethod;  
begin  
    writeln(Self)  
end;  
  
begin  
    MyPrint(1);  
    1.MyPrintEx;  
end.
```

Здесь MyPrint - это обычная процедура с параметром типа integer, MyPrintEx - метод расширения типа integer. При этом при вызове первый параметр MyPrintEx становится объектом, вызывающим MyPrintEx как метод.

Можно расширить тип последовательности, тогда все классы, являющиеся последовательностями (динамические одномерные массивы, списки List<T>, множества HashSet<T> и SortedSet<T>), получают этот метод. Например, в системном модуле PABCSystem так введен метод расширения ForEach для последовательностей:

```
procedure &ForEach<T>(Self: sequence of T; action: T ->  
    ()); extensionmethod;  
begin  
    foreach x: T in Self do  
        action(x);  
end;
```

С помощью методов расширения можно [перегружать операции](#):

```
function operator += <T>(a: List<T>; x: T): List<T>;  
extensionmethod;  
begin  
    a.Add(x);  
    Result := a;  
end;
```

В этом случае первый параметр не обязан иметь имя Self.

Для методов расширения имеется ряд ограничений:

- Методы расширения не могут быть виртуальными.
- Если метод расширения имеет то же имя, что и обычный метод, то предпочтение отдаётся обычному методу.

Атрибуты

Текст находится в разработке

t

E

Анонимные классы

Иногда необходимо сгенерировать объект класса на лету, не описывая класс. У такого класса нет имени (он анонимный), но известен набор полей.

Объект анонимного класса создаётся следующим образом:

```
var p := new class(Name := 'Иванов', Age := 20);  
Println(p.Name, p.Age);
```

У объекта p автоматически генерируются публичные поля Name и Age соответствующих типов.

Два объекта принадлежат к одному анонимному классу если они имеют одинаковый набор полей, и эти поля принадлежат к одинаковым типам. Например:

```
var p1 := new class(Name := 'Петров', Age := 21);  
p1 := p;
```

Если поля безымянного класса инициализируются переменными, то имена полей можно не писать - они генерируются автоматически и их имена и типы совпадают с именами и типами переменных.

Например:

```
var Name := 'Попова';  
var Age := 23;  
var p := new class(Name, Age);  
Println(p.Name, p.Age);
```

Поля безымянного класса можно также инициализировать переменной с составным именем, имеющим точечную нотацию. В этом случае в качестве имен полей берутся последние имена в точечной нотации. Например:

```
var d := new DateTime(2015, 5, 15);  
var p := new class(d.Day, d.Month, d.Year);  
Println(p.Day, p.Month, p.Year);  
Println(p);
```

Автоклассы

При описании класса перед словом **class** можно поставить слово **auto**. Такие классы называются автоклассами. Для автоклассов автоматически генерируется конструктор с параметрами, инициализирующими все поля класса, а также метод `ToString`, выводящий значения всех полей класса. Например:

```
type Person = auto class
    name: string;
    age: integer;
end;
var p := new Person('Иванов', 20); // конструктор
    автокласса генерируется автоматически
writeln(p); // вызывается сгенерированный автоматически
    метод ToString
```

Здесь в отличие от действия `writeln` по умолчанию выводятся значения не только публичных, а всех полей.

Обработка исключений: обзор

Когда во время выполнения программы происходит ошибка, генерируется так называемое *исключение*, которое можно *перехватить* и *обработать*. Исключение представляет собой объект класса, производного от класса `Exception`, создающийся при возникновении исключительной ситуации.

Имеется ряд [стандартных типов исключений](#). Можно также определять [пользовательские типы исключений](#).

Если исключение не обработать, то программа завершится с ошибкой. Для обработки исключений используется оператор [try...except](#).

Обычно исключения возбуждаются в подпрограммах, поскольку разработчик подпрограммы, как правило, не знает, как обработать ошибочную ситуацию. В месте вызова подпрограммы уже, как правило, известно, каким образом следует обрабатывать исключение. Например, пусть разработана следующая функция:

```
function mymod(a,b: integer): integer;  
begin  
    Result := a - (a div b) * b;  
end;
```

Если вызвать `mymod(1, 0)`, то будет возбуждено исключение `System.DivideByZeroException` целочисленного деления на 0.

Рассмотрим наивную попытку обработать ошибочную ситуацию внутри функции `mymod`:

```
function mymod(a,b: integer): integer;  
begin  
    if b = 0 then  
        writeln('Функция mymod: деление на 0');  
    Result := a - (a div b) * b;  
end;
```

Подобное решение является плохим, поскольку программист, разрабатывающий функцию `mymod`, не знает, как она будет использоваться. Например, при вызове функции `mymod` в цикле мы увидим на экране многократное сообщение об ошибке.

Простейший способ - оставить исходный вариант функции и обрабатывать исключение `System.DivideByZeroException`:

```
try
  readln(a,b);
  writeln(mymod(a,b) mod (a-1));
  ...
except
  on System.DivideByZeroException do
    writeln('Деление на 0');
end;
```

Отличие от вывода внутри функции состоит в том, что при разработке программы мы сами определяем действие, которое необходимо выполнять при обработке исключения. Это может быть специфическое сообщение об ошибке, вывод в файл ошибок или пустой оператор (в случае, когда требуется "беззвучно" погасить исключение).

Однако, данное решение обладает существенным недостатком: исключение `System.DivideByZeroException` будет возбуждено и при $a=1$ и не будет связано с функцией `mymod`. Для устранения подобного недостатка определим собственный класс исключения и возбудим его в функции `mymod`:

```
type MyModErrorException = class(System.Exception)
end;

function mymod(a,b: integer): integer;
begin
  if b = 0 then
    raise new MyModErrorException('Функция mymod:
деление на 0');
  Result := a - (a div b) * b;
end;
```

Тогда обработка ошибок будет выглядеть так:

```
try
  readln(a,b);
  writeln(mymod(a,b) mod (a-1));
  ...
except
  on System.DivideByZeroException do
    writeln('Деление на 0');
```

```

    on e: MyModErrorException do
        writeln(e.Message);
    else writeln('какое-то другое исключение')
end;

```

Если сделать MyModErrorException наследником класса System.ArithmeticException, как и System.DivideByZeroException, то последний код можно упростить:

```

type MyModErrorException =
class(System.ArithmeticException) end;
...
try
    readln(a,b);
    writeln(mymod(a,b) mod (a-1));
    ...
except
    on e: System.ArithmeticException do
        writeln(e.Message);
    else writeln('Какое-то другое исключение')
end;

```

Наконец, можно поступить следующим образом. Перехватим в функции mymod исключение System.DivideByZeroException и в ответ сгенерируем новое - MyModErrorException:

```

function mymod(a,b: integer): integer;
begin
    try
        Result := a - (a div b) * b;
    except
        on e: System.DivideByZeroException do
            raise new MyModErrorException('Функция mymod:
деление на 0');
        end;
    end;
end;

```

Стандартные классы исключений

Все классы исключений являются потомками класса `System.Exception`, включающего следующий интерфейс:

```
type Exception = class
    public
        constructor Create;
        constructor Create(message: string);
        property Message: string; // только на чтение
        property StackTrace: string; // только на чтение
    end;
```

Свойство `Message` возвращает сообщение, связанное с объектом исключения.

Свойство `StackTrace` возвращает стек вызовов подпрограмм на момент генерации исключения.

Ниже приводятся некоторые классы исключений, определенные в пространстве имен `System` и являющиеся производными от класса `System.SystemException`:

- `System.OutOfMemoryException` - недостаточно памяти для выполнения программы;
- `System.StackOverflowException` - переполнение стека (как правило, при многократных вложенных вызовах подпрограмм);
- `System.AccessViolationException` - попытка доступа к защищенной памяти;
- `System.ArgumentException` - неверное значение параметра подпрограммы;
- `System.ArithmeticException` - базовый класс всех арифметических исключений. Наследники:
 - `System.DivideByZeroException` - целочисленное деление на 0;
 - `System.OverflowException` - переполнение при выполнении арифметической операции или преобразования типов;
- `System.FormatException` - неверный формат параметра (например, при преобразовании строки в число);

`System.IndexOutOfRangeException` - выход за границы диапазона изменения индекса массива;

`System.InvalidCastException` - неверное приведение типов;

`System.NullReferenceException` - попытка вызвать метод для нулевого объекта или разыменовать нулевой указатель;

`System.IO.IOException` - ошибка ввода-вывода. Наследники:

`System.IO.IOException.DirectoryNotFoundException` - каталог не найден;

`System.IO.IOException.EndOfStreamException` - попытка чтения за концом потока;

`System.IO.IOException.FileNotFoundException` - файл не найден.

Исключения, определяемые пользователем

Для определения своего типа исключения достаточно породить класс - наследник класса `Exception`:

```
type MyException = class(Exception) end;
```

Тело класса-исключения может быть пустым, но, тем не менее, новое имя для типа исключения позволит его разграничить с остальными исключениями:

```
try ...  
except  
  on MyException do  
    writeln('Целочисленное деление на 0');  
  on Exception do  
    writeln('Файл отсутствует');  
end;
```

Исключение может содержать дополнительную информацию, связанную с точкой, в которой произошло исключение:

```
type  
  FileNotFoundException = class(Exception)  
    fname: string;  
    constructor Create(msg,fn: string);  
    begin  
      inherited Create(msg);  
      fname := fn;  
    end;  
  end;  
...  
procedure ReadFile(fname: string);  
begin  
  if not FileExists(fname) then  
    raise new FileNotFoundException('Файл не  
найден', fname);  
  end;  
...  
try  
  ...  
except
```

```
on e: FileNotFoundException do
    writeln('Файл '+e.fname+' не найден');
end;
```

Повторная генерация исключения

Для повторной генерации исключения в блоке **except** служит [оператор raise](#) без параметров:

```
raise;
```

Например:

```
try ...
except
    on FileNotFoundException do
        begin
            log.WriteLine('Файл не найден'); // запись в файл
ошибок
            raise;
        end;
    end;
end;
```

Примеры обработки исключений

Пример 1. Обработка неверного ввода данных.

Рассмотрим программу.

```
var i: integer;
begin
  readln(i);
  writeln(i);
  writeln('Выполнение программы продолжается');
end.
```

Если при вводе данных произойдет ошибка (например, мы введем не число), то программа завершится с ошибкой (ошибка ввода), и последующие операторы `writeln` не будут выполнены.

Перехватим исключение в блоке `try`:

```
var i: integer;
begin
  try
    readln(i);
    writeln(i);
  except
    writeln('Ошибка ввода');
  end;
  writeln('Выполнение программы продолжается');
end.
```

На этот раз при возникновении ошибки ввода программа не будет завершена, а выполнение будет передано в блок `except`, после чего выполнение программы продолжится дальше. Таким образом, в последней программе не выполнится лишь оператор `writeln(i)`.

Если в блоке `try` могут возникнуть различные исключения, то обычно используется вторая форма блока `except` с несколькими обработчиками исключений.

Пример 2. Обработка различных исключений.

```
var a,b: integer;
assign(f, 'a.txt');
try
  readln(a,b);
```

```

    reset(f);
    c:=a div b;
except
    on System.DivideByZeroException do
        writeln('Целочисленное деление на 0');
    on System.IO.IOException do
        writeln('Файл отсутствует');
end;

```

Часто необходимо совмещать обработку исключений и освобождение ресурсов независимо от того, произошло исключение или нет. В этом случае используются вложенные операторы **try ... except** и **try ... finally**.

Пример 3. Вложенные операторы **try ... except** и **try ... finally**.

```

assign(f, 'a.txt');
try
    reset(f);
    try
        try
            c:=a div b;
        except
            on System.DivideByZeroException do
                writeln('Целочисленное деление на 0');
            end;
        finally
            close(f);
        end;
    except
        on System.IO.IOException do
            writeln('Файл отсутствует');
        end;
end;

```

Обратим внимание, что в данном примере исключение, связанное с целочисленным делением на 0, обрабатывается в самом внутреннем блоке **try**, а исключение, связанное с отсутствующим файлом - в самом внешнем. При этом, если файл был открыт, то независимо от возникновения исключения деления на 0 он будет закрыт.

Интерфейсы: обзор

Интерфейс - это тип данных, содержащий набор заголовков методов и свойств, предназначенных для реализации некоторым классом. Интерфейсы описываются в разделе **type** следующим образом:

```
ИмяИнтерфейса = interface объявления методов и  
свойств  
end;
```

Для метода приводится только заголовок, для свойства после возвращаемого типа указываются необходимые модификаторы доступа `read` и `write`.

Например:

```
type  
  IShape = interface  
    procedure Draw;  
    property X: integer read;  
    property Y: integer read;  
  end;  
  ICloneable = interface  
    function Clone: Object;  
  end;
```

Поля и статические методы не могут входить в интерфейс.

Класс реализует интерфейс, если он реализует все методы и свойства интерфейса в **public**-секции. Если класс не реализует хотя бы один метод или свойство интерфейса, возникает ошибка компиляции. Класс может реализовывать также несколько интерфейсов. Список реализуемых интерфейсов указывается в скобках после ключевого слова **class** (если указано имя предка, то после имени предка).

Например:

```
type  
  Point = class(IShape, ICloneable)  
    private  
      xx,yy: integer;  
    public
```

```

constructor Create(x,y: integer);
begin
    xx := x; yy := y;
end;
procedure Draw; begin end;
property X: integer read xx;
property Y: integer read yy;
function Clone: Object;
begin
    Result := new Point(xx,yy);
end;
procedure Print;
begin
    write(xx, ' ',yy);
end;
end;

```

Интерфейсы можно наследовать друг от друга:

```

type
    IPosition = interface
        property X: integer read;
        property Y: integer read;
    end;
    IDrawable = interface
        procedure Draw;
    end;
    IShape = interface(IPosition, IDrawable)
    end;

```

Интерфейс по-существу представляет собой абстрактный класс без реализации входящих в него методов. Для интерфейсов, в частности, применимы все правила [приведения типов объектов](#): тип объекта, реализующего интерфейс, может быть неявно приведен к типу интерфейса, а обратное преобразование производится только явно и может вызвать исключение при невозможности преобразования:

```

var ip: IShape := new Point(20,30);
ip.Draw;
Point(ip).Print;

```

Все методы класса, реализующего интерфейс, являются виртуальными без использования ключевых слов **virtual** или

override. В частности, `ip.Draw` вызовет метод `Draw` класса `Point`. Однако, цепочка виртуальности таких методов обрывается. Чтобы продолжить цепочку виртуальности методов, реализующих интерфейс, в подклассах, следует использовать ключевое слово **virtual**:

```
type
  Point = class(IShape, ICloneable)
    ...
    function Clone: Object; virtual;
  begin
    Result := new Point(xx,yy);
  end;
end;
```

Для интерфейсов, как и для классов, можно также использовать операции **is** и **as**:

```
if ip is Point then
  ...
var p: Point := ip as Point;
if p<>nil then
  writeln('Преобразование успешно');
```

Реализация нескольких интерфейсов

Несколько интерфейсов могут содержать одинаковые методы или свойства. При наследовании от таких интерфейсов такие одинаковые методы или свойства сливаются в один:

```
type IShape = interface
    procedure Draw;
    property X: integer read;
    property Y: integer read;
end;
IBrush = interface
    procedure Draw;
    property Size: integer read;
end;
Brush = class(IShape, IBrush)
    // метод Draw реализуется единожды
end;
```

Чтобы решить проблему с одинаковыми именами в интерфейсах, в .NET классы могут реализовывать методы интерфейсов так называемым явным образом, так что вызов метода интерфейса для переменной класса возможен только после явного приведения к типу интерфейса. В **PascalABC.NET** такие классы определять нельзя, однако, пользоваться такими классами, реализованными в .NET, можно. Например, тип `integer` явно реализует интерфейс `IComparable`:

```
var i: integer := 1;
var res : integer := IComparable(i).CompareTo(2);
// i.CompareTo(2) - ошибка компиляции
```

Обобщенные типы: обзор

Обобщенным типом (generic) называется шаблон для создания класса, записи или интерфейса, параметризованный одним или несколькими типами. Класс (запись, интерфейс) образуется из шаблона класса (записи, интерфейса) подстановкой конкретных типов в качестве параметров. Параметры указываются после имени обобщенного типа в угловых скобках. Например, `Stack<T>` - шаблон класса списка элементов типа `T`, параметризованный типом `T`, а `Stack<integer>` - класс списка с элементами типа `integer`.

[Обобщённые подпрограммы описываются здесь.](#)

Для объявления шаблона класса используется следующий синтаксис:

```
type   Node<T> = class
    data: T;
    next: Node<T>;
public
    constructor Create(d: T; nxt: Node<T>);
    begin
        data := d;
        next := nxt;
    end;
end;
Stack<T> = class
    tp: Node<T>;
public
    procedure Push(x: T);
    begin
        tp := new Node<T>(x, tp);
    end;
    function Pop: T;
    begin
        Result := tp.data;
        tp := tp.next;
    end;
    function Top: T;
    begin
        Result := tp.data;
    end;
```

```
function IsEmpty: boolean;  
begin  
    Result := tp = nil;  
end;  
end;
```

Использование шаблона класса иллюстрируется ниже:

```
var  
    si: Stack<integer>;  
    sr: Stack<real>;  
begin  
    si := new Stack<integer>;  
    sr := new Stack<real>;  
    for var i := 1 to 10 do  
        si.Push(Random(100));  
    while not si.IsEmpty do  
        sr.Push(si.Pop);  
    while not sr.IsEmpty do  
        write(sr.Pop, ' ');  
end.
```

Подстановка конкретного типа-параметра в обобщенный тип называется инстанцированием.

Обобщенные подпрограммы: обзор

Обобщенной подпрограммой (generic) называется подпрограмма, параметризованная одним или несколькими типами. Подпрограмма образуется из обобщенной подпрограммы подстановкой конкретных типов в качестве параметров. Параметры указываются после имени подпрограммы в угловых скобках.

Например, следующая обобщённая функция параметризована одним параметром:

```
function FindFirstInArray<T>(a: array of T; val: T):
integer;
begin
  Result := -1;
  for var i:=0 to a.Length-1 do
    if a[i]=val then
      begin
        Result := i;
        exit;
      end;
  end;
var x: array of string;
begin
  SetLength(x,4);
  x[0] := 'Ваня';
  x[1] := 'Коля';
  x[2] := 'Сереза';
  x[3] := 'Саша';
  writeln(FindFirstInArray(x, 'Сереза'));
end.
```

При вызове обобщенной подпрограммы тип-параметр обобщения можно не указывать, поскольку компилятор **выводит** типы параметров шаблона по типам фактических параметров. В данном случае после выведения получено: T=string.

При выведении требуется точное соответствие типов, приведение типов не допускается. Например, при компиляции следующего кода

```
...
var x: array of real;
```

```

begin
  SetLength(x, 3);
  x[0] := 1;
  x[1] := 2.71;
  x[2] := 3.14;
  writeln(FindFirstInArray(x, 1));
end.

```

произойдет ошибка. Причина состоит в том, что первый параметр имеет тип `array of real`, а второй - тип `integer`, что не соответствует ни одному типу `T` в заголовке обобщенной функции. Для решения проблемы следует либо изменить тип второго параметра на `real`:

```
FindFirstInArray(x, 1.0)
```

либо явно после имени функции в угловых скобках указать имя типа, которым параметризован данный вызов:

```
FindFirstInArray<real>(x, 1)
```

Использование знака `&` здесь обязательно, поскольку в противном случае компилятор трактует знак `<` как "меньше".

Обобщёнными могут быть не только обычные подпрограммы, но и методы классов, а также методы другого обобщённого класса. Например:

```

type
  Pair<T, Q> = class
    first: T;
    second: Q;
    function ChangeSecond<S>(newval: S): Pair<T, S>;
  end;

function Pair<T, Q>.ChangeSecond<S>(newval: S):
Pair<T, S>;
begin
  result := new Pair<T, S>;
  result.first := first;
  result.second := newval;
end;

var
  x: Pair<integer, real>;
  y: Pair<integer, string>;
begin

```

```
x := new Pair<integer, real>;  
x.first := 3;  
y := x.ChangeSecond('abc');  
writeln(y.first, y.second);  
end.
```

По окончании работы данная программа выведет 3abc.

Обобщенные подпрограммы в качестве параметров

Обобщенная подпрограмма может выступать в качестве формального параметра другой обобщенной подпрограммы.

Например, в классе `System.Array` имеется несколько статических обобщенных методов с обобщенными подпрограммами в качестве параметров. Так, `System.Array.Find` имеет следующий прототип:

```
System.Array.FindAll<T>(a: array of T; pred: Predicate<T>): array of T;
```

и возвращает подмассив массива `a` элементов `T`, удовлетворяющих условию `pred`.

Приведем пример вызова этой функции:

```
function f(x: integer): boolean;
begin
    Result := ;
end;

var a := Seq(1,3,6,5,8);
var b := System.Array.FindAll(a, x -> x mod 2 = 0);
```

Здесь возвращается массив `b`, содержащий все четные значения массива `a` в том же порядке.

Ограничения на параметры обобщенных подпрограмм и классов

По умолчанию с переменными, имеющими тип параметра обобщенного класса или подпрограммы, внутри методов обобщенных классов и обобщенных подпрограмм можно делать лишь ограниченный набор действий: присваивать и сравнивать на равенство (отметим, что в .NET сравнение на равенство внутри обобщений запрещено!).

Например, данный код будет работать:

```
function Eq<T>(a,b: T): boolean;  
begin  
    Result := a = b;  
end;
```

Можно также использовать присваивание переменной, имеющей тип параметра обобщенного класса или подпрограммы, значение по умолчанию, используя конструкцию **default(T)** - значение по умолчанию для типа T (*nil* для ссылочных типов и нулевое значение для размерных типов):

```
procedure Def<T>(var a: T);  
begin  
    a := default(T);  
end;
```

Однако, данный код

```
function Sum<T>(a,b: T): T;  
begin  
    Result := a + b;  
end;
```

вызовет ошибку компиляции до инстанцирования (создания экземпляра с конкретным типом). Такое поведение в .NET кардинально отличается от шаблонов в C++, где в коде шаблона можно использовать любые операции с шаблонными параметрами, и ошибка может произойти только в момент инстанцирования с конкретным типом.

Чтобы разрешить использование некоторых действий с

переменными, имеющими тип параметра обобщенного класса или подпрограммы, используются ограничения на обобщенные параметры, задаваемые в секции **where** после заголовка подпрограммы или класса:

```
type
  MyPair<T> = class
    where T: System.ICloneable;
  private
    x,y: T;
  public
    constructor (x,y: T);
    begin
      Self.x := x;
      Self.y := y;
    end;
    function Clone: MyPair;
    begin
      Result := new MyPair<T>(x.Clone,y.Clone);
    end;
  end;
```

В секции **where** через запятую перечисляются следующие ограничения:

На 1 месте: слово **class** или слово **record** или имя класса-предка.

На 2 месте: список реализуемых интерфейсов через запятую.

На 3 месте: слово **constructor**, указывающее, что данный тип должен иметь конструктор по умолчанию.

При этом каждое из мест, кроме одного, может быть пустым.

Для каждого типа-параметра может быть своя секция **where**, каждая секция **where** завершается точкой с запятой.

Пример. Обобщенная функция поиска минимального элемента в массиве. Элементы должны реализовывать интерфейс **IComparable<T>**.

```
function MinElem<T>(a: array of T): T;
  where T: IComparable<T>;
begin
  var min := a[0];
  for var i := 1 to a.High do
```

```
    if a[i].CompareTo(min)<0 then
        min := a[i];
    Result := min;
end;
```

К сожалению, нет возможности использовать запись $a[i] < \text{min}$, поскольку операции не входят в интерфейсы.

Лямбда-выражения

Лямбда-выражение - это выражение специального вида, которое на этапе компиляции заменяется на имя подпрограммы, соответствующей лямбда-выражению и генерируемой компилятором "на лету".

Здесь излагается полный [синтаксис лямбда-выражений](#).

Здесь рассказывается о [захвате лямбда-выражением переменных](#) из внешнего контекста.

Лямбда-выражения запрещается использовать при инициализации полей класса или записи, внутри вложенных подпрограмм, в подпрограмме при наличии вложенной подпрограммы, в разделе инициализации модуля.

Лямбда-выражения запрещается использовать совместно с метками `label` и оператором `goto` в одной подпрограмме.

Синтаксис лямбда-выражений достаточно сложен и в данном пункте иллюстрируется на примерах.

Пример 1.

```
var f: integer -> integer := x -> x*x;  
f(2);
```

Запись `x -> x` является лямбда-выражением, представляющим собой функцию с одним параметром `x` типа `integer`, возвращающую `x*x` типа `integer`. По данной записи компилятор генерирует следующий код:

```
function #fun1(x: integer): integer;  
begin  
  Result := x*x;  
end;  
...  
var f: integer -> integer := #fun1;  
f(2);
```

Здесь `#fun1` - это имя, генерируемое компилятором. Кроме того, код функции `#fun1` также генерируется компилятором.

Пример 2. Фильтрация четных

Обычно лямбда-выражение передаётся как параметр подпрограммы. Например, в следующем коде

```
var a := Seq(3, 2, 4, 8, 5, 5);  
a.Where(x -> x mod 2 = 0).Print;
```

лямбда-выражение `x -> x mod 2 = 0` задаёт условие отбора чётных чисел из массива `a`.

Пример 3. Сумма квадратов

```
var a := Seq(1, 3, 5);  
writeln(a.Aggregate(0, (s, x) -> s + x * x));
```

Иногда необходимо явно задавать тип параметров в лямбда-выражении.

Пример 4. Выбор перегруженной версии процедуры с параметром-лямбдой.

```
procedure p(f: integer -> integer);  
begin  
  write(f(1));  
end;  
  
procedure p(f: real -> real);  
begin  
  write(f(2.5));  
end;  
  
begin  
  p((x: real) -> x * x);  
end.
```

В данном примере вызов `p(x -> x)` вызовет ошибку компиляции, потому что компилятор не может выбрать, какую версию процедуры `p` выбирать. Задание типа параметра лямбды помогает устранить эту неоднозначность.

Пример 5. Лямбда-процедура.

```
procedure p(a: integer -> ());  
begin  
  a(1)  
end;  
  
begin
```

```
p(procedure(x) -> write(x));  
end.
```

Захват переменных в лямбда-выражении

Лямбда-выражение может использовать переменные из внешнего контекста. Такие переменные называются захваченными лямбда-выражением.

Пример 1. Захват переменной в запросе Select.

```
begin var a := Seq(2,3,4);  
    var z := 1;  
    var q := a.Select(x->x+z);  
    q.Println;  
    z := 2;  
    q.Println;  
end.
```

Здесь лямбда-выражение `x->x+z` захватывает внешнюю переменную `z`. Важно заметить, что при изменении значения переменной `z` запрос `a.Select(x->x+z)`, хранящийся в переменной `q`, выполняется с новым значением `z`.

Пример 2. Накопление суммы во внешней переменной.

```
begin  
    var sum := 0;  
    var AddToSum: integer -> () := procedure (x) -> begin  
sum += x; end;  
  
    AddToSum(1);  
    AddToSum(3);  
    AddToSum(5);  
  
    writeln(sum);  
end.
```

Методы последовательностей

Все [последовательности](#) имеют множество методов обработки последовательностей, реализованных как [методы расширения](#).

Список методов последовательностей

- [Методы Print](#) (только PascalABC.NET)
- [Метод фильтрации Where](#)
- [Метод проецирования Select](#)
- [Метод проецирования SelectMany](#)
- [Методы Take, TakeWhile, Skip, SkipWhile](#)
- [Методы Sorted, SortedDescending](#) (только PascalABC.NET)
- [Методы OrderBy, OrderByDescending](#)
- [Методы ThenBy, ThenByDescending](#)
- [Метод ForEach](#) (только PascalABC.NET)
- [Метод Concat](#)
- [Метод JoinIntoString](#) (только PascalABC.NET)
- [Метод Zip](#)
- [Метод Distinct](#)
- [Методы Union, Intersect, Except](#)
- [Метод Reverse](#)
- [Метод SequenceEqual](#)
- [Методы First, FirstOrDefault](#)
- [Методы Last, LastOrDefault](#)
- [Методы Single, SingleOrDefault](#)
- [Метод DefaultIfEmpty](#)
- [Методы ElementAt, ElementAtOrDefault](#)
- [Методы Any, All](#)
- [Методы Count](#)
- [Метод Contains](#)
- [Метод Aggregate](#)
- [Методы Sum, Average](#)
- [Методы Min, Max](#)
- [Метод Join](#)
- [Метод GroupJoin](#)
- [Метод GroupBy](#)
- [Метод AsEnumerable](#)
- [Методы ToArray, ToList](#)
- [Метод ToDictionary](#)
- [Метод ToLookup](#)
- [Метод OfType](#)

- [Метод Cast](#)

Методы Print

Описание методов

Методы приведены для последовательности **sequence of T**.

function Print(delim: string := ' '): **sequence of T**;

Выводит последовательность на экран, используя delim в качестве разделителя.

function Println(delim: string := ' '): **sequence of T**;

Выводит последовательность на экран, используя delim в качестве разделителя, и переходит на новую строку.

Пример

```
begin  
  var a := Arr(1,3,5);  
  a.Println;  
  ReadLines('a.txt').Println(NewLine);  
end.
```

Метод фильтрации Where

Описание методов

Методы приведены для последовательности **sequence of T**.
function where(predicate: T->boolean): **sequence of T**;

Выполняет фильтрацию последовательности значений на основе заданного предиката. Возвращает подпоследовательность значений исходной последовательности, удовлетворяющих предикату.

function where(predicate: (T,integer)->boolean): **sequence of T**;

Выполняет фильтрацию последовательности значений на основе заданного предиката с учётом индекса элемента. Возвращает подпоследовательность значений исходной последовательности, удовлетворяющих предикату.

Пример

```
begin  
  var a := Arr(1,2,3,5,6);  
  a.Where(x -> x mod 2 = 0).Println; // 2 6  
end.
```

Метод проецирования Select

Описание методов

Методы приведены для последовательности **sequence of T**.

function Select<Res>(selector: T->Res): **sequence of Res**;

Проецирует каждый элемент последовательности на другой элемент с помощью функции selector. Возвращает последовательность элементов, полученных в результате проецирования.

function Select<Res>(selector: (T,integer)->Res): **sequence of Res**;

Проецирует каждый элемент последовательности на другой элемент с помощью функции selector, учитывающую индекс элемента. Возвращает последовательность элементов, полученных в результате проецирования.

Пример

```
begin  
  var a := Arr(1,2,3,4,5,6);  
  a.Select(x -> x*x).Println; // 1 4 9 16 25 36  
end.
```

Метод проецирования SelectMany

Описание методов

Методы приведены для последовательности **sequence of T**.

function SelectMany<Res>(selector: T->**sequence of Res**): **sequence of Res**; Проецирует каждый элемент последовательности в новую последовательность и объединяет результирующие последовательности в одну последовательность. Возвращает объединённую последовательность.

function SelectMany<Res>(selector: (T,integer)->**sequence of Res**): **sequence of Res**;

Проецирует каждый элемент последовательности в новую последовательность с учетом индекса элемента и объединяет результирующие последовательности в одну последовательность. Возвращает объединённую последовательность.

function SelectMany<Coll,Res>(collSelector: (T,integer)->**sequence of Coll**; resultSelector: (T,Coll)->Res): **sequence of Res**;

Проецирует каждый элемент последовательности в новую последовательность, объединяет результирующие последовательности в одну и вызывает функцию селектора результата для каждого элемента этой последовательности. Индекс каждого элемента исходной последовательности используется в промежуточной проецированной форме этого элемента. Возвращает объединённую последовательность.

function SelectMany<Coll,Res>(collSelector: T->**sequence of Coll**; resultSelector: (T,Coll)->Res): **sequence of Res**;

Проецирует каждый элемент последовательности в новую последовательность, объединяет результирующие последовательности в одну и вызывает функцию селектора результата для каждого элемента этой последовательности. Возвращает объединённую последовательность.

Пример

```
begin  
  var a := Arr(Arr(1,2,3),Arr(4,5,6),Arr(7,8,9));  
  a.SelectMany(x -> x).Println; // 1 2 3 4 5 6 7 8 9  
end.
```

Методы Take, TakeWhile, Skip, SkipWhile

Описание методов

Методы приведены для последовательности **sequence of T**.

function Take(count: integer): **sequence of T**; Возвращает последовательность из count элементов с начала последовательности.

function TakeWhile(predicate: T->boolean): **sequence of T**;

Возвращает цепочку элементов последовательности, удовлетворяющих указанному условию, до первого не удовлетворяющего.

function TakeWhile(predicate: (T,integer)->boolean): **sequence of T**;

Возвращает цепочку элементов последовательности, удовлетворяющих указанному условию, до первого не удовлетворяющего (учитывается индекс элемента).

function Skip(count: integer): **sequence of T**;

Пропускает count элементов в последовательности и возвращает остальные элементы.

function SkipWhile(predicate: T->boolean): **sequence of T**;

Пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы.

function SkipWhile(predicate: (T,integer)->boolean): **sequence of T**;

Пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы (учитывается индекс элемента).

Пример

begin

```
var a := Arr(1,2,3,4,5,6);
```

```
a.Take(3).Println; // 1 2 3
```

```
a.Skip(3).Println; // 4 5 6
```

```
a.Skip(2).Take(3).Println; // 3 4 5
```

```
a.TakeWhile(x -> x<3).Println; // 1 2
```

```
a.SkipWhile(x -> x<5).Println; // 5 6
```

end.

Методы Sorted, SortedDescending

Описание методов

Методы приведены для последовательности **sequence of T**.

function Sorted(): sequence of T; Возвращает отсортированную по возрастанию последовательность.

function SortedDescending(): sequence of T;
Возвращает отсортированную по убыванию последовательность.

Пример

```
begin  
  var a := Arr(6,2,7,4,8,1);  
  a.Sorted.Println; // 1 2 4 6 7 8  
  a.SortedDescending.Println; // 8 7 6 4 2 1  
end.
```

Методы OrderBy, OrderByDescending

Описание методов

Методы приведены для последовательности **sequence of T**.

function OrderBy<Key>(keySelector: T->Key):

System.Linq.IOrderedEnumerable<T>; Сортирует элементы

последовательности в порядке возрастания ключа и возвращает

отсортированную последовательность. keySelector - функция,

проектирующая элемент на ключ.

function OrderBy<Key>(keySelector: T->Key; comparer:

IComparer<Key>): System.Linq.IOrderedEnumerable<T>;

Сортирует элементы последовательности в порядке возрастания с

использованием компаратора comparer и возвращает отсортированную

последовательность. keySelector - функция, проектирующая элемент на

ключ.

function OrderByDescending<Key>(keySelector: T->Key):

System.Linq.IOrderedEnumerable<T>;

Сортирует элементы последовательности в порядке убывания ключа и

возвращает отсортированную последовательность. keySelector - функция,

проектирующая элемент на ключ.

function OrderByDescending<Key>(keySelector: T->Key;

comparer: IComparer<Key>):

System.Linq.IOrderedEnumerable<T>;

Сортирует элементы последовательности в порядке убывания с

использованием компаратора comparer и возвращает отсортированную

последовательность. keySelector - функция, проектирующая элемент на

ключ.

Пример

```
begin
  var a := Arr(('Иванов',20),('Попов',21),
('Авилов',28));
  a.OrderBy(t -> t[0]).Println;           //
(Авилов,28) (Иванов,20) (Попов,21)
  a.OrderByDescending(t -> t[1]).Println; //
(Авилов,28) (Попов,21) (Иванов,20)
end.
```

Методы ThenBy, ThenByDescending

Описание методов

Методы приведены для последовательности **sequence of T**.

function ThenBy<Key>(keySelector: T->Key):

System.Linq.IOrderedEnumerable<T>; Выполняет дополнительное упорядочение элементов последовательности в порядке возрастания ключа и возвращает отсортированную последовательность. keySelector - функция, проектирующая элемент на ключ.

function ThenBy<Key>(keySelector: T->Key; comparer: IComparer<Key>): System.Linq.IOrderedEnumerable<T>;

Выполняет дополнительное упорядочение элементов последовательности в порядке возрастания с использованием компаратора comparer и возвращает отсортированную последовательность. keySelector - функция, проектирующая элемент на ключ.

function ThenByDescending<Key>(keySelector: T->Key): System.Linq.IOrderedEnumerable<T>;

Выполняет дополнительное упорядочение элементов последовательности в порядке убывания ключа и возвращает отсортированную последовательность. keySelector - функция, проектирующая элемент на ключ.

function ThenByDescending<Key>(keySelector: T->Key; comparer: IComparer<Key>): System.Linq.IOrderedEnumerable<T>;

Выполняет дополнительное упорядочение элементов последовательности в порядке убывания с использованием компаратора comparer и возвращает отсортированную последовательность. keySelector - функция, проектирующая элемент на ключ.

Пример

```
begin  
  var a := Arr(('Иванов', 20), ('Попов', 21),  
    ('Иванов', 18), ('Авилов', 28), ('Иванов', 25));  
  a.OrderBy(t -> t[0]).ThenBy(t -> t[1]).Println;  
  // (Авилов,28) (Иванов,18) (Иванов,20) (Иванов,25)  
  (Попов,21)  
end.
```

Метод Concat

Описание методов

Методы приведены для последовательности **sequence of T**.
function Concat(second: **sequence of T**): **sequence of T**;

Соединяет две последовательности, дописывая вторую в конец первой и возвращая результирующую последовательность.

Пример

```
begin  
  var a1 := Lst(2,3,5);  
  var a2 := Seq(4,7,8);  
  a1.Concat(a2).Println; // 2 3 5 4 7 8  
end.
```

Метод Zip

Описание методов

Методы приведены для последовательности **sequence of T**.
function Zip<TSecond, Res>(second: **sequence of TSecond**;
resultSelector: (T, TSecond)->Res): **sequence of Res**;

Объединяет две последовательности, используя указанную функцию, принимающую по одному элементу каждой последовательности и возвращающую элемент результирующей последовательности.

Пример

```
begin  
  var a := Arr(1,2,3);  
  var b := Lst(4,5,6);  
  a.Zip(b,(x,y) -> x+y).Println; // 5 7 9  
end.
```

Метод Distinct

Описание методов

Методы приведены для последовательности **sequence of T**.

function Distinct(): sequence of T; Возвращает различающиеся элементы последовательности.

function Distinct(comparer: IEqualityComparer<T>): sequence of T;

Возвращает различающиеся элементы последовательности, используя для сравнения значений компаратор comparer.

Пример

```
begin  
  var a := Arr('aaa', 'bbb', 'ccc', 'aaa', 'ccc');  
  a.Distinct.Println; // aaa bbb ccc  
end.
```

Методы Union, Intersect, Except

Описание методов

Методы приведены для последовательности **sequence of T**.
function Union(second: **sequence of T**): **sequence of T**;

Находит объединение множеств, представленных двумя последовательностями.

function Union(second: **sequence of T**; comparer: **IEqualityComparer<T>**): **sequence of T**;

Находит объединение множеств, представленных двумя последовательностями, используя указанный компаратор.

function Intersect(second: **sequence of T**): **sequence of T**;

Находит пересечение множеств, представленных двумя последовательностями.

function Intersect(second: **sequence of T**; comparer: **IEqualityComparer<T>**): **sequence of T**;

Находит пересечение множеств, представленных двумя последовательностями, используя для сравнения значений указанный компаратор.

function Except(second: **sequence of T**): **sequence of T**;

Находит разность множеств, представленных двумя последовательностями.

function Except(second: **sequence of T**; comparer: **IEqualityComparer<T>**): **sequence of T**;

Находит разность множеств, представленных двумя последовательностями, используя для сравнения значений указанный компаратор.

Пример

```
begin  
  var a := Range(1,5);  
  var b := Range(3,7);  
  a.Union(b).Println;    // 1 2 3 4 5 6 7  
  a.Intersect(b).Println; // 3 4 5  
  a.Except(b).Println;  // 1 2  
end.
```

Метод Reverse

Описание методов

Методы приведены для последовательности **sequence of T**.
function Reverse(): sequence of T; Возвращает инвертированную последовательность.

Пример

```
begin  
  var a := Range(1,9);  
  a.Reverse.Println; // 9 8 7 6 5 4 3 2 1  
end.
```

Метод SequenceEqual

Описание методов

Методы приведены для последовательности **sequence of T**.

function SequenceEqual(second: **sequence of T**): boolean;

Определяет, совпадают ли две последовательности.

function SequenceEqual(second: **sequence of T**; comparer: IEqualityComparer<T>): boolean;

Определяет, совпадают ли две последовательности, используя для сравнения элементов указанный компаратор.

Пример

```
begin  
  var a := Arr(1,2,3);  
  var b := Lst(1,2,3);  
  a.SequenceEqual(b);  
end.
```

Методы First, FirstOrDefault

Описание методов

Методы приведены для последовательности **sequence of T**.

function First(): T; Возвращает первый элемент последовательности.

function First(predicate: T->boolean): T;

Возвращает первый элемент последовательности, удовлетворяющий указанному условию.

function FirstOrDefault(): T;

Возвращает первый элемент последовательности или значение по умолчанию, если последовательность не содержит элементов.

function FirstOrDefault(predicate: T->boolean): T;

Возвращает первый удовлетворяющий условию элемент последовательности или значение по умолчанию, если ни одного такого элемента не найдено.

Пример

```
begin  
  var a := Arr(1,2,3,4);  
  Println(a.Skip(2).First); // 3  
  Println(a.First(x -> x mod 2 = 0)); // 2  
  Println(a.FirstOrDefault(x -> x>5)); // 0  
end.
```

Методы Last, LastOrDefault

Описание методов

Методы приведены для последовательности **sequence of T**.

function Last(): T; Возвращает последний элемент последовательности.

function Last(predicate: T->boolean): T;

Возвращает последний элемент последовательности, удовлетворяющий указанному условию.

function LastOrDefault(): T;

Возвращает последний элемент последовательности или значение по умолчанию, если последовательность не содержит элементов.

function LastOrDefault(predicate: T->boolean): T;

Возвращает последний элемент последовательности, удовлетворяющий указанному условию, или значение по умолчанию, если ни одного такого элемента не найдено.

Пример

```
begin  
  var a := Arr(1,2,3,4);  
  Println(a.Last); // 4  
  Println(a.Last(x -> x mod 2 = 0)); // 4  
  Println(a.LastOrDefault(x -> x>5)); // 0  
end.
```

Методы Single, SingleOrDefault

Описание методов

Методы приведены для последовательности **sequence of T**.

function Single(): T; Возвращает единственный элемент последовательности и генерирует исключение, если число элементов последовательности отлично от 1.

function Single(predicate: T->boolean): T;

Возвращает единственный элемент последовательности, удовлетворяющий заданному условию, и генерирует исключение, если таких элементов больше одного.

function SingleOrDefault(): T;

Возвращает единственный элемент последовательности или значение по умолчанию, если последовательность пуста; если в последовательности более одного элемента, генерируется исключение.

function SingleOrDefault(predicate: T->boolean): T;

Возвращает единственный элемент последовательности, удовлетворяющий заданному условию, или значение по умолчанию, если такого элемента не существует; если условию удовлетворяет более одного элемента, генерируется исключение.

Пример

```
begin  
  var a := Arr(1,2,3,4);  
  Println(a.Single); // исключение  
  Println(a.Single(x -> x>3)); // 4  
  Println(a.SingleOrDefault(x -> x>5)); // 0  
end.
```

Метод `DefaultIfEmpty`

Описание методов

Методы приведены для последовательности `sequence of T`.

function `DefaultIfEmpty(): sequence of T`; Возвращает элементы указанной последовательности или одноэлементную коллекцию, содержащую значение параметра типа по умолчанию, если последовательность пуста.

function `DefaultIfEmpty(defaultValue: T): sequence of T`;
Возвращает элементы указанной последовательности или одноэлементную коллекцию, содержащую указанное значение, если последовательность пуста.

Пример

```
begin  
  var a := Arr(1,2,3,4);  
  a.Skip(4).DefaultIfEmpty.Println; // 0  
end.
```

Методы `ElementAt`, `ElementAtOrDefault`

Описание методов

Методы приведены для последовательности **sequence of T**.

function `ElementAt(index: integer): T;` Возвращает элемент по указанному индексу в последовательности.

function `ElementAtOrDefault(index: integer): T;`

Возвращает элемент по указанному индексу в последовательности или значение по умолчанию, если индекс вне допустимого диапазона.

Пример

```
begin  
  var a := Arr(1,2,3,4);  
  Println(a.ElementAt(2)); // 3  
  Println(a.ElementAtOrDefault(10)); // 0  
end.
```

Методы Any, All

Описание методов

Методы приведены для последовательности **sequence of T**.

function Any(): boolean; Проверяет, содержит ли последовательность какие-либо элементы.

function Any(predicate: T->boolean): boolean;

Проверяет, удовлетворяет ли какой-либо элемент последовательности заданному условию.

function All(predicate: T->boolean): boolean;

Проверяет, все ли элементы последовательности удовлетворяют условию.

Пример

```
begin  
  var a := Lst(1,3,5);  
  Println(a.All(x -> x mod 2 <> 0)); // True  
  Println(a.Any(x -> x mod 2 = 0));  // False  
end.
```

Методы Count

Описание методов

Методы приведены для последовательности **sequence of T**.

function Count(): integer; Возвращает количество элементов в последовательности.

function Count(predicate: T->boolean): integer;

 Возвращает число, представляющее количество элементов последовательности, удовлетворяющих заданному условию.

function LongCount(): int64;

 Возвращает значение типа Int64, представляющее общее число элементов в последовательности.

function LongCount(predicate: T->boolean): int64;

 Возвращает значение типа Int64, представляющее число элементов последовательности, удовлетворяющих заданному условию.

Пример

```
begin
```

```
  var a := Lst(1,3,5,6);
```

```
  Println(a.Count(x -> x mod 2 <> 0)); // 3
```

```
end.
```

Метод Contains

Описание методов

Методы приведены для последовательности **sequence of T**.

function Contains(value: T): boolean; Определяет, содержится ли указанный элемент в последовательности, используя компаратор проверки на равенство по умолчанию.

function Contains(value: T; comparer: IEqualityComparer<T>): boolean;

Определяет, содержит ли последовательность заданный элемент, используя указанный компаратор.

Пример

```
begin  
  var a := Lst(1,3,5,6);  
  Println(a.Contains(666)); // False  
  Println(666 in a); // False  
end.
```

Метод Aggregate

Описание методов

Методы приведены для последовательности **sequence of T**.

function Aggregate(func: (T,T)->T): T; Применяет к последовательности агрегатную функцию. Возвращает конечное агрегатное значение.

function Aggregate<Accum>(seed: T; func: (Accum,T)->Accum): T;

Применяет к последовательности агрегатную функцию. Указанное начальное значение используется в качестве исходного значения агрегатной операции. Возвращает конечное агрегатное значение.

function Aggregate<Accum,Res>(seed: T; func: (Accum,T)->Accum; resultSelector: Accum->Res): T;

Применяет к последовательности агрегатную функцию. Указанное начальное значение служит исходным значением для агрегатной операции, а указанная функция используется для выбора результирующего значения. Возвращает конечное агрегатное значение.

Пример

```
begin  
  var a := Seq(2,3,5,6);  
  Println(a.Aggregate(1, (p,x) -> p*x));  
end.
```

Методы Sum, Average

Описание методов

Методы приведены для последовательности **sequence of T**.

function Sum(): число; Вычисляет сумму последовательности значений числового типа.

function Sum(selector: T->число): число;

Вычисляет сумму последовательности значений числового типа, получаемой в результате применения функции преобразования к каждому элементу входной последовательности.

function Average(): real;

Вычисляет среднее для последовательности значений числового типа.

function Average(selector: T->число): real;

Вычисляет среднее для последовательности значений числового типа, получаемой в результате применения функции преобразования к каждому элементу входной последовательности.

Пример

```
begin  
  var a := Lst(1,3,5,6);  
  Println(a.Sum);  
  var b := Arr(('Иванов',20),('Попов',21),  
('Авилов',28));  
  Println(b.Average(x -> x[1]));  
end.
```

Методы Min, Max

Описание методов

Методы приведены для последовательности **sequence of T**.

function Min(): число; Вычисляет минимальный элемент последовательности значений числового типа.

function Min(selector: T->число): число;

Вызывает функцию преобразования для каждого элемента последовательности и возвращает минимальное значение числового типа.

function Max(): число;

Вычисляет максимальный элемент последовательности значений числового типа.

function Max(selector: T->число): число;

Вызывает функцию преобразования для каждого элемента последовательности и возвращает максимальное значение числового типа.

Пример

```
begin  
  var a := Lst(1,3,5,6);  
  Println(a.Min, a.Max);  
  var b := Arr(('Иванов',20), ('Попов',21),  
('Авилов',28));  
  Println(b.Min(x -> x[1]));  
end.
```

Методы Join

Описание методов

Методы приведены для последовательности **sequence of T**.

```
function Join<TInner,Key,Res>(inner: sequence of TInner;  
outerKeySelector: T->Key; innerKeySelector: TInner->TKey;  
resultSelector: (T,TInner)->Res): sequence of Res;
```

Объединяет две последовательности на основе сопоставления ключей в третью последовательность. Функция resultSelector задаёт проекцию элементов двух последовательностей с одинаковыми значениями ключа в элемент третьей последовательности.

```
function Join<TInner,Key,Res>(inner: sequence of TInner;  
outerKeySelector: T->Key; innerKeySelector: TInner->TKey;  
resultSelector: (T,TInner)->Res; comparer:  
System.Collections.Generic.IEqualityComparer<Key>):  
sequence of Res;
```

Объединяет две последовательности на основе сопоставления ключей в третью последовательность. Функция resultSelector задаёт проекцию элементов двух последовательностей с одинаковыми значениями ключа в элемент третьей последовательности. Для сравнения ключей используется компаратор comparer.

Пример

```
begin
  var people := Arr((1, 'Иванов'), (2, 'Попов'),
    (3, 'Сидоров'));
  var subjects := Arr((1, 'История'), (1, 'Математика'),
    (2, 'История')
    , (3, 'Математика'), (1, 'Русский'), (2, 'Физика'));

  people.Join(subjects, p->p[0], s->s[0], (p, s)->
    (p[1], s[1])).Println(NewLine);
end.
```

Вывод:

```
(Иванов, История)
(Иванов, Математика)
(Иванов, Русский)
(Попов, История)
(Попов, Физика)
(Сидоров, Математика)
```

Метод GroupJoin

Описание методов

Методы приведены для последовательности **sequence of T**.

```
function GroupJoin<TInner,Key,Res>(inner: sequence of  
TInner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultSelector: (T,sequence of TInner)->Res):
```

```
sequence of Res;    Объединяет две последовательности на основе равенства ключей и группирует результаты. Затем функция resultSelector проектирует ключ и последовательность соответствующих ему значений на элемент результирующей последовательности.
```

```
function GroupJoin<TInner,Key,Res>(inner: sequence of  
TInner; outerKeySelector: T->Key; innerKeySelector: TInner->TKey; resultSelector: (T,sequence of TInner)->Res;  
comparer: IEqualityComparer<Key>): sequence of Res;
```

Объединяет две последовательности на основе равенства ключей и группирует результаты. Для сравнения ключей используется указанный компаратор. Затем функция resultSelector проектирует ключ и последовательность соответствующих ему значений на элемент результирующей последовательности.

Пример

```
begin
  var people := Arr((1, 'Иванов'), (2, 'Попов'),
(3, 'Сидоров'));
  var subjects := Arr((1, 'История'), (1, 'Математика'),
(2, 'История')
, (3, 'Математика'), (1, 'Русский'), (2, 'Физика'));

  people.GroupJoin(subjects, p->p[0], s->s[0], (p, ss)->
(p[1], ss.Select(x->x[1]))).Println(NewLine);
end.
```

Вывод:

```
(Иванов, [История, Математика, Русский])
(Попов, [История, Физика])
(Сидоров, [Математика])
```

Метод GroupBy

Описание методов

Методы приведены для последовательности **sequence of T**.

function GroupBy<Key>(keySelector: T->Key):

IEnumerable<IGrouping<Key, T>>; Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и возвращает последовательность групп; каждая группа соответствует одному значению ключа.

function GroupBy<Key>(keySelector: T->Key; comparer: System.Collections.Generic.IEqualityComparer<Key>): IEnumerable<IGrouping<Key, T>>;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа, сравнивает ключи с помощью указанного компаратора и возвращает последовательность групп; каждая группа соответствует одному значению ключа.

function GroupBy<Key, Element>(keySelector: T->Key; elementSelector: T->Element): IEnumerable<IGrouping<Key, T>>;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и проецирует элементы каждой группы с помощью указанной функции. Возвращает последовательность групп; каждая группа соответствует одному значению ключа.

function GroupBy<Key, Element>(keySelector: T->Key; elementSelector: T->Element; comparer: IEqualityComparer<Key>): IEnumerable<IGrouping<Key, Element>>;

Группирует элементы последовательности в соответствии с функцией селектора ключа. Ключи сравниваются с помощью компаратора, элементы каждой группы проецируются с помощью указанной функции.

function GroupBy<Key, Res>(keySelector: T->Key; resultSelector: (Key, **sequence of T**)->Res): **sequence of Res**;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и создает результирующее значение для каждой группы и ее ключа.

function GroupBy<Key, Element, Res>(keySelector: T->Key; elementSelector: T->Element; resultSelector: (Key, **sequence**

of Element)->Res): sequence of Res;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и создает результирующее значение для каждой группы и ее ключа. Элементы каждой группы проецируются с помощью указанной функции.

function GroupBy<Key, Res>(keySelector: T->Key;
resultSelector: (Key, **sequence of** T)->Res; comparer:
IEqualityComparer<Key>): **sequence of** Res;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и создает результирующее значение для каждой группы и ее ключа. Ключи сравниваются с использованием заданного компаратора.

function GroupBy<Key, Element, Res>(keySelector: T->Key;
elementSelector: System.T->Element; resultSelector:
(Key, **sequence of** Element)->Res; comparer:
IEqualityComparer<Key>): **sequence of** Res;

Группирует элементы последовательности в соответствии с заданной функцией селектора ключа и создает результирующее значение для каждой группы и ее ключа. Значения ключей сравниваются с помощью указанного компаратора, элементы каждой группы проецируются с помощью указанной функции.

Пример

```
begin
  var a := Arr(('Иванов',3),('Попов',1),('Авилов',1),
    ('Козлов',3),('Ослов',2),('Рогов',1));
  var groups := a.GroupBy(s->s[1]);

  foreach var g in groups do
    begin
      Print(g.Key+':');
      g.Select(x->x[0]).Println;
    end;
end.
```

Вывод:

```
3: Иванов Козлов
1: Попов Авилов Рогов
2: Ослов
```

Метод AsEnumerable

Описание методов

Методы приведены для последовательности **sequence of T**.

function AsEnumerable(): **sequence of T**; Возвращает входные данные, приведенные к типу IEnumerable.

Пример

```
function Print<T>(Self: array of T): array of T;  
extensionmethod;  
begin  
    Self.AsEnumerable.Print;  
    Result := Self;  
end;  
  
begin  
    Arr(1,2,3).Print  
end.
```

Методы ToArray, ToList

Описание методов

Методы приведены для последовательности **sequence of T**.

function ToArray(): array of T; Создает массив из последовательности.

function ToList(): List<T>;

Создает список List из последовательности.

Пример

```
begin  
  var a := Arr(1,2,3);  
  a := a.Select(x->x*x).ToArray;  
  var l := Lst(1,2,3);  
  l := l.Select(x->x*x).ToList;  
end.
```

Метод ToDictionary

Описание методов

Методы приведены для последовательности **sequence of T**.

function ToDictionary<Key>(keySelector: T->Key):

Dictionary<Key, T>; Создает словарь Dictionary из

последовательности соответствии с заданной функцией селектора ключа.

function ToDictionary<Key>(keySelector: T->Key; comparer:

IEqualityComparer<Key>): Dictionary<Key, T>;

Создает словарь Dictionary из последовательности в соответствии с заданной функцией селектора ключа и компаратором ключей.

function ToDictionary<Key, Element>(keySelector: T->Key;

elementSelector: T->Element): Dictionary<Key, Element>;

Создает словарь Dictionary из последовательности в соответствии с заданными функциями селектора ключа и селектора элемента.

function ToDictionary<Key, Element>(keySelector: T->Key;

elementSelector: T->Element; comparer:

IEqualityComparer<Key>): Dictionary<Key, Element>;

Создает словарь Dictionary из последовательности в соответствии с заданным компаратором и функциями селектора ключа и селектора элемента.

Пример

```
begin  
  var a := Arr(('крокодил',3),('бегемот',1),  
('тигр',2));  
  var d := a.ToDictionary(x->x[1],x->x[0]);  
  d.Println; // (3,крокодил) (1,бегемот) (2,тигр)  
end.
```

Метод ToLookup

Описание методов

Методы приведены для последовательности **sequence of T**.

function ToLookup<Key>(keySelector: T->Key):

System.Linq.ILookup<Key, T>; Создает объект System.Linq.Lookup из последовательности в соответствии с заданной функцией селектора ключа.

function ToLookup<Key>(keySelector: T->Key; comparer: IEqualityComparer<Key>): System.Linq.ILookup<Key, T>;

Создает объект System.Linq.Lookup из последовательности в соответствии с заданной функцией селектора ключа и компаратором ключей.

function ToLookup<Key, Element>(keySelector: T->Key; elementSelector: T->Element):

System.Linq.ILookup<Key, Element>;

Создает объект System.Linq.Lookup из последовательности в соответствии с заданными функциями селектора ключа и селектора элемента.

function ToLookup<Key, Element>(keySelector: T->Key; elementSelector: T->Element; comparer: IEqualityComparer<Key>): System.Linq.ILookup<Key, Element>;

Создает объект System.Linq.Lookup из последовательности в соответствии с заданным компаратором и функциями селектора ключа и селектора элемента.

Пример

Без примера

Метод OfType

Описание методов

Методы приведены для последовательности **sequence of T**.
function OfType<Res>(): **sequence of Res**; Выполняет фильтрацию элементов объекта System.Collections.IEnumerable по заданному типу. Возвращает подпоследовательность данной последовательности, в которой все элементы принадлежат заданному типу.

Пример

```
begin  
  var a := new object[](1,2.5, 'd', 'ff', 3.4);  
  a.OfType<real>().Println;  
end.
```

Метод Cast

Описание методов

Методы приведены для последовательности **sequence of T**.
function Cast<Res>(): **sequence of Res**; Преобразовывает
элементы объекта System.Collections.IEnumerable в заданный тип.

Пример

```
begin  
  var a: sequence of integer;  
  var b: sequence of real;  
  a := Seq(1,3,5);  
  b := a.Cast<real>();  
end.
```

Метод JoinIntoString

Описание методов

Методы приведены для последовательности **sequence of T**.

```
function JoinIntoString(delim: string := ' '): string;
```

Преобразует элементы последовательности в строковое представление, после чего объединяет их в строку, используя `delim` в качестве разделителя.

Пример

```
begin  
  var a := Arr('aaa','bbb','ccc');  
  var s: string := a.JoinIntoString('');  
  Println(s); // aaabbbccc  
end.
```

Управление памятью

Все ссылочные типы в .NET находятся под управлением так называемого **сборщика мусора**. Это значит, что выделенная вызовом конструктора память никогда не возвращается явно вызовом деструктора. После того как объект становится не нужным, ему следует присвоить `nil`.

При нехватке динамической памяти выполнение программы приостанавливается, и запускается специальная процедура, называемая сборкой мусора. Она определяет все так называемые достижимые объекты. Если на данный объект более никто не указывает, то он считается недостижимым и будет собран сборщиком мусора. Время вызова сборщика мусора считается неопределенным.

Например, при выполнении участка кода

```
type Person = class
    ...
end;
var p: Person := new Person('Иванов', 20);
...
p := nil;
```

память, отведенная под `p`, после присваивания ей `nil` станет недостижимой и будет собрана в непредсказуемый момент.

Отметим, что динамическая память, выделяемая процедурой `New`, не находится под управлением сборщика мусора, поэтому нуждается в явном освобождении вызовом процедуры `Dispose`. Именно поэтому работа с обычными указателями считается в **PascalABC.NET** устаревшей и не рекомендуется к использованию.

Обзор системного модуля PABCSystem

Модуль PABCSystem называется **системным** и автоматически подключается первым к любой программе или модулю. Он содержит ряд процедур, функций, констант, типов, методов расширения, перегруженных операций.

Стандартные типы и константы

- [Стандартные константы](#)
- [Стандартные типы](#)

Стандартные подпрограммы

- [Общие подпрограммы](#)
- [Математические подпрограммы](#)
- [Подпрограммы ввода](#)
- [Подпрограммы вывода](#)
- [Общие подпрограммы для работы с файлами](#)
- [Подпрограммы для работы с текстовыми файлами](#)
- [Подпрограммы для работы с двоичными файлами](#)
- [Подпрограммы для работы с именами файлов](#)
- [Системные подпрограммы](#)
- [Подпрограммы для работы с символами](#)
- [Подпрограммы для работы со строками](#)
- [Подпрограммы для работы со стандартными множествами](#)
- [Подпрограммы для работы с динамическими массивами](#)

Генерация объектов структурированных типов

- [Подпрограммы для генерации последовательностей](#)
- [Подпрограммы для генерации бесконечных последовательностей](#)
- [Подпрограммы для генерации динамических массивов](#)
- [Подпрограммы для генерации матриц](#)
- [Подпрограммы для создания кортежей](#)
- [Короткие функции Lst, HSet, SSet, Dict, KV](#)

Методы расширения, определенные в модуле PABCSystem

- [Методы расширения типа sequence of T](#)
- [Методы расширения типа array of T](#)
- [Методы расширения типа array \[,\] of T](#)
- [Методы расширения типа List<T>](#)
- [Методы расширения типа integer](#)
- [Методы расширения типа BigInteger](#)
- [Методы расширения типа real](#)
- [Методы расширения типа char](#)
- [Методы расширения типа string](#)
- [Методы расширения типа Func](#)
- [Методы расширения типа Complex](#)
- [Методы расширения типа IDictionary](#)

Методы типов файлов

- [Общие методы файловых типов](#)
- [Методы текстовых файлов](#)
- [Методы типизированных файлов](#)
- [Методы двоичных файлов](#)
- [Методы расширения типизированных файлов](#)

Стандартные константы

`E = 2.718281828459045;` Константа E

`MaxByte = byte.MaxValue;`
Максимальное значение типа byte

`MaxDouble = real.MaxValue;`
Максимальное значение типа double

`MaxInt = integer.MaxValue;`
Максимальное значение типа integer

`MaxInt64 = int64.MaxValue;`
Максимальное значение типа int64

`MaxLongWord = longword.MaxValue;`
Максимальное значение типа longword

`MaxReal = real.MaxValue;`
Максимальное значение типа real

`MaxShortInt = shortint.MaxValue;`
Максимальное значение типа shortint

`MaxSingle = single.MaxValue;`
Максимальное значение типа single

`MaxSmallInt = smallint.MaxValue;`
Максимальное значение типа smallint

`MaxUInt64 = uint64.MaxValue;`
Максимальное значение типа uint64

`MaxWord = word.MaxValue;`
Максимальное значение типа word

`MinDouble = real.Epsilon;`
Минимальное положительное значение типа double

`MinReal = real.Epsilon;`
Минимальное положительное значение типа real

`MinSingle = single.Epsilon;`
Минимальное положительное значение типа single

`NewLine = System.Environment.NewLine;`
Константа перехода на новую строку

`Pi = 3.141592653589793;`
Константа Pi

Стандартные типы

`Action<T> = System.Action<T>;` Представляет действие с одним параметром

`Action0 = System.Action;`
Представляет действие без параметров

`Action2<T1, T2> = System.Action<T1, T2>;`
Представляет действие с двумя параметрами

`Action3<T1, T2, T3> = System.Action<T1, T2, T3>;`
Представляет действие с тремя параметрами

`BigInteger = System.Numerics.BigInteger;`
Представляет произвольно большое целое число

`cardinal = System.UInt32;`
`cardinal = longword`

`Comparer<T> = System.Collections.Generic.Comparer<T>;`
Представляет базовый класс для реализации интерфейса `IComparer`

`Complex = System.Numerics.Complex;`
Представляет комплексное число

`DateTime = System.DateTime;`
Представляет дату и время

`decimal = System.Decimal;`
Представляет 128-битное вещественное число

`Dictionary<Key, Value> =`
`System.Collections.Generic.Dictionary<Key, Value>;`
Представляет ассоциативный массив (набор пар Ключ-Значение), реализованный на базе хеш-таблицы

`double = System.Double;`
`double = real`

`Encoding = System.Text.Encoding;`
Тип кодировки символов

`Exception = System.Exception;`
Базовый тип исключений

`Func<T, Res> = System.Func<T, Res>;`
Представляет функцию с одним параметром

`Func0<Res> = System.Func<Res>;`

Представляет функцию без параметров

```
Func2<T1, T2, Res> = System.Func<T1, T2, Res>;
```

Представляет функцию с двумя параметрами

```
Func3<T1, T2, T3, Res> = System.Func<T1, T2, T3, Res>;
```

Представляет функцию с тремя параметрами

```
HashSet<T> = System.Collections.Generic.HashSet<T>;
```

Представляет множество значений, реализованное на базе хеш-таблицы

```
ICollection<T> = System.Collections.Generic.ICollection<T>;
```

Представляет интерфейс для коллекции

```
IComparable<T> = IComparable<T>;
```

Представляет базовый класс для реализации интерфейса IComparer

```
IComparer<T> = System.Collections.Generic.IComparer<T>;
```

Представляет интерфейс для сравнения двух элементов

```
IDictionary<Key, Value> =  
System.Collections.Generic.IDictionary<Key, Value>;
```

Представляет интерфейс для набора пар Ключ-Значение

```
IEnumerable<T> = System.Collections.Generic.IEnumerable<T>;
```

Представляет интерфейс, предоставляющий перечислитель для перебора элементов коллекции

```
IEnumerator<T> = System.Collections.Generic.IEnumerator<T>;
```

Представляет интерфейс для перебора элементов коллекции

```
IEqualityComparer<T> =  
System.Collections.Generic.IEqualityComparer<T>;
```

Представляет интерфейс для поддержки сравнения на равенство

```
ICollection<T> = System.Collections.Generic.ICollection<T>;
```

Представляет интерфейс для коллекции с доступом по индексу

```
IntFunc = Func<integer, integer>;
```

Представляет функцию с одним параметром целого типа, возвращающую целое

```
ISet<T> = System.Collections.Generic.ISet<T>;
```

Представляет интерфейс для множества

```
KeyValuePair<Key, Value> =  
System.Collections.Generic.KeyValuePair<Key, Value>;
```

Представляет пару Ключ-Значение для ассоциативного массива

`LinkedList<T> = System.Collections.Generic.LinkedList<T>;`

Представляет двусвязный список

`LinkedListNode<T> =`

`System.Collections.Generic.LinkedListNode<T>;`

Представляет узел двусвязного списка

`List<T> = System.Collections.Generic.List<T>;`

Представляет список на базе динамического массива

`longint = System.Int32;`

longint = integer

`Match = System.Text.RegularExpressions.Match;`

Представляет результаты из отдельного совпадения регулярного выражения

`MatchCollection =`

`System.Text.RegularExpressions.MatchCollection;`

Представляет набор успешных совпадений регулярного выражения

`MatchEvaluator =`

`System.Text.RegularExpressions.MatchEvaluator;`

Представляет метод, вызываемый при обнаружении совпадения в `Regex.Replace`

`Object = System.Object;`

Базовый тип объектов

`Predicate<T> = System.Predicate<T>;`

Представляет функцию с одним параметром, возвращающую `boolean`

`Predicate2<T1, T2> = function(x1: T1; x2: T2): boolean;`

Представляет функцию с двумя параметрами, возвращающую `boolean`

`Predicate3<T1, T2, T3> = function(x1: T1; x2: T2; x3: T3): boolean;`

Представляет функцию с тремя параметрами, возвращающую `boolean`

`Queue<T> = System.Collections.Generic.Queue<T>;`

Представляет очередь - набор элементов, реализованных по принципу "первый вошел-первый вышел"

`RealFunc = Func<real, real>;`

Представляет функцию с одним параметром вещественного типа, возвращающую вещественное

`Regex = System.Text.RegularExpressions.Regex;`

Представляет регулярное выражение

`RegexGroup = System.Text.RegularExpressions.Group;`
Представляет результаты из одной группы при выполнении `Regex.Match`

`RegexGroupCollection = System.Text.RegularExpressions.GroupCollection;`
Представляет результаты из набора групп при выполнении `Regex.Match`

`RegexOptions = System.Text.RegularExpressions.RegexOptions;`
Представляет параметры регулярного выражения

`ShortString = string[255];`
Представляет тип короткой строки фиксированной длины 255 символов

`SortedDictionary<Key, Value> = System.Collections.Generic.SortedDictionary<Key, Value>;`
Представляет ассоциативный массив, реализованный на базе бинарного дерева поиска

`SortedList<Key, Value> = System.Collections.Generic.SortedList<Key, Value>;`
Представляет ассоциативный массив (набор пар ключ-значение), реализованный на базе динамического массива пар

`SortedSet<T> = System.Collections.Generic.SortedSet<T>;`
Представляет множество значений, реализованное на базе бинарного дерева поиска

`Stack<T> = System.Collections.Generic.Stack<T>;`
Представляет стек - набор элементов, реализованных по принципу "последний вошел-первый вышел"

`StringBuilder = System.Text.StringBuilder;`
Представляет изменяемую строку символов

`StringFunc = Func<string, string>;`
Представляет функцию с одним параметром строкового типа, возвращающую строку

`Tuple = System.Tuple;`
Представляет кортеж

Общие подпрограммы

procedure Dec(**var** i: integer); Уменьшает значение
переменной i на 1

procedure Dec(**var** i: integer; n: integer);
 Уменьшает значение переменной i на n

procedure Dec(**var** e: перечислимый тип);
 Уменьшает значение перечислимого типа на 1

procedure Dec(**var** e: перечислимый тип; n: integer);
 Уменьшает значение перечислимого типа на n

function Eof: boolean;
 Возвращает True, если достигнут конец потока ввода

function Eoln: boolean;
 Возвращает True, если достигнут конец строки

procedure Inc(**var** i: integer);
 Увеличивает значение переменной i на 1

procedure Inc(**var** i: integer; n: integer);
 Увеличивает значение переменной i на n

procedure Inc(**var** e: перечислимый тип);
 Увеличивает значение перечислимого типа на 1

procedure Inc(**var** e: перечислимый тип; n: integer);
 Увеличивает значение перечислимого типа на n

function Ord(a: целое): целое;
 Возвращает порядковый номер значения a

function Ord(a: перечислимый тип): integer;
 Возвращает порядковый номер значения a

function Pred(x: целое): целое;
 Возвращает предшествующее x значение

function Pred(x: перечислимый тип): перечислимый тип;
 Возвращает предшествующее x значение

function Succ(x: целое): целое;
 Возвращает следующее за x значение

function Succ(x: перечислимый тип): перечислимый тип;
 Возвращает следующее за x значение

procedure Swap<T>(var a, b: T);

Меняет местами значения двух переменных

Подпрограммы ввода

procedure Read(a, b, ...); Вводит значения a, b, ... с клавиатуры

procedure Read(f: файл; a, b, ...);

Вводит значения a, b, ... из файла f

function ReadBoolean: boolean;

Возвращает значение типа boolean, введенное с клавиатуры

function ReadBoolean(prompt: string): boolean;

Выводит приглашение к вводу и возвращает значение типа boolean, введенное с клавиатуры

function ReadBoolean(f: Text): boolean;

Возвращает значение типа boolean, введенное из текстового файла f

function ReadChar: char;

Возвращает значение типа char, введенное с клавиатуры

function ReadChar(prompt: string): char;

Выводит приглашение к вводу и возвращает значение типа char, введенное с клавиатуры

function ReadChar(f: Text): char;

Возвращает значение типа char, введенное из текстового файла f

function ReadChar2: (char, char);

Возвращает кортеж из двух значений типа char, введенных с клавиатуры

function ReadChar2(prompt: string): (char, char);

Возвращает кортеж из двух значений типа char, введенных с клавиатуры

function ReadChar3: (char, char, char);

Возвращает кортеж из трёх значений типа char, введенных с клавиатуры

function ReadChar3(prompt: string): (char, char, char);

Возвращает кортеж из трёх значений типа char, введенных с клавиатуры

function ReadInteger: integer;

Возвращает значение типа integer, введенное с клавиатуры

function ReadInteger(prompt: string): integer;

Выводит приглашение к вводу и возвращает значение типа integer,

введенное с клавиатуры

```
function ReadInteger(f: Text): integer;
```

Возвращает значение типа integer, введенное из текстового файла f

```
function ReadInteger2: (integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры

```
function ReadInteger2(prompt: string): (integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры

```
function ReadInteger3: (integer, integer, integer);
```

Возвращает кортеж из трёх значений типа integer, введенных с клавиатуры

```
function ReadInteger3(prompt: string): (integer, integer, integer);
```

Возвращает кортеж из трёх значений типа integer, введенных с клавиатуры

```
procedure Readln(a, b, ...);
```

Вводит значения a, b, ... с клавиатуры и осуществляет переход на следующую строку

```
procedure Readln(f: Text; a, b, ...);
```

Вводит значения a, b, ... из текстового файла f и осуществляет переход на следующую строку

```
function ReadlnBoolean: boolean;
```

Возвращает значение типа boolean, введенное с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnBoolean(prompt: string): boolean;
```

Выводит приглашение к вводу и возвращает значение типа boolean, введенное с клавиатуры, и осуществляет переход на следующую строку ввода

```
function ReadlnBoolean(f: Text): boolean;
```

Возвращает значение типа boolean, введенное из текстового файла f, и осуществляет переход на следующую строку

```
function ReadlnChar: char;
```

Возвращает значение типа char, введенное с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnChar(prompt: string): char;
```

Выводит приглашение к вводу и возвращает значение типа char, введенное с клавиатуры,и осуществляет переход на следующую строку ввода

```
function ReadLnChar(f: Text): char;
```

Возвращает значение типа char, введенное из текстового файла f,и осуществляет переход на следующую строку

```
function ReadLnChar2: (char, char);
```

Возвращает кортеж из двух значений типа char, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnChar2(prompt: string): (char, char);
```

Возвращает кортеж из двух значений типа char, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnChar3: (char, char, char);
```

Возвращает кортеж из двух значений типа char, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnChar3(prompt: string): (char, char, char);
```

Возвращает кортеж из двух значений типа char, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnInteger: integer;
```

Возвращает значение типа integer, введенное с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnInteger(prompt: string): integer;
```

Выводит приглашение к вводу и возвращает значение типа integer, введенное с клавиатуры,и осуществляет переход на следующую строку ввода

```
function ReadLnInteger(f: Text): integer;
```

Возвращает значение типа integer, введенное из текстового файла f,и осуществляет переход на следующую строку

```
function ReadLnInteger2: (integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnInteger2(prompt: string): (integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnInteger3: (integer, integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnInteger3(prompt: string): (integer, integer, integer);
```

Возвращает кортеж из двух значений типа integer, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnReal: real;
```

Возвращает значение типа real, введенное с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnReal(prompt: string): real;
```

Выводит приглашение к вводу и возвращает значение типа real, введенное с клавиатуры, и осуществляет переход на следующую строку ввода

```
function ReadlnReal(f: Text): real;
```

Возвращает значение типа real, введенное из текстового файла f, и осуществляет переход на следующую строку

```
function ReadlnReal2: (real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnReal2(prompt: string): (real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnReal3: (real, real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnReal3(prompt: string): (real, real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnString: string;
```

Возвращает значение типа string, введенное с клавиатуры, и переходит на следующую строку ввода

```
function ReadlnString(prompt: string): string;
```

Выводит приглашение к вводу и возвращает значение типа string, введенное с клавиатуры, и осуществляет переход на следующую строку ввода

```
function ReadlnString(f: Text): string;
```

Возвращает значение типа string, введенное из текстового файла f, и осуществляет переход на следующую строку

```
function ReadLnString2: (string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnString2(prompt: string): (string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnString3: (string, string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadLnString3(prompt: string): (string, string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры, и переходит на следующую строку ввода

```
function ReadReal: real;
```

Возвращает значение типа real, введенное с клавиатуры

```
function ReadReal(prompt: string): real;
```

Выводит приглашение к вводу и возвращает значение типа real, введенное с клавиатуры

```
function ReadReal(f: Text): real;
```

Возвращает значение типа real, введенное из текстового файла f

```
function ReadReal2: (real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры

```
function ReadReal2(prompt: string): (real, real);
```

Возвращает кортеж из двух значений типа real, введенных с клавиатуры

```
function ReadReal3: (real, real, real);
```

Возвращает кортеж из трёх значений типа real, введенных с клавиатуры

```
function ReadReal3(prompt: string): (real, real, real);
```

Возвращает кортеж из трёх значений типа real, введенных с клавиатуры

```
function ReadString: string;
```

Возвращает значение типа string, введенное с клавиатуры

```
function ReadString(prompt: string): string;
```

Выводит приглашение к вводу и возвращает значение типа string, введенное с клавиатуры

```
function ReadString(f: Text): string;
```

Возвращает значение типа string, введенное из текстового файла f

```
function ReadString2: (string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры

```
function ReadString2(prompt: string): (string, string);
```

Возвращает кортеж из двух значений типа string, введенных с клавиатуры

```
function ReadString3: (string, string, string);
```

Возвращает кортеж из трёх значений типа string, введенных с клавиатуры

```
function ReadString3(prompt: string): (string, string, string);
```

Возвращает кортеж из трёх значений типа string, введенных с клавиатуры

```
function TryRead(var x: integer): boolean;
```

Вводит числовое значение x клавиатуры. Возвращает False если при вводе произошла ошибка

Подпрограммы вывода

procedure Print(a,b,...); Выводит значения a,b,... на экран, после каждого значения выводит пробел

procedure Print(f: Text; a,b,...);

Выводит значения a,b,... в текстовый файл f, после каждого значения выводит пробел

procedure Println(a,b,...);

Выводит значения a,b,... на экран, после каждого значения выводит пробел и переходит на новую строку

procedure Println(f: Text; a,b,...);

Выводит значения a,b,... в текстовый файл f, после каждого значения выводит пробел и переходит на новую строку

procedure Write(a,b,...);

Выводит значения a,b,... на экран

procedure Write(f: файл; a,b,...);

Выводит значения a,b,... в файл f

procedure WriteFormat(formatstr: string; **params** args: **array of object**);

Выводит значения args согласно форматной строке formatstr

procedure WriteFormat(f: Text; formatstr: string; **params** args: **array of object**);

Выводит значения args в текстовый файл f согласно форматной строке formatstr

procedure Writeln(a,b,...);

Выводит значения a,b,... на экран и осуществляет переход на новую строку

procedure Writeln(f: Text; a,b,...);

Выводит значения a,b,... в текстовый файл f и осуществляет переход на новую строку

procedure WritelnFormat(formatstr: string; **params** args: **array of object**);

Выводит значения args согласно форматной строке formatstr и осуществляет переход на новую строку

procedure WritelnFormat(f: Text; formatstr: string; **params** args: **array of object**);

Выводит значения `args` в текстовый файл `f` согласно форматной строке `formatstri` и осуществляет переход на новую строку

Математические подпрограммы

- function** Abs(x: число): число; Возвращает модуль числа x
- function** ArcCos(x: real): real;
Возвращает арккосинус числа x
- function** ArcSin(x: real): real;
Возвращает арксинус числа x
- function** ArcTan(x: real): real;
Возвращает арктангенс числа x
- function** Ceil(x: real): integer;
Возвращает наименьшее целое \geq x
- function** Cos(x: real): real;
Возвращает косинус числа x
- function** Cosh(x: real): real;
Возвращает гиперболический косинус числа x
- function** DegToRad(x: real): real;
Переводит градусы в радианы
- function** Exp(x: real): real;
Возвращает экспоненту числа x
- function** Floor(x: real): integer;
Возвращает наибольшее целое \leq x
- function** Frac(x: real): real;
Возвращает дробную часть числа x
- function** Int(x: real): real;
Возвращает целую часть числа x
- function** Ln(x: real): real;
Возвращает натуральный логарифм числа x
- function** Log(x: real): real;
Возвращает натуральный логарифм числа x
- function** Log10(x: real): real;
Возвращает десятичный логарифм числа x
- function** Log2(x: real): real;
Возвращает логарифм числа x по основанию 2
- function** LogN(base, x: real): real;
Возвращает логарифм числа x по основанию base

function Max(a: число, b: число): число;
Возвращает максимальное из чисел a,b

function Min(a: число, b: число): число;
Возвращает минимальное из чисел a,b

function Odd(i: целое): boolean;
Возвращает True, если i нечетно, и False в противном случае

function Power(x, y: real): real;
Возвращает x в степени y

function Power(x: real; n: integer): real;
Возвращает x в целой степени n

function Power(x: BigInteger; y: integer): BigInteger;
Возвращает x в степени y

function RadToDeg(x: real): real;
Переводит радианы в градусы

function Random(maxValue: integer): integer;
Возвращает случайное целое в диапазоне от 0 до maxValue-1

function Random(maxValue: real): real;
Возвращает случайное вещественное в диапазоне [0,maxValue)

function Random(a, b: integer): integer;
Возвращает случайное целое в диапазоне от a до b

function Random(a, b: real): real;
Возвращает случайное вещественное в диапазоне [a,b)

function Random: real;
Возвращает случайное вещественное в диапазоне [0..1)

function Random2(maxValue: integer): (integer, integer);
Возвращает кортеж из двух случайных целых в диапазоне от 0 до maxValue-1

function Random2(maxValue: real): (real, real);
Возвращает кортеж из двух случайных вещественных в диапазоне [0,maxValue)

function Random2(a, b: integer): (integer, integer);
Возвращает кортеж из двух случайных целых в диапазоне от a до b

function Random2(a, b: real): (real, real);
Возвращает кортеж из двух случайных вещественных в диапазоне [a,b)

function Random2: (real, real);

Возвращает кортеж из двух случайных вещественных в диапазоне [0..1)

function Random3(maxValue: integer): (integer, integer, integer);

Возвращает кортеж из трех случайных целых в диапазоне от 0 до maxValue-1

function Random3(maxValue: real): (real, real, real);

Возвращает кортеж из трех случайных вещественных в диапазоне [0,maxValue)

function Random3(a, b: integer): (integer, integer, integer);

Возвращает кортеж из трех случайных целых в диапазоне от a до b

function Random3(a, b: real): (real, real, real);

Возвращает кортеж из трех случайных вещественных в диапазоне [a,b)

function Random3: (real, real, real);

Возвращает кортеж из трех случайных вещественных в диапазоне [0..1)

procedure Randomize(seed: integer);

Инициализирует датчик псевдослучайных чисел, используя значение seed. При одном и том же seed генерируются одинаковые псевдослучайные последовательности

procedure Randomize;

Инициализирует датчик псевдослучайных чисел

function Round(x: real): integer;

Возвращает x, округленное до ближайшего целого. Если вещественное находится посередине между двумя целыми, то округление осуществляется к ближайшему четному (банковское округление): Round(2.5)=2, Round(3.5)=4

function Round(x: real; digits: integer): real;

Возвращает x, округленное до ближайшего вещественного с digits знаками после десятичной точки

function RoundBigInteger(x: real): BigInteger;

Возвращает x, округленное до ближайшего длинного целого

function Sign(x: число): число;

Возвращает знак числа x

function Sin(x: real): real;

Возвращает синус числа x

function Sinh(x: real): real;

Возвращает гиперболический синус числа x

function Sqr(x: число): число;

Возвращает квадрат числа x

function Sqrt(x: real): real;

Возвращает квадратный корень числа x

function Tan(x: real): real;

Возвращает тангенс числа x

function Tanh(x: real): real;

Возвращает гиперболический тангенс числа x

function Trunc(x: real): integer;

Возвращает целую часть вещественного числа x

function TruncBigInteger(x: real): BigInteger;

Возвращает целую часть вещественного числа x как длинное целое

Системные подпрограммы

procedure Assert(cond: boolean; sourceFile: string := ''; line: integer := 0); Выводит в специальном окне стек вызовов подпрограмм если условие не выполняется

procedure Assert(cond: boolean; message: string; sourceFile: string := ''; line: integer := 0);

Выводит в специальном окне диагностическое сообщение и стек вызовов подпрограмм если условие не выполняется

function ChangeFileNameExtension(name, newext: string): string;

Изменяет расширение файла с именем name на newext

procedure ChDir(s: string);

Меняет текущий каталог

function CreateDir(s: string): boolean;

Создает каталог. Возвращает True, если каталог успешно создан

function DeleteFile(s: string): boolean;

Удаляет файл. Если файл не может быть удален, то возвращает False

function DiskFree(diskname: string): int64;

Возвращает свободное место в байтах на диске с именем diskname

function DiskFree(disk: integer): int64;

Возвращает свободное место в байтах на диске disk. disk=0 - текущий диск, disk=1 - диск A: , disk=2 - диск B: и т.д.

function DiskSize(diskname: string): int64;

Возвращает размер в байтах на диске с именем diskname

function DiskSize(disk: integer): int64;

Возвращает размер в байтах на диске disk. disk=0 - текущий диск, disk=1 - диск A: , disk=2 - диск B: и т.д.

procedure Dispose<T>(var p: ^T);

Освобождает динамическую память, на которую указывает p

function EnumerateAllDirectories(path: string): sequence of string;

Возвращает последовательность имен каталогов по заданному пути, включая подкаталоги

function EnumerateAllFiles(path: string; searchPattern: string := '*.*'): sequence of string;

Возвращает последовательность имен файлов по заданному пути, соответствующих шаблону поиска, включая подкаталоги

function EnumerateDirectories(path: string): **sequence of** string;

Возвращает последовательность имен каталогов по заданному пути

function EnumerateFiles(path: string; searchPattern: string := '*.*'): **sequence of** string;

Возвращает последовательность имен файлов по заданному пути, соответствующих шаблону поиска

procedure Exec(filename: string);

Запускает программу или документ с именем filename

procedure Exec(filename: string; args: string);

Запускает программу или документ с именем filename и параметрами командной строки args

procedure Execute(filename: string);

Запускает программу или документ с именем filename

procedure Execute(filename: string; args: string);

Запускает программу или документ с именем filename и параметрами командной строки args

function FileExists(name: string): boolean;

Возвращает True, если файл с именем name существует

function GetCurrentDir: string;

Возвращает текущий каталог

function GetDir: string;

Возвращает текущий каталог

function GetEXEFileName: string;

Возвращает имя запущенного .exe-файла

procedure Halt(exitCode: integer);

Завершает работу программы, возвращая код ошибки exitCode

procedure Halt;

Завершает работу программы

function Milliseconds: integer;

Возвращает количество миллисекунд с момента начала работы программы

function MillisecondsDelta: integer;

Возвращает количество миллисекунд с момента последнего вызова

Milliseconds или MillisecondsDelta

procedure Mkdir(s: string);

Создает каталог

procedure New<T>(var p: ^T);

Выделяет динамическую память размера sizeof(T) и возвращает в переменной p указатель на нее. Тип T должен быть размерным

function ParamCount: integer;

Возвращает количество параметров командной строки

function ParamStr(i: integer): string;

Возвращает i-тый параметр командной строки

function PascalABCVersion: string;

Возвращает версию PascalABC.NET

function PointerToString(p: pointer): string;

Преобразует указатель к строковому представлению

function RemoveDir(s: string): boolean;

Удаляет каталог. Возвращает True, если каталог успешно удален

function RenameFile(name, newname: string): boolean;

Переименовывает файл name, давая ему новое имя newname. Возвращает True, если файл успешно переименован

procedure Rmdir(s: string);

Удаляет каталог

function SetCurrentDir(s: string): boolean;

Устанавливает текущий каталог. Возвращает True, если каталог успешно удален

procedure Sleep(ms: integer);

Делает паузу на ms миллисекунд

Общие подпрограммы для работы с файлами

procedure Assign(f: файл; name: string); Связывает
файловую переменную с файлом на диске

procedure AssignFile(f: файл; name: string);
 Связывает файловую переменную с файлом на диске

procedure Close(f: файл);
 Закрывает файл

procedure CloseFile(f: файл);
 Закрывает файл

function Eof(f: файл): boolean;
 Возвращает True, если достигнут конец файла

procedure Erase(f: файл);
 Удаляет файл, связанный с файловой переменной

procedure Rename(f: файл; newname: string);
 Переименовывает файл, связанный с файловой переменной, давая ему
имя newname.

Подпрограммы для работы с текстовыми файлами

procedure Append(f: Text); Открывает текстовый файл на дополнение в кодировке Windows

procedure Append(f: Text; en: Encoding);
Открывает текстовый файл на дополнение в указанной кодировке

procedure Append(f: Text; name: string);
Связывает файловую переменную f с именем файла name и открывает текстовый файл на дополнение в кодировке Windows

procedure Append(f: Text; name: string; en: Encoding);
Связывает файловую переменную f с именем файла name и открывает текстовый файл на дополнение в указанной кодировке

function Eoln(f: Text): boolean;
Возвращает True, если в файле достигнут конец строки

procedure Flush(f: Text);
Записывает содержимое буфера файла на диск

function OpenAppend(fname: string): Text;
Возвращает текстовый файл с именем fname, открытый на дополнение в кодировке Windows

function OpenAppend(fname: string; en: Encoding): Text;
Возвращает текстовый файл с именем fname, открытый на дополнение в указанной кодировке

function OpenRead(fname: string): Text;
Возвращает текстовый файл с именем fname, открытый на чтение в кодировке Windows

function OpenRead(fname: string; en: Encoding): Text;
Возвращает текстовый файл с именем fname, открытый на чтение в указанной кодировке

function OpenWrite(fname: string): Text;
Возвращает текстовый файл с именем fname, открытый на запись в кодировке Windows

function OpenWrite(fname: string; en: Encoding): Text;
Возвращает текстовый файл с именем fname, открытый на запись в указанной кодировке

function ReadAllLines(path: string): array of string;

Открывает файл, считывает из него строки в кодировке Windows в виде массива строк, после чего закрывает файл

```
function ReadAllLines(path: string; en: Encoding): array of  
string;
```

Открывает файл, считывает из него строки в указанной кодировке в виде массива строк, после чего закрывает файл

```
function ReadAllText(path: string): string;
```

Открывает файл, считывает его содержимое в кодировке Windows в виде строки, после чего закрывает файл

```
function ReadAllText(path: string; en: Encoding): string;
```

Открывает файл, считывает его содержимое в указанной кодировке в виде строки, после чего закрывает файл

```
function ReadLines(path: string): sequence of string;
```

Открывает файл, считывает из него строки в кодировке Windows и закрывает файл. В каждый момент в памяти хранится только текущая строка

```
function ReadLines(path: string; en: Encoding): sequence of  
string;
```

Открывает файл, считывает из него строки в указанной кодировке и закрывает файл. В каждый момент в памяти хранится только текущая строка

```
procedure Reset(f: Text);
```

Открывает текстовый файл на чтение в кодировке Windows

```
procedure Reset(f: Text; en: Encoding);
```

Открывает текстовый файл на чтение в указанной кодировке

```
procedure Reset(f: Text; name: string);
```

Связывает файловую переменную f с именем файла name и открывает текстовый файл на чтение в кодировке Windows

```
procedure Reset(f: Text; name: string; en: Encoding);
```

Связывает файловую переменную f с именем файла name и открывает текстовый файл на чтение в указанной кодировке

```
procedure Rewrite(f: Text);
```

Открывает текстовый файл на запись в кодировке Windows. Если файл существовал - он обнуляется, если нет - создается пустой

```
procedure Rewrite(f: Text; en: Encoding);
```

Открывает текстовый файл на запись в указанной кодировке. Если

файл существовал - он обнуляется, если нет - создается пустой

```
procedure Rewrite(f: Text; name: string);
```

Связывает файловую переменную с именем файла name и открывает текстовый файл f на запись в кодировке Windows. Если файл существовал - он обнуляется, если нет - создается пустой

```
procedure Rewrite(f: Text; name: string; en: Encoding);
```

Связывает файловую переменную f с именем файла name и открывает текстовый файл f на запись в указанной кодировке. Если файл существовал - он обнуляется, если нет - создается пустой

```
function SeekEof(f: Text): boolean;
```

Пропускает пробельные символы, после чего возвращает True, если достигнут конец файла

```
function SeekEoln(f: Text): boolean;
```

Пропускает пробельные символы, после чего возвращает True, если в файле достигнут конец строки

```
procedure WriteAllLines(path: string; ss: array of string);
```

Создает новый файл, записывает в него строки из массива в кодировке Windows, после чего закрывает файл

```
procedure WriteAllLines(path: string; ss: array of string;  
en: Encoding);
```

Создает новый файл, записывает в него строки из массива в указанной кодировке, после чего закрывает файл

```
procedure WriteAllText(path: string; s: string);
```

Создает новый файл, записывает в него строку в кодировке Windows, после чего закрывает файл

```
procedure WriteAllText(path: string; s: string; en:  
Encoding);
```

Создает новый файл, записывает в него строку в указанной кодировке, после чего закрывает файл

```
procedure WriteLines(path: string; ss: sequence of string);
```

Создает новый файл, записывает в него строки из последовательности в кодировке Windows, после чего закрывает файл

```
procedure WriteLines(path: string; ss: sequence of string;  
en: Encoding);
```

Создает новый файл, записывает в него строки из последовательности в указанной кодировке, после чего закрывает файл

Подпрограммы для работы с типизированными файлами

function CreateBinary<T>(fname: string): **file of T**;

Создаёт или обнуляет типизированный файл и возвращает значение для инициализации файловой переменной

function CreateFile<T>(fname: string): **file of T**;

Создаёт или обнуляет типизированный файл и возвращает значение для инициализации файловой переменной

function CreateFileInteger(fname: string): **file of integer**;

Создаёт или обнуляет типизированный файл целых и возвращает значение для инициализации файловой переменной

function CreateFileReal(fname: string): **file of real**;

Создаёт или обнуляет типизированный файл вещественных и возвращает значение для инициализации файловой переменной

function OpenBinary<T>(fname: string): **file of T**;

Открывает типизированный файл и возвращает значение для инициализации файловой переменной

function OpenFile<T>(fname: string): **file of T**;

Открывает типизированный файл и возвращает значение для инициализации файловой переменной

function OpenFileInteger(fname: string): **file of integer**;

Открывает типизированный файл целых и возвращает значение для инициализации файловой переменной

function OpenFileReal(fname: string): **file of real**;

Открывает типизированный файл вещественных и возвращает значение для инициализации файловой переменной

procedure WriteElements<T>(fname: string; ss: **sequence of T**);

Открывает типизированный файл, возвращает последовательность его элементов и закрывает его

Подпрограммы для работы с двоичными файлами

function FilePos(f: двоичный файл): int64; Возвращает текущую позицию файлового указателя в двоичном файле

function FileSize(f: двоичный файл): int64;
 Возвращает количество элементов в двоичном файле

procedure Reset(f: двоичный файл);

Открывает двоичный файл на чтение и запись. Двоичный файл - это либо типизированный файл file of T, либо бестиповой файл file

procedure Reset(f: двоичный файл; name: string);

Связывает файловую переменную f с файлом name на диске и открывает двоичный файл на чтение и запись. Двоичный файл - это либо типизированный файл file of T, либо бестиповой файл file

procedure Rewrite(f: двоичный файл);

Открывает двоичный файл на чтение и запись, при этом обнуляя его содержимое. Если файл существовал, он обнуляется. Двоичный файл - это либо типизированный файл file of T, либо бестиповой файл file

procedure Rewrite(f: двоичный файл; name: string);

Связывает файловую переменную f с файлом name на диске и открывает двоичный файл на чтение и запись, при этом обнуляя его содержимое. Двоичный файл - это либо типизированный файл file of T, либо бестиповой файл file

procedure Seek(f: двоичный файл; n: int64);

Устанавливает текущую позицию файлового указателя в двоичном файле на элемент с данным номером

procedure Truncate(f: двоичный файл);

Усекает двоичный файл, отбрасывая все элементы с позиции файлового указателя. Двоичный файл - это либо типизированный файл file of T, либо бестиповой файл file

Подпрограммы для работы с именами файлов

function ExpandFileName(fname: string): string;

Возвращает полное имя файла fname

function ExtractFileDir(fname: string): string;

Выделяет имя диска и путь из полного имени файла fname

function ExtractFileDrive(fname: string): string;

Выделяет путь из полного имени файла fname

function ExtractFileExt(fname: string): string;

Выделяет расширение из полного имени файла fname

function ExtractFileName(fname: string): string;

Выделяет имя файла из полного имени файла fname

function ExtractFilePath(fname: string): string;

Выделяет путь из полного имени файла fname

Подпрограммы для работы с символами

function Chr(a: word): char; Преобразует код в символ в кодировке Unicode

function ChrAnsi(a: byte): char;
 Преобразует код в символ в кодировке Windows

function ChrUnicode(a: word): char;
 Преобразует код в символ в кодировке Unicode

procedure Dec(var c: char);
 Уменьшает код символа c на 1

procedure Dec(var c: char; n: integer);
 Уменьшает код символа c на n

procedure Inc(var c: char);
 Увеличивает код символа c на 1

procedure Inc(var c: char; n: integer);
 Увеличивает код символа c на n

function LowCase(ch: char): char;
 Преобразует символ в нижний регистр

function LowerCase(ch: char): char;
 Преобразует символ в нижний регистр

function Ord(a: char): word;
 Преобразует символ в код в кодировке Unicode

function OrdAnsi(a: char): byte;
 Преобразует символ в код в кодировке Windows

function OrdUnicode(a: char): word;
 Преобразует символ в код в кодировке Unicode

function Pred(x: char): char;
 Возвращает предшествующий x символ

function Succ(x: char): char;
 Возвращает следующий за x символ

function UpCase(ch: char): char;
 Преобразует символ в верхний регистр

function UpperCase(ch: char): char;
 Преобразует символ в верхний регистр

Подпрограммы для работы со строками

function CompareStr(s1, s2: string): integer; Сравнивает строки. Возвращает значение < 0 если s1<s2, > 0 если s1>s2 и = 0 если s1=s2

function Concat(s1,s2,...): string;

Возвращает строку, являющуюся результатом слияния строк s1,s2,...

function Concat(s1, s2: string): string;

Возвращает строку, являющуюся результатом слияния строк s1 и s2

function Copy(s: string; index, count: integer): string;

Возвращает подстроку строки s длины count с позиции index

procedure Delete(var s: string; index, count: integer);

Удаляет из строки s count символов с позиции index

function FloatToStr(a: real): string;

Преобразует вещественное число к строковому представлению

function Format(formatstring: string; params pars: array of object): string;

Возвращает отформатированную строку, построенную по форматной строке и списку форматируемых параметров

procedure Insert(source: string; var s: string; index: integer);

Вставляет подстроку source в строку s с позиции index

function IntToStr(a: integer): string;

Преобразует целое число к строковому представлению

function IntToStr(a: int64): string;

Преобразует целое число к строковому представлению

function LastPos(subs, s: string): integer;

Возвращает позицию последнего вхождения подстроки subs в строке s. Если не найдена, возвращает 0

function LastPos(subs, s: string; from: integer): integer;

Возвращает позицию последнего вхождения подстроки subs в строке s начиная с позиции from. Если не найдена, возвращает 0

function LeftStr(s: string; count: integer): string;

Возвращает первые count символов строки s

function Length(s: string): integer;

Возвращает длину строки

```
function LowerCase(s: string): string;
```

Возвращает строку в нижнем регистре

```
function Pos(subs, s: string; from: integer := 1): integer;
```

Возвращает позицию подстроки subs в строке s. Если не найдена, возвращает 0

```
function PosEx(subs, s: string; from: integer := 1):  
integer;
```

Возвращает позицию подстроки subs в строке s начиная с позиции from. Если не найдена, возвращает 0

```
function ReadIntegerFromString(s: string; var from:  
integer): integer;
```

Считывает целое из строки начиная с позиции from и устанавливает from за считанным значением

```
function ReadRealFromString(s: string; var from: integer):  
real;
```

Считывает вещественное из строки начиная с позиции from и устанавливает from за считанным значением

```
function ReadWordFromString(s: string; var from: integer):  
string;
```

Считывает из строки последовательность символов до пробельного символа начиная с позиции from и устанавливает from за считанным значением

```
function ReverseString(s: string): string;
```

Возвращает инвертированную строку

```
function ReverseString(s: string; index, length: integer):  
string;
```

Возвращает инвертированную строку в диапазоне длины length начиная с индекса index

```
function RightStr(s: string; count: integer): string;
```

Возвращает последние count символов строки s

```
procedure SetLength(var s: string; n: integer);
```

Устанавливает длину строки s равной n

```
procedure Str(i: целое; var s: string);
```

Преобразует целое значение i к строковому представлению и записывает результат в s

procedure Str(r: real; var s: string);

Преобразует вещественное значение r к строковому представлению и записывает результат в s

procedure Str(r: single; var s: string);

Преобразует вещественное значение r к строковому представлению и записывает результат в s

function StringIsEmpty(s: string; var from: integer):
boolean;

Возвращает True если достигнут конец строки или в строке остались только пробельные символы и False в противном случае

function StringOfChar(ch: char; count: integer): string;

Возвращает строку, состоящую из count символов ch

function StrToFloat(s: string): real;

Преобразует строковое представление вещественного числа к числовому значению

function StrToInt(s: string): integer;

Преобразует строковое представление целого числа к числовому значению

function StrToInt64(s: string): int64;

Преобразует строковое представление целого числа к числовому значению

function StrToReal(s: string): real;

Преобразует строковое представление вещественного числа к числовому значению

function Trim(s: string): string;

Возвращает строку с удаленными начальными и конечными пробелами

function TrimLeft(s: string): string;

Возвращает строку с удаленными начальными пробелами

function TrimRight(s: string): string;

Возвращает строку с удаленными конечными пробелами

function TryReadIntegerFromString(s: string; var from:
integer; var res: integer): boolean;

Считывает целое из строки начиная с позиции from и устанавливает from за считанным значением. Возвращает True если считывание успешно и False в противном случае

```
function TryReadRealFromString(s: string; var from: integer; var res: real): boolean;
```

Считывает вещественное из строки начиная с позиции from и устанавливает from за считанным значением. Возвращает True если считывание успешно и False в противном случае

```
function TryStrToFloat(s: string; var value: real): boolean;
```

Преобразует строковое представление s вещественного числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function TryStrToFloat(s: string; var value: single): boolean;
```

Преобразует строковое представление s вещественного числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function TryStrToInt(s: string; var value: integer): boolean;
```

Преобразует строковое представление s целого числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function TryStrToInt64(s: string; var value: int64): boolean;
```

Преобразует строковое представление s целого числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function TryStrToReal(s: string; var value: real): boolean;
```

Преобразует строковое представление s вещественного числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function TryStrToSingle(s: string; var value: single): boolean;
```

Преобразует строковое представление s вещественного числа к числовому значению и записывает его в value. При невозможности преобразования возвращается False

```
function UpperCase(s: string): string;
```

Возвращает строку в верхнем регистре

```
procedure Val(s: string; var value: число; var err:
```

`integer);`

Преобразует строковое представление `s` целого числа к числовому значению и записывает его в переменную `value`. Если преобразование успешно, то `err=0`, иначе `err>0`

Подпрограммы для работы с динамическими массивами

function Copy(a: array of T): array of T; Создает копию динамического массива

function High(a: array of T): integer;
Возвращает верхнюю границу динамического массива

function Length(a: array of T): integer;
Возвращает длину динамического массива

function Length(a: array of T; dim: integer): integer;
Возвращает длину динамического массива по размерности dim

function Low(a: array of T): integer;
Возвращает 0

procedure Reverse<T>(a: array of T);
Изменяет порядок элементов в динамическом массиве на противоположный

procedure Reverse<T>(a: array of T; index, count: integer);
Изменяет порядок элементов на противоположный в диапазоне динамического массива длины count, начиная с индекса index

procedure Reverse<T>(a: List<T>);
Изменяет порядок элементов в списке на противоположный

procedure Reverse<T>(a: List<T>; index, count: integer);
Изменяет порядок элементов на противоположный в диапазоне списка длины count, начиная с индекса index

procedure SetLength(var a: array of T);
Устанавливает длину одномерного динамического массива. Старое содержимое сохраняется

procedure SetLength(var a: array of T; n1, n2, ...: integer);
Устанавливает размеры n-мерного динамического массива. Старое содержимое сохраняется

procedure Shuffle<T>(a: array of T);
Перемешивает динамический массив случайным образом

procedure Shuffle<T>(l: List<T>);
Перемешивает список случайным образом

procedure Sort<T>(a: **array of** T);

Сортирует динамический массив по возрастанию

procedure Sort<T>(a: **array of** T; cmp: (T,T)->integer);

Сортирует динамический массив по критерию сортировки, задаваемому функцией сравнения cmp

procedure Sort<T>(a: **array of** T; less: (T,T)->boolean);

Сортирует динамический массив по критерию сортировки, задаваемому функцией сравнения less

procedure Sort<T>(l: List<T>);

Сортирует список по возрастанию

procedure Sort<T>(l: List<T>; cmp: (T,T)->integer);

Сортирует список по критерию сортировки, задаваемому функцией сравнения cmp

procedure Sort<T>(l: List<T>; less: (T,T)->boolean);

Сортирует список по критерию сортировки, задаваемому функцией сравнения less

Подпрограммы для работы со стандартными множествами

procedure Exclude(**var** s: **set of** T; element: T); Удаляет
элемент element из множества s

procedure Include(**var** s: **set of** T; element: T);
Добавляет элемент element во множество s

Подпрограммы для работы с комплексными числами

function Abs(c: Complex): Complex; Возвращает модуль комплексного числа

function Conjugate(c: Complex): Complex;
 Возвращает комплексно сопряженное число

function Cos(c: Complex): Complex;
 Возвращает косинус комплексного числа

function Cplx(re, im: real): Complex;
 Конструирует комплексное число с вещественной частью re и мнимой частью im

function CplxFromPolar(magnitude, phase: real): Complex;
 Конструирует комплексное число по полярным координатам

function Exp(c: Complex): Complex;
 Возвращает экспоненту комплексного числа

function Ln(c: Complex): Complex;
 Возвращает натуральный логарифм комплексного числа

function Log(c: Complex): Complex;
 Возвращает натуральный логарифм комплексного числа

function Log10(c: Complex): Complex;
 Возвращает десятичный логарифм комплексного числа

function Power(c, power: Complex): Complex;
 Возвращает степень комплексного числа

function Sin(c: Complex): Complex;
 Возвращает синус комплексного числа

function Sqrt(c: Complex): Complex;
 Возвращает квадратный корень из комплексного числа

Подпрограммы для генерации последовательностей

function Partition(a, b: real; n: integer): **sequence of real**; Возвращает последовательность вещественных в точках разбиения отрезка [a,b] на n равных частей

function Range(a, b: integer): **sequence of integer**; Возвращает последовательность целых от a до b

function Range(c1, c2: char): **sequence of char**; Возвращает последовательность символов от c1 до c2

function Range(a, b: real; n: integer): **sequence of real**; Возвращает последовательность вещественных в точках разбиения отрезка [a,b] на n равных частей (Используйте Partition)

function Range(a, b, step: integer): **sequence of integer**; Возвращает последовательность целых от a до b с шагом step

function ReadSeqInteger(n: integer): **sequence of integer**; Возвращает последовательность из n целых, введенных с клавиатуры

function ReadSeqInteger(prompt: string; n: integer): **sequence of integer**; Выводит приглашение к вводу и возвращает последовательность из n целых, введенных с клавиатуры

function ReadSeqIntegerWhile(cond: integer->boolean): **sequence of integer**; Возвращает последовательность целых, вводимых с клавиатуры пока выполняется определенное условие

function ReadSeqIntegerWhile(prompt: string; cond: integer->boolean): **sequence of integer**; Выводит приглашение к вводу и возвращает последовательность целых, вводимых с клавиатуры пока выполняется определенное условие

function ReadSeqReal(n: integer): **sequence of real**; Возвращает последовательность из n вещественных, введенных с клавиатуры

function ReadSeqReal(prompt: string; n: integer): **sequence of real**; Выводит приглашение к вводу и возвращает последовательность из n вещественных, введенных с клавиатуры

```
function ReadSeqRealWhile(cond: real->boolean): sequence of  
real;
```

Возвращает последовательность вещественных, вводимых с клавиатуры пока выполняется определенное условие

```
function ReadSeqRealWhile(prompt: string; cond: real->  
>boolean): sequence of real;
```

Выводит приглашение к вводу и возвращает последовательность вещественных, вводимых с клавиатуры пока выполняется определенное условие

```
function ReadSeqString(n: integer): sequence of string;
```

Возвращает последовательность из n строк, введенных с клавиатуры

```
function ReadSeqString(prompt: string; n: integer):  
sequence of string;
```

Выводит приглашение к вводу и возвращает последовательность из n строк, введенных с клавиатуры

```
function ReadSeqStringWhile(cond: string->boolean):  
sequence of string;
```

Возвращает последовательность строк, вводимых с клавиатуры пока выполняется определенное условие

```
function ReadSeqStringWhile(prompt: string; cond: string->  
>boolean): sequence of string;
```

Выводит приглашение к вводу и возвращает последовательность строк, вводимых с клавиатуры пока выполняется определенное условие

```
function Seq<T>(params a: array of T): sequence of T;
```

Возвращает последовательность указанных элементов

```
function SeqFill<T>(count: integer; x: T): sequence of T;
```

Возвращает последовательность из count элементов x

```
function SeqGen<T>(count: integer; f: integer->T): sequence  
of T;
```

Возвращает последовательность из count элементов, заполненных значениями f(i)

```
function SeqGen<T>(count: integer; f: integer->T; from:  
integer): sequence of T;
```

Возвращает последовательность из count элементов, заполненных значениями f(i), начиная с i=from

```
function SeqGen<T>(count: integer; first: T; next: T->T):  
sequence of T;
```

Возвращает последовательность из count элементов, начинающуюся с first, с функцией next перехода от предыдущего к следующему

```
function SeqGen<T>(count: integer; first, second: T; next:  
(T,T) ->T): sequence of T;
```

Возвращает последовательность из count элементов, начинающуюся с first и second, с функцией next перехода от двух предыдущих к следующему

```
function SeqRandom(n: integer := 10; a: integer := 0; b:  
integer := 100): sequence of integer;
```

Возвращает последовательность из n случайных целых элементов

```
function SeqRandomInteger(n: integer := 10; a: integer :=  
0; b: integer := 100): sequence of integer;
```

Возвращает последовательность из n случайных целых элементов

```
function SeqRandomReal(n: integer := 10; a: real := 0; b:  
real := 10): sequence of real;
```

Возвращает последовательность из n случайных вещественных элементов

```
function SeqWhile<T>(first: T; next: T->T; pred: T->  
boolean): sequence of T;
```

Возвращает последовательность элементов с начальным значением first, функцией next перехода от предыдущего к следующему и условием pred продолжения последовательности

```
function SeqWhile<T>(first, second: T; next: (T,T) ->T;  
pred: T->boolean): sequence of T;
```

Возвращает последовательность элементов, начинающуюся с first и second, с функцией next перехода от двух предыдущих к следующему и условием pred продолжения последовательности

Подпрограммы для создания динамических массивов

function Arr<T>(params a: array of T): array of T;

Возвращает массив, заполненный указанными значениями

function Arr<T>(a: sequence of T): array of T;

Возвращает массив, заполненный значениями из последовательности

function ArrFill<T>(count: integer; x: T): array of T;

Возвращает массив из count элементов x

function ArrGen<T>(count: integer; gen: integer->T): array of T;

Возвращает массив из count элементов, заполненных значениями gen(i)

function ArrGen<T>(count: integer; gen: integer->T; from: integer): array of T;

Возвращает массив из count элементов, заполненных значениями gen(i), начиная с i=from

function ArrGen<T>(count: integer; first: T; next: T->T): array of T;

Возвращает массив из count элементов, начинающихся с first, с функцией next перехода от предыдущего к следующему

function ArrGen<T>(count: integer; first, second: T; next: (T,T) ->T): array of T;

Возвращает массив из count элементов, начинающихся с first и second, с функцией next перехода от двух предыдущих к следующему

function ArrRandom(n: integer := 10; a: integer := 0; b: integer := 100): array of integer;

Возвращает массив размера n, заполненный случайными целыми значениями

function ArrRandomInteger(n: integer := 10; a: integer := 0; b: integer := 100): array of integer;

Возвращает массив размера n, заполненный случайными целыми значениями

function ArrRandomReal(n: integer := 10; a: real := 0; b: real := 10): array of real;

Возвращает массив размера n, заполненный случайными

вещественными значениями

function ReadArrInteger(n: integer): **array of integer**;

Возвращает массив из n целых, введенных с клавиатуры

function ReadArrInteger(prompt: string; n: integer): **array of integer**;

Выводит приглашение к вводу и возвращает массив из n целых, введенных с клавиатуры

function ReadArrReal(n: integer): **array of real**;

Возвращает массив из n вещественных, введенных с клавиатуры

function ReadArrReal(prompt: string; n: integer): **array of real**;

Выводит приглашение к вводу и возвращает массив из n вещественных, введенных с клавиатуры

function ReadArrString(n: integer): **array of string**;

Возвращает массив из n строк, введенных с клавиатуры

function ReadArrString(prompt: string; n: integer): **array of string**;

Выводит приглашение к вводу и возвращает массив из n строк, введенных с клавиатуры

Подпрограммы для создания двумерных динамических массивов

function Matr<T>(m,n: integer; **params** data: **array of T**): **array [,] of T**;
Возвращает двумерный массив размера m x n, заполненный указанными значениями по строкам

function MatrFill<T>(m, n: integer; x: T): **array [,] of T**;
Возвращает двумерный массив размера m x n, заполненный элементами x

function MatrGen<T>(m, n: integer; gen: (integer,integer)->T): **array [,] of T**;

Возвращает двумерный массив размера m x n, заполненный элементами x

function MatrRandom(m: integer := 5; n: integer := 5; a: integer := 0; b: integer := 100): **array [,] of integer**;

Возвращает двумерный массив размера m x n, заполненный случайными целыми значениями

function MatrRandomInteger(m: integer := 5; n: integer := 5; a: integer := 0; b: integer := 100): **array [,] of integer**;

Возвращает двумерный массив размера m x n, заполненный случайными целыми значениями

function MatrRandomReal(m: integer := 5; n: integer := 5; a: real := 0; b: real := 10): **array [,] of real**;

Возвращает двумерный массив размера m x n, заполненный случайными вещественными значениями

function ReadMatrInteger(m, n: integer): **array [,] of integer**;

Возвращает матрицу m на n целых, введенных с клавиатуры

function ReadMatrReal(m, n: integer): **array [,] of real**;

Возвращает матрицу m на n вещественных, введенных с клавиатуры

function Transpose<T>(a: **array [,] of T**): **array [,] of T**;

Транспонирует двумерный массив

Подпрограммы для создания кортежей

function Rec(x1: T1, x2: T2, ...): (T1, T2, ...);

кортеж из элементов разных типов

Возвращает

Короткие функции Lst, LLst, HSet, SSet, Dict, KV

function Dict<TKey, TVal>(params pairs: array of KeyValuePair<TKey, TVal>): Dictionary<TKey, TVal>;

Возвращает словарь пар элементов (ключ, значение)

function Dict<TKey, TVal>(params pairs: array of (TKey, TVal)): Dictionary<TKey, TVal>;

Возвращает словарь пар элементов (ключ, значение)

function HSet<T>(params a: array of T): HashSet<T>;

Возвращает множество на базе хеш таблицы, заполненное указанными значениями

function HSet<T>(a: sequence of T): HashSet<T>;

Возвращает множество на базе хеш таблицы, заполненное значениями из последовательности

function KV<TKey, TVal>(key: TKey; value: TVal): KeyValuePair<TKey, TVal>;

Возвращает пару элементов (ключ, значение)

function LLst<T>(params a: array of T): LinkedList<T>;

Возвращает двусвязный список, заполненный указанными значениями

function LLst<T>(a: sequence of T): LinkedList<T>;

Возвращает двусвязный список, заполненный значениями из последовательности

function Lst<T>(params a: array of T): List<T>;

Возвращает список, заполненный указанными значениями

function Lst<T>(a: sequence of T): List<T>;

Возвращает список, заполненный значениями из последовательности

function SSet<T>(params a: array of T): SortedSet<T>;

Возвращает множество на базе бинарного дерева поиска, заполненное значениями из последовательности

function SSet<T>(a: sequence of T): SortedSet<T>;

Возвращает множество на базе бинарного дерева поиска, заполненное значениями из последовательности

Генерация бесконечных последовательностей

function Cycle<T>(Self: **sequence of** T): **sequence of** T;

Повторяет последовательность бесконечное число раз

function Step(Self: integer): **sequence of** integer;

Возвращает бесконечную последовательность целых от текущего значения с шагом 1

function Step(Self: integer; step: integer): **sequence of** integer;

Возвращает бесконечную последовательность целых от текущего значения с шагом step

function Step(Self: real; step: real): **sequence of** real;

Возвращает бесконечную последовательность вещественных от текущего значения с шагом step

Методы расширения последовательностей

function AdjacentGroup<T>(Self: **sequence of T**): **sequence of array of T**; Группирует одинаковые подряд идущие элементы,

получая последовательность массивов

function Batch<T>(Self: **sequence of T**; size: integer): **sequence of sequence of T**;

Разбивает последовательность на серии длины size

function Batch<T, Res>(Self: **sequence of T**; size: integer; proj: Func<IEnumerable<T>, Res>): **sequence of Res**;

Разбивает последовательность на серии длины size и применяет проекцию к каждой серии

function Cartesian<T, T1>(Self: **sequence of T**; b: **sequence of T1**): **sequence of (T, T1)**;

Декартово произведение последовательностей

function Cartesian<T, T1, T2>(Self: **sequence of T**; b: **sequence of T1**; func: (T,T1)->T2): **sequence of T2**;

Декартово произведение последовательностей

procedure ForEach<T>(Self: **sequence of T**; action: T -> ());

Применяет действие к каждому элементу последовательности

procedure ForEach<T>(Self: **sequence of T**; action: (T,integer) -> ());

Применяет действие к каждому элементу последовательности, зависящее от номера элемента

function Incremental(Self: **sequence of integer**): **sequence of integer**;

Возвращает последовательность разностей соседних элементов исходной последовательности

function Incremental(Self: **array of integer**): **sequence of integer**;

Возвращает последовательность разностей соседних элементов исходной последовательности

function Incremental(Self: List<integer>): **sequence of integer**;

Возвращает последовательность разностей соседних элементов исходной последовательности

function Incremental(Self: LinkedList<integer>): **sequence of integer**;

Возвращает последовательность разностей соседних элементов исходной последовательности

```
function Incremental(Self: sequence of real): sequence of real;
```

Возвращает последовательность разностей соседних элементов исходной последовательности

```
function Incremental(Self: array of real): sequence of real;
```

Возвращает последовательность разностей соседних элементов исходной последовательности

```
function Incremental(Self: List<real>): sequence of real;
```

Возвращает последовательность разностей соседних элементов исходной последовательности

```
function Incremental(Self: LinkedList<real>): sequence of real;
```

Возвращает последовательность разностей соседних элементов исходной последовательности

```
function Incremental<T, T1>(Self: sequence of T; func: (T,T)->T1): sequence of T1;
```

Возвращает последовательность разностей соседних элементов исходной последовательности. В качестве функции разности используется func

```
function Incremental<T, T1>(Self: sequence of T; func: (T,T,integer)->T1): sequence of T1;
```

Возвращает последовательность разностей соседних элементов исходной последовательности. В качестве функции разности используется func

```
function Interleave<T>(Self: sequence of T; a: sequence of T): sequence of T;
```

Чередует элементы двух последовательностей

```
function Interleave<T>(Self: sequence of T; a, b: sequence of T): sequence of T;
```

Чередует элементы трех последовательностей

```
function Interleave<T>(Self: sequence of T; a, b, c: sequence of T): sequence of T;
```

Чередует элементы четырех последовательностей

```
function JoinIntoString<T>(Self: sequence of T; delim: string): string;
```

Преобразует элементы последовательности в строковое

представление, после чего объединяет их в строку, используя delim в качестве разделителя

function JoinIntoString<T>(Self: **sequence of** T): string;

Преобразует элементы последовательности в строковое представление, после чего объединяет их в строку, используя пробел в качестве разделителя

function LastMaxBy<T, TKey>(Self: **sequence of** T; selector: T->TKey): T;

Возвращает последний элемент последовательности с максимальным значением ключа

function LastMinBy<T, TKey>(Self: **sequence of** T; selector: T->TKey): T;

Возвращает последний элемент последовательности с минимальным значением ключа

function MaxBy<T, TKey>(Self: **sequence of** T; selector: T->TKey): T;

Возвращает первый элемент последовательности с максимальным значением ключа

function MinBy<T, TKey>(Self: **sequence of** T; selector: T->TKey): T;

Возвращает первый элемент последовательности с минимальным значением ключа

function Numerate<T>(Self: **sequence of** T): **sequence of** (integer, T);

Нумерует последовательность с единицы

function Numerate<T>(Self: **sequence of** T; from: integer): **sequence of** (integer, T);

Нумерует последовательность с номера from

function Order<T>(Self: **sequence of** T): **sequence of** T;

Возвращает отсортированную по возрастанию последовательность

function OrderDescending<T>(Self: **sequence of** T): **sequence of** T;

Возвращает отсортированную по убыванию последовательность

function Pairwise<T>(Self: **sequence of** T): **sequence of** (T, T);

Превращает последовательность в последовательность пар соседних элементов

function Pairwise<T, Res>(Self: **sequence of** T; func: (T,T)-

>Res): sequence of Res;

Превращает последовательность в последовательность пар соседних элементов, применяет func к каждой паре полученных элементов и получает новую последовательность

function Partition<T>(Self: sequence of T; cond: T->boolean): (sequence of T, sequence of T);

Разделяет последовательности на две по заданному условию

function Partition<T>(Self: sequence of T; cond: (T,integer)->boolean): (sequence of T, sequence of T);

Разделяет последовательности на две по заданному условию, в котором участвует индекс

function Print<T>(Self: sequence of T; delim: string): sequence of T;

Выводит последовательность на экран, используя delim в качестве разделителя

function Print<T>(Self: sequence of T): sequence of T;

Выводит последовательность на экран, используя пробел в качестве разделителя

function PrintLines<T>(Self: sequence of T): sequence of T;

Выводит последовательность, каждый элемент выводится на новой строке

function PrintLines<T,T1>(Self: sequence of T; map: T->T1): sequence of T;

Выводит последовательность, каждый элемент отображается с помощью функции map и выводится на новой строке

function Println<T>(Self: sequence of T; delim: string): sequence of T;

Выводит последовательность на экран, используя delim в качестве разделителя, и переходит на новую строку

function Println<T>(Self: sequence of T): sequence of T;

Выводит последовательность на экран, используя пробел качестве разделителя, и переходит на новую строку

function SkipLast<T>(self: sequence of T; count: integer := 1): sequence of T;

Возвращает последовательность без последних count элементов

function Slice<T>(Self: sequence of T; from, step: integer): sequence of T;

Возвращает срез последовательности от номера from с шагом step > 0

function Slice<T>(Self: **sequence of T**; from, step, count: integer): **sequence of T**;

Возвращает срез последовательности от номера from с шагом step > 0 длины не более count

function Sorted<T>(Self: **sequence of T**): **sequence of T**;

Возвращает отсортированную по возрастанию последовательность

function SortedDescending<T>(Self: **sequence of T**): **sequence of T**;

Возвращает отсортированную по убыванию последовательность

function SplitAt<T>(Self: **sequence of T**; ind: integer): (**sequence of T**, **sequence of T**);

Разбивает последовательности на две в позиции ind

function Tabulate<T, T1>(Self: **sequence of T**; F: T->T1): **sequence of (T, T1)**;

Табулирует функцию последовательностью

function TakeLast<T>(Self: **sequence of T**; count: integer): **sequence of T**;

Возвращает последние count элементов последовательности

function ToHashSet<T>(Self: **sequence of T**): HashSet<T>;

Возвращает множество HashSet по данной последовательности

function ToLinkedList<T>(Self: **sequence of T**): LinkedList<T>;

Возвращает LinkedList по данной последовательности

function ToSortedSet<T>(Self: **sequence of T**): SortedSet<T>;

Возвращает множество SortedSet по данной последовательности

function UnZipTuple<T, T1>(Self: **sequence of (T, T1)**): (**sequence of T**, **sequence of T1**);

Разъединяет последовательность двухэлементных кортежей на две последовательности

function UnZipTuple<T, T1, T2>(Self: **sequence of (T, T1, T2)**): (**sequence of T**, **sequence of T1**, **sequence of T2**);

Разъединяет последовательность трехэлементных кортежей на три последовательности

function UnZipTuple<T, T1, T2, T3>(Self: **sequence of (T, T1, T2, T3)**): (**sequence of T**, **sequence of T1**, **sequence of T2**, **sequence of T3**);

Разъединяет последовательность четырехэлементных кортежей на четыре последовательности

```
function WriteLine(Self: sequence of string; fname: string): sequence of string;
```

Выводит последовательность строк в файл

```
function ZipTuple<T, T1>(Self: sequence of T; a: sequence of T1): sequence of (T, T1);
```

Объединяет две последовательности в последовательность двухэлементных кортежей

```
function ZipTuple<T, T1, T2>(Self: sequence of T; a: sequence of T1; b: sequence of T2): sequence of (T, T1, T2);
```

Объединяет три последовательности в последовательность трехэлементных кортежей

```
function ZipTuple<T, T1, T2, T3>(Self: sequence of T; a: sequence of T1; b: sequence of T2; c: sequence of T3): sequence of (T, T1, T2, T3);
```

Объединяет четыре последовательности в последовательность четырехэлементных кортежей

Методы расширения одномерных динамических массивов

function AdjacentFind<T>(Self: **array of T**; start: integer := 0): integer; / Находит первую пару подряд идущих одинаковых элементов и возвращает индекс первого элемента пары. Если не найден, возвращается -1

function AdjacentFind<T>(Self: **array of T**; eq: (T,T)->boolean; start: integer := 0): integer;

Находит первую пару подряд идущих одинаковых элементов, используя функцию сравнения eq, и возвращает индекс первого элемента пары. Если не найден, возвращается -1

function BinarySearch<T>(Self: **array of T**; x: T): integer;

Выполняет бинарный поиск в отсортированном массиве

function ConvertAll<T, T1>(Self: **array of T**; converter: T->T1): **array of T1**;

Преобразует элементы массива и возвращает преобразованный массив

function ConvertAll<T, T1>(Self: **array of T**; converter: (T,integer)->T1): **array of T1**;

Преобразует элементы массива и возвращает преобразованный массив

procedure Fill<T>(Self: **array of T**; f: integer -> T);

Заполняет элементы массива значениями, вычисляемыми по некоторому правилу

function Find<T>(Self: **array of T**; p: T->boolean): T;

Выполняет поиск первого элемента в массиве, удовлетворяющего предикату. Если не найден, возвращается нулевое значение соответствующего типа

function FindAll<T>(Self: **array of T**; p: T->boolean): **array of T**;

Возвращает в виде массива все элементы, удовлетворяющие предикату

function FindIndex<T>(Self: **array of T**; p: T->boolean): integer;

Выполняет поиск индекса первого элемента в массиве, удовлетворяющего предикату. Если не найден, возвращается -1

function FindIndex<T>(Self: **array of T**; start: integer; p: T->boolean): integer;

Выполняет поиск индекса первого элемента в массиве, удовлетворяющего предикату, начиная с индекса start. Если не найден, возвращается -1

```
function FindLast<T>(Self: array of T; p: T->boolean): T;
```

Выполняет поиск последнего элемента в массиве, удовлетворяющего предикату. Если не найден, возвращается нулевое значение соответствующего типа

```
function FindLastIndex<T>(Self: array of T; p: T->boolean):  
integer;
```

Выполняет поиск индекса последнего элемента в массиве, удовлетворяющего предикату. Если не найден, возвращается -1

```
function FindLastIndex<T>(self: array of T; start: integer;  
p: T->boolean): integer;
```

Выполняет поиск индекса последнего элемента в массиве, удовлетворяющего предикату, в диапазоне индексов от 0 до start. Если не найден, возвращается -1

```
function High(Self: System.Array);
```

Возвращает индекс последнего элемента массива

```
function Indexes<T>(Self: array of T): sequence of integer;
```

Возвращает последовательность индексов одномерного массива

```
function IndexesOf<T>(Self: array of T; cond: T->boolean):  
sequence of integer;
```

Возвращает последовательность индексов элементов одномерного массива, удовлетворяющих условию

```
function IndexesOf<T>(Self: array of T; cond: (T,integer) -  
>boolean): sequence of integer;
```

Возвращает последовательность индексов элементов одномерного массива, удовлетворяющих условию

```
function IndexMax<T>(self: array of T; index: integer :=  
0): integer; where T: IComparable<T>;
```

Возвращает индекс первого максимального элемента начиная с позиции index

```
function IndexMin<T>(Self: array of T; index: integer :=  
0): integer; where T: IComparable<T>;
```

/ Возвращает индекс первого минимального элемента начиная с позиции index

```
function IndexOf<T>(Self: array of T; x: T): integer;
```

Возвращает индекс первого вхождения элемента или -1 если элемент

не найден

```
function IndexOf<T>(Self: array of T; x: T; start: integer): integer;
```

Возвращает индекс первого вхождения элемента начиная с индекса start или -1 если элемент не найден

```
function LastIndexMax<T>(Self: array of T): integer; where T: IComparable<T>;
```

Возвращает индекс последнего минимального элемента

```
function LastIndexMax<T>(Self: array of T; index: integer): integer; where T: IComparable<T>;
```

Возвращает индекс последнего минимального элемента в диапазоне [0,index]

```
function LastIndexMin<T>(Self: array of T): integer; where T: IComparable<T>;
```

Возвращает индекс последнего минимального элемента

```
function LastIndexMin<T>(Self: array of T; index: integer): integer; where T: IComparable<T>;
```

Возвращает индекс последнего минимального элемента в диапазоне [0,index]

```
function LastIndexOf<T>(Self: array of T; x: T): integer;
```

Возвращает индекс последнего вхождения элемента или -1 если элемент не найден

```
function LastIndexOf<T>(Self: array of T; x: T; start: integer): integer;
```

Возвращает индекс последнего вхождения элемента начиная с индекса start или -1 если элемент не найден

```
function Low(Self: System.Array);
```

Возвращает индекс первого элемента массива

```
function Max<T>(Self: array of T): T; where T: IComparable<T>;
```

Возвращает максимальный элемент

```
function Max(Self: array of integer): integer;
```

Возвращает максимальный элемент

```
function Max(Self: array of real): real;
```

Возвращает максимальный элемент

```
function Min<T>(Self: array of T): T; where T: IComparable<T>;
```

Возвращает минимальный элемент

function Min(Self: **array of integer**): integer;

Возвращает минимальный элемент

function Min(Self: **array of real**): real;

Возвращает минимальный элемент

procedure Replace<T>(Self: **array of T**; oldValue, newValue: T);

Заменяет в массиве все вхождения одного значения на другое

function Shuffle<T>(Self: **array of T**): **array of T**;

Перемешивает элементы массива случайным образом

function Slice<T>(Self: **array of T**; from, step: integer): **array of T**;

Возвращает срез массива от индекса from с шагом step

function Slice<T>(Self: **array of T**; from, step, count: integer): **array of T**;

Возвращает срез массива от индекса from с шагом step длины не более count

procedure Sort<T>(Self: **array of T**);

Сортирует массив по возрастанию

procedure Sort<T>(Self: **array of T**; cmp: (T, T) ->integer);

Сортирует массив по возрастанию, используя cmp в качестве функции сравнения элементов

procedure Transform<T>(self: **array of T**; f: T -> T);

Преобразует элементы массива по заданному правилу

Методы расширения двумерных динамических массивов

function Col<T>(Self: **array** [,] **of** T; k: **integer**): **array of** T;
к-тый столбец двумерного массива

function ColCount<T>(Self: **array** [,] **of** T): **integer**;

Количество столбцов в двумерном массиве

function Cols<T>(Self: **array** [,] **of** T): **sequence of sequence of** T;

Возвращает последовательность столбцов двумерного массива

function ColSeq<T>(Self: **array** [,] **of** T; k: **integer**): **sequence of** T;

к-тый столбец двумерного массива как последовательность

function ConvertAll<T, T1>(Self: **array** [,] **of** T; converter: T->T1): **array** [,] **of** T1;

Преобразует элементы двумерного массива и возвращает преобразованный массив

function ConvertAll<T, T1>(Self: **array** [,] **of** T; converter: (T, **integer**, **integer**)->T1): **array** [,] **of** T1;

Преобразует элементы двумерного массива и возвращает преобразованный массив

function ElementsByCol<T>(Self: **array** [,] **of** T): **sequence of** T;

Возвращает по заданному двумерному массиву последовательность его элементов по столбцам

function ElementsByRow<T>(Self: **array** [,] **of** T): **sequence of** T;

Возвращает по заданному двумерному массиву последовательность его элементов по строкам

function ElementsWithIndexes<T>(Self: **array** [,] **of** T): **sequence of** (T, **integer**, **integer**);

Возвращает по заданному двумерному массиву последовательность (a[i,j],i,j)

procedure Fill<T>(Self: **array** [,] **of** T; f: (**integer**, **integer**) ->T);

Заполняет элементы двумерного массива значениями, вычисляемыми по некоторому правилу

```
function Print<T>(Self: array [,] of T; w: integer := 4):  
array [,] of T;
```

Вывод двумерного массива, w - ширина поля вывода

```
function Print(Self: array [,] of real; w: integer := 7; f:  
integer := 2): array [,] of real;
```

Вывод двумерного вещественного массива по формату :w:f

```
function Println<T>(Self: array [,] of T; w: integer := 4):  
array [,] of T;
```

Вывод двумерного массива и переход на следующую строку, w - ширина поля вывода

```
function Println(Self: array [,] of real; w: integer := 7;  
f: integer := 2): array [,] of real;
```

Вывод двумерного вещественного массива по формату :w:f и переход на следующую строку

```
function Row<T>(Self: array [,] of T; k: integer): array of  
T;
```

k-тая строка двумерного массива

```
function RowCount<T>(Self: array [,] of T): integer;
```

Количество строк в двумерном массиве

```
function Rows<T>(Self: array [,] of T): sequence of  
sequence of T;
```

Возвращает последовательность строк двумерного массива

```
function RowSeq<T>(Self: array [,] of T; k: integer):  
sequence of T;
```

k-тая строка двумерного массива как последовательность

```
procedure SetCol<T>(Self: array [,] of T; k: integer; a:  
array of T);
```

Меняет столбец k двумерного массива на другой столбец

```
procedure SetCol<T>(Self: array [,] of T; k: integer; a:  
sequence of T);
```

Меняет столбец k двумерного массива на другой столбец

```
procedure SetRow<T>(Self: array [,] of T; k: integer; a:  
array of T);
```

Меняет строку k двумерного массива на другую строку

```
procedure SetRow<T>(Self: array [,] of T; k: integer; a:  
sequence of T);
```

Меняет строку k двумерного массива на другую строку

```
procedure SwapCols<T>(Self: array [,] of T; k1, k2:  
integer);
```

Меняет местами два столбца двумерного массива с номерами k1 и k2

```
procedure SwapRows<T>(Self: array [,] of T; k1, k2:  
integer);
```

Меняет местами две строки двумерного массива с номерами k1 и k2

```
procedure Transform<T>(Self: array [,] of T; f: T->T);
```

Преобразует элементы двумерного массива по заданному правилу

```
procedure Transform<T>(Self: array [,] of T; f:  
(T, integer, integer)->T);
```

Преобразует элементы двумерного массива по заданному правилу

Методы расширения списков

function AdjacentFind<T>(Self: IList<T>; start: integer := 0): integer; Находит первую пару подряд идущих одинаковых элементов и возвращает индекс первого элемента пары. Если не найден, возвращается -1

function AdjacentFind<T>(Self: IList<T>; eq: (T,T)->boolean; start: integer := 0): integer;

Находит первую пару подряд идущих одинаковых элементов, используя функцию сравнения eq, и возвращает индекс первого элемента пары. Если не найден, возвращается -1

procedure Fill<T>(Self: IList<T>; f: integer->T);

Заполняет элементы массива или списка значениями, вычисляемыми по некоторому правилу

function IndexMax<T>(self: IList<T>; index: integer := 0): integer; **where** T: IComparable<T>;

Возвращает индекс первого максимального элемента начиная с позиции index

function IndexMin<T>(Self: IList<T>; index: integer := 0): integer; **where** T: IComparable<T>;

Возвращает индекс первого минимального элемента начиная с позиции index

function LastIndexMax<T>(Self: IList<T>): integer; **where** T: IComparable<T>;

Возвращает индекс последнего максимального элемента

function LastIndexMax<T>(Self: IList<T>; index: integer): integer; **where** T: IComparable<T>;

Возвращает индекс последнего минимального элемента в диапазоне [0,index-1]

function LastIndexMin<T>(Self: IList<T>): integer; **where** T: IComparable<T>;

Возвращает индекс последнего минимального элемента

function LastIndexMin<T>(Self: IList<T>; index: integer): integer; **where** T: IComparable<T>;

Возвращает индекс последнего минимального элемента в диапазоне [0,index-1]

function RemoveLast<T>(Self: List<T>): List<T>;

Удаляет последний элемент. Если элементов нет, генерирует исключение

```
procedure Replace<T>(Self: IList<T>; oldValue, newValue: T);
```

Заменяет в массиве или списке все вхождения одного значения на другое

```
function Shuffle<T>(Self: List<T>): List<T>;
```

Перемешивает элементы списка случайным образом

```
function Slice<T>(Self: List<T>; from, step: integer): List<T>;
```

Возвращает срез списка от индекса from с шагом step

```
function Slice<T>(Self: List<T>; from, step, count: integer): List<T>;
```

Возвращает срез списка от индекса from с шагом step длины не более count

```
procedure Transform<T>(Self: IList<T>; f: T->T);
```

Преобразует элементы массива или списка по заданному правилу

```
procedure Transform<T>(Self: IList<T>; f: (T, integer)->T);
```

Преобразует элементы массива или списка по заданному правилу

Методы расширения типа `integer`

function `Between(Self: integer; a, b: integer): boolean;`

Возвращает `True` если значение находится между двумя другими

function `Downto(Self: integer; n: integer): sequence of integer;`

Генерирует последовательность целых от текущего значения до `n` в убывающем порядке

function `InRange(Self: integer; a,b: integer): boolean;`

Возвращает `True` если значение находится между двумя другими

function `IsEven(Self: integer): boolean;`

Возвращает, является ли целое четным

function `IsOdd(Self: integer): boolean;`

Возвращает, является ли целое нечетным

function `Range(Self: integer): sequence of integer;`

Возвращает последовательность чисел от 1 до данного

function `Sqr(Self: integer): integer;`

Возвращает квадрат числа

function `Sqrt(Self: integer): real;`

Возвращает квадратный корень числа

function `Times(Self: integer): sequence of integer;`

Возвращает последовательность целых $0,1,\dots,n-1$

function `To(Self: integer; n: integer): sequence of integer;`

Генерирует последовательность целых от текущего значения до `n`

Методы расширения типа BigInteger

function Sqrt(Self: BigInteger): real; Возвращает
квадратный корень числа

Методы расширения типа `real`

function `Between(Self: real; a, b: real): boolean;`

Возвращает `True` если значение находится между двумя другими

function `InRange(Self: real; a,b: real): boolean;`

Возвращает `True` если значение находится между двумя другими

function `Round(Self: real): integer;`

Возвращает число, округленное до ближайшего целого

function `Round(Self: real; digits: integer): real;`

Возвращает `x`, округленное до ближайшего вещественного с `digits` знаками после десятичной точки

function `RoundBigInteger(Self: real): BigInteger;`

Возвращает число, округленное до ближайшего длинного целого

function `Sqr(Self: real): real;`

Возвращает квадрат числа

function `Sqrt(Self: real): real;`

Возвращает квадратный корень числа

function `ToString(Self: real; frac: integer): string;`

Возвращает вещественное, отформатированное к строке с `frac` цифрами после десятичной точки

function `Trunc(Self: real): integer;`

Возвращает целую часть вещественного числа

function `TruncBigInteger(Self: real): BigInteger;`

Возвращает целую часть вещественного числа как длинное целое

Методы расширения типа char

function Between(Self: char; a, b: char): boolean;

Возвращает True если значение находится между двумя другими

function Code(Self: char): integer;

Код символа в кодировке Unicode

function InRange(Self: char; a,b: char): boolean;

Возвращает True если значение находится между двумя другими

function IsDigit(Self: char);

Является ли символ цифрой

function IsLetter(Self: char): boolean;

Является ли символ буквой

function IsLower(Self: char): boolean;

Принадлежит ли символ к категории букв нижнего регистра

function IsUpper(Self: char): boolean;

Принадлежит ли символ к категории букв верхнего регистра

function Pred(Self: char);

Предыдущий символ

function Succ(Self: char);

Следующий символ

function ToDigit(Self: char): integer;

Преобразует символ в цифру

function ToLower(Self: char): char;

Преобразует символ в нижний регистр

function ToUpper(Self: char): char;

Преобразует символ в верхний регистр

Методы расширения типа string

function Between(Self: string; a, b: string): boolean;

Возвращает True если значение находится между двумя другими

function InRange(Self: string; a,b: string): boolean;

Возвращает True если значение находится между двумя другими

function Inverse(Self: string): string;

Возвращает инверсию строки

function IsMatch(Self: string; reg: string; options: RegexOptions := RegexOptions.None): boolean;

Удовлетворяет ли строка регулярному выражению

function Left(Self: string; length: integer): string;

Возвращает подстроку, полученную вырезанием из строки length самых левых символов

function Matches(Self: string; reg: string; options: RegexOptions := RegexOptions.None): **sequence of** Match;

Ищет в указанной строке все вхождения регулярного выражения и возвращает их в виде последовательности элементов типа Match

function MatchValue(Self: string; reg: string; options: RegexOptions := RegexOptions.None): string;

Ищет в указанной строке первое вхождение регулярного выражения и возвращает его в виде строки

function MatchValues(Self: string; reg: string; options: RegexOptions := RegexOptions.None): **sequence of** string;

Ищет в указанной строке все вхождения регулярного выражения и возвращает их в виде последовательности строк

function ReadInteger(Self: string; **var** from: integer): integer;

Считывает целое из строки начиная с позиции from и устанавливает from за считанным значением

function ReadReal(Self: string; **var** from: integer): real;

Считывает вещественное из строки начиная с позиции from и устанавливает from за считанным значением

function ReadWord(Self: string; **var** from: integer): string;

Считывает слово из строки начиная с позиции from и устанавливает from за считанным значением

function RegexReplace(Self: string; reg, repl: string; options: RegexOptions := RegexOptions.None): string;

Заменяет в указанной строке все вхождения регулярного выражения указанной строкой замены и возвращает преобразованную строку

```
function RegexReplace(Self: string; reg: string; repl: Match->string; options: RegexOptions := RegexOptions.None): string;
```

Заменяет в указанной строке все вхождения регулярного выражения указанным преобразованием замены и возвращает преобразованную строку

```
function Remove(Self: string; params targets: array of string): string;
```

Удаляет в строке все вхождения указанных строк

```
function Right(Self: string; length: integer): string;
```

Возвращает подстроку, полученную вырезанием из строки length самых правых символов

```
function Slice(Self: string; from, step: integer): string;
```

Возвращает срез строки от индекса from с шагом step

```
function Slice(Self: string; from, step, count: integer): string;
```

Возвращает срез строки от индекса from с шагом step длины не более count

```
function ToBigInteger(Self: string): BigInteger;
```

Преобразует строку в BigInteger

```
function ToInteger(Self: string): integer;
```

Преобразует строку в целое

```
function ToInteger(Self: string; defaultvalue: integer): integer;
```

Преобразует строку в целое При невозможности преобразования возвращается defaultvalue

```
function ToIntegers(Self: string): array of integer;
```

Преобразует строку в массив целых

```
function ToReal(Self: string): real;
```

Преобразует строку в вещественное

```
function ToReal(Self: string; defaultvalue: real): real;
```

Преобразует строку в вещественное При невозможности преобразования возвращается defaultvalue

```
function ToReals(Self: string): array of real;
```

Преобразует строку в массив вещественных

```
function ToWords(Self: string; params delim: array of
```

char): **array of string**;

Преобразует строку в массив слов

function TryToInteger(Self: string; **var** value: integer):
boolean;

Преобразует строку в целое и записывает его в value. При
невозможности преобразования возвращается False

function TryToReal(Self: string; **var** value: real): boolean;

Преобразует строку в вещественное и записывает его в value. При
невозможности преобразования возвращается False

Методы расширения типа Func

```
function Compose<T1, T2, TResult>(Self: T2->TResult;  
composer: T1->T2): T1->TResult;    Суперпозиция функций
```

Методы расширения типа `Complex`

`function Conjugate(Self: Complex): Complex;`
комплексно сопряженное значение

Возвращает

Методы расширения словарей

function Get<Key, Value>(Self: IDictionary<Key, Value>; K: Key): Value; Возвращает в словаре значение, связанное с указанным ключом, а если такого ключа нет, то значение по умолчанию

Общие методы файловых типов

Для всех стандартных файловых типов (`Text`, `file`, `file of T`) определены следующие методы:

procedure `Write(a, b, ...);` Записывает в файл значения `a`, `b`, ...

function `Name: string;`

Возвращает имя файла

function `FullName: string;`

Возвращает полное имя файла

function `Eof: boolean;`

Возвращает `True`, если достигнут конец файла, и `False` в противном случае

procedure `Close;`

Закрывает файл

procedure `Erase;`

Удаляет файл

procedure `Rename(newname: string);`

Переименовывает файл, давая ему имя `newname`

Методы текстовых файлов

Для текстовых файлов (тип `Text`) определены следующие методы:

procedure `Write(a,b,...);` Записывает в файл значения a, b, ...

procedure `Writeln(a,b,...);`

Записывает в файл значения a, b, ... и осуществляет переход на следующую строку

procedure `Print(a,b,...);`

Записывает в файл значения a, b, ..., разделяя их пробелами

procedure `Println(a,b,...);`

Записывает в файл значения a, b, ..., разделяя их пробелами, и осуществляет переход на следующую строку

function `ReadInteger: integer;`

Возвращает значение типа `integer`, введенное из текстового файла

function `ReadReal: real;`

Возвращает значение типа `real`, введенное из текстового файла

function `ReadChar: char;`

Возвращает значение типа `char`, введенное из текстового файла

function `ReadString: string;`

Возвращает значение типа `string`, введенное из текстового файла, без перехода на следующую строку

function `ReadBoolean: boolean;`

Возвращает значение типа `boolean`, введенное из текстового файла

function `ReadWord: string;`

Возвращает слово, введенное из текстового файла

function `ReadlnInteger: integer;`

Возвращает значение типа `integer`, введенное из текстового файла, и переходит на следующую строку

function `ReadlnReal: real;`

Возвращает значение типа `real`, введенное из текстового файла, и переходит на следующую строку

function `ReadlnChar: char;`

Возвращает значение типа `char`, введенное из текстового файла, и переходит на следующую строку

function `ReadlnString: string;`

Возвращает значение типа `string`, введенное из текстового файла, и

переходит на следующую строку

function ReadLnBoolean: boolean;

Возвращает значение типа boolean, введенное из текстового файла, и переходит на следующую строку

function ReadLnWord: string;

Возвращает слово, введенное из текстового файла, и переходит на следующую строку

function Eof: boolean;

Возвращает True, если достигнут конец файла, и False в противном случае

function Eoln: boolean;

Возвращает True, если достигнут конец строки, и False в противном случае

function SeekEof: boolean;

Пропускает пробельные символы, после чего возвращает True, если достигнут конец файла

function SeekEoln: boolean;

Пропускает пробельные символы, после чего возвращает True, если достигнут конец строки в файле

function Name: string;

Возвращает имя файла

function FullName: string;

Возвращает полное имя файла

procedure Reset;

Устанавливает файловый указатель на начало файла

procedure Close;

Закрывает файл

procedure Flush;

Записывает содержимое буфера файла на диск

function ReadToEnd: string;

Возвращает в виде строки содержимое файла от текущего положения до конца

procedure Erase;

Удаляет файл (файл должен быть закрыт)

procedure Rename(newname: string);

Переименовывает файл, давая ему имя newname (файл должен быть закрыт)

Методы двоичных файлов

Для двоичных файлов **file** определены следующие методы:

function Position: int64; Возвращает или устанавливает текущую позицию файлового указателя в двоичном файле

function Size: int64;

Возвращает количество элементов в двоичном файле

procedure Seek(n: int64);

Устанавливает текущую позицию файлового указателя в двоичном файле на байт с номером n

procedure Truncate;

Усекает двоичный файл, отбрасывая все элементы с позиции файлового указателя

Методы типизированных файлов

Для типизированных файлов `file of T` определены следующие методы:

function `Position: int64;` Возвращает текущую позицию файлового указателя в типизированном файле

function `Size: int64;`

 Возвращает количество элементов в типизированном файле

procedure `Seek(n: int64);`

 Устанавливает текущую позицию файлового указателя в типизированном файле на элемент с номером `n`

procedure `Truncate;`

 Усекает типизированный файл, отбрасывая все элементы с позиции файлового указателя

Методы расширения типизированных файлов

function Read<T>(Self: **file of T**): T; Читывает и возвращает следующий элемент типизированного файла

function Read2<T>(Self: **file of T**): (T,T);

Считывает и возвращает два следующих элемента типизированного файла в виде кортежа

function Read3<T>(Self: **file of T**): (T,T,T);

Считывает и возвращает три следующих элемента типизированного файла в виде кортежа

function ReadElements<T>(Self: **file of T**): **sequence of T**;

Возвращает последовательность элементов открытого типизированного файла от текущего элемента до конечного

function ReadElements<T>(fname: string): **sequence of T**;

Открывает типизированный файл, возвращает последовательность его элементов и закрывает его

procedure Reset<T>(Self: **file of T**);

Открывает существующий типизированный файл

procedure Rewrite<T>(Self: **file of T**);

Создает новый или обнуляет существующий типизированный файл

function Seek<T>(Self: **file of T**; n: int64): **file of T**;

Устанавливает текущую позицию файлового указателя в типизированном файле на элемент с номером n

procedure Write<T>(Self: **file of T**; **params** vals: **array of T**);

Записывает данные в типизированный файл

OpenMP: обзор

OpenMP – [открытый стандарт для распараллеливания программ](#) на многопроцессорных системах с общей памятью (например, на многоядерных процессорах). OpenMP реализует параллельные вычисления с помощью многопоточности: главный поток создает набор подчиненных потоков, и задача распределяется между ними.

OpenMP представляет собой набор директив компилятора, которые управляют процессом автоматического выделения потоков и данными, требуемыми для работы этих потоков.

В системе PascalABC.NET реализованы следующие элементы OpenMP:

- Конструкции для создания и распределения работы между потоками (директивы `parallel for` и `parallel sections`)
- Конструкции для синхронизации потоков (директива `critical`)

Директивы имеют следующий вид:

```
{$omp directive-name [опция[[,] опция]...]}
```

Здесь `$omp` означает то, что это директива OpenMP, `directive-name` – имя директивы, например `parallel`, после чего могут быть опции.

Директива относится к тому оператору, перед которым она находится.

Примеры использования OpenMP находятся в папке `Samples/OMPSamples`

Ниже приводится описание директив.

[Директива `parallel for`](#)

[Редукция в директиве `parallel for`](#)

[Параллельные секции и директива `parallel sections`](#)

[Синхронизация и директива `critical`](#)

Директива `parallel for`

Директива `parallel for` обеспечивает распараллеливание следующего за ней цикла.

```
{$omp parallel for}
  for var i: integer:=1 to 10 do
    тело цикла
```

Здесь будет создано несколько потоков и разные итерации цикла будут распределены по этим потокам. Количество потоков, как правило, совпадает с количеством ядер процессора, но в некоторых случаях могут быть отличия, например, если поток ожидает ввод данных от пользователя, могут создаваться дополнительные потоки, чтобы по возможности задействовать все доступные ядра.

Все переменные, описанные вне параллельного цикла, будут разделяемыми, то есть, если в теле цикла есть обращение к таким переменным, все потоки будут обращаться к одной и той же ячейке памяти. Все переменные, объявленные внутри цикла, будут частными, то есть у каждого потока будет своя копия этой переменной.

Опция `private` позволяет переменные, описанные вне цикла, сделать частными. Опция записывается так:

```
{$omp parallel for private(список переменных)}
```

Список переменных – одна или несколько переменных через запятую.

```
var a,b: integer;
{$omp parallel for private(a, b)}
for var i: integer:=1 to 10 do
  a := ...
```

В этом случае переменные `a` и `b` будут частными, и присваивание этим переменным в одном потоке не будет влиять на другие потоки.

Ограничение: счетчики распараллеливаемого цикла и вложенных циклов должны быть объявлены в заголовке цикла.

Не все циклы можно распараллеливать. Если на разных итерациях происходит обращение к одной и той же переменной и при этом ее

значение меняется – распараллеливание такого цикла приведет к ошибкам, при разных запусках могут получаться разные результаты в зависимости от того, в каком порядке происходили обращения к этой переменной.

```
{$omp parallel for}
  for var i:=1 to 2 do
    a[i] := a[i+1];
```

Здесь на первой итерации происходит чтение второго элемента массива, а на второй итерации – запись этого же элемента. Если первая итерация выполнится раньше второй – в первый элемент массива запишется значение из второго, а если позже – то из третьего элемента массива.

```
var a:integer;
{$omp parallel for}
for var i:=1 to 10 do
begin
  a := i;
  ... := a; //к этому моменту a может быть изменено
           другим потоком
end;
```

Значение переменной a после этого цикла может быть любым в диапазоне от 1 до 10.

Наиболее эффективно распараллеливаются циклы, каждая итерация которых выполняется достаточно долго. Если тело цикла состоит из небольшого количества простых операторов, затраты на создание потоков и распределение нагрузки между ними могут превысить выигрыш от параллельного выполнения цикла.

Пример параллельного перемножения матриц

Перемножение матриц - классический пример иллюстрации параллельности. Вычисление различных элементов матрицы происходит независимо, поэтому не надо предусматривать никаких средств синхронизации.

```
uses Arrays;

procedure ParallelMult(a,b,c: array [,] of real; n:
integer);
```

```

begin
  {$omp parallel for }
  for var i:=0 to n-1 do
    for var j:=0 to n-1 do
      begin
        c[i,j]:=0;
        for var l:=0 to n-1 do
          c[i,j]:=c[i,j]+a[i,l]*b[l,j];
        end;
      end;
    end;
  end;

procedure Mult(a,b,c: array [,] of real; n: integer);
begin
  for var i:=0 to n-1 do
    for var j:=0 to n-1 do
      begin
        c[i,j]:=0;
        for var l:=0 to n-1 do
          c[i,j]:=c[i,j]+a[i,l]*b[l,j];
        end;
      end;
    end;
  end;

const n = 400;

begin
  var a := Arrays.CreateRandomRealMatrix(n,n);
  var b := Arrays.CreateRandomRealMatrix(n,n);
  var c := new real[n,n];
  ParallelMult(a,b,c,n);
  writeln('Параллельное перемножение матриц:
  ',Milliseconds,' миллисекунд');
  var d := Milliseconds;
  Mult(a,b,c,n);
  writeln('Непараллельное перемножение матриц:
  ',Milliseconds-d,' миллисекунд');
end.

```

Редукция в директиве `parallel for`

Часто в цикле накапливается значение некоторой переменной, перед циклом эта переменная инициализируется, а на каждой итерации к ней добавляется некоторое значение или умножается на некоторое значение. Эта переменная должна быть объявлена вне цикла, а значит, будет общей. В таком случае возможны ошибки при параллельном выполнении:

```
var a: integer:=0;
{$omp parallel for}
for var i:integer:=1 to 100 do
  a := a+1;
```

Два потока могут считать старое значение, затем первый поток прибавит единицу и запишет в переменную `a`, затем второй поток прибавит единицу к старому значению и запишет результат в переменную `a`. При этом изменения, сделанные первым потоком, будут потеряны. Правильная работа программы возможна при некоторых запусках, но возможны и ошибки.

Опция `reduction` позволяет обеспечить правильное накопление результата:

```
{$omp parallel for reduction(действие : список
переменных)}
```

При этом все переменные из списка будут объявлены частными, таким образом, разные потоки будут работать со своими экземплярами переменных. Эти экземпляры будут инициализированы в зависимости от действия, а в конце цикла новое значение переменной будет получено из значения этой переменной до цикла и всех частных копий применением действия из опции.

```
var a: integer := 1;
{$omp parallel for reduction(+:a)}
for var i: integer:=1 to 2 do
  a := a+1;
```

Здесь начальное значение переменной `a` – единица, для действия `+` локальные копии будут инициализированы нулями, будет выполнено две итерации и у каждого потока локальная копия переменной `a`

примет значение 1. После завершения цикла к начальному значению (1) будут прибавлены обе локальные копии, и результирующее значение переменной a будет равно 3, так же как и при последовательном выполнении.

В таблице приведены допустимые операторы редукции и значения, которыми инициализируются локальные копии переменной редукции:

Оператор раздела reduction	Инициализированное значение
+	0
*	1
-	0
and (побитовый)	~0 (каждый бит установлен)
or (побитовый)	0
xor (побитовый)	0
and (логический)	true
or (логический)	false

Параллельные секции и директива `parallel sections`

Директива `parallel sections` обеспечивает параллельное выполнение нескольких операторов, простых или составных.

```
{$omp parallel sections}
begin
  секция 1;
  секция 2;
  ...;
end;
```

Каждый оператор в блоке `begin ... end`, следующем за директивой является отдельной секцией.

```
{$omp parallel sections}
begin
  оператор 1;
  оператор 2;
  begin
    оператор 3;
    оператор 4;
    оператор 5;
  end;
end;
```

Здесь описаны три параллельные секции, первая – оператор 1, вторая – оператор 2 и третья – блок `begin ... end`, состоящий из операторов 3-5.

Все переменные, описанные вне параллельных секций, будут разделяемыми, то есть, если в секциях есть обращение к таким переменным, то потоки, выполняющие эти секции, будут обращаться к одной и той же ячейке памяти. Все переменные, объявленные внутри секции, будут доступны только в той секции, в которой они объявлены.

Корректная работа параллельных секций возможна, только если секции независимы друг от друга – если они могут выполняться в любом порядке, не обращаются к одним и тем же переменным и не изменяют их.

Синхронизация и директива `critical`

Директива `critical` исключает параллельное выполнение следующего за ней оператора.

```
{$omp critical имя}  
оператор;
```

Этот оператор образует критическую секцию – участок кода, который не может выполняться одновременно несколькими потоками.

Только критические секции с одинаковыми именами не могут выполняться одновременно. Если один поток уже выполняет критическую секцию, а второй пытается войти в секцию с таким же именем, он будет заблокирован до тех пор, пока первый поток не выйдет за пределы критической секции.

Критические секции можно использовать при обращении к общим переменным, чтобы избежать потерь данных.

```
var a:integer:=0;  
{$omp parallel for}  
for var i:integer:=1 to 100 do  
  {$omp critical}  
  a:=a+1;
```

Здесь критическую секцию можно использовать вместо редукции. Весь оператор `a:=a+1` выполнится одним потоком и только потом – другим. Однако использование критических секций обычно снижает эффективность за счет последовательного выполнения этих участков. В этом примере все тело цикла является критической секцией, поэтому весь цикл будет выполнен последовательно.

Но не во всех случаях использование критических секций помогает обеспечить корректную работу параллельных конструкций.

```
var a:integer := 0;  
{$omp parallel sections}  
begin  
  {$omp critical}  
  a:=1;  
  {$omp critical}  
  a:=a+1;  
end;
```

Значение переменной a зависит от того, в каком порядке выполняются секции. Если первая секция выполнится раньше, значение a будет равно двум, иначе – единице.

При использовании критических секций возможно возникновение взаимоблокировок. Например, первый поток выполняет код, содержащий критическую секцию А, внутри которой есть критическая секция В. Второй поток выполняет код, содержащий критическую секцию В, внутри которой есть критическая секция А. Возможен такой порядок выполнения: первый поток входит в секцию А и не успевает войти в секцию В. Второй поток входит в секцию В и не может войти в секцию А, так как эта секция уже выполняется другим потоком. Первый поток не может продолжить выполнение, так как секция В уже выполняется другим потоком. Оба потока оказываются заблокированными.

Директивы компилятора

Директивы компилятора - это специальные команды компилятору в ходе компиляции, записанные в тексте программы внутри последовательности {\$... }. Фигурные скобки обозначают комментарий, но наличие знака \$ после { говорит о том, что внутри комментария располагается директива компилятора.

Общий вид директивы компилятора:

```
{$Имядирективы параметры}
```

Список директив компилятора

{\$apptype <тип приложения>} - задание типа приложения (windows/console).

{\$reference <имя файла>} - подключение библиотеки.

{\$gendoc <параметр>} - генерация документации в XML формате. Параметры: **true**, **false**.

{\$mainresource <имя файла>} - подключение .res файла в качестве неуправляемого ресурса

{\$resource <имя файла>} - подключение файла в качестве управляемого ресурса

{\$region <имя региона>} - начало региона (используется в редакторе в режиме сворачивания кода

{\$endregion} - конец региона

{\$product <название продукта>} - название продукта

{\$version <версия продукта>} - версия продукта

{\$company <компания>} - компания

{\$copyright <копирайт>} - копирайт

{\$strademark <торговая марка>} - торговая марка

{\$include <имя файла>} - включение в текст программы содержимого указанного файла.

{\$define <идентификатор>} - определение имени, используемого в директивах **\$ifdef**, **\$ifndef**.

{\$undef <идентификатор>} - исключение имени, используется для отмены действия директивы **\$define**.

{\$ifdef <идентификатор>} - начало блока условной компиляции (проверяется условие: "идентификатор определен").

{\$ifndef <идентификатор>} - начало блока условной компиляции (проверяется условие: "идентификатор не определен").

{\$else} - директива "иначе" в блоке условной компиляции.

{\$endif} - завершение блока условной компиляции.

{\$faststrings} - строки с быстрым доступом к символам на запись, но со ссылочной семантикой.

{\$string_nullbased+} - включение строк, индексируемых с 0.

{\$string_nullbased-} - выключение строк, индексируемых с 0.

Директивы **\$ifdef**, **\$ifndef** совместно с директивами **\$else** и **\$endif** управляют условной компиляцией частей исходного файла. Каждой директиве **\$ifdef**, **\$ifndef** должна соответствовать завершающая ее директива **\$endif**. Между директивами **\$ifdef**, **\$ifndef** и **\$endif** допускается произвольное количество блоков условной компиляции (в том числе вложенных) и не более одной директивы **\$else**.

Пример. Использование директив условной компиляции.

```
{$define DEBUG}
begin
  {$ifndef DEBUG}
    writeln('Имя DEBUG не определено');
  {$else}
    writeln('Имя DEBUG определено');
  {$endif}
end.
```

Модуль GraphABC

Модуль [GraphABC](#) представляет собой простую графическую библиотеку и предназначен для создания несобытийных графических и анимационных программ в процедурном и частично в объектном стиле. Рисование осуществляется в специальном *графическом окне*, возможность рисования в нескольких окнах отсутствует. Кроме этого, в модуле [GraphABC](#) определены простейшие события мыши и клавиатуры, позволяющие создавать элементарные событийные приложения. Основная сфера использования модуля [GraphABC](#) - обучение.

Модуль [GraphABC](#) основан на графической библиотеке GDI+, но запоминает текущие перо, кисть и шрифт, что позволяет не передавать их в качестве параметров при вызове графических примитивов. К свойствам пера, кисти и шрифта можно получать доступ как в процедурном, так и в объектном стиле. Например, для доступа к цвету текущего пера используется процедура [SetPenColor\(c\)](#) и функция [PenColor](#), а также свойство [Pen.Color](#).

В модуле [GraphABC](#) можно управлять самим графическим окном и компонентом [GraphABCControl](#), на котором осуществляется рисование. По умолчанию компонент [GraphABCControl](#) занимает всю клиентскую часть графического окна, однако, на графическое окно можно добавить элементы управления, уменьшив область, занимаемую графическим компонентом (например, так сделано в модулях [Robot](#) и [Drawman](#)).

Для работы с рисунками используется класс [Picture](#), позволяющий рисовать на себе те же графические примитивы, что и на экране.

Режим блокировки рисования на экране ([LockDrawing](#)) позволяет осуществлять прорисовку лишь во внеэкранный буфере, после чего с помощью метода [Redraw](#) восстанавливать все графическое окно. Данный метод используется для ускорения анимации и создания анимации без мерцания.

В модуле [GraphABC](#) определен ряд констант, типов, процедур, функций и классов для рисования в *графическом окне*. Они подразделяются на следующие группы:

[Графические примитивы](#) [Функции для работы с цветом](#)
[Цветовые константы](#)
[Действия с пером: процедуры и функции](#)
[Действия с пером: объект Pen](#)
[Стиль пера](#)
[Действия с кистью: процедуры и функции](#)
[Действия с кистью: объект Brush](#)
[Стили кисти](#)
[Стили штриховки кисти](#)
[Действия со шрифтом: процедуры и функции](#)
[Действия со шрифтом: объект Font](#)
[Стили шрифта](#)
[Действия с рисунками: класс Picture](#)
[Действия с графическим окном: процедуры и функции](#)
[Действия с графическим окном: объект Window](#)
[Действия с системой координат: процедуры и функции](#)
[Действия с системой координат: объект Coordinate](#)
[Блокировка рисования и ускорение анимации](#)
[Режимы рисования](#)
[События GraphABC](#)
[Виртуальные коды клавиш](#)
[Перенаправление ввода-вывода](#)

Типы и переменные модуля GraphABC

`Color = System.Drawing.Color;` Тип цвета

`Point = System.Drawing.Point;`

 Тип точки

`GraphABCException = class(Exception) end;`

 Тип исключения GraphABC

`RedrawProc: procedure;`

 Процедурная переменная перерисовки графического окна. Если равна `nil`, то используется стандартная перерисовка

`DrawInBuffer: boolean;`

 Следует ли рисовать во внеэкранным буфере

Графические примитивы

Графические примитивы представляют собой процедуры, осуществляющие рисование в графическом окне. Рисование осуществляется текущим пером (линии), текущей кистью (заливка замкнутых областей) и текущим шрифтом (вывод строк).

procedure SetPixel(x,y: integer; c: Color);

Закрашивает пиксел с координатами (x,y) цветом c

procedure PutPixel(x,y: integer; c: Color);

Закрашивает пиксел с координатами (x,y) цветом c

function GetPixel(x,y: integer): Color;

Возвращает цвет пиксела с координатами (x,y)

procedure MoveTo(x,y: integer);

Устанавливает текущую позицию рисования в точку (x,y)

procedure LineTo(x,y: integer);

Рисует отрезок от текущей позиции до точки (x,y). Текущая позиция переносится в точку (x,y)

procedure LineTo(x,y: integer; c: Color);

Рисует отрезок от текущей позиции до точки (x,y) цветом c. Текущая позиция переносится в точку (x,y)

procedure Line(x1,y1,x2,y2: integer);

Рисует отрезок от точки (x1,y1) до точки (x2,y2)

procedure Line(x1,y1,x2,y2: integer; c: Color);

Рисует отрезок от точки (x1,y1) до точки (x2,y2) цветом c

procedure FillCircle(x,y,r: integer);

Заполняет внутренность окружности с центром (x,y) и радиусом r

procedure DrawCircle(x,y,r: integer);

Рисует окружность с центром (x,y) и радиусом r

procedure FillEllipse(x1,y1,x2,y2: integer);

Заполняет внутренность эллипса, ограниченного прямоугольником, заданным координатами противоположных вершин (x1,y1) и (x2,y2)

procedure DrawEllipse(x1,y1,x2,y2: integer);

Рисует границу эллипса, ограниченного прямоугольником, заданным координатами противоположных вершин (x1,y1) и (x2,y2)

procedure FillRectangle(x1,y1,x2,y2: integer);

Заполняет внутренность прямоугольника, заданного координатами

противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure FillRect(x_1, y_1, x_2, y_2 : integer);

Заполняет внутренность прямоугольника, заданного координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure DrawRectangle(x_1, y_1, x_2, y_2 : integer);

Рисует границу прямоугольника, заданного координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure FillRoundRect(x_1, y_1, x_2, y_2, w, h : integer);

Заполняет внутренность прямоугольника со скругленными краями; (x_1, y_1) и (x_2, y_2) задают пару противоположных вершин, а w и h – ширину и высоту эллипса, используемого для скругления краев

procedure DrawRoundRect(x_1, y_1, x_2, y_2, w, h : integer);

Рисует границу прямоугольника со скругленными краями; (x_1, y_1) и (x_2, y_2) задают пару противоположных вершин, а w и h – ширину и высоту эллипса, используемого для скругления краев

procedure Circle(x, y, r : integer);

Рисует заполненную окружность с центром (x, y) и радиусом r

procedure Ellipse(x_1, y_1, x_2, y_2 : integer);

Рисует заполненный эллипс, ограниченный прямоугольником, заданным координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure Rectangle(x_1, y_1, x_2, y_2 : integer);

Рисует заполненный прямоугольник, заданный координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure RoundRect(x_1, y_1, x_2, y_2, w, h : integer);

Рисует заполненный прямоугольник со скругленными краями; (x_1, y_1) и (x_2, y_2) задают пару противоположных вершин, а w и h – ширину и высоту эллипса, используемого для скругления краев

procedure Arc(x, y, r, a_1, a_2 : integer);

Рисует дугу окружности с центром в точке (x, y) и радиусом r , заключенной между двумя лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure FillPie(x, y, r, a_1, a_2 : integer);

Заполняет внутренность сектора окружности, ограниченного дугой с центром в точке (x, y) и радиусом r , заключенной между двумя лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure DrawPie(x, y, r, a_1, a_2 : integer);

Рисует сектор окружности, ограниченный дугой с центром в точке (x,y) и радиусом r, заключенной между двумя лучами, образующими углы a1 и a2 с осью OX (a1 и a2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure Pie(x,y,r,a1,a2: integer);

Рисует заполненный сектор окружности, ограниченный дугой с центром в точке (x,y) и радиусом r, заключенной между двумя лучами, образующими углы a1 и a2 с осью OX (a1 и a2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure DrawPolygon(points: array of Point);

Рисует замкнутую ломаную по точкам, координаты которых заданы в массиве points

procedure FillPolygon(points: array of Point);

Заполняет многоугольник, координаты вершин которого заданы в массиве points

procedure Polygon(points: array of Point);

Рисует заполненный многоугольник, координаты вершин которого заданы в массиве points

procedure Polyline(points: array of Point);

Рисует ломаную по точкам, координаты которых заданы в массиве points

procedure Curve(points: array of Point);

Рисует кривую по точкам, координаты которых заданы в массиве points

procedure DrawClosedCurve(points: array of Point);

Рисует замкнутую кривую по точкам, координаты которых заданы в массиве points

procedure FillClosedCurve(points: array of Point);

Заполняет замкнутую кривую по точкам, координаты которых заданы в массиве points

procedure ClosedCurve(points: array of Point);

Рисует заполненную замкнутую кривую по точкам, координаты которых заданы в массиве points

procedure TextOut(x,y: integer; s: string);

Выводит строку s в прямоугольник к координатами левого верхнего угла (x,y)

procedure TextOut(x,y: integer; n: integer);

Выводит целое n в прямоугольник к координатами левого верхнего угла (x,y)

procedure TextOut(x,y : integer; r : real);

Выводит вещественное r в прямоугольник к координатами левого верхнего угла (x,y)

procedure DrawTextCentered($x,y,x1,y1$: integer; s : string);

Выводит строку s , отцентрированную в прямоугольнике с координатами $(x,y,x1,y1)$

procedure DrawTextCentered($x,y,x1,y1$: integer; n : integer);

Выводит целое значение n , отцентрированное в прямоугольнике с координатами $(x,y,x1,y1)$

procedure DrawTextCentered($x,y,x1,y1$: integer; r : real);

Выводит вещественное значение r , отцентрированное в прямоугольнике с координатами $(x,y,x1,y1)$

procedure FloodFill(x,y : integer; c : Color);

Заливает область одного цвета цветом c , начиная с точки (x,y) .

Функции для работы с цветом

Тип цвета `Color` является синонимом `System.Drawing.Color`.

function `RGB(r,g,b: byte): Color;`

Возвращает цвет, который содержит красную (r), зеленую (g) и синюю (b) составляющие (r,g и b - в диапазоне от 0 до 255)

function `ARGB(a,r,g,b: byte): Color;`

Возвращает цвет, который содержит красную (r), зеленую (g) и синюю (b) составляющие и прозрачность (a) (a,r,g,b - в диапазоне от 0 до 255)

function `RedColor(r: byte): Color;`

Возвращает красный цвет с интенсивностью r (r - в диапазоне от 0 до 255)

function `GreenColor(g: byte): Color;`

Возвращает зеленый цвет с интенсивностью g (g - в диапазоне от 0 до 255)

function `BlueColor(b: byte): Color;`

Возвращает синий цвет с интенсивностью b (b - в диапазоне от 0 до 255)

function `clRandom: Color;`

Возвращает случайный цвет

function `GetRed(c: Color): integer;`

Возвращает красную составляющую цвета

function `GetGreen(c: Color): integer;`

Возвращает зеленую составляющую цвета

function `GetBlue(c: Color): integer;`

Возвращает синюю составляющую цвета

function `GetAlpha(c: Color): integer;`

Возвращает составляющую прозрачности цвета

Цветовые константы

clAquamarine	clAzure	clBeige
clBisque	clBlack	clBlanchedAlmond
clBlue	clBlueViolet	clBrown
clBurlyWood	clCadetBlue	clChartreuse
clChocolate	clCoral	clCornflowerBlue
clCornsilk	clCrimson	clCyan
clDarkBlue	clDarkCyan	clDarkGoldenrod
clDarkGray	clDarkGreen	clDarkKhaki
clDarkMagenta	clDarkOliveGreen	clDarkOrange
clDarkOrchid	clDarkRed	clDarkTurquoise
clDarkSeaGreen	clDarkSlateBlue	clDarkSlateGray
clDarkViolet	clDeepPink	clDarkSalmon
clDeepSkyBlue	clDimGray	clDodgerBlue
clFirebrick	clFloralWhite	clForestGreen
clFuchsia	clGainsboro	clGhostWhite
clGold	clGoldenrod	clGray
clGreen	clGreenYellow	clHoneydew
clHotPink	clIndianRed	clIndigo
clIvory	clKhaki	clLavender
clLavenderBlush	clLawnGreen	clLemonChiffon
clLightBlue	clLightCoral	clLightCyan
clLightGray	clLightGreen	clLightGoldenrodYellow
clLightPink	clLightSalmon	clLightSeaGreen
clLightSkyBlue	clLightSlateGray	clLightSteelBlue
clLightYellow	clLime	clLimeGreen
clLinen	clMagenta	clMaroon
clMediumBlue	clMediumOrchid	clMediumAquamarine
clMediumPurple	clMediumSeaGreen	clMediumSlateBlue
clMoneyGreen	clPlum	clMistyRose
clNavy	clMidnightBlue	clMintCream
clMediumSpringGreen	clMoccasin	clNavajoWhite
clMediumTurquoise	clOldLace	clOlive
clOliveDrab	clOrange	clOrangeRed

clOrchid	clPaleGoldenrod	clPaleGreen
clPaleTurquoise	clPaleVioletRed	clPapayaWhip
clPeachPuff	clPeru	clPink
clMediumVioletRed	clPowderBlue	clPurple
clRed	clRosyBrown	clRoyalBlue
clSaddleBrown	clSalmon	clSandyBrown
clSeaGreen	clSeaShell	clSienna
clSilver	clSkyBlue	clSlateBlue
clSlateGray	clSnow	clSpringGreen
clSteelBlue	clTan	clTeal
clThistle	clTomato	clTransparent
clTurquoise	clViolet	clWheat
clWhite	clWhiteSmoke	clYellow
clYellowGreen		

Подпрограммы для работы с пером

Рисование линий осуществляется текущим пером. Доступ к свойствам текущего пера можно осуществлять как в процедурном, так и в объектно-ориентированном стиле.

Процедуры и функции для доступа к свойствам пера сгруппированы парами: если `Prop` - имя свойства пера, то функция `PenProp` возвращает значение этого свойства, а процедура `SetPenProp(p)` устанавливает это свойство:

```
procedure SetPenColor(c: Color);
```

Устанавливает цвет текущего пера

```
function PenColor: Color;
```

Возвращает цвет текущего пера

```
procedure SetPenWidth(width: integer);
```

Устанавливает ширину текущего пера

```
function PenWidth: integer;
```

Возвращает ширину текущего пера

```
procedure SetPenStyle(style: DashStyle);
```

Устанавливает стиль текущего пера. Константы стилей пера приведены [здесь](#)

```
function PenStyle: DashStyle;
```

Возвращает стиль текущего пера. Константы стилей пера приведены [здесь](#)

```
procedure SetPenMode(m: integer);
```

Устанавливает режим текущего пера

```
function PenMode: integer;
```

Возвращает режим текущего пера

```
function PenX: integer;
```

Возвращают x-координату текущей позиции рисования

```
function PenY: integer;
```

Возвращают y-координату текущей позиции рисования

Кроме этого, можно изменять свойства текущего пера через [объект Pen](#).

Текущее перо Pen

Объект текущего пера возвращается функцией `Pen` и имеет тип `GraphABCPen`:

```
function Pen: GraphABCPen;
```

Класс `GraphABCPen` имеет следующий интерфейс:

```
type GraphABCPen = class  
  property NETPen: System.Drawing.Pen;  
  property Color: GraphABC.Color;  
  property Width: integer;  
  property Style: DashStyle;  
  property Mode: integer;  
  property X: integer;  
  property Y: integer;  
end;
```

Свойства класса `GraphABCPen` описаны в следующей таблице:

property NETPen: System.Drawing.Pen;

Текущее перо .NET. Служит для более тонкой настройки свойств пера

property Color: GraphABC.Color;

Цвет пера

property Width: integer;

Ширина пера

property Style: [DashStyle](#);

Стиль пера. Константы стилей пера приведены [здесь](#)

property Mode: integer;

Режим пера

property X: integer;

Координата X пера (только чтение)

property Y: integer;

Координата Y пера (только чтение)

Пример.

```
Pen.Color := clRed;  
Pen.Style := psDot;
```

Кроме этого, можно изменять свойства текущего пера, используя соответствующие [процедуры и функции](#).

Стили пера

Стили пера задаются перечислимым типом `DashStyle`. Кроме того, для стилей пера определены следующие константы:

```
psSolid = DashStyle.Solid;
```

Сплошное перо



```
psClear = DashStyle.Clear;
```

Прозрачное перо

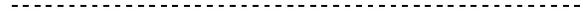
```
psDash = DashStyle.Dash;
```

Штриховое перо



```
psDot = DashStyle.Dot;
```

Пунктирное перо



```
psDashDot = DashStyle.DashDot;
```

Штрихпунктирное перо



```
psDashDotDot =  
DashStyle.DashDotDot;
```

Альтернативное штрихпунктирное перо



Подпрограммы для работы с кистью

Рисование внутренностей замкнутых областей осуществляется текущей кистью. Доступ к свойствам текущей кисти можно осуществлять как в процедурном, так и в объектно-ориентированном стиле.

Процедуры и функции для доступа к свойствам кисти сгруппированы парами: если `Prop` - имя свойства кисти, то функция `PenProp` возвращает значение этого свойства, а процедура `SetPenProp(p)` устанавливает это свойство:

procedure `SetBrushColor(c: Color);` Устанавливает цвет текущей кисти

function `BrushColor: Color;`
Возвращает цвет текущей кисти

procedure `SetBrushStyle(bs: BrushStyleType);`
Устанавливает стиль текущей кисти. Константы стилей кисти приведены [здесь](#)

function `BrushStyle: BrushStyleType;`
Возвращает стиль текущей кисти. Константы стилей кисти приведены [здесь](#)

procedure `SetBrushHatch(bh: HatchStyle);`
Устанавливает штриховку текущей кисти. Константы стилей штриховки кисти приведены [здесь](#)

function `BrushHatch: HatchStyle;`
Возвращает штриховку текущей кисти. Константы стилей штриховки кисти приведены [здесь](#)

procedure `SetHatchBrushBackgroundColor(c: Color);`
Устанавливает цвет заднего плана текущей штриховой кисти

function `HatchBrushBackgroundColor: Color;`
Возвращает цвет заднего плана текущей штриховой кисти

procedure `SetGradientBrushSecondColor(c: Color);`
Устанавливает второй цвет текущей градиентной кисти

function `GradientBrushSecondColor: Color;`
Возвращает второй цвет текущей градиентной кисти

Кроме этого, можно изменять свойства текущей кисти через [объект Brush](#).

Текущая графическая кисть Brush

Объект текущей кисти возвращается функцией `Brush` и имеет тип `GraphABCBush`:

```
function Brush: GraphABCBush;
```

Класс `GraphABCBush` имеет следующий интерфейс:

```
type GraphABCBush = class  
  property NETBrush: System.Drawing.Brush;  
  property Color: GraphABC.Color;  
  property Style: BrushStyleType;  
  property Hatch: HatchStyle;  
  property HatchBackgroundColor: GraphABC.Color;  
  property GradientSecondColor: GraphABC.Color;  
end;
```

Свойства класса `GraphABCBush` описаны в следующей таблице:

```
property NETBrush: System.Drawing.Brush;  
    Текущая кисть .NET  
property Color: GraphABC.Color;  
    Цвет кисти  
property Style: BrushStyleType;  
    Стиль кисти  
property Hatch: HatchStyle;  
    Штриховка кисти  
property HatchBackgroundColor: GraphABC.Color;  
    Цвет заднего плана штриховой кисти  
property GradientSecondColor: GraphABC.Color;  
    Второй цвет градиентной кисти
```

Пример.

```
Brush.Color := clRed;  
Brush.Style := bsHatch;  
Brush.Hatch := bhBackwardDiagonal;
```

Кроме того, можно изменять свойства текущей кисти, используя соответствующие [процедуры и функции](#).

Стили кисти

Стили кисти задаются перечислимым типом `BrushStyleType`:

```
type BrushStyleType = (bsSolid, bsClear, bsHatch,  
bsGradient);
```

Константы имеют следующий смысл:

`bsSolid`

Сплошная кисть (по умолчанию)

`bsClear`

Прозрачная кисть

`bsHatch`

Штриховая кисть

`bsGradient`

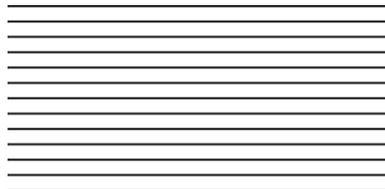
Градиентная кисть

Для всех кистей используется свойство `Color`. Для штриховой кисти дополнительно можно устанавливать свойства `Hatch` и `HatchBackgroundColor`, для градиентной - свойство `GradientSecondColor`.

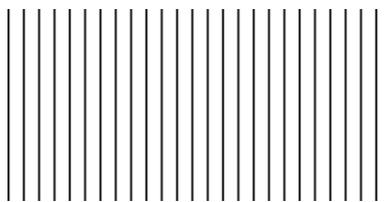
Стили штриховки кисти

Стили штриховки кисти задаются перечислимым типом `HatchStyle`. Кроме того, для стилей штриховки кисти определены следующие константы:

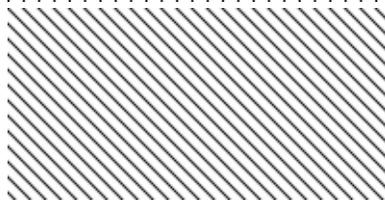
`bhHorizontal`



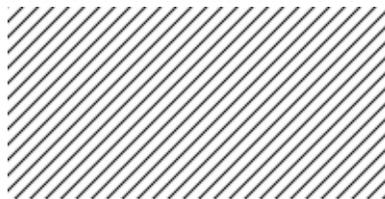
`bhVertical`



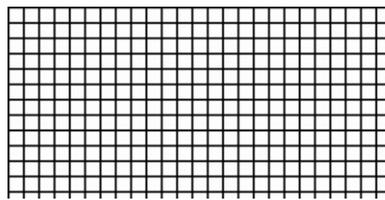
`bhForwardDiagonal`



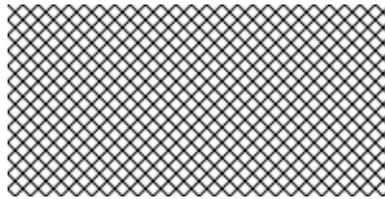
`bhBackwardDiagonal`



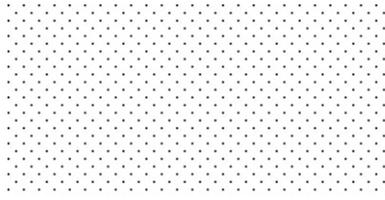
`bhCross`



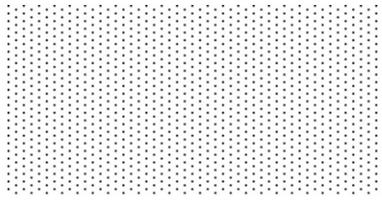
`bhDiagonalCross`



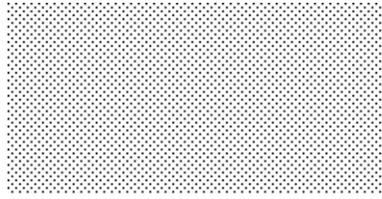
`bhPercent05`



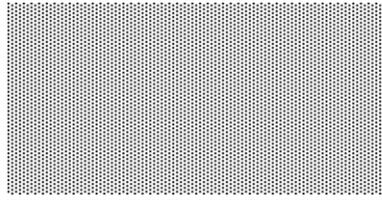
bhPercent10



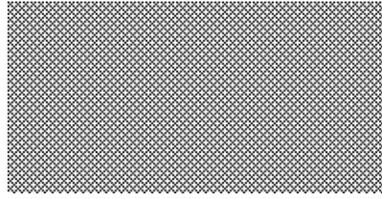
bhPercent20



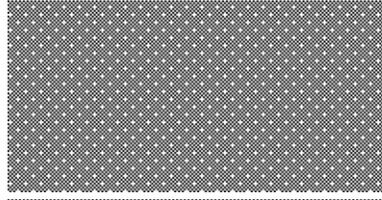
bhPercent25



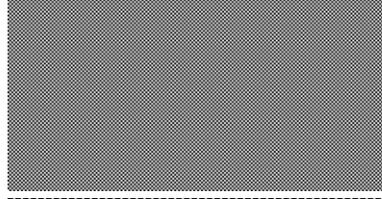
bhPercent30



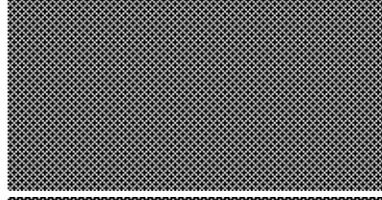
bhPercent40



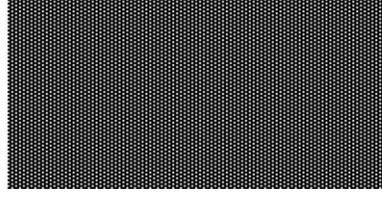
bhPercent50



bhPercent60

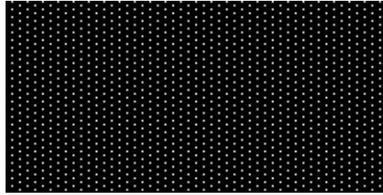
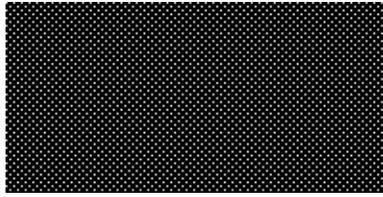


bhPercent70

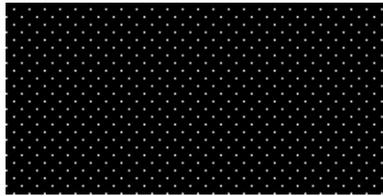


bhPercent75

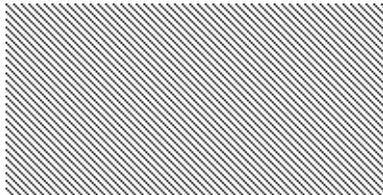
bhPercent80



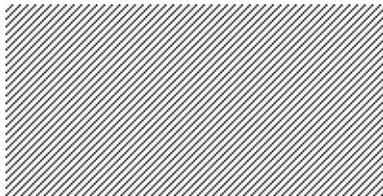
bhPercent90



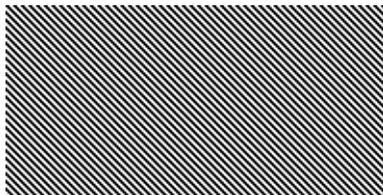
bhLightDownwardDiagonal



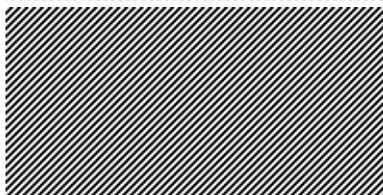
bhLightUpwardDiagonal



bhDarkDownwardDiagonal



bhDarkUpwardDiagonal

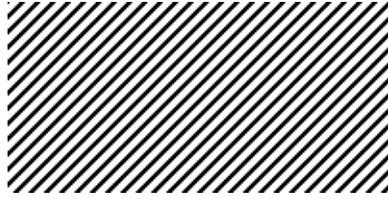


bhWideDownwardDiagonal

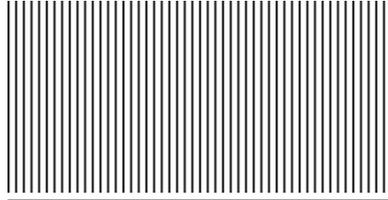


bhWideUpwardDiagonal

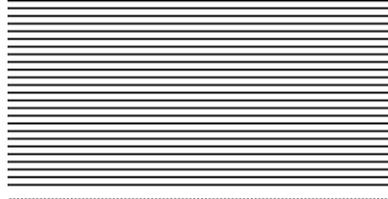
bhLightVertical



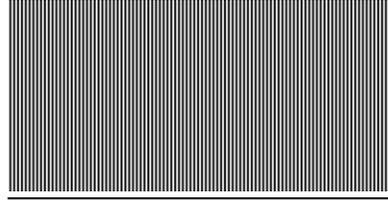
bhLightHorizontal



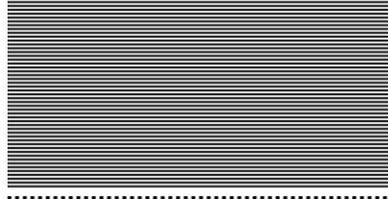
bhNarrowVertical



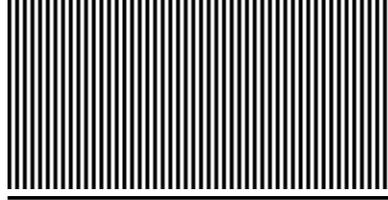
bhNarrowHorizontal



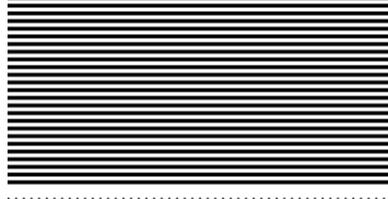
bhDarkVertical



bhDarkHorizontal



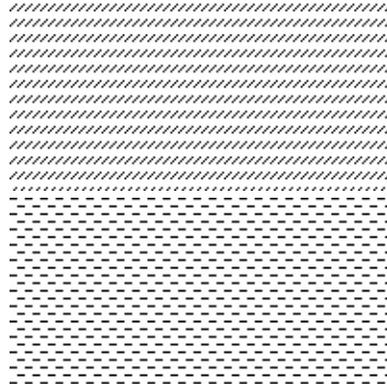
bhDashedDownwardDiagonal



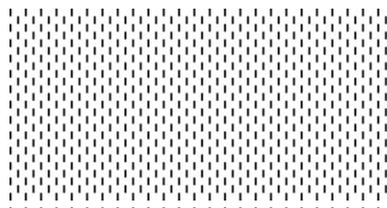
bhDashedUpwardDiagonal



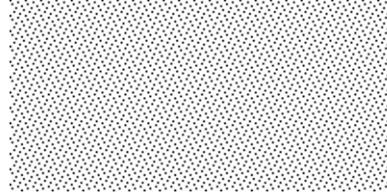
bhDashedHorizontal



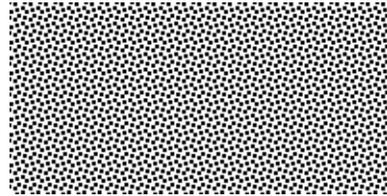
bhDashedVertical



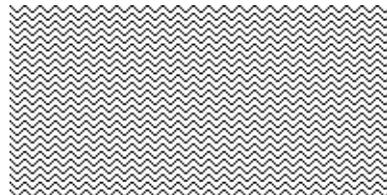
bhSmallConfetti



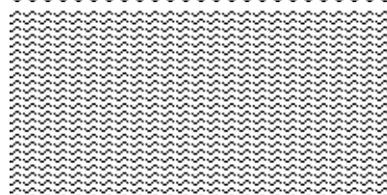
bhLargeConfetti



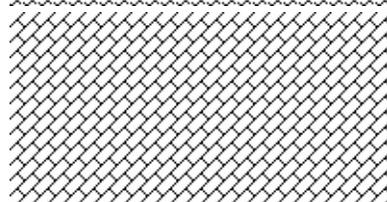
bhZigZag



bhWave

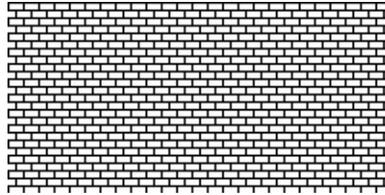


bhDiagonalBrick

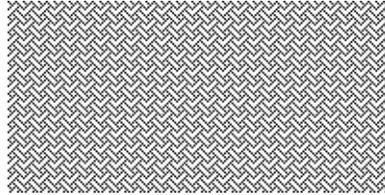


bhHorizontalBrick

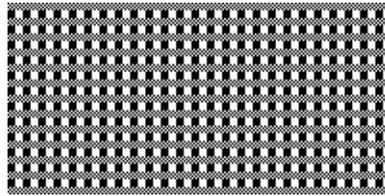
bhWeave



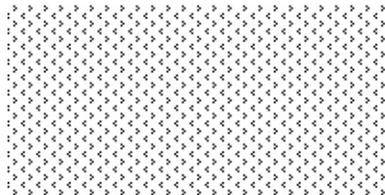
bhPlaid



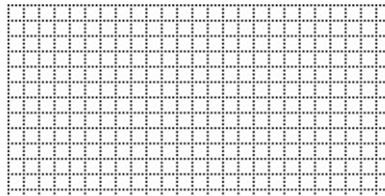
bhDivot



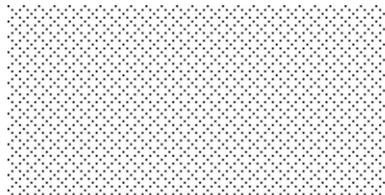
bhDottedGrid



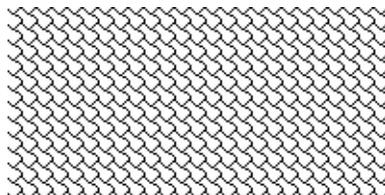
bhDottedDiamond



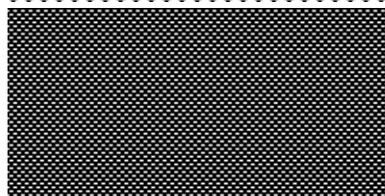
bhShingle



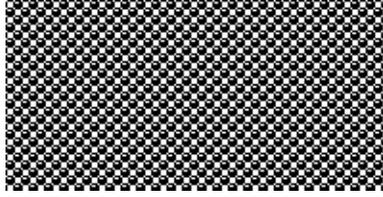
bhTrellis



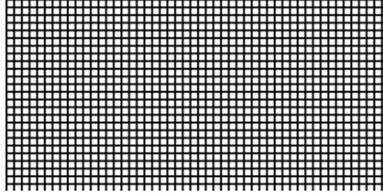
bhSphere



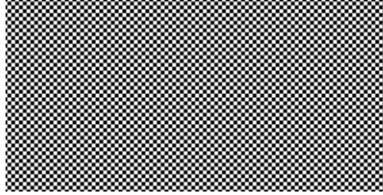
bhSmallGrid



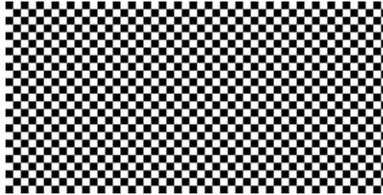
bhSmallCheckerBoard



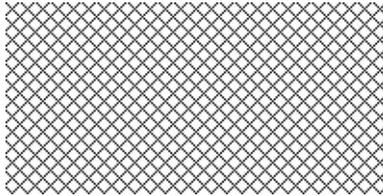
bhLargeCheckerBoard



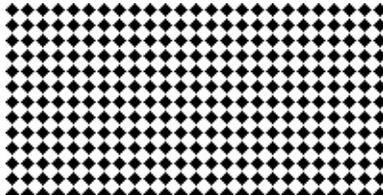
bhOutlinedDiamond



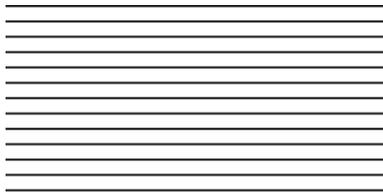
bhSolidDiamond



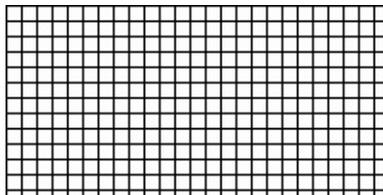
bhMin

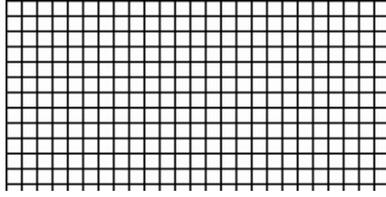


bhLargeGrid



bhMax





Подпрограммы для работы со шрифтом

Вывод текста осуществляется текущим шрифтом. Доступ к свойствам текущего шрифта можно осуществлять как в процедурном, так и в объектно-ориентированном стиле.

Процедуры и функции для доступа к свойствам шрифта сгруппированы парами: если `Prop` - имя свойства пера, то функция `PenProp` возвращает значение этого свойства, а процедура `SetPenProp(p)` устанавливает это свойство:

procedure `SetFontSize(size: integer);` Устанавливает размер текущего шрифта в пунктах

function `FontSize: integer;`

Возвращает размер текущего шрифта в пунктах

procedure `SetFontName(name: string);`

Устанавливает имя текущего шрифта

function `FontName: string;`

Возвращает имя текущего шрифта

procedure `SetFontColor(c: Color);`

Устанавливает цвет текущего шрифта

function `FontColor: Color;`

Возвращает цвет текущего шрифта

procedure `SetFontStyle(fs: integer);`

Устанавливает стиль текущего шрифта

function `FontStyle: integer;`

Возвращает стиль текущего шрифта

Можно также изменять свойства текущего пера через [объект Font](#).

Кроме того, для определения ширины и высоты строки при текущих настройках шрифта используются следующие функции:

function `TextWidth(s: string): integer;`

Возвращает ширину строки `s` в пикселях при текущих настройках шрифта

function `TextHeight(s: string): integer;`

Возвращает высоту строки `s` в пикселях при текущих настройках шрифта

Текущий шрифт Font

Объект текущего шрифта возвращается функцией `Font` и имеет тип `GraphABCFont`:

```
function Font: GraphABCFont;
```

Класс `GraphABCFont` имеет следующий интерфейс: Кроме этого, можно изменять свойства текущего пера через объект `Font`.

```
type GraphABCFont = class  
  property NETFont: System.Drawing.Font;  
  property Color: GraphABC.Color;  
  property Style: integer;  
  property Size: integer;  
  property Name: string;  
end;
```

Свойства класса `GraphABCFont` описаны в следующей таблице:

```
property NETFont: System.Drawing.Font;
```

Текущий шрифт .NET

```
property Color: GraphABC.Color;
```

Цвет шрифта

```
property Style: FontStyleType;
```

Стиль шрифта

```
property Size: integer;
```

Размер шрифта в пунктах

```
property Name: string;
```

Наименование шрифта

Кроме этого, можно изменять свойства текущего шрифта, используя соответствующие [процедуры и функции](#).

Стили шрифта

Стиль шрифта задается перечислимым типом `FontStyleType`, который содержит следующие константы:

- `fsNormal` – обычный; `fsBold` – жирный;
- `fsItalic` – наклонный;
- `fsBoldItalic` – жирный наклонный;
- `fsUnderline` – подчеркнутый;
- `fsBoldUnderline` – жирный подчеркнутый;
- `fsItalicUnderline` – наклонный подчеркнутый;
- `fsBoldItalicUnderline` – жирный наклонный подчеркнутый.

Класс `Picture` графического рисунка

Класс `Picture` представляет собой графический рисунок модуля `GraphABC` и является надстройкой над типом `System.Drawing.Bitmap`. Он имеет свойство прозрачности, которое можно включать/выключать, а также возможность непосредственного рисования на себе всех графических примитивов.

Конструкторы класса `Picture`

constructor `Create(w, h: integer);` Создает рисунок размера `w` на `h` пикселей
constructor `Create(fname: string);`
Создает рисунок из файла с именем `fname`
constructor `Create(r: System.Drawing.Rectangle);`
Создает рисунок из прямоугольника `r` графического окна

Свойства класса `Picture`

property `Width: integer;`
Ширина рисунка в пикселах
property `Height: integer;`
Высота рисунка в пикселах
property `Transparent: boolean;`
Прозрачность рисунка; прозрачный цвет задается свойством `TransparentColor`
property `TransparentColor: Color;`
Прозрачный цвет рисунка. Должна быть установлена прозрачность `Transparent = True`

Методы класса `Picture`

procedure `Load(fname: string);`
Загружает рисунок из файла с именем `fname`
procedure `Save(fname: string);`
Сохраняет рисунок в файл с именем `fname`
procedure `SetSize(w, h: integer);`
Устанавливает размер рисунка `w` на `h` пикселей
function `Intersect(p: Picture): boolean;`

Возвращает **True**, если изображение данного рисунка пересекается с изображением рисунка **p**, и **False** в противном случае. Для проверки пересечения оба объекта рисуются на белом фоне, и прямоугольник пересечения попиксельно проверяется на пересечение. К сожалению, при таком алгоритме любые белые пиксели считаются не принадлежащими объекту. Поэтому для корректной работы этого метода не следует использовать белый цвет для внутренней области объекта.

procedure Draw(*x,y*: integer);

Выводит рисунок в позиции (*x,y*)

procedure Draw(*x,y*: integer; *g*: Graphics);

Выводит рисунок в позиции (*x,y*) на поверхность рисования *g*

procedure Draw(*x,y,w,h*: integer);

Выводит рисунок в позиции (*x,y*), масштабируя его к размеру (*w,h*)

procedure Draw(*x,y,w,h*: integer; *g*: Graphics);

Выводит рисунок в позиции (*x,y*), масштабируя его к размеру (*w,h*), на поверхность рисования *g*

procedure Draw(*x,y*: integer; *r*: System.Drawing.Rectangle);
// *r* - part of Picture

Выводит часть рисунка, заключенную в прямоугольнике *r*, в позиции (*x,y*)

procedure Draw(*x,y*: integer; *r*: System.Drawing.Rectangle;
g: Graphics);

Выводит часть рисунка, заключенную в прямоугольнике *r*, в позиции (*x,y*) на поверхность рисования *g*

procedure Draw(*x,y,w,h*: integer; *r*:
System.Drawing.Rectangle); // *r* - part of Picture

Выводит часть рисунка, заключенную в прямоугольнике *r*, в позиции (*x,y*), масштабируя его к размеру (*w,h*)

procedure Draw(*x,y,w,h*: integer; *r*:
System.Drawing.Rectangle; *g*: Graphics);

Выводит часть рисунка, заключенную в прямоугольнике *r*, в позиции (*x,y*), масштабируя его к размеру (*w,h*), на поверхность рисования *g*

procedure CopyRect(*dst*: System.Drawing.Rectangle; *p*:
Picture; *src*: System.Drawing.Rectangle);

Копирует прямоугольник *src* рисунка *p* в прямоугольник *dst* текущего рисунка

procedure CopyRect(*dst*: System.Drawing.Rectangle; *bmp*:
Bitmap; *src*: System.Drawing.Rectangle);

Копирует прямоугольник `src` битового образа `bmp` в прямоугольник `dst` текущего рисунка

procedure FlipHorizontal;

Зеркально отображает рисунок относительно горизонтальной оси симметрии

procedure FlipVertical;

Зеркально отображает рисунок относительно вертикальной оси симметрии

procedure SetPixel(x,y: integer; c: Color);

Закрашивает пиксел (x,y) рисунка цветом c

procedure PutPixel(x,y: integer; c: Color);

Закрашивает пиксел (x,y) рисунка цветом c

function GetPixel(x,y: integer): Color;

Возвращает цвет пиксела (x,y) рисунка

procedure Line(x1,y1,x2,y2: integer);

Выводит на рисунке отрезок от точки (x1,y1) до точки (x2,y2)

procedure Line(x1,y1,x2,y2: integer; c: Color);

Выводит на рисунке отрезок от точки (x1,y1) до точки (x2,y2) цветом c

procedure FillCircle(x,y,r: integer);

Заполняет на рисунке внутренность окружности с центром (x,y) и радиусом r

procedure DrawCircle(x,y,r: integer);

Выводит на рисунке окружность с центром (x,y) и радиусом r

procedure FillEllipse(x1,y1,x2,y2: integer);

Заполняет на рисунке внутренность эллипса, ограниченного прямоугольником, заданным координатами противоположных вершин (x1,y1) и (x2,y2)

procedure DrawEllipse(x1,y1,x2,y2: integer);

Выводит на рисунке границу эллипса, ограниченного прямоугольником, заданным координатами противоположных вершин (x1,y1) и (x2,y2)

procedure FillRectangle(x1,y1,x2,y2: integer);

Заполняет на рисунке внутренность прямоугольника, заданного координатами противоположных вершин (x1,y1) и (x2,y2)

procedure FillRect(x1,y1,x2,y2: integer);

Заполняет на рисунке внутренность прямоугольника, заданного

координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure DrawRectangle(x_1, y_1, x_2, y_2 : integer);

Выводит на рисунке границу прямоугольника, заданного координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure Circle(x, y, r : integer);

Выводит на рисунке заполненную окружность с центром (x, y) и радиусом r

procedure Ellipse(x_1, y_1, x_2, y_2 : integer);

Выводит на рисунке заполненный эллипс, ограниченный прямоугольником, заданным координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure Rectangle(x_1, y_1, x_2, y_2 : integer);

Выводит на рисунке заполненный прямоугольник, заданный координатами противоположных вершин (x_1, y_1) и (x_2, y_2)

procedure RoundRect(x_1, y_1, x_2, y_2, w, h : integer);

Выводит на рисунке заполненный прямоугольник со скругленными краями; (x_1, y_1) и (x_2, y_2) задают пару противоположных вершин, а w и h – ширину и высоту эллипса, используемого для скругления краев

procedure Arc(x, y, r, a_1, a_2 : integer);

Выводит на рисунке дугу окружности с центром в точке (x, y) и радиусом r , заключенной между двумя лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure FillPie(x, y, r, a_1, a_2 : integer);

Заполняет на рисунке внутренность сектора окружности, ограниченного дугой с центром в точке (x, y) и радиусом r , заключенной между двумя лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure DrawPie(x, y, r, a_1, a_2 : integer);

Выводит на рисунке сектор окружности, ограниченный дугой с центром в точке (x, y) и радиусом r , заключенной между двумя лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure Pie(x, y, r, a_1, a_2 : integer);

Выводит на рисунке заполненный сектор окружности, ограниченный дугой с центром в точке (x, y) и радиусом r , заключенной между двумя

лучами, образующими углы a_1 и a_2 с осью OX (a_1 и a_2 – вещественные, задаются в градусах и отсчитываются против часовой стрелки)

procedure DrawPolygon(points: array of Point);

Выводит на рисунке замкнутую ломаную по точкам, координаты которых заданы в массиве `points`

procedure FillPolygon(points: array of Point);

Заполняет на рисунке многоугольник, координаты вершин которого заданы в массиве `points`

procedure Polygon(points: array of Point);

Выводит на рисунке заполненный многоугольник, координаты вершин которого заданы в массиве `points`

procedure Polyline(points: array of Point);

Выводит на рисунке ломаную по точкам, координаты которых заданы в массиве `points`

procedure Curve(points: array of Point);

Выводит на рисунке кривую по точкам, координаты которых заданы в массиве `points`

procedure DrawClosedCurve(points: array of Point);

Выводит на рисунке замкнутую кривую по точкам, координаты которых заданы в массиве `points`

procedure FillClosedCurve(points: array of Point);

Заполняет на рисунке замкнутую кривую по точкам, координаты которых заданы в массиве `points`

procedure ClosedCurve(points: array of Point);

Выводит на рисунке заполненную замкнутую кривую по точкам, координаты которых заданы в массиве `points`

procedure TextOut(x,y: integer; s: string);

Выводит на рисунке строку `s` в прямоугольник к координатами левого верхнего угла (x,y)

procedure FloodFill(x,y: integer; c: Color);

Заливает на рисунке область одного цвета цветом `c`, начиная с точки (x,y) .

procedure Clear;

Очищает рисунок белым цветом

procedure Clear(c: Color);

Очищает рисунок цветом `c`

Подпрограммы для работы с графическим окном

Доступ к свойствам графического окна можно осуществлять как в процедурном, так и в объектно-ориентированном стиле.

Процедуры и функции для доступа к свойствам окна сгруппированы парами: если `Prop` - имя свойства кисти, то функция `PenProp` возвращает значение этого свойства, а процедура `SetPenProp(p)` устанавливает это свойство:

function `WindowWidth: integer;` Возвращает ширину клиентской части графического окна в пикселах

function `WindowHeight: integer;`

 Возвращает высоту клиентской части графического окна в пикселах

function `WindowLeft: integer;`

 Возвращает отступ графического окна от левого края экрана в пикселах

function `WindowTop: integer;`

 Возвращает отступ графического окна от верхнего края экрана в пикселах

function `WindowIsFixedSize: boolean;`

 Возвращает `True`, если графическое окно имеет фиксированный размер, и `False` в противном случае

procedure `SetWindowWidth(w: integer);`

 Устанавливает ширину клиентской части графического окна в пикселах

procedure `SetWindowHeight(h: integer);`

 Устанавливает высоту клиентской части графического окна в пикселах

procedure `SetWindowLeft(l: integer);`

 Устанавливает отступ графического окна от левого края экрана в пикселах

procedure `SetWindowTop(t: integer);`

 Устанавливает отступ графического окна от верхнего края экрана в пикселах

procedure `SetWindowIsFixedSize(b: boolean);`

 Устанавливает, имеет ли графическое окно фиксированный размер

function `WindowCaption: string;`

Возвращает заголовок графического окна
function WindowTitle: string;

Возвращает заголовок графического окна
procedure SetWindowCaption(s: string);

Устанавливает заголовок графического окна
procedure SetWindowTitle(s: string);

Устанавливает заголовок графического окна
procedure SetWindowSize(w,h: integer);

Устанавливает размеры клиентской части графического окна в пикселах
procedure SetWindowPos(l,t: integer);

Устанавливает отступ графического окна от левого верхнего края экрана в пикселах
procedure ClearWindow;

Очищает графическое окно белым цветом
procedure ClearWindow(c: Color);

Очищает графическое окно цветом с
procedure InitWindow(Left,Top,Width,Height: integer;
BackColor: Color := clWhite);

Устанавливает ширину и высоту клиентской части графического окна в пикселах
procedure SaveWindow(fname: string);

Сохраняет содержимое графического окна в файл с именем fname
procedure LoadWindow(fname: string);

Восстанавливает содержимое графического окна из файла с именем fname
procedure FillWindow(fname: string);

Заполняет содержимое графического окна обоями из файла с именем fname
procedure CloseWindow;

Закрывает графическое окно и завершает приложение
procedure CenterWindow;

Центрирует графическое окно по центру экрана
function WindowCenter: Point;

Возвращает центр графического окна
procedure MaximizeWindow;

Максимизирует графическое окно

procedure MinimizeWindow;

Сворачивает графическое окно

procedure NormalizeWindow;

Возвращает графическое окно к нормальному размеру

Кроме того, можно возвращать размеры экрана Screen, а также размеры и положение графического компонента GraphBox, на котором осуществляется рисование:

function GraphBoxWidth: integer;

Возвращает ширину графического компонента в пикселах (по умолчанию совпадает с WindowWidth)

function GraphBoxHeight: integer;

Возвращает высоту графического компонента в пикселах (по умолчанию совпадает с WindowHeight)

function GraphBoxLeft: integer;

Возвращает отступ графического компонента от левого края окна в пикселах

function GraphBoxTop: integer;

Возвращает отступ графического компонента от верхнего края окна в пикселах

function ScreenWidth: integer;

Возвращает ширину экрана в пикселях

function ScreenHeight: integer;

Возвращает высоту экрана в пикселях

Можно также изменять свойства графического окна через [объект Window](#).

Класс `GraphABCWindow` графического окна

Класс `GraphABCWindow` представляет собой графическое окно.

Функция

```
function Window: GraphABCWindow;
```

возвращает объект текущего графического окна.

Свойства класса `GraphABCWindow`

property `Left: integer;` Отступ графического окна от левого края экрана в пикселах

property `Top: integer;`
Отступ графического окна от верхнего края экрана в пикселах

property `Width: integer;`
Ширина клиентской части графического окна в пикселах

property `Height: integer;`
Высота клиентской части графического окна в пикселах

property `Caption: string;`
Заголовок графического окна

property `Title: string;`
Заголовок графического окна

property `IsFixedSize: boolean;`
Имеет ли графическое окно фиксированный размер

Методы класса `GraphABCWindow`

procedure `Clear;`
Очищает графическое окно белым цветом

procedure `Clear(c: Color);`
Очищает графическое окно цветом `c`

procedure `SetSize(w,h: integer);`
Устанавливает размеры клиентской части графического окна в пикселах

procedure `SetPos(l,t: integer);`
Устанавливает отступ графического окна от левого верхнего края экрана в пикселах

procedure `Init(Left,Top,Width,Height: integer; BackColor: Color := clWhite);`

Устанавливает положение, размеры и цвет графического окна
procedure Save(fname: string);
Сохраняет содержимое графического окна в файл с именем fname
procedure Load(fname: string);
Восстанавливает содержимое графического окна из файла с именем
fname
procedure Fill(fname: string);
Заполняет содержимое графического окна обоями из файла с именем
fname
procedure Close;
Закрывает графическое окно и завершает приложение
procedure Minimize;
Сворачивает графическое окно
procedure Maximize;
Максимизирует графическое окно
procedure Normalize;
Возвращает графическое окно к нормальному размеру
procedure CenterOnScreen;
Центрирует графическое окно по центру экрана
function Center: Point;
Возвращает центр графического окна

Кроме того, можно изменять свойства графического окна, используя соответствующие [процедуры и функции](#).

Подпрограммы для работы с координатами графического окна

Доступ к свойствам координат графического окна можно осуществлять как в процедурном, так и в объектно-ориентированном стиле.

Процедуры и функции для изменения системы координат окна приведены ниже:

procedure SetCoordinateOrigin(x0,y0: integer);

Устанавливает начало координат в точку (x0,y0)

procedure SetCoordinateScale(sx,sy: real);

Устанавливает масштаб системы координат

procedure SetCoordinateAngle(a: real);

Устанавливает поворот системы координат

Можно также изменять свойства системы координат графического окна через [объект Coordinate](#).

Класс GraphABCCoordinate графического окна

Объект текущей системы координат возвращается функцией `Coordinate` и имеет тип `GraphABCCoordinate`:

```
function Coordinate: GraphABCCoordinate;
```

Класс `GraphABCCoordinate` представляет тип системы координат для графического окна.

Свойства класса GraphABCCoordinate

```
property OriginX: integer;      X-координата начала координат  
относительно левого верхнего угла окна
```

```
property OriginY: integer;
```

```
      Y-координата начала координат относительно левого верхнего угла  
окна
```

```
property Origin: Point;
```

```
      Координаты начала координат относительно левого верхнего угла  
окна
```

```
property Angle: real;
```

```
      Угол поворота системы координат
```

```
property ScaleX: real;
```

```
      Масштаб системы координат по оси X
```

```
property ScaleY: real;
```

```
      Масштаб системы координат по оси Y
```

```
property Scale: real;
```

```
      Масштаб системы координат по обоим осям
```

```
property Matrix: System.Drawing.Drawing2D.Matrix;
```

```
      Матрица 3x3 преобразований координат
```

Методы класса GraphABCCoordinate

```
procedure SetTransform(x0, y0, angle, sx, sy: real);
```

```
      Устанавливает параметры системы координат
```

```
procedure SetOrigin(x0, y0: integer);
```

```
      Устанавливает начало системы координат
```

```
procedure SetScale(sx, sy: real);
```

```
      Устанавливает масштаб системы координат
```

```
procedure SetScale(scale: real);
```

Устанавливает масштаб системы координат

procedure SetMathematic;

Устанавливает правую систему координат (ось OY направлена вверх, ось OX - вправо)

procedure SetStandard;

Устанавливает левую систему координат (ось OY направлена вниз, ось OX - вправо)

Кроме того, можно изменять свойства системы координат, используя соответствующие [процедуры и функции](#).

Подпрограммы блокировки рисования

procedure Redraw; Перерисовывает содержимое графического окна. Вызывается в паре с `LockDrawing`

procedure LockDrawing;

 Блокирует рисование на графическом окне. Перерисовка графического окна выполняется с помощью `Redraw`

procedure UnlockDrawing;

 Снимает блокировку рисования на графическом окне и осуществляет его перерисовку

Блокировка вывода в графическое окно `LockDrawing` с последующим вызовом `Redraw` используется для простейшего создания анимации без мерцания.

Пример. [Анимация без мерцания](#)

Подпрограммы режимов рисования

procedure SetSmoothing(sm: boolean); Устанавливает режим
сглаживания

procedure SetSmoothingOn;
 Включает режим сглаживания

procedure SetSmoothingOff;
 Выключает режим сглаживания

function SmoothingIsOn: boolean;
 Возвращает True, если режим сглаживания установлен

procedure SetCoordinateOrigin(x0,y0: integer);
 Устанавливает начало координат в точку (x0,y0)

События модуля GraphABC

Графическая программа продолжает выполняться даже после того как выполнены все операторы основной программы. Графическое окно реагирует на события мыши, клавиатуры, событие изменения размеров окна и событие закрытия окна. Данные события определены в модуле `GraphABC` и представляют собой процедурные переменные:

OnMouseDown: procedure (x,y,mousebutton: integer);

Событие нажатия на кнопку мыши. (x,y) - координаты курсора мыши в момент наступления события, `mousebutton = 1`, если нажата левая кнопка мыши, и 2, если нажата правая кнопка мыши

OnMouseUp: procedure (x,y,mousebutton: integer);

Событие отжатия кнопки мыши. (x,y) - координаты курсора мыши в момент наступления события, `mousebutton = 1`, если отжата левая кнопка мыши, и 2, если отжата правая кнопка мыши

OnMouseMove: procedure (x,y,mousebutton: integer);

Событие перемещения мыши. (x,y) - координаты курсора мыши в момент наступления события, `mousebutton = 0`, если кнопка мыши не нажата, 1, если нажата левая кнопка мыши, и 2, если нажата правая кнопка мыши.

OnKeyDown: procedure (key: integer);

Событие нажатия клавиши. `key` - виртуальный код нажатой клавиши

OnKeyUp: procedure (key: integer);

Событие отжатия клавиши. `key` - виртуальный код отжатой клавиши

OnKeyPress: procedure (ch: char);

Событие нажатия символьной клавиши. `ch` - символ, генерируемый нажатой символьной клавишей

OnResize: procedure;

Событие изменения размера графического окна

OnClose: procedure;

Событие закрытия графического окна

Если переменной-событию присвоена некоторая процедура, она называется **обработчиком** данного события и автоматически вызывается при наступлении указанного события.

Параметры `x` и `y` в обработчиках `OnMouseDown`, `OnMouseUp` и `OnMouseMove` определяют координаты курсора мыши в момент наступления события, параметр `mousebutton` равен 0, если кнопка мыши не нажата, 1, если нажата левая кнопка мыши, и 2, если нажата правая кнопка мыши. Параметр `key` в обработчиках `OnKeyDown` и `OnKeyUp` определяет [виртуальный код нажатой клавиши](#). Параметр `ch` в обработчике `OnKeyPress` определяет нажатый символ.

Пример 1. [Рисование мышью в окне.](#)

Пример 2. [Перемещение окна с помощью клавиатуры.](#)

Виртуальные коды клавиш

VK_Left	VK_Up	VK_Right
VK_Down	VK_PageUp	VK_PageDown
VK_Prior	VK_Next	VK_Home
VK_End	VK_Insert	VK_Delete
VK_Enter	VK_Return	VK_Back
VK_Tab	VK_ShiftKey	VK_ControlKey
VK_F1	VK_F2	VK_F3
VK_F4	VK_F5	VK_F6
VK_F7	VK_F8	VK_F9
VK_F10	VK_F11	VK_F12
VK_Menu	VK_Pause	VK_CapsLock
VK_Capital	VK_PrintScreen	VK_Help
VK_Space	VK_A	VK_B
VK_C	VK_D	VK_E
VK_F	VK_G	VK_H
VK_I	VK_J	VK_K
VK_L	VK_M	VK_N
VK_O	VK_P	VK_Q
VK_R	VK_S	VK_T
VK_U	VK_V	VK_W
VK_X	VK_Y	VK_Z
VK_LWin	VK_RWin	VK_Apps
VK_Sleep	VK_LineFeed	VK_Numpad0
VK_Numpad1	VK_Numpad2	VK_Numpad3
VK_Numpad4	VK_Numpad5	VK_Numpad6
VK_Numpad7	VK_Numpad8	VK_Numpad9
VK_Multiply	VK_Add	VK_Separator
VK_Subtract	VK_Decimal	VK_Divide
VK_NumLock	VK_Scroll	VK_LShiftKey
VK_RShiftKey	VK_LControlKey	VK_RControlKey
VK_LMenu	VK_RMenu	VK_KeyCode
VK_Shift	VK_Control	VK_Alt
VK_Modifiers	VK_Select	VK_Print

VK_Snapshot

Модуль ABCObjects: обзор

Модуль `ABCObjects` реализует векторные графические объекты с возможностью масштабирования, наложения друг на друга, создания составных графических объектов и многократного их вложения друг в друга. Каждый векторный графический объект корректно себя перерисовывает при перемещении, изменении размеров и частичном перекрытии другими объектами.

Модуль `ABCObjects` предназначен для раннего обучения основам объектно-ориентированного программирования, а также для реализации графических и анимационных проектов средней сложности. Он реализован на основе модуля `GraphABC`.

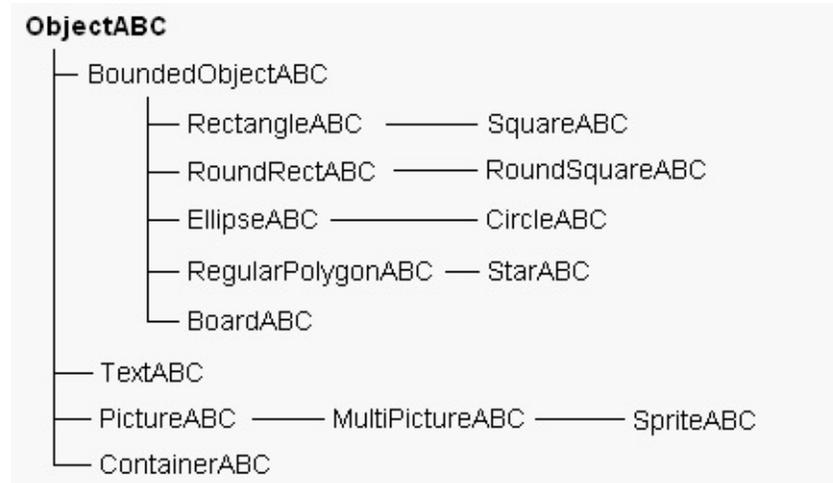
На основе модуля `ABCObjects` созданы модули `ABCSprites`, `ABCButtons`, `ABCChessObjects`, `ABCHouse`, `ABCRobots`, `ABCAdditionalObjects`.

Следующие темы помогут изучить возможности модуля `ABCObjects`:

- [ABCObjects: быстрое введение](#)
- [Диаграмма классов ABCObjects](#)
- Классы [ObjectABC](#), [BoundedObjectABC](#)
- Классы [RectangleABC](#), [SquareABC](#), [EllipseABC](#), [CircleABC](#), [RoundRectABC](#), [RoundSquareABC](#), [TextABC](#)
- Классы [RegularPolygonABC](#), [StarABC](#)
- Классы [PictureABC](#), [MultiPictureABC](#)
- [Мультирисунки](#)
- Классы [BoardABC](#), [ObjectBoardABC](#)
- [Массив графических объектов Objects](#)
- Класс [ContainerABC](#)
- [Контейнеры графических объектов](#)
- [Переменные, процедуры и функции модуля ABCObjects](#)
- [Ускорение перерисовки графических объектов](#)
- [Совмещение графического вывода модулей ABCObjects и GraphABC](#)

Диаграмма классов

На рисунке приведена диаграмма классов модуля `ABCObjects`.



Класс `SpriteABC` описан в модуле `ABCSprites`, однако, приведен на диаграмме как один из важнейших.

Класс ObjectABC

Класс `ObjectABC` является базовым классом для всех графических объектов `ABCObjects`. Его основными потомками, определенными в модуле `ABCObjects`, являются следующие классы:

`BoundedObjectABC`, `RectangleABC`, `SquareABC`, `EllipseABC`, `CircleABC`, `TextABC`, `RegularPolygonABC`, `StarABC`, `PictureABC`, `MultiPictureABC`, `BoardABC` и `ContainerABC`. Класс `ObjectABC` - абстрактный: объекты этого класса не создаются.

Конструкторы класса ObjectABC

constructor `Create(x,y,w,h: integer; cl: GColor);`

Создает графический объект размера (`w`, `h`) цвета `cl` с координатами левого верхнего угла (`x`, `y`)

constructor `Create(g: ObjectABC);`

Создает графический объект - копию объекта `g`

Свойства класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края

property `Top: integer;`

Отступ графического объекта от верхнего края

property `Width: integer;`

Ширина графического объекта

property `Height: integer;`

Высота графического объекта

property `dx: integer;`

`x`-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property `dy: integer;`

`y`-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property `Center: Point;`

Центр графического объекта
property Position: Point;
Левый верхний угол графического объекта
property Visible: boolean;
Видим ли графический объект
property Color: GColor;
Цвет графического объекта
property FontColor: GColor;
Цвет шрифта графического объекта
property Text: string;
Текст внутри графического объекта
property TextVisible: boolean;
Видимость текста внутри графического объекта
property TextScale: real;
Масштаб текста относительно размеров графического объекта,
 $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale} = 1$ текст занимает всю ширину или высоту
объекта. По умолчанию $\text{TextScale} = 0.8$
property FontName: string;
Имя шрифта для вывода свойства Text
property FontStyle: FontStyleType;
Стиль шрифта для вывода свойства Text
property Number: integer;
Целое число, выводимое в центре графического объекта. Для вывода
используется свойство Text
property RealNumber: real;
Вещественное число, выводимое в центре графического объекта. Для
вывода используется свойство Text. Вещественное число выводится с
одним знаком после десятичной точки
property Owner: ContainerABC;
Владелец графического объекта, ответственный также за перерисовку
графического объекта внутри себя (по умолчанию nil)

Методы класса ObjectABC

procedure MoveTo(x, y: integer);
Перемещает левый верхний угол графического объекта к точке (x,y)
procedure MoveOn(a, b: integer);
Перемещает графический объект на вектор (a,b)

procedure Move; virtual;

Перемещает графический объект на вектор, задаваемый свойствами dx,dy

procedure Scale(f: real); virtual;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x,y: integer): boolean; virtual;

Возвращает True, если точка (x,y) находится внутри графического объекта, и False в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает True, если изображение данного графического объекта пересекается с изображением графического объекта g, и False в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает True, если прямоугольник графического объекта пересекается прямоугольником r, и False в противном случае

function Clone0: ObjectABC; virtual;

Возвращает клон графического объекта

function Clone: ObjectABC;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); virtual;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте g: Graphics

destructor Destroy;

Уничтожает графический объект

Класс BoundedObjectABC

Класс `BoundedObjectABC` является непосредственным потомком класса `ObjectABC` и базовым классом для всех замкнутых графических объектов. Его основными потомками являются следующие классы: `RectangleABC`, `SquareABC`, `EllipseABC`, `CircleABC`, `RegularPolygonABC`, `StarABC`, `BoardABC`. Класс `BoundedObjectABC` - абстрактный: объекты этого класса не создаются.

Свойства класса BoundedObjectABC

property `BorderColor: GColor;` Цвет границы
property `BorderWidth: integer;`
Ширина границы
property `Bordered: boolean;`
Имеет ли объект границу (по умолчанию `True`)
property `Filled: boolean;`
Заполнена ли внутренность объекта (по умолчанию `True`)

Методы класса BoundedObjectABC

procedure `SetDrawSettings;`
Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`
Отступ графического объекта от левого края
property `Top: integer;`
Отступ графического объекта от верхнего края
property `width: integer;`
Ширина графического объекта
property `Height: integer;`
Высота графического объекта
property `dx: integer;`
x-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации,

связанной с объектом

property `dy: integer;`

у-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property `Center: Point;`

Центр графического объекта

property `Position: Point;`

Левый верхний угол графического объекта

property `Visible: boolean;`

Видим ли графический объект

property `Color: GColor;`

Цвет графического объекта

property `FontColor: GColor;`

Цвет шрифта графического объекта

property `Text: string;`

Текст внутри графического объекта

property `TextVisible: boolean;`

Видимость текста внутри графического объекта

property `TextScale: real;`

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При `TextScale=1` текст занимает всю ширину или высоту объекта. По умолчанию `TextScale=0.8`

property `FontName: string;`

Имя шрифта для вывода свойства `Text`

property `FontStyle: FontStyleType;`

Стиль шрифта для вывода свойства `Text`

property `Number: integer;`

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property `RealNumber: real;`

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property `Owner: ContainerABC;`

Владелец графического объекта, ответственный также за перерисовку

графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure `MoveTo(x, y: integer);`

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure `MoveOn(a, b: integer);`

Перемещает графический объект на вектор (a, b)

procedure `Move; override;`

Перемещает графический объект на вектор, задаваемый свойствами `dx, dy`

procedure `Scale(f: real); override;`

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure `ToFront;`

Переносит графический объект на передний план

procedure `ToBack;`

Переносит графический объект на задний план

function `Bounds: System.Drawing.Rectangle;`

Возвращает прямоугольник, определяющий границы графического объекта

function `PtInside(x, y: integer): boolean; override;`

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function `Intersect(g: ObjectABC): boolean;`

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function `IntersectRect(r: System.Drawing.Rectangle): boolean;`

Возвращает `True`, если прямоугольник графического объекта пересекается прямоугольником `r`, и `False` в противном случае

function `Clone0: ObjectABC; override;`

Возвращает клон графического объекта

procedure `Draw(x, y: integer; g: Graphics); override;`

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте `g: Graphics`

destructor Destroy;

Уничтожает графический объект

Класс RectangleABC

Класс `RectangleABC` является потомком класса `BoundedObjectABC` и представляет графический объект "Прямоугольник".

Конструкторы класса RectangleABC

constructor `Create(x,y,w,h: integer; cl: GColor);`

Создает прямоугольник размера (w,h) цвета cl с координатами левого верхнего угла (x,y)

constructor `Create(g: RectangleABC);`

Создает прямоугольник - копию прямоугольника g

Методы класса RectangleABC

function `Clone: RectangleABC;`

Возвращает клон прямоугольника

Свойства, унаследованные от класса BoundedObjectABC

property `BorderColor: GColor;`

Цвет границы

property `BorderWidth: integer;`

Ширина границы

property `Bordered: boolean;`

Имеет ли объект границу (по умолчанию `True`)

property `Filled: boolean;`

Заполнена ли внутренность объекта (по умолчанию `True`)

Методы, унаследованные от класса BoundedObjectABC

procedure `SetDrawSettings;`

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края

property `Top: integer;`

Отступ графического объекта от верхнего края

property `Width: integer;`

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

х-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

у-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale**=1 текст занимает всю ширину или высоту объекта. По умолчанию **TextScale**=0.8

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство **Text**

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; override;

Перемещает графический объект на вектор, задаваемый свойствами `dx`, `dy`

procedure Scale(f: real); override;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; override;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает `True`, если прямоугольник графического объекта

пересекается прямоугольником `r`, и `False` в противном случае

function `Clone0: ObjectABC; override;`

Возвращает клон графического объекта

procedure `Draw(x,y: integer; g: Graphics); override;`

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте `g: Graphics`

destructor `Destroy;`

Уничтожает графический объект

Класс SquareABC

Класс SquareABC является потомком класса RectangleABC и представляет графический объект "Квадрат".

Конструкторы класса SquareABC

constructor Create(x,y,w: integer; c1: GColor); Создает квадрат размера w цвета c1 с координатами левого верхнего угла (x,y)

constructor Create(g: SquareABC);
Создает квадрат - копию квадрата g

Методы класса SquareABC

function Clone: SquareABC;
Возвращает клон квадрата

Свойства, унаследованные от класса BoundedObjectABC

property BorderColor: GColor;
Цвет границы

property BorderWidth: integer;
Ширина границы

property Bordered: boolean;
Имеет ли объект границу (по умолчанию True)

property Filled: boolean;
Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;
Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;
Отступ графического объекта от левого края

property Top: integer;
Отступ графического объекта от верхнего края

property Width: integer;
Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

х-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

у-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale**=1 текст занимает всю ширину или высоту объекта. По умолчанию **TextScale**=0.8

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство **Text**

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; override;

Перемещает графический объект на вектор, задаваемый свойствами `dx`, `dy`

procedure Scale(f: real); override;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; override;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает `True`, если прямоугольник графического объекта

пересекается прямоугольником *r*, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(*x,y*: integer; *g*: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте *g*: Graphics

destructor Destroy;

Уничтожает графический объект

Класс `EllipseABC`

Класс `EllipseABC` является потомком класса `BoundedObjectABC` и представляет графический объект "Эллипс". Большинство свойств и методов унаследовано от классов `ObjectABC` и `BoundedObjectABC`.

Конструкторы класса `EllipseABC`

constructor `Create(x,y,w,h: integer; cl: GColor);`

Создает эллипс размера (w, h) цвета cl с координатами левого верхнего угла (x, y)

constructor `Create(g: EllipseABC);`

Создает эллипс - копию эллипса g

Методы класса `EllipseABC`

function `Clone: EllipseABC;`

Возвращает клон эллипса

Свойства, унаследованные от класса `BoundedObjectABC`

property `BorderColor: GColor;`

Цвет границы

property `BorderWidth: integer;`

Ширина границы

property `Bordered: boolean;`

Имеет ли объект границу (по умолчанию `True`)

property `Filled: boolean;`

Заполнена ли внутренность объекта (по умолчанию `True`)

Методы, унаследованные от класса `BoundedObjectABC`

procedure `SetDrawSettings;`

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса `ObjectABC`

property `Left: integer;`

Отступ графического объекта от левого края

property `Top: integer;`

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

х-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

у-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale} = 1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale} = 0.8$

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; override;

Перемещает графический объект на вектор, задаваемый свойствами `dx`, `dy`

procedure Scale(f: real); override;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; override;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle):

boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone@: ObjectABC; override;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); override;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс CircleABC

Класс `CircleABC` является потомком класса `EllipseABC` и представляет графический объект "Круг". Большинство свойств и методов унаследовано от классов `ObjectABC` и `BoundedObjectABC`.

Конструкторы класса CircleABC

constructor `Create(x,y,r: integer; cl: GColor);` Создает круг радиуса `r` цвета `cl` с координатами центра `(x,y)`

constructor `Create(g: CircleABC);`

Создает круг - копию круга `g`

procedure `Scale(f: real);`

Масштабирует круг в `f` раз (`f>1` - увеличение, `0<f<1` - уменьшение)

function `Clone: CircleABC;`

Возвращает клон круга

Свойства класса CircleABC

property `Radius: integer;`

Радиус круга

Свойства, унаследованные от класса BoundedObjectABC

property `BorderColor: GColor;`

Цвет границы

property `BorderWidth: integer;`

Ширина границы

property `Bordered: boolean;`

Имеет ли объект границу (по умолчанию `True`)

property `Filled: boolean;`

Заполнена ли внутренность объекта (по умолчанию `True`)

Методы, унаследованные от класса BoundedObjectABC

procedure `SetDrawSettings;`

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края
property Top: integer;

Отступ графического объекта от верхнего края
property Width: integer;

Ширина графического объекта
property Height: integer;

Высота графического объекта
property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale} = 1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale} = 0.8$

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства Text

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство Text

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство Text. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию nil)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; **override**;

Возвращает True, если точка (x, y) находится внутри графического объекта, и False в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает True, если изображение данного графического объекта пересекается с изображением графического объекта g, и False в

противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle):
boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс RoundRectABC

Класс RoundRectABC является потомком класса BoundedObjectABC и представляет графический объект "Прямоугольник со скругленными краями".

Конструкторы класса RoundRectABC

constructor Create(x,y,w,h,rr: integer; cl: GColor);

Создает прямоугольник со скругленными краями размера (w,h), цветом cl, радиусом скругления r и координатами левого верхнего угла (x,y)

constructor Create(g: RoundRectABC);

Создает прямоугольник со скругленными краями - копию прямоугольника со скругленными краями g

Свойства класса RoundRectABC

property Radius: integer;

Радиус скругления углов

Методы класса RoundRectABC

function Clone: RoundRectABC;

Возвращает клон прямоугольника со скругленными краями

Свойства, унаследованные от класса BoundedObjectABC

property BorderColor: GColor;

Цвет границы

property Borderwidth: integer;

Ширина границы

property Bordered: boolean;

Имеет ли объект границу (по умолчанию True)

property Filled: boolean;

Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale=1** текст занимает всю ширину или высоту объекта. По умолчанию **TextScale=0.8**

property FontName: string;

Имя шрифта для вывода свойства Text

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства Text

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство Text

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство Text. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию nil)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; **override**;

Возвращает True, если точка (x, y) находится внутри графического объекта, и False в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает **True**, если изображение данного графического объекта пересекается с изображением графического объекта **g**, и **False** в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

```
function IntersectRect(r: System.Drawing.Rectangle):  
boolean;
```

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

```
function Clone0: ObjectABC; override;
```

Возвращает клон графического объекта

```
procedure Draw(x,y: integer; g: Graphics); override;
```

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

```
destructor Destroy;
```

Уничтожает графический объект

Класс RoundSquareABC

Класс RoundSquareABC является потомком класса RoundRectABC и представляет графический объект "Квадрат со скругленными краями".

Конструкторы класса RoundSquareABC

constructor Create(x,y,w,r: integer; cl: GColor);

Создает квадрат со скругленными краями размера w, цвета cl с радиусом скругления r и координатами левого верхнего угла (x, y)

constructor Create(g: RoundSquareABC);

Создает квадрат со скругленными краями - копию квадрата со скругленными краями g

Методы класса RoundSquareABC

function Clone: RoundSquareABC;

Возвращает клон квадрата со скругленными краями

Свойства, унаследованные от класса BoundedObjectABC

property BorderColor: GColor;

Цвет границы

property BorderWidth: integer;

Ширина границы

property Bordered: boolean;

Имеет ли объект границу (по умолчанию True)

property Filled: boolean;

Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale**=1 текст занимает всю ширину или высоту объекта. По умолчанию **TextScale**=0.8

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; override;

Перемещает графический объект на вектор, задаваемый свойствами `dx`, `dy`

procedure Scale(f: real); override;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; override;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle):

boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; override;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); override;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс TextABC

Класс `TextABC` является потомком класса `ObjectABC` и представляет графический объект "Текст".

Конструкторы класса TextABC

constructor `Create(x,y,pt: integer; cl: GColor; txt: string);` Создает текстовый объект с текстом `txt` размера `pt` пунктов, цветом `cl` и координатами левого верхнего угла (`x,y`)

constructor `Create(g: TextABC);`
Создает текстовый объект - копию текстового объекта `g`

Свойства класса TextABC

property `FontSize: integer;`

Размер шрифта в пунктах

property `TransparentBackground: boolean;`

Прозрачен ли фон текстового объекта

property `BackgroundColor: GColor;`

Цвет фона текстового объекта

Методы класса TextABC

function `Clone: TextABC;`

Возвращает клон текстового объекта

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края

property `Top: integer;`

Отступ графического объекта от верхнего края

property `Width: integer;`

Ширина графического объекта

property `Height: integer;`

Высота графического объекта

property `dx: integer;`

`x`-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property `dy: integer;`

у-координата вектора перемещения объекта при вызове метода `Move`. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property `Center: Point;`

Центр графического объекта

property `Position: Point;`

Левый верхний угол графического объекта

property `Visible: boolean;`

Видим ли графический объект

property `Color: GColor;`

Цвет графического объекта

property `FontColor: GColor;`

Цвет шрифта графического объекта

property `Text: string;`

Текст внутри графического объекта

property `TextVisible: boolean;`

Видимость текста внутри графического объекта

property `TextScale: real;`

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При `TextScale=1` текст занимает всю ширину или высоту объекта. По умолчанию `TextScale=0.8`

property `FontName: string;`

Имя шрифта для вывода свойства `Text`

property `FontStyle: FontStyleType;`

Стиль шрифта для вывода свойства `Text`

property `Number: integer;`

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property `RealNumber: real;`

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property `Owner: ContainerABC;`

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; **override**;

Возвращает True, если точка (x, y) находится внутри графического объекта, и False в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает True, если изображение данного графического объекта пересекается с изображением графического объекта g, и False в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает True, если прямоугольник графического объекта пересекается прямоугольником r, и False в противном случае

function Clone: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x, y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте g: Graphics

destructor Destroy;

Уничтожает графический объект

Класс RegularPolygonABC

Класс RegularPolygonABC является потомком класса BoundedObjectABC и представляет графический объект "Правильный многоугольник".

Конструкторы класса RegularPolygonABC

constructor Create(x,y,r,n: integer; c1: GColor);

Создает правильный многоугольник с n вершинами, радиусом r, цветом c1 и координатами центра (x, y)

constructor Create(g: RegularPolygonABC);

Создает правильный многоугольник - копию правильного многоугольника g

Свойства класса RegularPolygonABC

property Count: integer;

Количество вершин правильного многоугольника

property Radius: integer;

Радиус многоугольника

property Angle: real;

Угол поворота (в градусах)

Методы класса RegularPolygonABC

function Clone: RegularPolygonABC;

Возвращает клон правильного многоугольника

Свойства, унаследованные от класса BoundedObjectABC

property BorderColor: GColor;

Цвет границы

property BorderWidth: integer;

Ширина границы

property Bordered: boolean;

Имеет ли объект границу (по умолчанию True)

property Filled: boolean;

Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale}=1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale}=0.8$

property `FontName: string;`

Имя шрифта для вывода свойства `Text`

property `FontStyle: FontStyleType;`

Стиль шрифта для вывода свойства `Text`

property `Number: integer;`

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property `RealNumber: real;`

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property `Owner: ContainerABC;`

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure `MoveTo(x, y: integer);`

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure `MoveOn(a, b: integer);`

Перемещает графический объект на вектор (a, b)

procedure `Move; override;`

Перемещает графический объект на вектор, задаваемый свойствами `dx, dy`

procedure `Scale(f: real); override;`

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure `ToFront;`

Переносит графический объект на передний план

procedure `ToBack;`

Переносит графический объект на задний план

function `Bounds: System.Drawing.Rectangle;`

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x,y: integer): boolean; **override**;

Возвращает **True**, если точка (x, y) находится внутри графического объекта, и **False** в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает **True**, если изображение данного графического объекта пересекается с изображением графического объекта **g**, и **False** в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс StarABC

Класс `StarABC` является потомком класса `RegularPolygonABC` и представляет графический объект "Правильная звезда".

Конструкторы класса StarABC

constructor `Create(x,y,r,r1,nn: integer; cl: GColor);`

Создает звезду с nn вершинами, радиусом r, внутренним радиусом r1, цветом cl и координатами центра (x,y)

constructor `Create(g: StarABC);`

Создает звезду - копию звезды g

Свойства класса SquareABC

property `InternalRadius: integer;`

Внутренний радиус

Методы класса SquareABC

function `Clone: StarABC;`

Возвращает клон звезды

Свойства, унаследованные от класса BoundedObjectABC

property `BorderColor: GColor;`

Цвет границы

property `BorderWidth: integer;`

Ширина границы

property `Bordered: boolean;`

Имеет ли объект границу (по умолчанию True)

property `Filled: boolean;`

Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure `SetDrawSettings;`

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

х-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

у-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale=1** текст занимает всю ширину или высоту объекта. По умолчанию **TextScale=0.8**

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства `Text`

property `Number: integer;`

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property `RealNumber: real;`

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property `Owner: ContainerABC;`

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure `MoveTo(x, y: integer);`

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure `MoveOn(a, b: integer);`

Перемещает графический объект на вектор (a, b)

procedure `Move; override;`

Перемещает графический объект на вектор, задаваемый свойствами `dx, dy`

procedure `Scale(f: real); override;`

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure `ToFront;`

Переносит графический объект на передний план

procedure `ToBack;`

Переносит графический объект на задний план

function `Bounds: System.Drawing.Rectangle;`

Возвращает прямоугольник, определяющий границы графического объекта

function `PtInside(x, y: integer): boolean; override;`

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function `Intersect(g: ObjectABC): boolean;`

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта `g`, и `False` в противном случае. Белый цвет считается прозрачным и не

принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle):
boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс PictureABC

Класс `PictureABC` является потомком класса `ObjectABC` и представляет графический объект "Рисунок".

Конструкторы класса PictureABC

constructor `Create(x,y: integer; fname: string);` Создает рисунок с координатами левого верхнего угла (x,y), считывая его из файла `fname`

constructor `Create(x,y: integer; p: Picture);`
Создает рисунок с координатами левого верхнего угла (x,y), считывая его из объекта `p`

constructor `Create(g: PictureABC);`
Создает рисунок - копию рисунка `g`

Свойства класса PictureABC

property `Transparent: boolean;`
Прозрачен ли рисунок

property `TransparentColor: GColor;`
Цвет, считающийся прозрачным

property `ScaleX: real;`
Масштаб рисунка по оси X относительно исходного изображения. При отрицательных значениях происходит зеркальное отражение относительно вертикальной оси

property `ScaleY: real;`
Масштаб рисунка по оси Y относительно исходного изображения. При отрицательных значениях происходит зеркальное отражение относительно вертикальной оси

Методы класса PictureABC

procedure `ChangePicture(fname: string);`
Меняет изображение рисунка, считывая его из файла `fname`

procedure `ChangePicture(p: Picture);`
Меняет изображение рисунка, считывая его из объекта `p`

procedure `FlipVertical;`
Зеркально отображает рисунок относительно вертикальной оси

procedure `FlipHorizontal;`

Зеркально отображает рисунок относительно горизонтальной оси
procedure Save(fname: string);

Сохраняет рисунок в файл fname

function Clone: PictureABC;

Возвращает клон рисунка

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale}=1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale}=0.8$

property FontName: string;

Имя шрифта для вывода свойства Text

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства Text

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство Text

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство Text. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию nil)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x,y: integer): boolean; **override**;

Возвращает **True**, если точка (x, y) находится внутри графического объекта, и **False** в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает **True**, если изображение данного графического объекта пересекается с изображением графического объекта **g**, и **False** в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Класс MultiPictureABC

Класс `MultiPictureABC` является потомком класса `PictureABC` и представляет графический объект "Набор рисунков", содержащий несколько изображений, одно из которых рисуется на экране.

Конструкторы класса MultiPictureABC

constructor `Create(x,y: integer; fname: string);` Создает набор рисунков, состоящий из одного рисунка, загружая его из файла с именем `fname`. После создания рисунок отображается на экране в позиции `(x,y)`. Остальные рисунки добавляются методом `Add`

constructor `Create(x,y: integer; p: Picture);`

Создает набор рисунков, состоящий из одного рисунка, хранящегося в переменной `p`. После создания рисунок отображается на экране в позиции `(x,y)`. Остальные рисунки добавляются методом `Add`

constructor `Create(x,y,w: integer; p: Picture);`

Создает набор рисунков из объекта `p` типа `Picture`. Объект `p` должен хранить последовательность изображений одного размера, расположенных по горизонтали. Каждое изображение считается имеющим ширину `w`. Если ширина рисунка в объекте `p` не кратна `w`, то возникает исключение. После создания первый рисунок из набора отображается на экране в позиции `(x,y)`

constructor `Create(x,y,w: integer; fname: string);`

Создает набор рисунков, загружая его из файла `fname`. Файл должен хранить последовательность изображений одного размера, расположенных по горизонтали. Каждое изображение считается имеющим ширину `w`. Если ширина рисунка в файле `fname` не кратна `w`, то возникает исключение. После создания первый рисунок из набора отображается на экране в позиции `(x,y)`

constructor `Create(g: MultiPictureABC);`

Создает набор рисунков - копию набора рисунков `g`

Свойства класса MultiPictureABC

property `CurrentPicture: integer;`

Номер текущего рисунка

property `Count: integer;`

Количество рисунков в наборе

Методы класса MultiPictureABC

procedure Add(fname: string);

Добавляет рисунок к набору, загружая его из файла fname. Рисунок должен иметь те же размеры, что и все рисунки из набора

procedure ChangePicture(fname: string);

Меняет набор рисунков на набор, состоящий из одного рисунка, загружая его из файла с именем fname

procedure ChangePicture(w: integer; fname: string);

Меняет набор рисунков на набор, загружая его из файла с именем fname. Файл должен хранить последовательность изображений одного размера, расположенных по горизонтали. Каждое изображение считается имеющим ширину w

procedure NextPicture;

Циклически переходит к следующему рисунку из набора

procedure PrevPicture;

Циклически переходит к предыдущему рисунку из набора

function Clone: MultiPictureABC;

Возвращает клон набора рисунков

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации,

связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale} = 1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale} = 0.8$

property FontName: string;

Имя шрифта для вывода свойства `Text`

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства `Text`

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a,b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x,y: integer): boolean; **override**;

Возвращает True, если точка (x, y) находится внутри графического объекта, и False в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает True, если изображение данного графического объекта пересекается с изображением графического объекта g, и False в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает True, если прямоугольник графического объекта пересекается прямоугольником r, и False в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте g: Graphics

destructor Destroy;

Уничтожает графический объект

Класс BoardABC

Класс `BoardABC` является потомком класса `BoundedObjectABC` и представляет графический объект "Доска". Большинство свойств и методов унаследовано от классов `ObjectABC` и `BoundedObjectABC`.

Конструкторы класса BoardABC

constructor `Create(x, y, nx, ny, szx, szy: integer; cl: GColor);`

Создает доску `nx` на `ny` клеток цвета `cl` с размером клетки (`szx, szy`) в позиции (`x, y`).

constructor `Create(g: BoardABC);`

Создает доску - копию доски `g`

Свойства класса BoardABC

property `DimX: integer;`

Количество клеток доски по горизонтали

property `DimY: integer;`

Количество клеток доски по вертикали

property `CellSizeX: integer;`

Размер клетки по горизонтали

property `CellSizeY: integer;`

Размер клетки по вертикали

Методы класса BoardABC

function `Clone: BoardABC;`

Возвращает клон доски

Свойства, унаследованные от класса BoundedObjectABC

property `BorderColor: GColor;`

Цвет границы

property `BorderWidth: integer;`

Ширина границы

property `Bordered: boolean;`

Имеет ли объект границу (по умолчанию `True`)

property `Filled: boolean;`

Заполнена ли внутренность объекта (по умолчанию `True`)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale}=1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale}=0.8$

property `FontName: string;`

Имя шрифта для вывода свойства `Text`

property `FontStyle: FontStyleType;`

Стиль шрифта для вывода свойства `Text`

property `Number: integer;`

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property `RealNumber: real;`

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property `Owner: ContainerABC;`

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса `ObjectABC`

procedure `MoveTo(x,y: integer);`

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure `MoveOn(a,b: integer);`

Перемещает графический объект на вектор (a, b)

procedure `Move; override;`

Перемещает графический объект на вектор, задаваемый свойствами `dx, dy`

procedure `Scale(f: real); override;`

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure `ToFront;`

Переносит графический объект на передний план

procedure `ToBack;`

Переносит графический объект на задний план

function `Bounds: System.Drawing.Rectangle;`

Возвращает прямоугольник, определяющий границы графического объекта

function `PtInside(x,y: integer): boolean; override;`

Возвращает **True**, если точка (x, y) находится внутри графического объекта, и **False** в противном случае

```
function Intersect(g: ObjectABC): boolean;
```

Возвращает **True**, если изображение данного графического объекта пересекается с изображением графического объекта **g**, и **False** в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

```
function IntersectRect(r: System.Drawing.Rectangle):  
boolean;
```

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

```
function Clone0: ObjectABC; override;
```

Возвращает клон графического объекта

```
procedure Draw(x,y: integer; g: Graphics); override;
```

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

```
destructor Destroy;
```

Уничтожает графический объект

Класс ObjectBoardABC

Класс ObjectBoardABC является потомком класса BoardABC и представляет графический объект "Доска с объектами".

Конструкторы класса ObjectBoardABC

constructor Create(x, y, nx, ny, szx, szy: integer; cl: GColor);

Создает доску с объектами nx на ny клеток цвета cl с размером клетки (szx, szy) в позиции (x, y).

constructor Create(g: ObjectBoardABC);

Создает доску с объектами - копию доски g

Методы класса ObjectBoardABC

procedure DestroyObject(x, y: integer);

Удаляет объект в клетке с координатами (x, y)

property Items[x, y: integer]: ObjectABC **read** GetObject
write SetObject; **default**;

Объект в клетке с координатами (x, y)

procedure SwapObjects(x1, y1, x2, y2: integer);

Меняет местами объекты в клетках с координатами (x1, y1) и (x2, y2)

function Clone: ObjectBoardABC;

Возвращает клон доски с объектами

Свойства, унаследованные от класса BoardABC

property DimX: integer;

Количество клеток доски по горизонтали

property DimY: integer;

Количество клеток доски по вертикали

property CellSizeX: integer;

Размер клетки по горизонтали

property CellSizeY: integer;

Размер клетки по вертикали

Свойства, унаследованные от класса BoundedObjectABC

property BorderColor: GColor;

Цвет границы

property BorderWidth: integer;

Ширина границы

property Bordered: boolean;

Имеет ли объект границу (по умолчанию True)

property Filled: boolean;

Заполнена ли внутренность объекта (по умолчанию True)

Методы, унаследованные от класса BoundedObjectABC

procedure SetDrawSettings;

Защищенный метод. Устанавливает атрибуты пера и кисти перед рисованием

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale}=1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale}=0.8$

property FontName: string;

Имя шрифта для вывода свойства Text

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства Text

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство Text

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство Text. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию nil)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз ($f > 1$ - увеличение, $0 < f < 1$ - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x,y: integer): boolean; **override**;

Возвращает **True**, если точка (x, y) находится внутри графического объекта, и **False** в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает **True**, если изображение данного графического объекта пересекается с изображением графического объекта **g**, и **False** в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником **r**, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте **g: Graphics**

destructor Destroy;

Уничтожает графический объект

Мультирисунки

Мультирисунок представляет собой объект класса `MultiPictureABC` и содержит несколько картинок одинакового размера (кадры мультирисунка), одна из которых отображается на экране.

Мультирисунки удобно использовать для графических объектов, имеющих несколько состояний. Например, для игрового объекта в мультирисунке хранятся все его повороты: вверх, вниз, вправо, влево. Мультирисунки удобно использовать также для создания спрайтов - анимационных рисунков с автоматически меняющимися кадрами. Однако, для спрайтов предназначен специальный класс `SpriteABC`, расположенный в модуле `ABCSprites`.

Рассмотрим создание мультирисунка из четырех рисунков, каждый из которых находится в отдельном файле:

```
uses ABCObjects, GraphABC;  
var p: MultiPictureABC;  
begin  p := new MultiPictureABC(50,50, 'multi1.bmp');  
      p.Add('multi2.bmp');  
      p.Add('multi3.bmp');  
      p.Add('multi2.bmp');  
end.
```

После запуска программы в графическом окне отображается рисунок из первого кадра.

Для смены рисунка достаточно изменить свойство `CurrentPicture`:

```
p.CurrentPicture := 2;
```

Можно также циклически перейти к следующему рисунку:

```
p.NextPicture;
```

или к предыдущему рисунку:

```
p.PrevPicture;
```

Нетрудно организовать анимацию, состоящую в последовательной циклической смене рисунков:

```
while True do  
begin
```

```
p.NextPicture;  
Sleep(100);  
end;
```

Мультирисунок после создания хранится не в виде последовательности отдельных рисунков, а в виде одного "длинного" рисунка, в котором все кадры-рисунки расположены последовательно по горизонтали. Такой "длинный" рисунок можно сохранить в файл для последующего использования:

```
p.Save('multipic.gif');
```

После этого мультирисунок можно создавать непосредственно из файла с "длинным" рисунком, указывая в качестве дополнительного параметра ширину одного кадра:

```
p := new MultiPictureABC(50,50,100,'multipic.gif');
```

Третий параметр здесь указывает ширину одного кадра.

Следует еще раз отметить, что спрайты `SpriteABC` полностью перекрывают по возможностям мультирисунки (они имеют состояния, задаваемые строковыми константами, а также автоматически анимируются по таймеру, причем, скорость анимации можно задавать индивидуально для каждого спрайта). Однако, спрайты немного сложнее создавать и, кроме того, они требуют достаточно больших ресурсов процессора при анимации.

Массив всех графических объектов `Objects`

В модуле `ABCObjects` определен динамический массив `Objects`, который хранит все созданные графические объекты. Он принадлежит к типу `ObjectsABCArray`, определенному в `ABCObjects`. Для массива `Objects` определены всего 2 операции: `Objects.Count` возвращает количество графических объектов, а `Objects[i]` возвращает *i*-тый графический объект (типа `ObjectABC`, нумерация осуществляется с 0).

Использование массива `Objects` позволяет единообразно обращаться ко всем графическим объектам, вызывая для них любые методы и обращаясь к любым свойствам класса `ObjectABC`.

Пример 1. Броуновское движение объектов.

```
uses ABCObjects;
var i:integer;
begin  for i:=1 to 30 do
    new
    CircleABC(Random(WindowWidth),Random(WindowHeight),20,clRa
    while True do
        for i:=0 to Objects.Count-1 do
            Objects[i].MoveOn(Random(3)-1,Random(3)-1);
        end.
```

В этом примере создается 30 объектов `CircleABC`. Поскольку все они сохраняются в массиве `Objects`, при их создании результат вызова конструктора не присваивается никакой переменной. После создания все объекты начинают перемещаться в бесконечном цикле на случайный вектор, совершая броуновское движение.

Пример 2. Изменение свойств объектов заданного типа.

```
uses ABCObjects;
var i:integer;
begin
    for i:=1 to 30 do
        case Random(2) of
            0: new
            CircleABC(Random(WindowWidth),Random(WindowHeight),20,clRa
```

```

    1: new
RegularPolygonABC(Random(WindowWidth),Random(WindowHeight))
end;
while True do
  for i:=0 to Objects.Count-1 do
    if Objects[i] is RegularPolygonABC then
      RegularPolygonABC(Objects[i]).Angle :=
RegularPolygonABC(Objects[i]).Angle + 1;
    end.
  end.
end.

```

В этом примере создается 30 объектов `CircleABC` или `RegularPolygonABC`. Затем в бесконечном цикле возвращаются только объекты `RegularPolygonABC`. Вращение достигается увеличением свойства `Angle`, которое определено только в классе `RegularPolygonABC`. Для установки принадлежности объекта к классу `RegularPolygonABC` используется операция `is`, после чего объект `Objects[i]` преобразуется к типу `RegularPolygonABC` при помощи операции приведения типа.

Пример 3. Удаление всех графических объектов, пересекшихся с объектом `p`:

```

for i:=Objects.Count-1 downto 0 do
  if (Objects[i]<>p) and (p.Intersects(Objects[i])) then
    Objects[i].Destroy;
  end.
end.

```

В играх часто нужно удалить все графические объекты, которые "настиг" объект `p`. Для этого следует перебрать все графические объекты за исключением самого `p` и проверить их на предмет пересечения с `p`. Удаление осуществляется вызовом деструктора соответствующего объекта. Отметим также, что в результате удаления количество объектов уменьшается, поэтому следует перебирать объекты от конца к началу.

Переменные, процедуры и функции модуля ABCObjects

procedure LockDrawingObjects; Блокирует рисование
графических объектов. Возможна лишь перерисовка всего экрана вместе
со всеми графическими объектами на нем вызовом RedrawObjects

procedure UnLockDrawingObjects;
Разблокирует рисование графических объектов

procedure RedrawObjects;
Перерисовывает все графическое окно вместе со всеми графическими
объектами на нем

procedure ToFront(g: ObjectABC);
Переносит графический объект g на передний план

procedure ToBack(g: ObjectABC);
Переносит графический объект g на задний план

function ObjectsCount: integer;
Количество графических объектов

function ObjectUnderPoint(x,y: integer): ObjectABC;
Графический объект под точкой (x, y)

function ObjectUnderPoint(p: Point): ObjectABC;
Графический объект под точкой p

procedure SwapPositions(o1,o2: ObjectABC);
Поменять позиции графических объектов o1 и o2

function UIElementUnderPoint(x,y: integer): UIElementABC;
Элемент управления ABCObject под точкой (x, y)

var Objects: ObjectsABCArray;
Массив графических объектов

UIElementABC
Класс элемента управления ABCObject

Ускорение перерисовки графических объектов

Для обеспечения быстрой перерисовки используется следующий прием: в начале программы вызывается `LockDrawingObjects`, а при необходимости перерисовать весь экран – специальная процедура `RedrawObjects`. Отключается режим вызовом `UnLockDrawingObjects`.

При наличии большого количества объектов, каждый из которых движется, такой прием может ощутимо ускорить анимацию. Причина здесь кроется в следующем: по умолчанию при каждом движении объекта он перерисовывается в своем прямоугольнике, при этом в этом прямоугольнике перерисовываются все объекты. Если в графическом окне 100 объектов, и каждый из них переместился, то происходит $100 \times 100 = 10000$ перерисовок объектов. После вызова `LockDrawingObjects` перерисовки не происходит, а в результате вызова `RedrawObjects` каждый объект прорисовывается только один раз, то есть происходит всего 100 перерисовок объектов.

Следует обратить внимание, что в модуле `GraphABC` имеются родственные процедуры `LockDrawing`, `UnLockDrawing`, и `Redraw`. Однако, они отвечают за *растровую перерисовку без мерцания*, в то время как процедуры `LockDrawingObjects`, `UnLockDrawingObjects`, и `RedrawObjects` отвечают только за *ускорение перерисовки векторной графики* `ABCObjects` (отсутствие мерцания векторных объектов при их изменении обеспечивается автоматически).

Совмещение графического вывода модулей ABCObjects и GraphABC

Использование объектов `ABCObjects` можно совмещать с выводом в графическое окно с помощью процедур рисования модуля `GraphABC`. При этом, все объекты `ABCObjects` располагаются поверх изображения, нарисованного в графическом окне вызовом процедур рисования модуля `GraphABC`. Это можно использовать, например, для задания фоновой картинке, на которой затем можно размещать объекты `ABCObjects`:

```
FillWindow('aqua.jpg');c := new  
CircleABC(100,100,50,c1Green);  
r := new RectangleABC(300,300,100,50,c1Blue);
```

Следует иметь в виду, что прорисовка при изменении свойств объектов `ABCObjects` происходит корректно, но если рисовать вызовом процедур модуля `GraphABC`, то графические объекты `ABCObjects` будут затираться. Для восстановления картинке следует либо передвинуть объекты `ABCObjects`, либо вызвать процедуру `RedrawObjects`, перерисовывающую все графическое окно.

Модуль ABCSprites

Модуль `ABCSprites` реализует спрайты - анимационные объекты с автоматически меняющимися кадрами. Спрайт представляется классом `SpriteABC` и является разновидностью мультикартинки `MultiPictureABC`, однако, обладает двумя дополнительными возможностями:

1. Спрайты автоматически анимируются в цикле, что управляется специальным таймером. Можно регулировать скорость анимации каждого спрайта, а также останавливать/запускать все спрайты.
2. Спрайты могут иметь состояния, задаваемые строками. Каждое состояние имеет свой независимый набор кадров, меняющихся циклически. Например, игровой объект в состоянии "Идти" имеет три кадра, а в состоянии "Сидеть" - один кадр (в этом состоянии анимация отсутствует). Переключая состояния, можно моделировать различное поведение игрового объекта.

Кроме того, анимацию всех спрайтов можно выключить/включить вызовом следующих процедур:

procedure `StartSprites;` Стартует анимацию всех спрайтов
procedure `StopSprites;`
 Останавливает анимацию всех спрайтов

Класс SpriteABC

Класс `SpriteABC` является потомком класса `MultiPictureABC` и представляет графический объект "Спрайт", автоматически анимирующий на экране последовательность рисунков. Спрайты также могут иметь несколько состояний, каждое из которых представляет собой анимацию рисунков.

Конструкторы класса `SpriteABC`

constructor `Create(x,y: integer; fname: string);` Создает спрайт, загружая его из файла с именем `fname`. Имя `fname` может быть либо именем графического файла, либо именем информационного файла спрайта с расширением `.spinf`. Если имя является именем графического файла, то создается спрайт с одним кадром. Остальные кадры добавляются методом `Add`. После этого при необходимости добавляются состояния методом `AddStates` и вызывается метод `CheckStates`. Если файл имеет расширение `.spinf`, то он содержит информацию о кадрах и состояниях спрайта и должен сопровождаться соответствующим графическим файлом. После создания спрайт отображается на экране в позиции `(x,y)`

constructor `Create(x,y,w: integer; fname: string);`

Создает спрайт, загружая его из файла `fname`. Файл должен хранить рисунок, представляющий собой последовательность кадров одного размера, расположенных по горизонтали. Каждый кадр считается имеющим ширину `w`. Если ширина рисунка в файле `fname` не кратна `w`, то возникает исключение. После этого при необходимости добавляются состояния методом `AddStates` и вызывается метод `CheckStates`. После создания спрайт отображается на экране в позиции `(x,y)`

constructor `Create(x,y,w: integer; p: Picture);`

Создает спрайт, загружая его из объекта `p: Picture`. Он должен хранить рисунок, представляющий собой последовательность кадров одного размера, расположенных по горизонтали. Каждый кадр считается имеющим ширину `w`. Если ширина рисунка не кратна `w`, то возникает исключение. После этого при необходимости добавляются состояния методом `AddStates` и вызывается метод `CheckStates`. После создания спрайт отображается на экране в позиции `(x,y)`

constructor `Create(g: SpriteABC);`

Создает спрайт - копию спрайта `g`

Свойства класса SpriteABC

property StateName: string;

Имя состояния

property State: integer;

Номер состояния (от 1 до StateCount)

property StateCount: integer;

Количество состояний. Свойство доступно только на чтение

property Speed: integer;

Скорость спрайта (1..10)

property Active: boolean;

Активность спрайта: True, если спрайт активен (т.е. происходит его анимация), и False в противном случае

property Frame: integer;

Текущий кадр в текущем состоянии

Методы класса SpriteABC

procedure AddState(name: string; count: integer);

Добавляет состояние к спрайту. После добавления всех состояний следует вызвать CheckStates

procedure CheckStates;

Проверяет корректность набора состояний. Вызывается после добавления всех состояний

procedure SaveWithInfo(fname: string);

Сохраняет графический и информационный файлы спрайта. Имя fname задает имя графического файла. Информационный файл сохраняется в тот же каталог, что и графический, имеет то же имя и расширение .spinf

procedure NextFrame;

Переходит к следующему кадру в текущем состоянии

procedure NextTick;

Переходит к следующему тикку таймера; если он равен ticks, то он сбрасывается в 1 и вызывается NextFrame

function FrameCount: integer;

Возвращает количество кадров в текущем состоянии

function FrameBeg: integer;

Возвращает начальный кадр в текущем состоянии

function Clone: SpriteABC;

Возвращает клон объекта

Свойства, унаследованные от класса MultiPictureABC

property CurrentPicture: integer;

Номер текущего рисунка

property Count: integer;

Количество рисунков в наборе

Методы, унаследованные от класса MultiPictureABC

procedure Add(fname: string);

Добавляет рисунок к спрайту, загружая его из файла fname. Рисунок должен иметь те же размеры, что и все рисунки из набора

procedure ChangePicture(fname: string);

Меняет набор рисунков на набор, состоящий из одного рисунка, загружая его из файла с именем fname

procedure ChangePicture(w: integer; fname: string);

Меняет набор рисунков на набор, загружая его из файла с именем fname. Файл должен хранить последовательность изображений одного размера, расположенных по горизонтали. Каждое изображение считается имеющим ширину w

procedure NextPicture;

Циклически переходит к следующему рисунку из набора

procedure PrevPicture;

Циклически переходит к предыдущему рисунку из набора

function Clone: MultiPictureABC;

Возвращает клон набора рисунков

Свойства, унаследованные от класса ObjectABC

property Left: integer;

Отступ графического объекта от левого края

property Top: integer;

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**.

По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

у-координата вектора перемещения объекта при вызове метода **Move**. По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При **TextScale=1** текст занимает всю ширину или высоту объекта. По умолчанию **TextScale=0.8**

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство **Text**

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство **Text**. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; override;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); override;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; override;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта g, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle): boolean;

Возвращает `True`, если прямоугольник графического объекта пересекается прямоугольником r, и `False` в противном случае

function Clone0: ObjectABC; override;

Возвращает клон графического объекта

procedure Draw(x, y: integer; g: Graphics); override;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте `g: Graphics`

destructor `Destroy;`

Уничтожает графический объект

Модуль Timers

Модуль `Timers` содержит класс `Timer`, позволяющий выполнять определенные действия через равные промежутки времени. В конструкторе класса `Timer` указывается промежуток времени и имя процедуры без параметров - обработчика события таймера, вызываемой через указанный промежуток времени.

Класс `Timer` имеет следующий интерфейс:

```
type   Timer = class
  constructor (ms: integer; TimerProc: procedure);
  procedure Start;
  procedure Stop;
  property Enabled: boolean read write;
  property Interval: integer read write;
end;
```

Члены класса `Timer` описаны в следующей таблице:

constructor (ms: integer; TimerProc: **procedure**);

Создает таймер, выполняющий каждые `ms` миллисекунд действие, содержащееся в процедуре без параметров `TimerProc`, называемой обработчиком таймера. Созданный таймер необходимо запустить, вызвав метод `Start`

procedure Start;

Запускает таймер

procedure Stop;

Останавливает таймер

property Enabled: boolean **read write**;

Запущен ли таймер

property Interval: integer **read write**;

Промежуток времени между вызовами обработчика таймера

[Пример использования таймера.](#)

Что такое исполнители

Исполнителем будем называть устройство, способное выполнять определенный набор команд. Обычно с исполнителем связана некоторая среда, в которой он работает.

Традиционно концепция исполнителей используется для быстрого обучения основным конструкциям языка программирования при проведении занятий в средних классах школы. В **PascalABC.NET** реализованы исполнители [Робот](#) и [Чертежник](#), описанные в учебнике А. Г. Кушниренко, Г. В. Лебедева и Я. Н. Зайдельмана «Информатика 7–9 классы», М., 2001. Следует отметить, что данный учебник уже не используется для обучения в школе, однако он является наиболее удачным и по интеграции исполнителей в процесс обучения начальному программированию, и по набору задач.

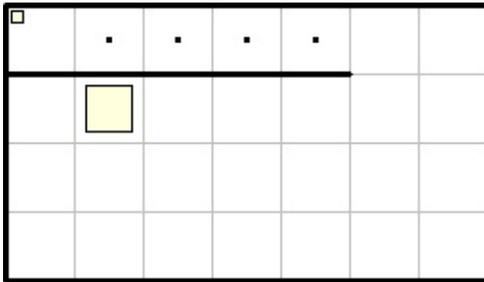
Кроме того, исполнители в **PascalABC.NET** активно используются в [системе проверяемых заданий](#) — одной из ключевых учебных особенностей системы **PascalABC.NET**.

См. также [пример выполнения задания a1 для исполнителя Робот](#).

Исполнитель Робот

Исполнитель Робот действует на прямоугольном клеточном поле. Между некоторыми клетками, а также по периметру поля находятся *стены*. Основная цель Робота — закрасить указанные клетки и переместиться в конечную клетку.

Исполнитель Робот и поле, на котором он работает, отображаются следующим образом:



Здесь большой желтый квадрат изображает Робота, маленький желтый квадрат в левом верхнем углу клетки — конечное положение Робота, черными точками помечены клетки, которые надо закрасить.

Команды исполнителя Робот содержатся в модуле `Robot`:

- `Right` – перемещает Робота вправо; `Left` – перемещает Робота влево;
- `Up` – перемещает Робота вверх;
- `Down` – перемещает Робота вниз;
- `Paint` – закрашивает текущую ячейку;
- `WallFromLeft` – возвращает `True` если слева от Робота стена;
- `WallFromRight` – возвращает `True` если справа от Робота стена;
- `WallFromUp` – возвращает `True` если сверху от Робота стена;
- `WallFromDown` – возвращает `True` если снизу от Робота стена;
- `FreeFromLeft` – возвращает `True` если слева от Робота свободно;
- `FreeFromRight` – возвращает `True` если справа от Робота свободно;
- `FreeFromUp` – возвращает `True` если сверху от Робота свободно;
- `FreeFromDown` – возвращает `True` если снизу от Робота свободно;
- `CellIsPainted` – возвращает `True` если ячейка, в которой находится Робот, закрашена;
- `CellIsFree` – возвращает `True` если ячейка, в которой находится Робот, не закрашена.

Для вызова задания для исполнителя Робот используется следующий шаблон программы:

```
uses Robot;  
begin  
  Task('c1');  
end.
```

Здесь `Task` — процедура, содержащаяся в модуле `Robot` и вызывающая задание с указанным именем.

Имеются следующие группы заданий для исполнителя Робот:

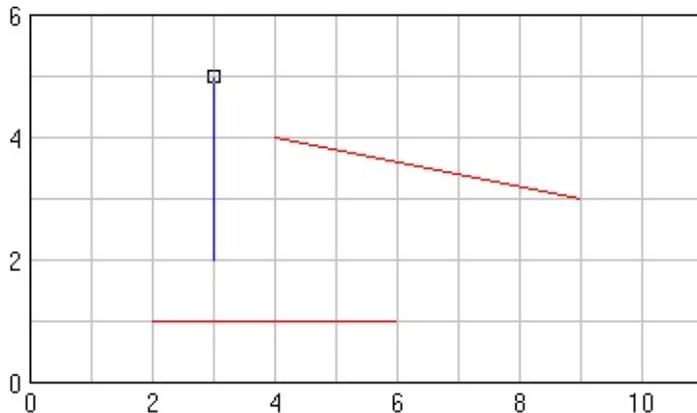
- `a` – вводные задания;
- `c` – цикл с параметром;
- `if` – логические выражения;
- `w` – циклы с условием;
- `cif` – циклы + логические выражения;
- `count` – переменные-счетчики;
- `cc` – вложенные циклы;
- `p` – процедуры без параметров;
- `pp` – процедуры с параметрами.

Для создания стандартного поля размера 9 x 11 используется процедура `StandardField` без параметров, а для создания поля размера N x M — процедура `Field(N, M)`. Робот при этом помещается в центр поля.

Исполнитель Чертежник

Исполнитель Чертежник предназначен для построения рисунков и чертежей на плоскости с координатами. Чертежник имеет перо, которое он может поднимать, опускать и перемещать. При перемещении опущенного пера за ним остается след.

Исполнитель Чертежник и поле, на котором он работает, отображаются на экране следующим образом:



Здесь маленький квадрат изображает Чертежника, красным цветом изображены отрезки, которые надо нарисовать, а синим — уже нарисованные Чертежником отрезки. Когда перо Чертежника опущено, он изображается квадратом меньшего размера.

Команды исполнителя Чертежник содержатся в модуле `Drawman`:

- `ToPoint(x, y)` – перемещает перо Чертежника в точку (x, y) ;
- `OnVector(a, b)` – перемещает перо Чертежника на вектор (a, b) ;
- `PenUp` – поднимает перо Чертежника;
- `PenDown` – опускает перо Чертежника.

Для вызова задания для исполнителя Чертежник используется следующий шаблон программы:

```
uses Drawman;  
begin  
  Task('c1');  
end.
```

В конце программы перо Чертежника должно быть поднято и находиться в начале координат.

Здесь `Task` — процедура, содержащаяся в модуле `Drawman` и вызывающая задание с указанным именем.

Имеются следующие группы заданий для исполнителя Чертежник:

- `a` – вводные задания;
- `c` – цикл с параметром;
- `cc` – вложенные циклы;
- `p` – процедуры без параметров;
- `pp` – процедуры с параметрами.

Для создания произвольного поля размера 20 x 30 используется процедура `StandardField` без параметров, а для создания поля размера N x M — процедура `Field(N, M)`.

Что такое проверяемые задания

Задания, проверяемые автоматически, являются мечтой любого преподавателя. Несомненно, что при автоматической проверке можно проверить только сам факт выполнения задания, но не те знания, которые получил ученик. Поэтому крайне желательно, чтобы выполнение проверяемых заданий проходило под контролем преподавателя.

Опыт авторов системы **PascalABC.NET** и [электронного задачника Programming Taskbook](#) свидетельствует, что обучение программированию с помощью проверяемых заданий, сочетаемое с другими методиками (обучение на примерах, поиск ошибок в программах, ручная и автоматическая трассировка, создание творческих программных проектов и т. д.), позволяет существенно ускорить сам процесс обучения и облегчить труд преподавателя.

Итак, что такое **проверяемые задания** и как их выполнять в системе **PascalABC.NET**? Для того чтобы это узнать, проще всего ознакомиться с несколькими примерами выполнения типовых заданий.

[Пример 1. Задание Begin3 из электронного задачника Programming Taskbook.](#)

Выполняя задание `Begin3`, следует обратить внимание на одну важную особенность работы с электронным задачником: при его подключении процедуры ввода-вывода `read` и `write` начинают работать иначе: процедура `read` получает исходные данные непосредственно от задачника (который генерирует их, используя датчик случайных чисел и учитывая особенности выполняемого задания), а процедура `write` пересылает результаты задачнику для проверки их правильности. Исходные данные, подготовленные задачником, и результаты, полученные программой, отображаются в соответствующих разделах окна задачника; ученик не должен заботиться об их форматировании и поясняющих подписях, поскольку за эти действия отвечает сам задачник. Важно учитывать, что при каждом запуске программы учащегося ей будут предлагаться *различные варианты исходных данных*, поэтому для решения

задания необходимо разработать алгоритм, правильно обрабатывающий любые наборы допустимых исходных данных.

[Пример 2. Задание a1 для исполнителя Робот.](#)

Исполнитель Робот является бесценным источником заданий для быстрого обучения основным конструкциям программирования школьников средних классов.

[Пример 3. Задание String9 из электронного задачника **Programming Taskbook**.](#)

Данное задание знакомит с особенностями выполнения заданий на обработку символов и строк. Следует обратить внимание на диагностику ошибки, связанной с выводом строки, длина которой превышает длину строки — правильного решения.

[Пример 4. Задание File48 из электронного задачника **Programming Taskbook**.](#)

Это задание знакомит с особенностями выполнения заданий на обработку типизированных файлов. Исходные файлы при вызове задания создаются на диске, после чего могут обрабатываться по стандартным правилам. Содержимое файлов, созданных в процессе выполнения задания, проверяется электронным задачником; после проверки исходные и результирующие файлы автоматически удаляются. Следует обратить внимание на то, как содержимое файлов отображается в окне задачника, а также на обработку ошибок времени выполнения, возникающих при выполнении заданий. В примере описаны типичные ошибки при работе с файлами и способы их устранения.

[Пример 5. Задания группы Dynamic из электронного задачника **Programming Taskbook**.](#)

В данном примере описываются задания из раздела, посвященного указателям и линейным динамическим структурам данных. При вызове задания в динамической памяти создаются все исходные структуры данных (в виде односвязных или двусвязных списков), а указатели на элементы этих структур выступают в качестве входных данных задания. Содержимое всех структур данных отображается в окне задачника в наглядном виде. Созданные программой ученика

структуры данных анализируются задачиком и после анализа все структуры данных удаляются из памяти.

[Пример 6. Задания группы Tree из электронного задачника Programming Taskbook.](#)

В данном примере описываются задания на обработку иерархических структур данных — деревьев. Подобные структуры, как и линейные динамические структуры, рассмотренные в предыдущем примере, создаются задачиком автоматически и в наглядном виде отображаются в его окне. Для доступа к ним также используются указатели.

[Пример 7. Задания групп ExamBegin и ExamTaskC из электронного задачника Programming Taskbook.](#)

Пример содержит описание процесса выполнения заданий, связанных с ЕГЭ по информатике. Основной особенностью этих заданий является то, что при их выполнении требуется использовать стандартные процедуры ввода-вывода языка Pascal (а не их специальные модификации, реализованные для задачника **Programming Taskbook**). Рассматриваются задания из группы ExamBegin, посвященные базовым алгоритмам, и задания из группы ExamTaskC, посвященные обработке сложных наборов данных.

Все примеры, связанные с электронным задачиком **Programming Taskbook**, иллюстрируются изображениями окна задачника в [режиме с динамической компоновкой](#), появившемся в версии 4.11.

Задание Begin3 из электронного задачника Programming Taskbook

Задание Begin3 относится к самой первой группе задачника, посвященной знакомству с вводом-выводом и оператором присваивания. Приведем формулировку этого задания.

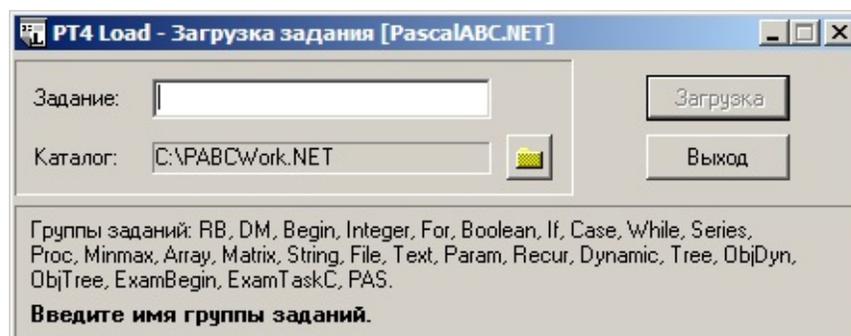
Begin3. Даны стороны прямоугольника a и b . Найти его площадь $S=a*b$ и периметр $P=2*(a+b)$.

Далее по шагам опишем сценарий выполнения задания в системе **PascalABC.NET**.

Шаг 1. Для выполнения заданий из электронного задачника подключим к программе модуль **PT4** и вызовем в начале программы процедуру **Task**, передав ей в качестве параметра имя задания:

```
uses PT4;  
begin  
    Task('Begin3');  
  
end.
```

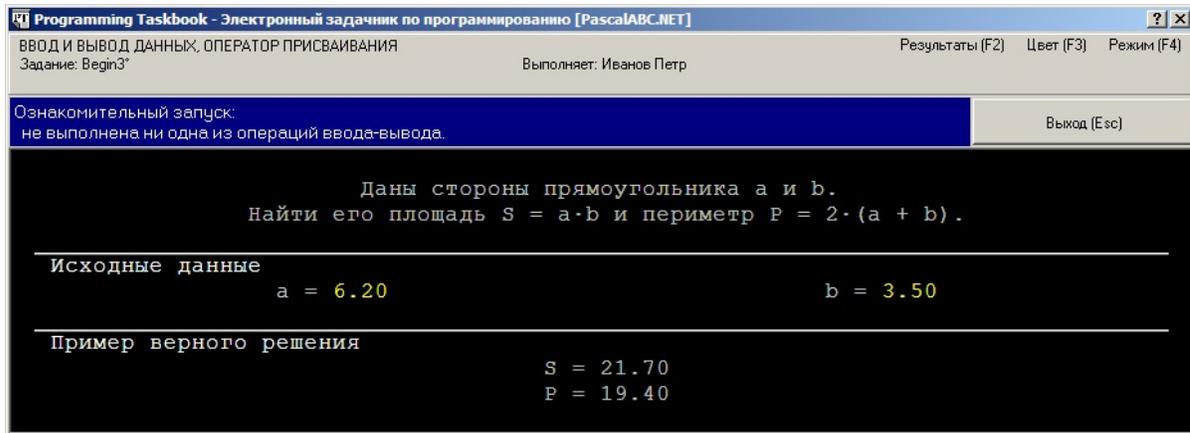
Заметим, что эту программу не обязательно набирать вручную. Для автоматической генерации данной программы выполним следующие действия. Нажмем кнопку . На экране появится окно программного модуля **PT4Load**, позволяющего создать программу-заготовку для требуемого задания:



Наберем в поле ввода «Задание» текст **Begin3** и нажмем **Enter**. После этого указанный выше текст сгенерируется автоматически и сохранится в файле **Begin3.pas** в рабочем каталоге системы

PascalABC.NET (по умолчанию это каталог C:\PABCWork.NET).

Запустим программу (нажав клавишу **F9**), чтобы увидеть на экране текст задания:

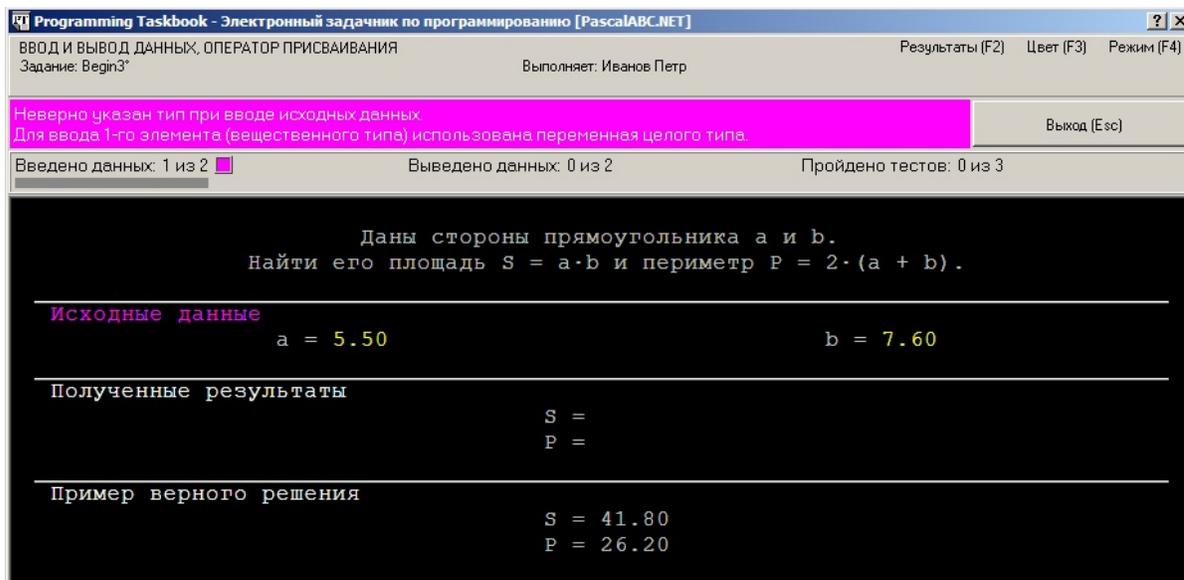


Запуск нашей программы был признан *ознакомительным* (и поэтому правильность решения не анализировалась), так как в ходе ее выполнения не было совершено ни одной операции ввода-вывода.

Шаг 2. Переменные **a** и **b** считываются из полей ввода, выделенных желтым цветом, с помощью обычной процедуры **read**. Закроем окно задачника и изменим программу следующим образом:

```
uses PT4;  
var a, b: integer;  
begin  
  Task('Begin3');  
  read(a, b);  
end.
```

После запуска этого вариант программы окно задачник примет вид:



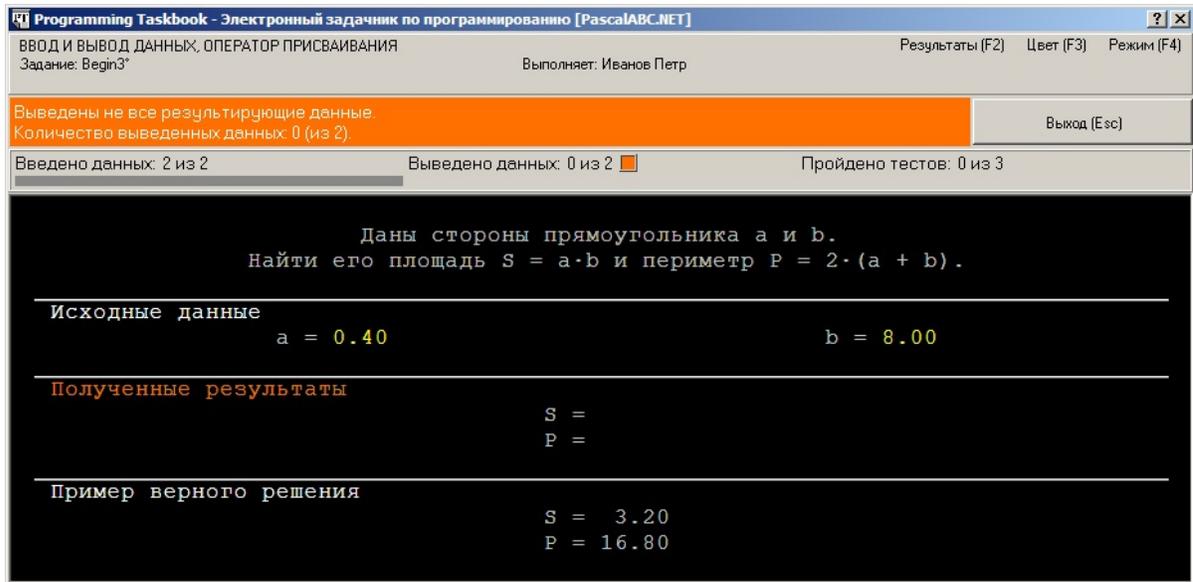
Как видим, исходные данные поменялись и появилось сообщение об ошибке — переменные `a` и `b` должны быть объявлены как вещественные. Если запуск программы не является ознакомительным, то в окне задачника появляется раздел с полученными результатами (в нашем случае он не содержит ни одного числа), а также панель индикаторов, показывающих количество введенных и выведенных данных, а также число успешных тестовых запусков программы.

Шаг 3. Исправим тип исходных данных:

```

uses PT4;
var a, b: real;
begin
  Task('Begin3');
  read(a, b);
end.
  
```

Вновь запустим программу:

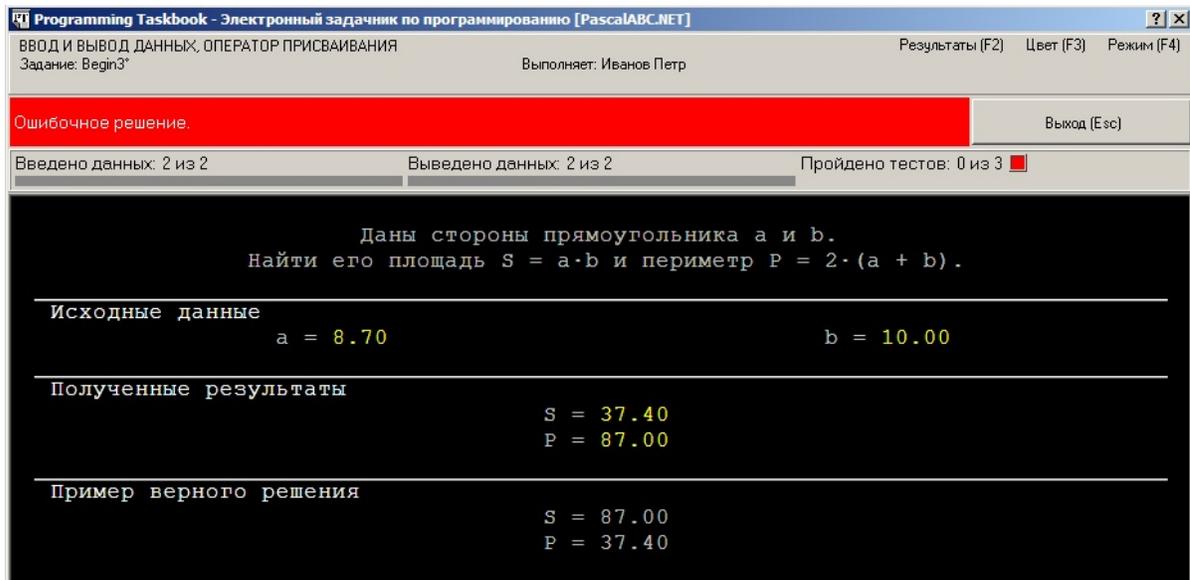


Электронный задачник предупреждает, что не выведен результат. Обратите внимание на то, что для сообщений о различных видах ошибок используются разные цвета (например, фиолетовый для ошибок, связанных с несоответствием типов, и оранжевый для ошибок, связанных с вводом/выводом недостаточного числа данных).

Шаг 4. Проведем вычисления и выведем результирующие данные, но в неверном порядке:

```
uses PT4;  
var a, b, S, P: real;  
begin  
  Task('Begin3');  
  read(a, b);  
  S := a * b;  
  P := 2 * (a + b);  
  write(P, S)  
end.
```

Вновь запустим программу:

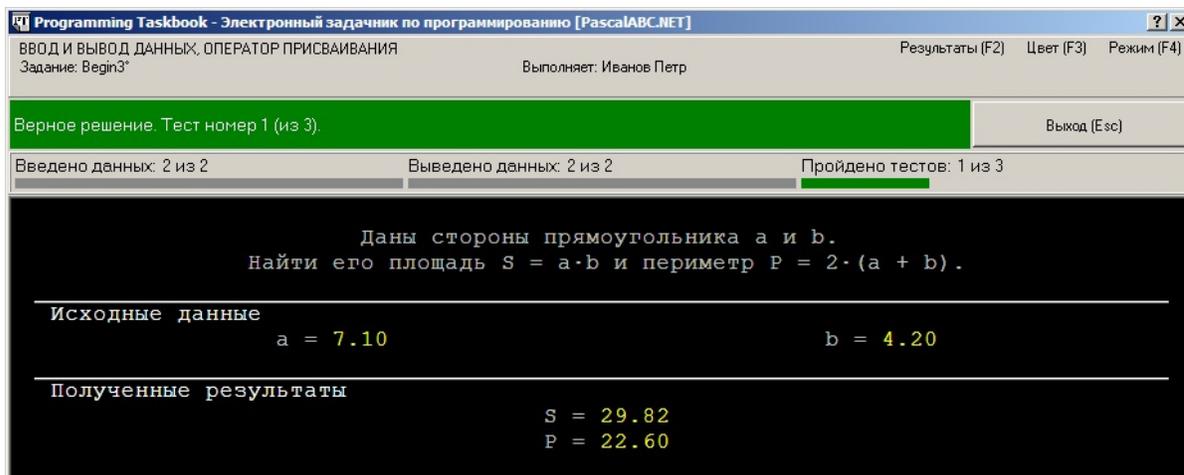


Мы видим, что все требуемые исходные данные введены, все результаты выведены, однако полученные результаты имеют неверные значения. В случае ошибочного решения окно задачника всегда содержит раздел с примером верного решения, чтобы можно было сравнить полученные и правильные результаты.

Шаг 5. Исправим ошибку:

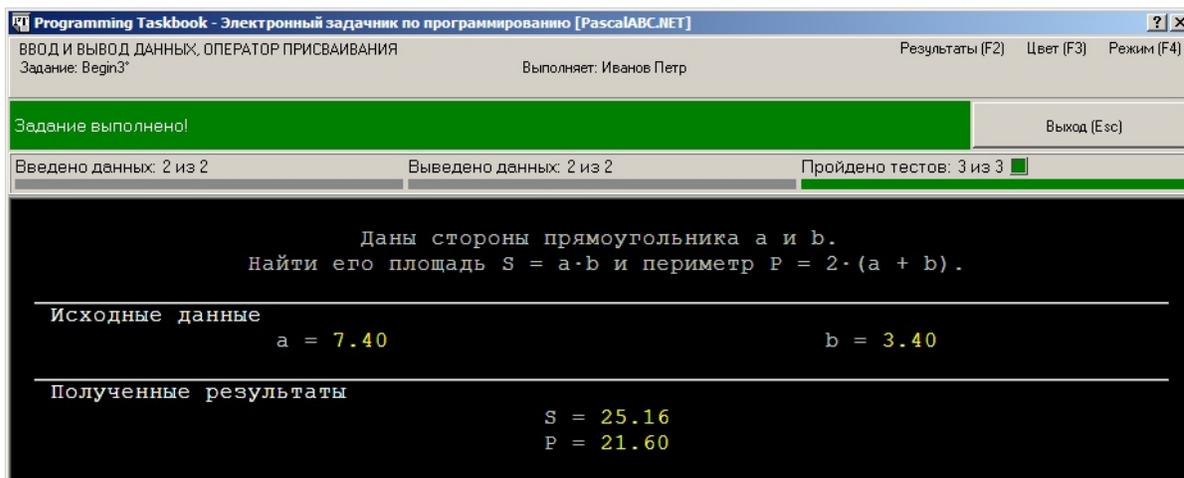
```
uses PT4;  
var a, b, S, P: real;  
begin  
  Task('Begin3');  
  read(a, b);  
  S := a * b;  
  P := 2 * (a + b);  
  write(S, P)  
end.
```

После запуска программы увидим следующее окно:



Если решение является правильным, то дополнительный раздел с примером правильного решения в окне не выводится.

Шаг 6. Чтобы задание считалось выполненным, запустим программу еще два раза. В результате в окне задачника появится сообщение «Задание выполнено!»:

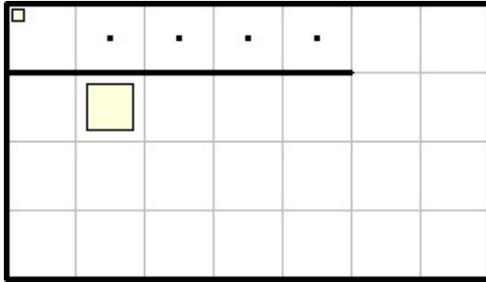


Поздравляем! Задание **Begin3** под руководством электронного задачника выполнено! Результаты выполнения заданий можно просмотреть, щелкнув мышью на метке «Результаты (F2)», расположенной в правом верхнем углу окна задачника, или нажав клавишу **F2**. После закрытия окна задачника и возврата в среду PascalABC.NET результаты можно отобразить на экране, нажав кнопку  или нажав клавиатурную комбинацию **Shift+Ctrl+R**.

Задание a1 для исполнителя Робот

Задание a1 относится к вводным заданиям, посвященным знакомству с основными командами исполнителя Робот. Приведем формулировку этого задания.

a1. Закрасить помеченные клетки.

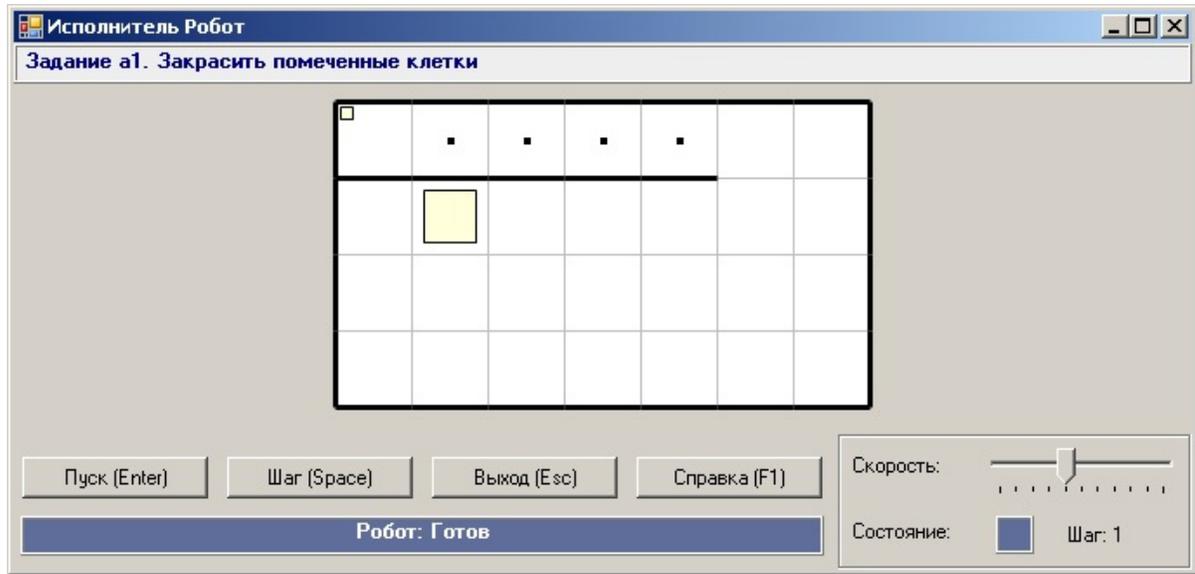


Опишем сценарий решения задания.

Шаг 1. Подключим к программе модуль `Robot` и вызовем в начале программы процедуру `Task`, передав ей в качестве параметра имя задания:

```
uses Robot;  
begin  
  Task('a1');  
  
end.
```

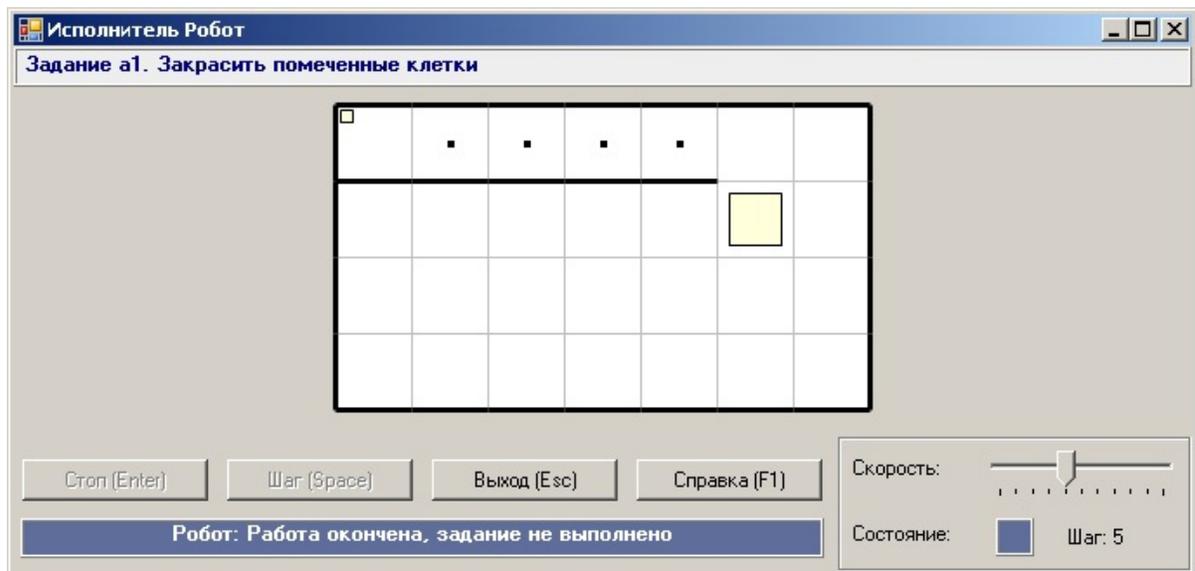
Чтобы не набирать указанный текст, нажмем кнопку  и в появившемся окне наберем имя задания `RBa1` (префикс `RB` означает, что решается задача для Робота). Запустим программу (нажав клавишу **F9**), чтобы увидеть окно Робота с графическим изображением задания:



Шаг 2. Наберем несколько команд Робота:

```
uses Robot;
begin
  Task('a1');
  Right;Right;Right;Right;
end.
```

Запустим программу, после чего нажмем **Enter** или кнопку «Пуск» чтобы Робот начал выполнять заложенную в него программу:



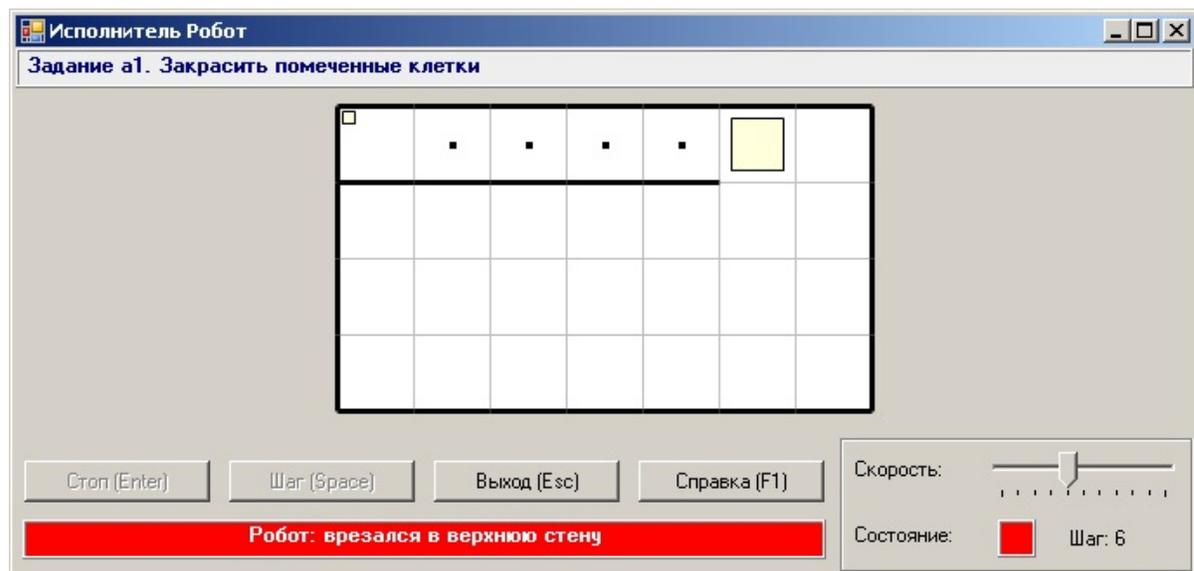
После окончания движения Робота осуществляется проверка, все ли помеченные клетки закрашены и находится ли Робот в конечной

клетке. Если проверка дает отрицательный ответ, то задание не считается выполненным.

Шаг 3. Выполним неверную команду, в результате которой Робот врежется в стенку:

```
uses Robot;  
begin  
  Task('a1');  
  Right;Right;Right;Right;  
  Up;Up;Left;  
end.
```

После запуска программы и нажатия **Enter** получим следующее окно:

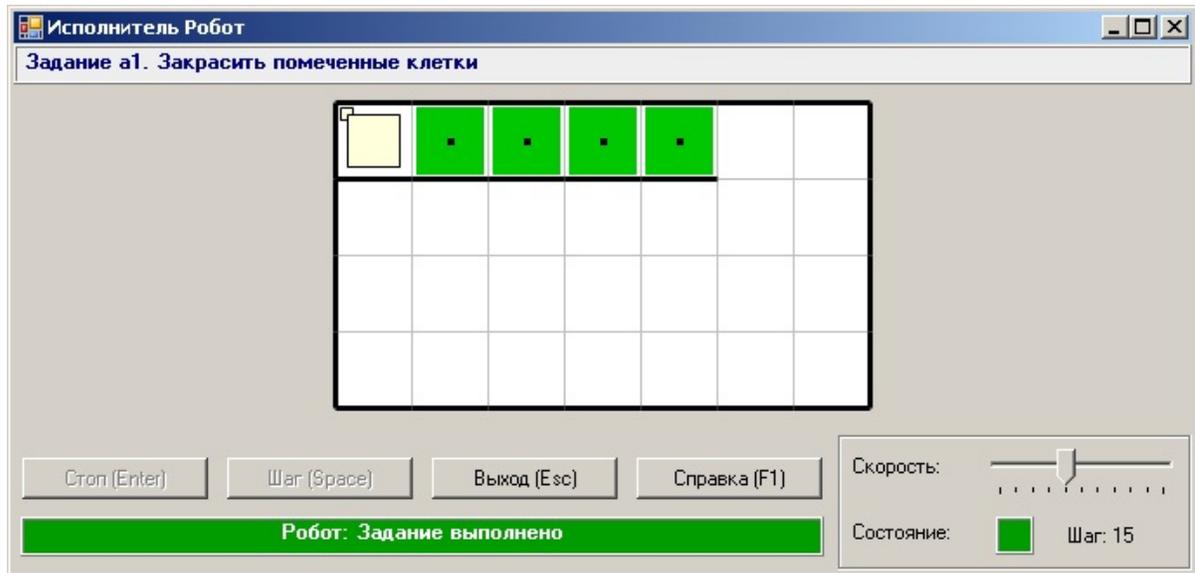


Заметим, что квадратный индикатор состояния Робота окрасился в красный цвет, а последняя команда **Left** не выполнялась, так как после фатальной ошибки Робот прекратил выполнение задания.

Шаг 4. Исправим ошибку и выполним задание до конца:

```
uses Robot;  
begin  
  Task('a1');  
  Right;Right;Right;Right;  
  Up;  
  Left;Paint;
```

```
Left;Paint;  
Left;Paint;  
Left;Paint;  
Left;  
end.
```



Последний рисунок не нуждается в комментариях.

Заметим, что сведения о выполненных заданиях можно просмотреть, нажав кнопку .

Задание на обработку строк

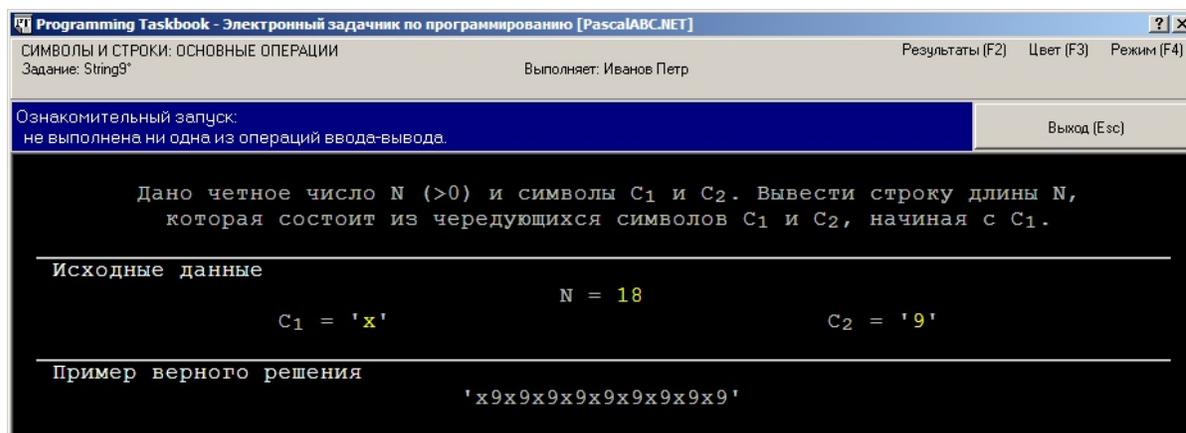
Создание программы-заготовки и знакомство с заданием

В качестве примера задания на обработку строк рассмотрим задание String9.

Программу-заготовку для решения этого задания можно создать с помощью команды меню «Модули | Создать шаблон программы», кнопки  или клавиатурной комбинации **Shift+Ctrl+L**. Эта заготовка будет иметь следующий вид:

```
uses PT4;  
  
begin  
    Task('String9');  
  
end.
```

После запуска данной программы на экране появится [ОКНО ЗАДАЧНИКА](#):



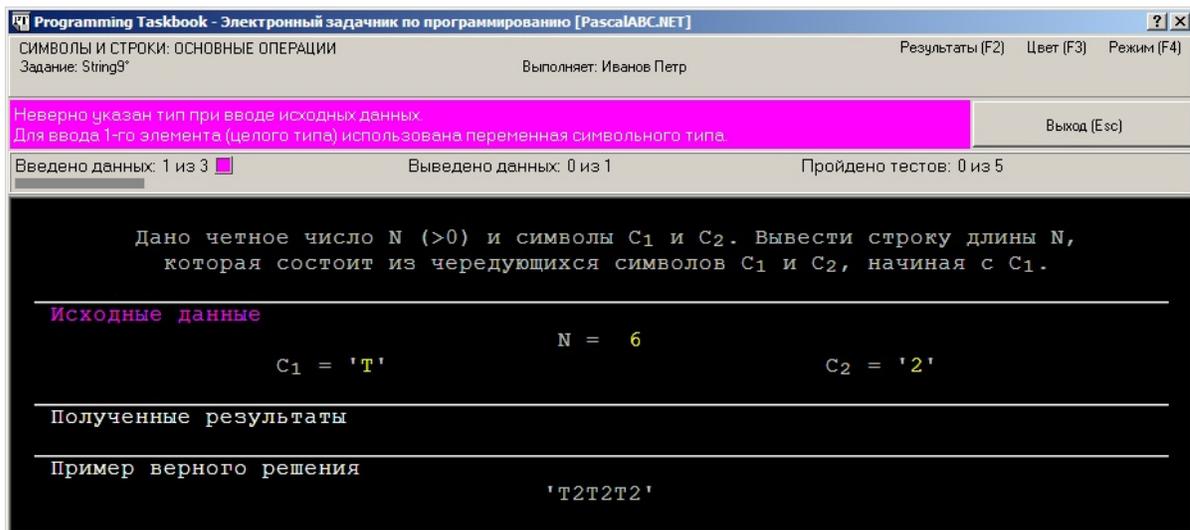
Данные типа `char` и `string` в окне задачника заключаются в *апострофы*; это позволяет, в частности, отличить числовые данные (например, `9`) от символьных и строковых данных, содержащих цифры (например, `'9'`). Кроме того, апострофы дают возможность увидеть *пробелы*, находящиеся в начале или конце строк.

Ввод исходных данных

Добавим в программу фрагмент, обеспечивающий ввод исходных данных (мы намеренно ввели данные не в том порядке, в котором они указаны в окне задачника):

```
uses PT4;
var
  n: integer;
  c1, c2: char;
begin
  Task('String9');
  read(c1, c2, n);
end.
```

Запуск нового варианта программы уже не будет считаться ознакомительным, поскольку в программе выполняется ввод исходных данных. Так как порядок ввода исходных данных является ошибочным, этот вариант решения будет признан неверным и приведет к сообщению «*Неверно указан тип при вводе исходных данных*»:



Общее правило, определяющее порядок ввода и вывода данных для задачника **Programming Taskbook** гласит: *ввод и вывод данных производится по строкам (слева направо), а строки просматриваются сверху вниз*. Иными словами, данные, отображаемые в окне задачника, вводятся и выводятся в том порядке, в котором читается обычный текст.

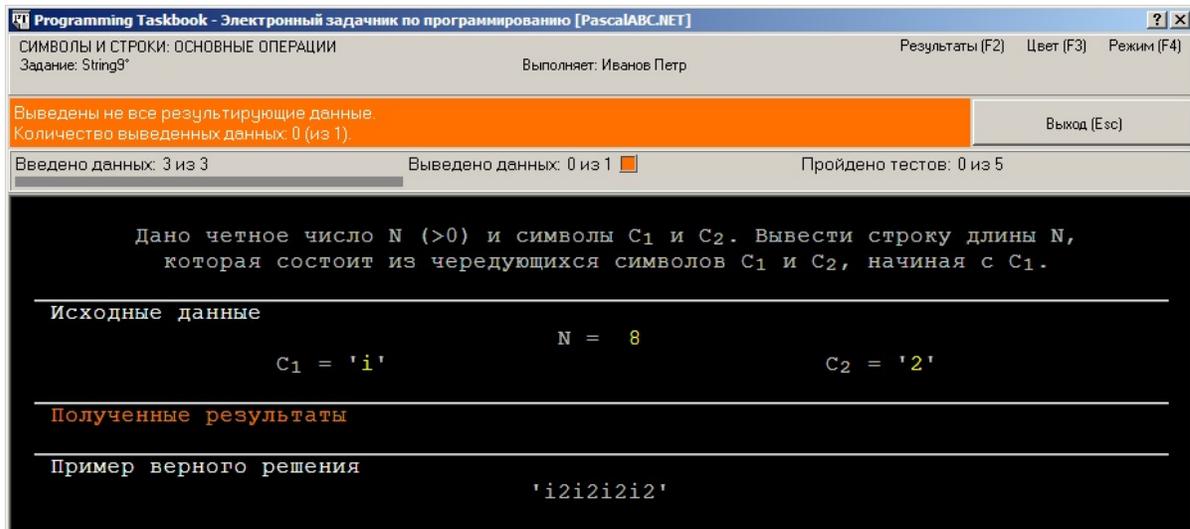
На панели индикаторов, расположенной под информационной панелью, отмечено, что введен всего один элемент исходных данных

(из трех), хотя в процедуре `read` нашей программы были указаны три переменные. Это объясняется тем, что при вводе первой из этих переменных (`c1`) была обнаружена ошибка несоответствия типа, поэтому задачник не стал анализировать остальные данные, которые программа пыталась ввести. Здесь проявляется еще одно правило задачника **Programming Taskbook**: *при обнаружении первой ошибки ввода-вывода анализ оставшихся исходных данных и результатов не проводится.*

Исправим нашу программу, изменив порядок параметров в процедуре ввода:

```
read(n, c1, c2);
```

Теперь с вводом данных проблем не возникает, однако из-за того что в программе отсутствует вывод результирующих данных, решение по-прежнему считается ошибочным:



Данная ошибка, в отличие от предыдущей, связана не с вводом исходных данных, а с выводом результатов. Это отмечается в окне задачника двумя способами: цветной маркер ошибки располагается рядом с индикатором вывода, и этим же цветом выделяется заголовок раздела результатов (который в данном случае не содержит никаких данных).

Формирование и вывод требуемой строки

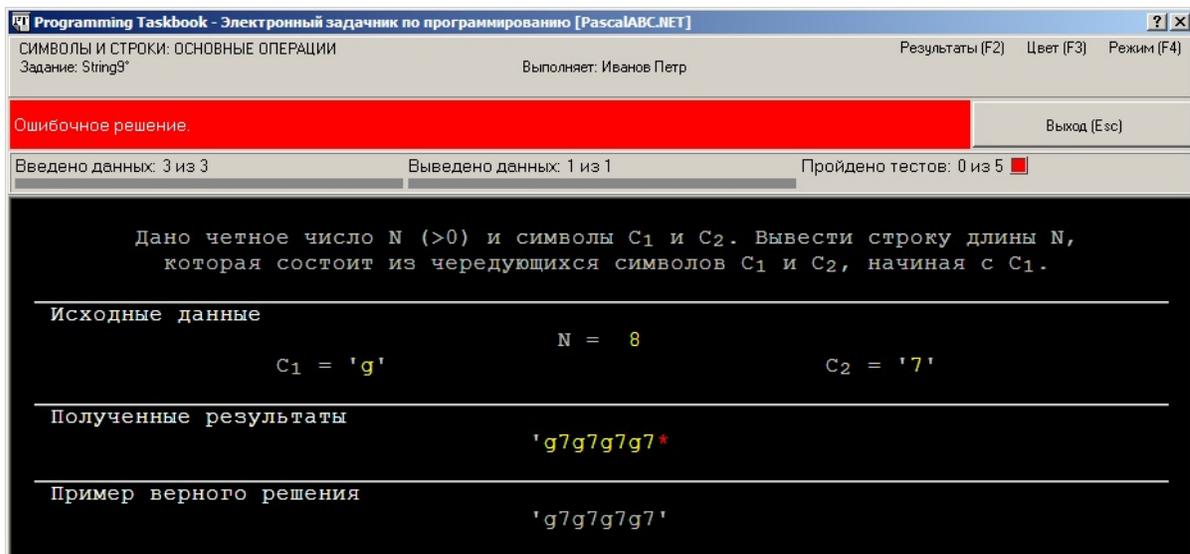
Для формирования нужной строки воспользуемся операцией +

(сцепления строк):

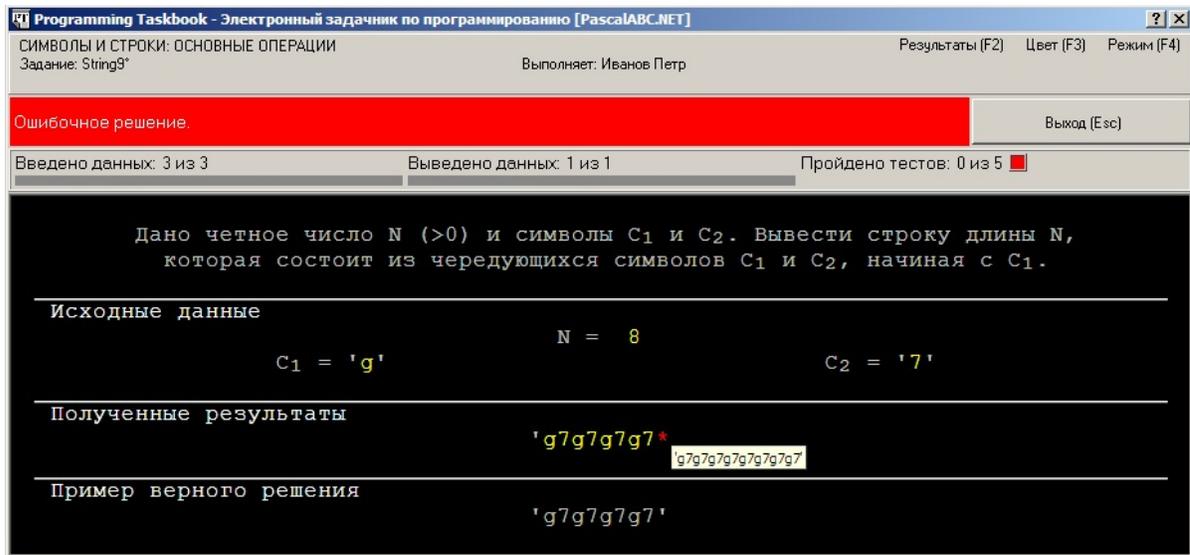
```
uses PT4;
var
  n, i: integer;
  c1, c2: char;
  s: string;
begin
  Task('String9');
  read(n, c1, c2);
  s := '';
  for i := 1 to n do
    s := s + c1 + c2;
  write(s);
end.
```

Заметим, что в операторе `s := ''` нет необходимости, так как все глобальные переменные в Паскале автоматически инициализируются «нулевыми» значениями (для числовых данных это обычные нули, для символов — это символ `#0`, для строк — пустая строка `''`).

Результат выполнения этой программы будет следующим:



Красная звездочка, расположенная в конце выведенной строки, означает, что длина полученной строки превышает длину верной строки. Для того чтобы увидеть на экране всю полученную строку, достаточно подвести курсор мыши к строке со звездочкой; при этом полный текст строки появится во всплывающей подсказке:



Замечание. Красная звездочка может появиться и при выводе ошибочных числовых данных. Например, если ожидается целое число в диапазоне от 1 до 99, а получено число 10000, то на экране изобразится первая цифра этого большого числа, за которой будет указана красная звездочка: 1*.

Верное решение

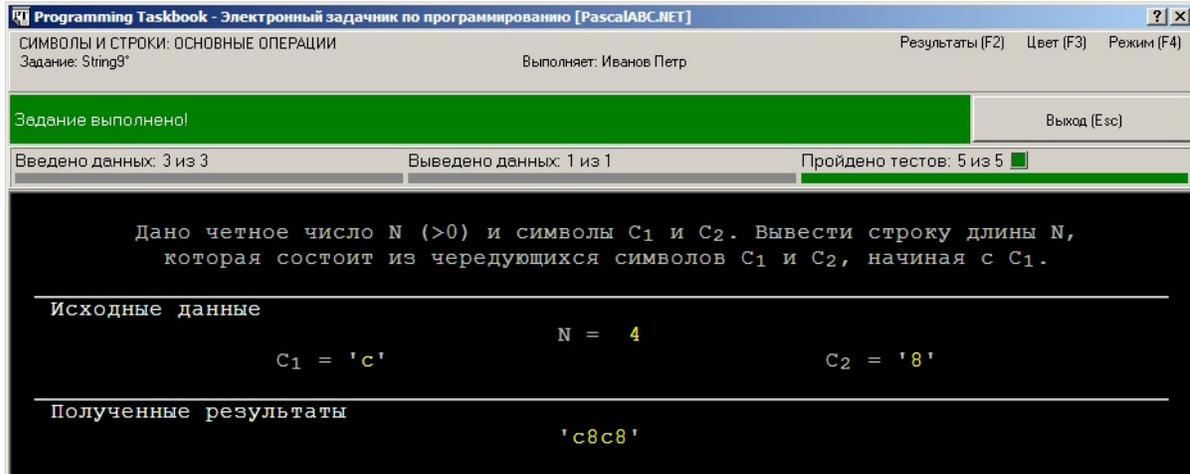
Ошибка в предыдущей программе возникла из-за неверного указания количества итераций цикла. Действительно, на каждой итерации к строке добавляется по два символа, поэтому после n итераций строка будет содержать $2*n$ символов (а не n , как требуется в задании).

Для исправления ошибки достаточно вдвое уменьшить число итераций:

```
uses PT4;
var
  n, i: integer;
  c1, c2: char;
  s: string;
begin
  Task('String9');
  read(n, c1, c2);
  s := '';
  for i := 1 to n div 2 do
    s := s + c1 + c2;
```

```
write(s);  
end.
```

После запуска этого варианта мы получим сообщение «*Верное решение. Тест номер 1 (из 5)*», а после пяти подобных запусков — сообщение «*Задание выполнено!*»:



Замечание. Приведем другой, более быстрый, способ решения задания String9, в котором результирующая строка заполняется *ПОСИМВОЛЬНО*, как обычный массив:

```
uses PT4;  
var  
  n, i: integer;  
  c1, c2: char;  
  s: string;  
begin  
  Task('String9');  
  read(n, c1, c2);  
  SetLength(s, n);  
  for i := 1 to n div 2 do  
  begin  
    s[2 * i - 1] := c1;  
    s[2 * i] := c2;  
  end;  
  write(s);  
end.
```

Обратите внимание на процедуру `SetLength(s, n)`, которая обеспечивает правильную настройку *длины* результирующей строки `s`. Без вызова этой процедуры программа работала бы неверно, так как любая глобальная строковая переменная инициализируется

пустой строкой, а при работе с отдельными символами строки корректировка ее длины не производится.

Просмотр результатов выполнения задания

Щелкнув мышью на метке «Результаты (F2)», расположенной в правом верхнем углу окна задачника, или нажав клавишу **F2**, мы можем вывести на экран *окно результатов*, в котором будет перечислены все наши попытки решения задачи:

```
String9    a08/09 11:17 Ознакомительный запуск.  
String9    a08/09 11:18 Неверно указан тип при вводе исходных данн  
String9    a08/09 11:21 Выведены не все результирующие данные.  
String9    a08/09 11:26 Ошибочное решение.  
String9    a08/09 11:29 Задание выполнено!
```

Для закрытия окна результатов достаточно нажать клавишу **Esc**.
Окно результатов можно отобразить на экране и после закрытия окна задачника и возврата в среду **PascalABC.NET**. Для этого надо использовать команду меню «Модули | Просмотреть результаты», кнопку  или клавиатурную комбинацию **Shift+Ctrl+R**.

Задание на обработку файлов

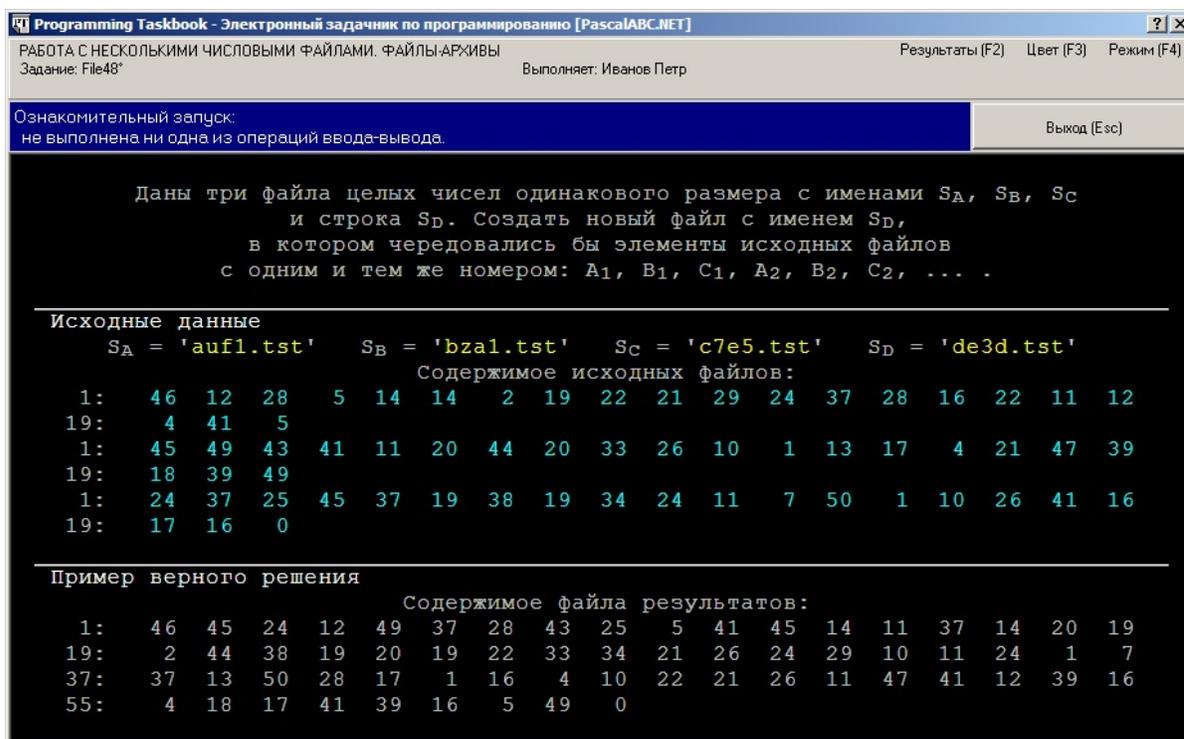
Создание программы-заготовки и знакомство с заданием

В качестве примера задания на обработку файлов рассмотрим задание File48.

Напомним, что программу-заготовку для решения этого задания можно создать с помощью команды меню «Модули | Создать шаблон программы», кнопки  или клавиатурной комбинации **Shift+Ctrl+L**. Эта заготовка будет иметь следующий вид:

```
uses PT4;  
  
begin  
  Task('File48');  
  
end.
```

После запуска данной программы на экране появится [ОКНО](#) [задачника](#):



В первой строке раздела исходных данных указаны имена трех

исходных файлов (S_A , S_B и S_C) и одного результирующего (S_D). В последующих строках раздела исходных данных показано содержимое исходных файлов. Элементы файлов отображаются бирюзовым цветом, чтобы подчеркнуть их отличие от обычных исходных данных (желтого цвета) и комментариев (светло-серого цвета).

Поскольку размер файлов, как правило, превышает количество элементов, которое может уместиться на одной экранной строке, для отображения содержимого файла может отводиться более одной экранной строки. Слева от каждой строки с содержимым файла указывается порядковый номер файлового элемента, значение которого указано первым в этой строке (элементы нумеруются от 1).

Запуск нашей программы признан *ознакомительным* (и поэтому правильность решения не анализировалась), так как в ходе ее выполнения не было введено ни одного элемента исходных данных. При ознакомительном запуске раздел результатов не отображается, однако приводится *пример верного решения*, т. е. те числа, которые должны содержаться в результирующем файле при правильной обработке исходных файлов.

Ввод части исходных данных

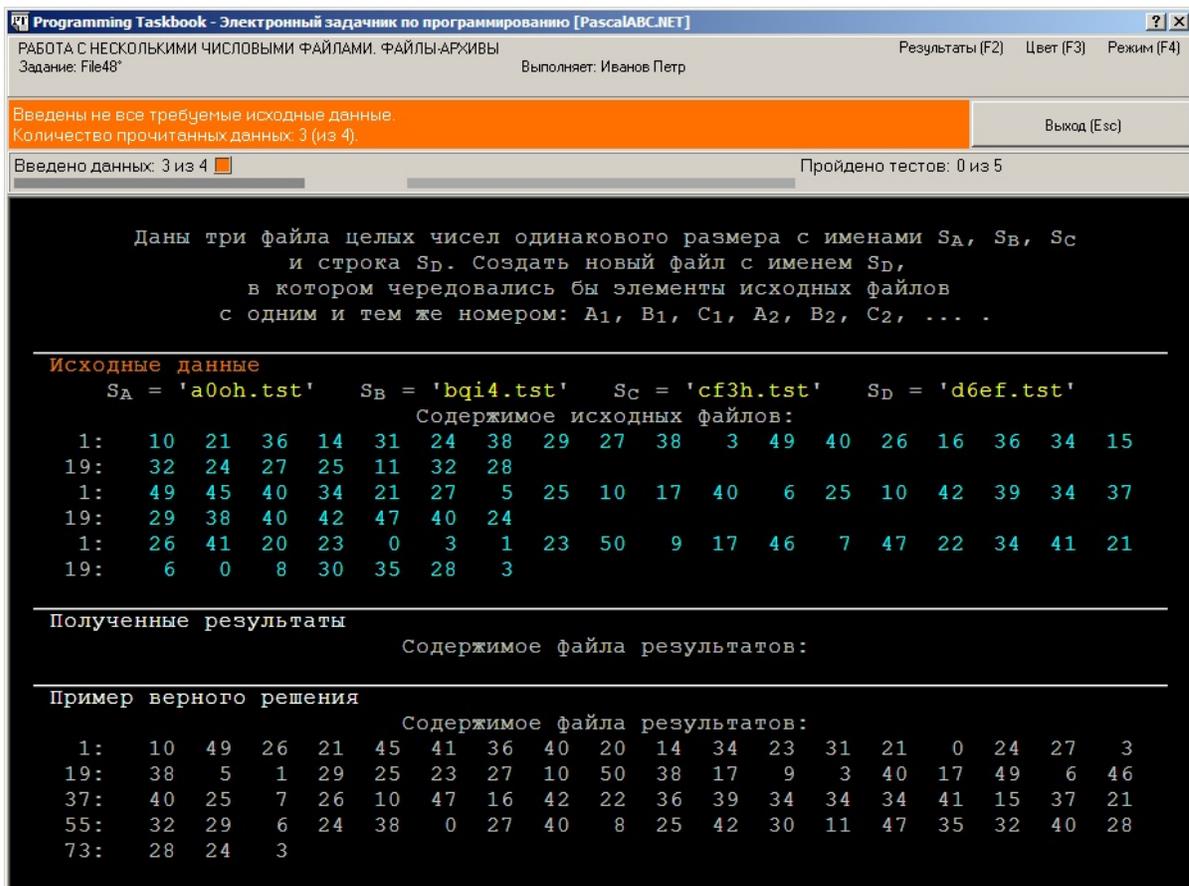
Добавим в программу фрагмент, позволяющий ввести имена исходных файлов и связать с этими файлами соответствующие файловые переменные. Поскольку мы собираемся работать с четырьмя файлами одного типа, удобно предусмотреть *массив* для хранения всех файловых переменных:

```
uses PT4;
var
  i: integer;
  s: string;
  f: array [1..4] of file of integer;
begin
  Task('File48');
  for i := 1 to 3 do
    begin
      read(s);
      Assign(f[i], s);
    end;
```

end.

Мы намеренно ограничились *тремя* итерациями цикла, оставив неп прочитанным имя результирующего файла. Считывание имен файлов производится в одну и ту же переменную `s`, поскольку после связывания файла, имеющего имя `s`, с соответствующей файловой переменной (процедурой `Assign`) все остальные действия с данным файлом в нашей программе будут осуществляться с использованием файловой переменной, без обращения к имени файла.

Запуск нового варианта программы уже не будет считаться ознакомительным, поскольку в программе выполняется ввод исходных данных. Так как имя результирующего файла осталось неп прочитанным, этот вариант решения будет признан неверным и приведет к сообщению «Введены не все требуемые исходные данные»:



При этом на экране появится раздел результатов (кроме

комментария он пока ничего не содержит), а также панель индикаторов. Первый из индикаторов (индикатор ввода) показывает количество введенных исходных данных. Обратите внимание на то, что второй индикатор (индикатор вывода) является неактивным: он выделяется серым цветом более светлого оттенка и не содержит текстового заголовка. Это объясняется тем, что индикатор вывода показывает количество результирующих данных, полученных задачиком от программы, а в нашем случае программа не должна передавать задачику никакие данные; вместо этого ей необходимо создать файл и заполнить его требуемыми значениями. Для заданий подобного типа (обычно это задания, связанные с обработкой файлов) индикатор вывода не используется.

Ввод всех исходных данных без создания требуемого файла

Изменим программу, заменив в заголовке цикла число 3 на 4, и вновь запустим программу. Теперь все данные, необходимые для выполнения задания, введены в программу (это видно по индикатору ввода). Однако задание не выполнено, поскольку результирующий файл не создан. Поэтому решение опять признано ошибочным с диагностикой «*Результирующий файл не найден*»:

```

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
РАБОТА С НЕСКОЛЬКИМИ ЧИСЛОВЫМИ ФАЙЛАМИ. ФАЙЛЫ-АРХИВЫ
Задание: File48*
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)
Результирующий файл не найден.
Введено данных: 4 из 4
Пройдено тестов: 0 из 5

```

Даны три файла целых чисел одинакового размера с именами S_A , S_B , S_C и строка S_D . Создать новый файл с именем S_D , в котором чередовались бы элементы исходных файлов с одним и тем же номером: $A_1, B_1, C_1, A_2, B_2, C_2, \dots$.

Исходные данные

$S_A = 'apby.tst'$ $S_B = 'bb0r.tst'$ $S_C = 'c9ev.tst'$ $S_D = 'dl4g.tst'$

Содержимое исходных файлов:

```

1: 39 37 0 22 2 9 23 40 45 1 50 11 11 5 9 20 34 19
19: 22 37 29 19 8 10 45 32 6
1: 26 0 22 42 40 15 4 5 27 43 9 12 33 14 45 43 17 32
19: 1 29 40 19 38 44 11 48 47
1: 14 21 22 42 40 12 32 31 35 23 22 8 33 5 31 27 7 49
19: 17 8 44 35 1 37 6 48 6

```

Полученные результаты

Содержимое файла результатов:

Пример верного решения

Содержимое файла результатов:

```

1: 39 26 14 37 0 21 0 22 22 22 42 42 2 40 40 9 15 12
19: 23 4 32 40 5 31 45 27 35 1 43 23 50 9 22 11 12 8
37: 11 33 33 5 14 5 9 45 31 20 43 27 34 17 7 19 32 49
55: 22 1 17 37 29 8 29 40 44 19 19 35 8 38 1 10 44 37
73: 45 11 6 32 48 48 6 47 6

```

Пример программы, приводящей к ошибке времени выполнения

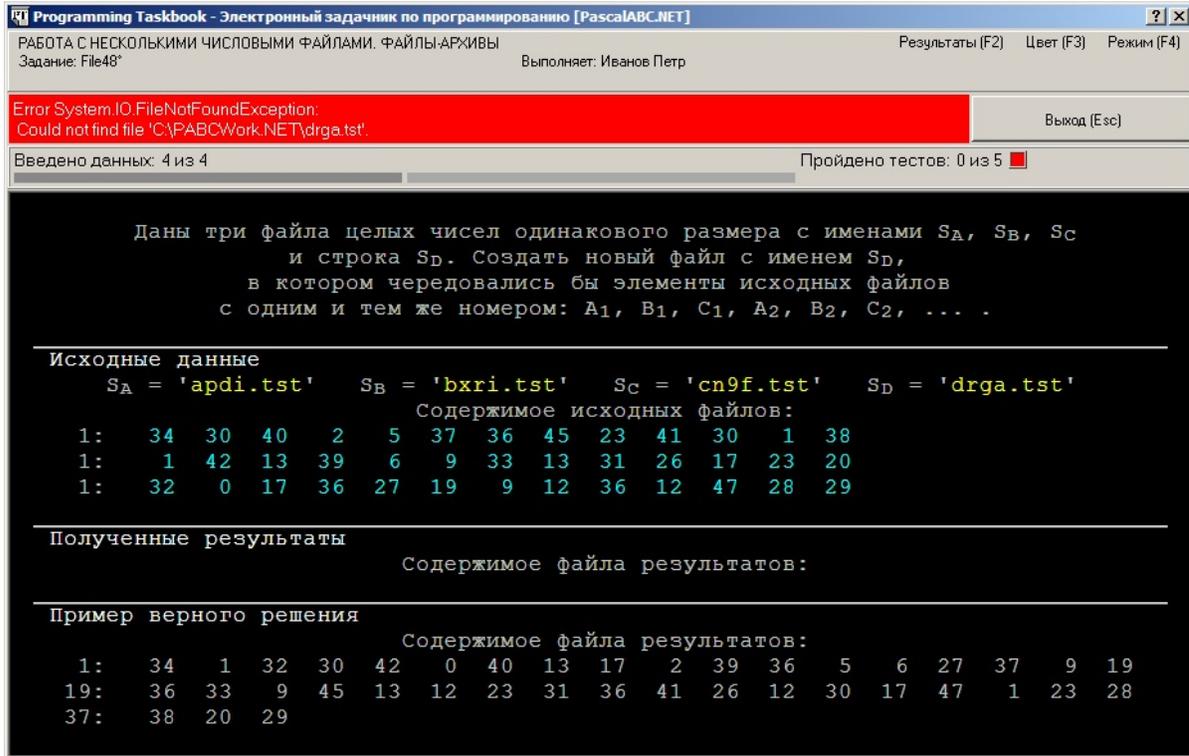
Добавим в тело цикла после процедуры `Assign` вызов процедуры `Reset`, обеспечивающий открытие существующего файла:

```

uses PT4;
var
  i: integer;
  s: string;
  f: array [1..4] of file of integer;
begin
  Task('File48');
  for i := 1 to 4 do
  begin
    read(s);
    Assign(f[i], s);
    Reset(f[i]);
  end;
end.

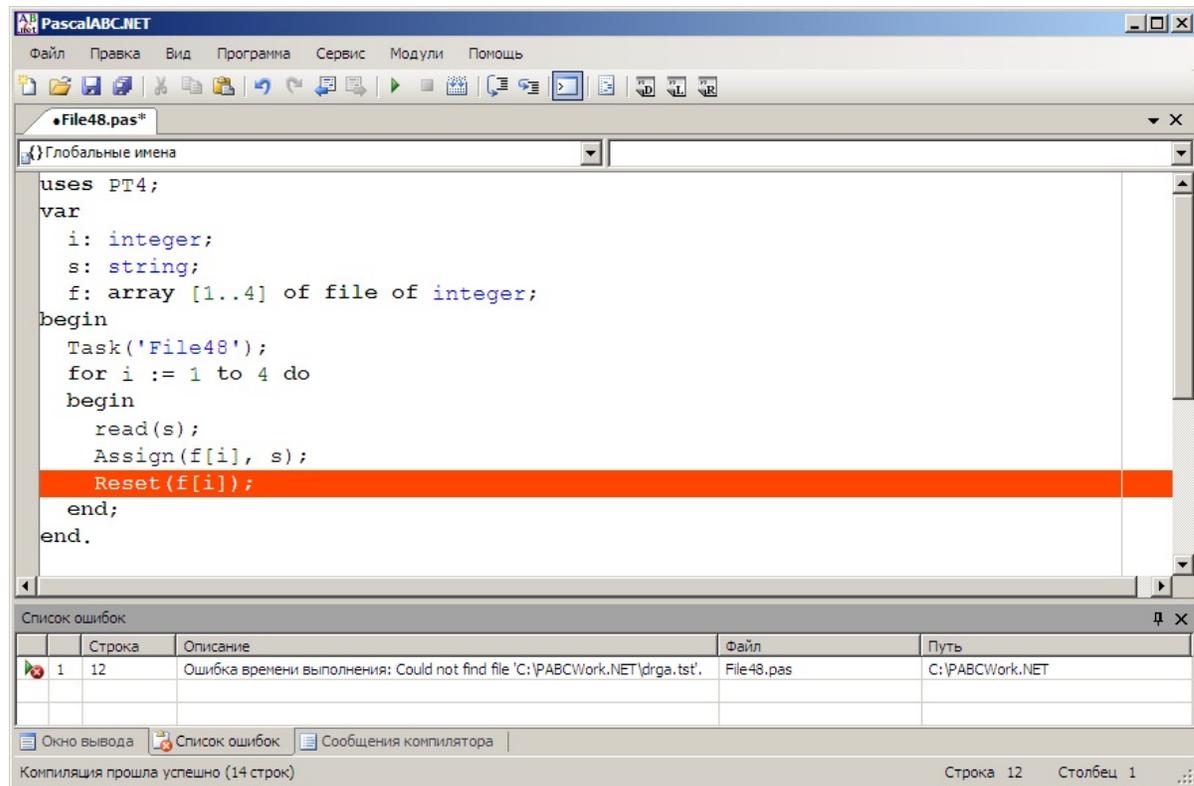
```

Теперь запуск программы приведет к сообщению об ошибке «*Error System.IO.FileNotFoundException*»:



Сообщение, начинающееся со слова *Error*, означает, что при работе программы произошла *ошибка времени выполнения* (Runtime Error). После слова *Error* указывается имя этой ошибки (в данном случае *System.IO.FileNotFoundException*, то есть ошибка ввода-вывода, связанная с тем, что файл не найден) и краткое ее описание на английском языке.

Сообщение об ошибке времени выполнения появится и в разделе «Список ошибок» окна **PascalABC.NET**:



Создание пустого результирующего файла

Для того чтобы избежать ошибки времени выполнения, отсутствующий файл результатов следует открыть не процедурой **Reset**, а процедурой **Rewrite**, которая и обеспечит создание этого файла. Далее, после завершения работы с файлами, открытыми в программе, их необходимо закрыть процедурой **Close**. Добавим в программу соответствующие операторы:

```
uses PT4;
var
  i: integer;
  s: string;
  f: array [1..4] of file of integer;
begin
  Task('File48');
  for i := 1 to 4 do
  begin
    read(s);
    Assign(f[i], s);
    if i < 4 then Reset(f[i])
    else Rewrite(f[i]);
  end;
end;
```

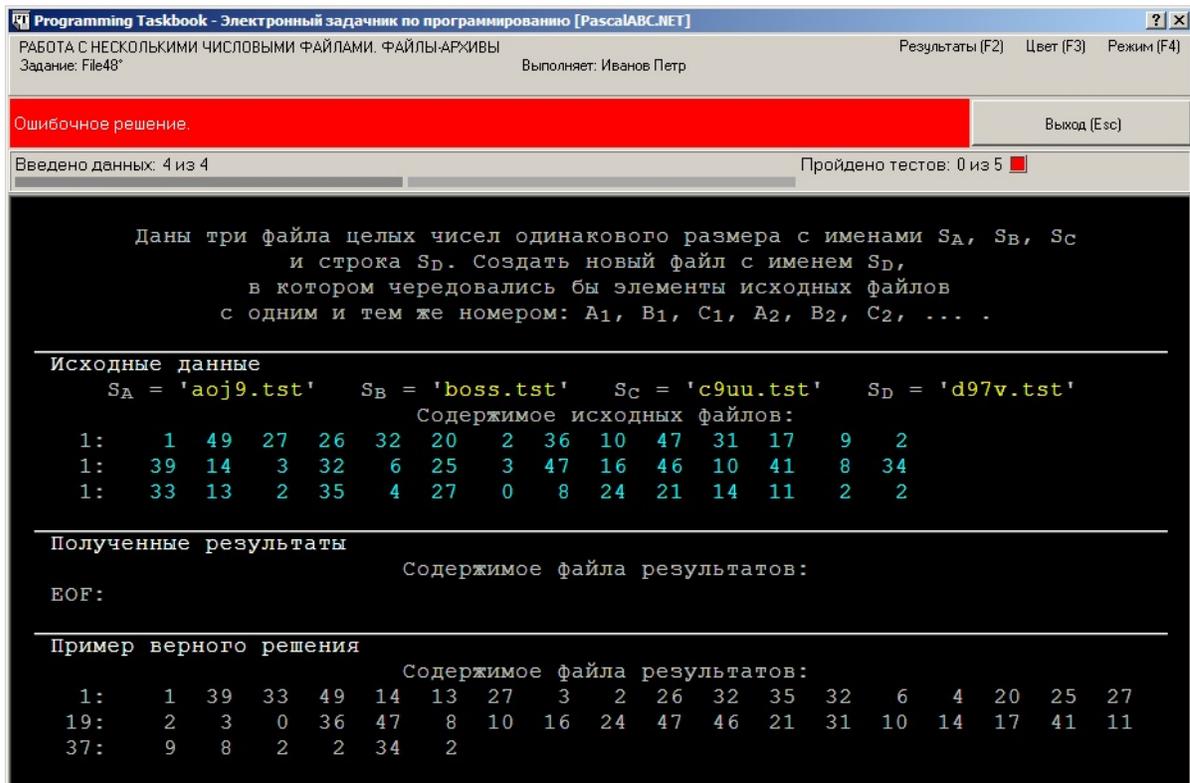
```

{ * }
for i := 1 to 4 do
  Close(f[i]);
end.

```

Комментарий { * } расположен в том месте программы, в котором можно выполнять операции ввода-вывода для всех четырех файлов: они уже открыты процедурами `Reset` или `Rewrite` и еще не закрыты процедурой `Close`.

Запуск этого варианта программы не приведет к ошибке времени выполнения; более того, результирующий файл будет создан. Однако созданный файл останется пустым, то есть не содержащим ни одного элемента. Поэтому при запуске программы на информационной панели появится сообщение «*Ошибочное решение*», а в строке, которая должна содержать элементы результирующего файла, появится текст `EOF`: (особое значение `EOF` для указателя текущей файловой позиции означает, что данный файл существует, но не содержит ни одного элемента):



Пример программы, использующей неправильные типы для

файловых данных

Во всех ранее рассмотренных вариантах программы мы не использовали операции ввода-вывода для файлов. Поэтому тип файлов не играл никакой роли: вместо типа `file of integer` мы могли использовать любой другой файловый тип, например, `file of real`, и результат выполнения программы был бы тем же самым.

Тип файловых элементов становится принципиально важным, если в программе используются операции ввода-вывода для данного файла. Чтобы продемонстрировать это на примере нашей программы, внесем в нее следующие изменения: в описании массива `f` файловых переменных тип `integer` заменим на `real`, в раздел описаний добавим описание переменной `a` типа `real`, в раздел операторов (в позицию, помеченную комментарием `{ * }`) добавим следующий фрагмент:

```
for i := 1 to 3 do
begin
  read(f[i], a);
  write(f[4], a);
end;
```

Данный фрагмент обеспечивает считывание *одного* элемента для каждого из трех исходных файлов и запись этих элементов в результирующий файл (в требуемом порядке). Подчеркнем, что мы *неправильно* указали типы файлов; тем не менее, компиляция программы пройдет успешно, а после ее запуска не произойдет ошибок времени выполнения.

Результат работы программы будет неожиданным:

```

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
РАБОТА С НЕСКОЛЬКИМИ ЧИСЛОВЫМИ ФАЙЛАМИ. ФАЙЛЫ-АРХИВЫ
Задание: File48*
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ошибочное решение.
Вывод (Esc)

Введено данных: 4 из 4
Пройдено тестов: 0 из 5

Даны три файла целых чисел одинакового размера с именами SA, SB, SC
и строка SD. Создать новый файл с именем SD,
в котором чередовались бы элементы исходных файлов
с одним и тем же номером: A1, B1, C1, A2, B2, C2, ...

Исходные данные
SA = 'a4pm.tst'   SB = 'bvbn.tst'   SC = 'c44g.tst'   SD = 'deis.tst'
Содержимое исходных файлов:
1:  33  1  38  1  10  17  27  29  19  39  21  12  18  5
1:  40  34  38  43  9  11  48  44  17  48  44  35  44  28
1:  13  6  44  17  15  38  8  25  23  38  26  40  19  13

Полученные результаты
Содержимое файла результатов:
1:  33  1  40  34  13  6

Пример верного решения
Содержимое файла результатов:
1:  33  40  13  1  34  6  38  38  44  1  43  17  10  9  15  17  11  38
19: 27  48  8  29  44  25  19  17  23  39  48  38  21  44  26  12  35  40
37: 18  44  19  5  28  13

```

Судя по экранной строке с содержимым результирующего файла, в него будут записаны не три, а *шесть элементов*, по два начальных элемента из каждого исходного файла. Объясняется это тем, что после связывания файлов с файловыми переменными типа `file of real` элементами файлов стали считаться *вещественные числа* (занимающие в памяти по 8 байтов), тогда как «на самом деле», то есть по условию задания, элементами файлов являются целые числа (занимающие в памяти по 4 байта). Поэтому считывание из файла и последующая запись в файл одного «вещественного элемента» фактически приводит к считыванию и записи блока данных размером 8 байтов, содержащего два последовательных целочисленных элемента исходного файла.

Итак, мы выяснили, что ошибки, связанные с несоответствием типов файлов, не выявляются при компиляции и не всегда приводят к ошибкам времени выполнения. Это следует иметь в виду, и при появлении «странных» результирующих данных начинать поиск ошибки с проверки типов файловых переменных.

Исправление ошибки, связанной с неверными типами

файловых данных

Заменяем в нашей программе все описания `real` на `integer`:

```
uses PT4;
var
  i: integer;
  s: string;
  f: array [1..4] of file of integer;
  a: integer;
begin
  Task('File48');
  for i := 1 to 4 do
    begin
      read(s);
      Assign(f[i], s);
      if i < 4 then Reset(f[i])
      else Rewrite(f[i]);
    end;
  for i := 1 to 3 do
    begin
      read(f[i], a);
      write(f[4], a);
    end;
  for i := 1 to 4 do
    Close(f[i]);
  end.
```

Мы получим все еще неверное, но вполне «понятное» решение: первые три элемента результирующего файла совпадают с контрольными (то есть «правильными»), а прочие элементы отсутствуют:

```

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
РАБОТА С НЕСКОЛЬКИМИ ЧИСЛОВЫМИ ФАЙЛАМИ. ФАЙЛЫ-АРХИВЫ
Задание: File48*
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ошибочное решение.
Вывод (Esc)

Введено данных: 4 из 4
Пройдено тестов: 0 из 5

Даны три файла целых чисел одинакового размера с именами SA, SB, SC
и строка SD. Создать новый файл с именем SD,
в котором чередовались бы элементы исходных файлов
с одним и тем же номером: A1, B1, C1, A2, B2, C2, ...

Исходные данные
SA = 'a10c.tst'   SB = 'b7wl.tst'   SC = 'cnvj.tst'   SD = 'dflt.tst'
Содержимое исходных файлов:
1:  48  25   6  29   9  13  31  32   6   0  18  25   1  23  33  41  35  22
19:  22  49  15   4  38   2
1:  27  21  10  37  10   5  36  12  42  42  40  42  45  20   8   4  20   7
19:  30  21  12  39  37  21
1:  33  31  47  50  20  21  15  23  26  14  47   9  13  22  26  38  20  43
19:  47  18  19  24  43  28

Полученные результаты
Содержимое файла результатов:
1:  48  27  33

Пример верного решения
Содержимое файла результатов:
1:  48  27  33  25  21  31   6  10  47  29  37  50   9  10  20  13   5  21
19:  31  36  15  32  12  23   6  42  26   0  42  14  18  40  47  25  42   9
37:   1  45  13  23  20  22  33   8  26  41   4  38  35  20  20  22   7  43
55:  22  30  47  49  21  18  15  12  19   4  39  24  38  37  43   2  21  28

```

Верное решение

Приведем, наконец, верное решение задания [File48](#):

```

uses PT4;
var
  i, a: integer;
  s: string;
  f: array [1..4] of file of integer;
begin
  Task('File48');
  for i := 1 to 4 do
  begin
    read(s);
    Assign(f[i], s);
    if i < 4 then Reset(f[i])
    else Rewrite(f[i]);
  end;
  while not Eof(f[1]) do
  for i := 1 to 3 do
  begin

```

```

        read(f[i], a);
        write(f[4], a);
    end;
for i := 1 to 4 do
    Close(f[i]);
end.

```

От предыдущего варианта данное решение отличается добавлением заголовка цикла **while not Eof(f[1]) do**, который обеспечивает считывание всех элементов из исходных файлов (напомним, что по условию задания все исходные файлы имеют *одинаковый размер*) и запись их в результирующий файл в нужном порядке. После запуска этого варианта мы получим сообщение «Верное решение. Тест номер 1 (из 5)», а после пяти подобных запусков — сообщение «Задание выполнено!»:

The screenshot shows the 'Programming Taskbook' application window. The title bar reads 'Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]'. The main window contains a green bar with the text 'Задание выполнено!' and a 'Выход (Esc)' button. Below this, a progress bar shows 'Введено данных: 4 из 4' and 'Пройдено тестов: 5 из 5'. The main content area displays the following text:

Даны три файла целых чисел одинакового размера с именами S_A , S_B , S_C и строка S_D . Создать новый файл с именем S_D , в котором чередовались бы элементы исходных файлов с одним и тем же номером: $A_1, B_1, C_1, A_2, B_2, C_2, \dots$.

Исходные данные

$S_A = 'at82.tst'$ $S_B = 'b47u.tst'$ $S_C = 'caci.tst'$ $S_D = 'd36a.tst'$

Содержимое исходных файлов:

1:	38	37	11	14	34	5	9	10	14	11	18	8	47	46	2	0	16	43
19:	34	8																
1:	45	14	39	4	14	29	32	16	37	8	15	9	20	49	18	18	16	18
19:	6	31																
1:	21	34	12	10	16	22	18	48	18	9	32	23	19	17	7	39	17	37
19:	45	30																

Полученные результаты

Содержимое файла результатов:

1:	38	45	21	37	14	34	11	39	12	14	4	10	34	14	16	5	29	22
19:	9	32	18	10	16	48	14	37	18	11	8	9	18	15	32	8	9	23
37:	47	20	19	46	49	17	2	18	7	0	18	39	16	16	17	43	18	37
55:	34	6	45	8	31	30												

Просмотр результатов выполнения задания

Щелкнув мышью на метке «Результаты (F2)», расположенной в правом верхнем углу окна задачника, или нажав клавишу **F2**, мы можем вывести на экран *окно результатов*, в котором будет перечислены все наши попытки решения задачи:

```
File48      a08/09 12:43 Ознакомительный запуск.  
File48      a08/09 12:50 Введены не все требуемые исходные данные.  
File48      a08/09 12:52 Результирующий файл не найден.  
File48      a08/09 12:53 Error System.IO.FileNotFoundException.  
File48      a08/09 12:57 Ошибочное решение.--3  
File48      a08/09 13:06 Задание выполнено!
```

Для закрытия окна результатов достаточно нажать клавишу **Esc**.
Окно результатов можно отобразить на экране и после закрытия
окна задачника и возврата в среду PascalABC.NET. Для этого надо
использовать команду меню «Модули | Просмотреть результаты»,
кнопку  или клавиатурную комбинацию **Shift+Ctrl+R**.

Задания на указатели и динамические структуры данных

Пример 1. Анализ существующей динамической структуры

В заданиях группы Dynamic мы встречаемся с двумя новыми видами данных: это *динамические структуры*, реализованные в виде цепочек связанных друг с другом записей типа **TNode**, и *указатели* типа **PNode** на записи **TNode**: **PNode = ^TNode**. Типы **TNode** и **PNode** не являются стандартными типами языка Паскаль; они определены в задачнике Programming Taskbook следующим образом (приводятся только те поля записи **TNode**, которые используются при выполнении заданий группы Dynamic):

```
type
  PNode = ^TNode;
  TNode = record
    Data: integer;
    Next: PNode;
    Prev: PNode;
    . . .
end;
```

На примере задания Dynamic2 рассмотрим особенности, связанные с использованием этих новых типов данных.

Создание программы-заготовки и знакомство с заданием

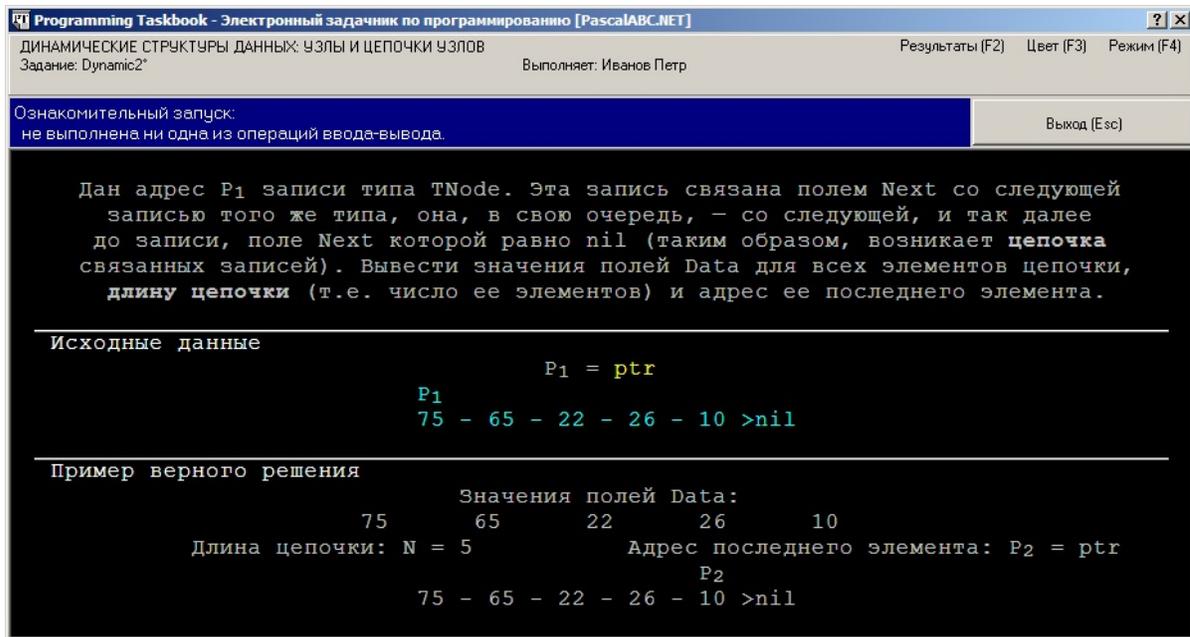
Программа-заготовка для задания Dynamic2, созданная с помощью команды меню «Модули | Создать шаблон программы», кнопки  или клавиатурной комбинации **Shift+Ctrl+L**, имеет следующий вид:

```
uses PT4;

begin
  Task('Dynamic2');

end.
```

После запуска данной программы на экране появится [окно задачника](#):



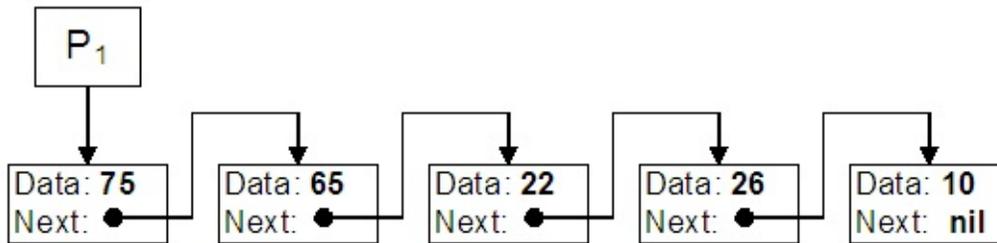
Это окно содержит в качестве исходных и результирующих данных новые элементы: динамические структуры и указатели.

Начнем с описания того, как отображается на экране *динамическая структура*. Для ее вывода используются две экранные строки; в первой строке отображаются имена указателей, связанных с данной структурой, а во второй — содержимое элементов этой структуры, то есть значения их полей **Data** и способ связи между ними. Вся информация о динамической структуре отображается бирюзовым цветом (подобно информации об элементах файлов).

Рассмотрим в качестве примера динамическую структуру, указанную на рисунке:

P_1
75 - 65 - 22 - 26 - 10 >nil

Этот текст означает, что структура состоит из 5 элементов, причем ее первый элемент имеет поле **Data**, равное 75, и связан с помощью своего поля **Next** со вторым элементом, поле **Data** которого равно 65, и так далее до последнего, пятого элемента, поле **Data** которого равно 10, а поле **Next** равно **nil**, что является признаком завершения структуры. Таким образом, текст, описывающий данную динамическую структуру, является максимально упрощенным вариантом следующей схемы:



Поскольку эта структура указана в разделе исходных данных, следовательно, после инициализации задания она уже *существует* и размещается в некоторой области динамической памяти (подобно тому, как исходные файлы после инициализации задания размещаются в каталоге учащегося).

Как получить доступ к этой существующей динамической структуре? Здесь также уместна аналогия с файлами. Для доступа к внешнему файлу необходимо знать его *имя*, и в любом задании на обработку файлов имена исходных файлов входят в набор исходных данных. Для доступа к данным, размещенным в динамической памяти, необходимо знать их *адрес*, и поэтому в любом задании на обработку динамических структур в набор исходных данных входят *указатели*, содержащие адреса этих структур.

Из текста, описывающего динамическую структуру, видно, что на ее первый элемент указывает указатель с именем P_1 , который также содержится в наборе исходных данных. Описание этого указателя имеет вид

$P_1 = ptr$

Здесь текст $P_1 =$ является *комментарием* и выделяется, как обычный комментарий, светло-серым цветом, а текст ptr означает, что этот элемент исходных данных является *указателем*, который надо ввести в программу с помощью процедуры ввода `read`.

Замечание. Может возникнуть вопрос: почему вместо условного текста «ptr» не отображается «настоящее» значение указателя (то есть некоторый четырехбайтный адрес)? Это связано с тем, что, даже выведя это значение на экран, мы не сможем определить, с какими данными связан этот адрес, поэтому подобная информация на экране будет излишней.

Итак, слово `ptr` в разделе исходных или результирующих данных означает, что соответствующий элемент данных является *указателем*, причем непустым (для пустого указателя используется слово `nil`). Определить, с каким элементом динамической структуры данных связан непустой указатель, можно по экранной информации об этой динамической структуре. Разумеется, при чтении указателя программа учащегося получит «настоящий» адрес, с помощью которого она сможет обратиться к исходной динамической структуре. Аналогично, создав (или преобразовав) некоторую динамическую структуру, программа учащегося должна передать задачнику некоторый адрес, связанный в этой структурой (используя процедуру вывода `write`). Зная этот адрес, задачник сможет проверить правильность созданной структуры.

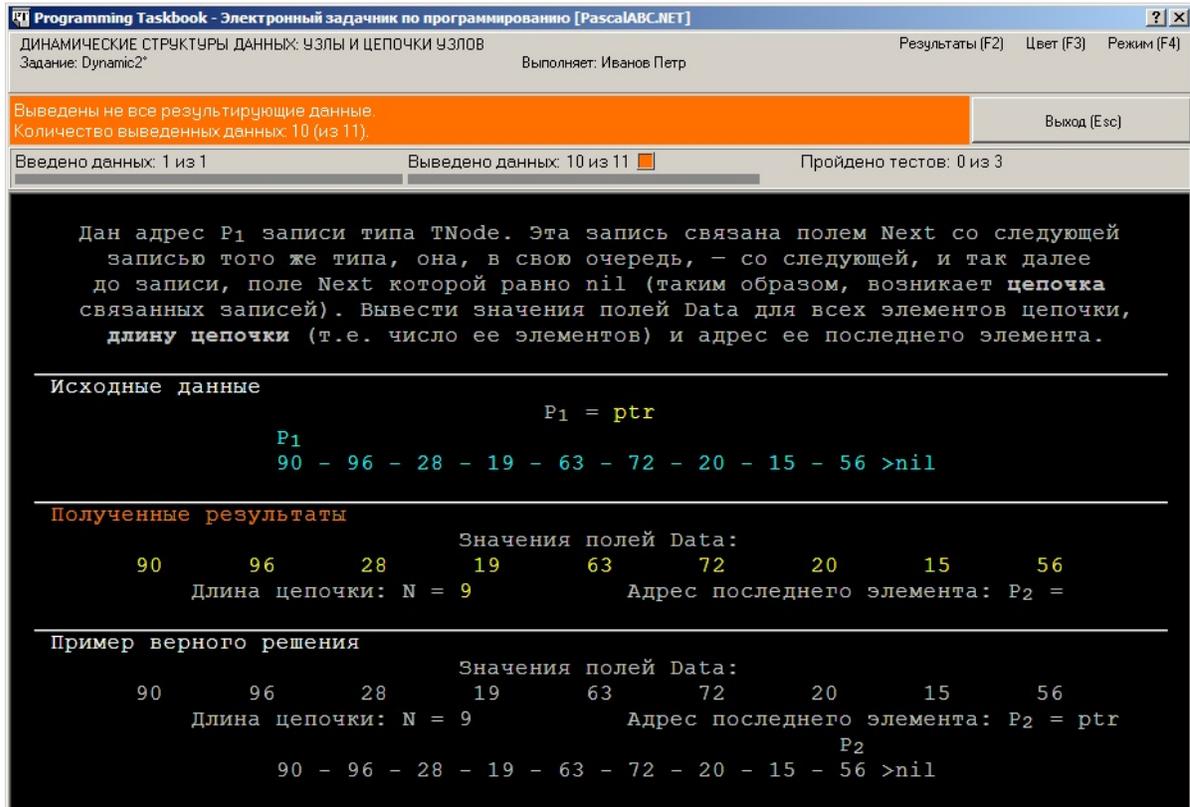
Приступаем к решению

Вернемся к заданию `Dynamic2`. В нем не требуется ни создавать, ни преобразовывать исходную структуру данных; ее необходимо лишь проанализировать, а именно, определить значения всех ее элементов, подсчитать количество элементов и, кроме того, вывести указатель на последний элемент этой структуры.

Приведем вначале неполное решение задачи, выводящее все необходимые данные, кроме указателя на последний элемент:

```
uses PT4;
var
  p1: PNode;
  n: integer;
begin
  Task('Dynamic2');
  read(p1);
  n := 0;
  while p1 <> nil do
    begin
      write(p1^.Data);
      n := n + 1;
      p1 := p1^.Next;
    end;
  write(n);
end.
```

После запуска программы можно убедиться, что все числовые результирующие данные найдены правильно, однако из-за того, что не выведен указатель на последний элемент, решение признано ошибочным с диагностикой «Выведены не все результирующие данные»:



Добавим в конец программы оператор

```
write(p1);
```

После запуска нового варианта программы все требуемые данные будут выведены, однако результирующее значение указателя будет равно `nil`. Это связано с тем, что после завершения цикла `while` в переменной `p1` содержится *нулевой указатель*, а не указатель на последний элемент динамической структуры:

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ: УЗЛЫ И ЦЕПОЧКИ УЗЛОВ Результаты (F2) Цвет (F3) Режим (F4)
 Задание: Dynamic2* Выполняет: Иванов Петр

Ошибочное решение. Выход (Esc)

Введено данных: 1 из 1 Выведено данных: 9 из 9 Пройдено тестов: 0 из 3

```

Дан адрес P1 записи типа TNode. Эта запись связана полем Next со следующей
записью того же типа, она, в свою очередь, - со следующей, и так далее
до записи, поле Next которой равно nil (таким образом, возникает цепочка
связанных записей). Вывести значения полей Data для всех элементов цепочки,
длину цепочки (т.е. число ее элементов) и адрес ее последнего элемента.
  
```

Исходные данные

```

                                P1 = ptr
P1
19 - 96 - 72 - 22 - 37 - 98 - 39 >nil
  
```

Полученные результаты

```

                                Значения полей Data:
    19      96      72      22      37      98      39
Длина цепочки: N = 7      Адрес последнего элемента: P2 = nil

    19 - 96 - 72 - 22 - 37 - 98 - 39 >nil
  
```

Пример верного решения

```

                                Значения полей Data:
    19      96      72      22      37      98      39
Длина цепочки: N = 7      Адрес последнего элемента: P2 = ptr
                                P2
    19 - 96 - 72 - 22 - 37 - 98 - 39 >nil
  
```

Правильное решение

Для того чтобы получить правильное решение, опишем вспомогательную переменную `p2`, в которой будем сохранять адрес элемента, *предшествующего* элементу с адресом `p1`. После завершения цикла `while` в этой переменной будет храниться адрес последнего элемента динамической структуры:

```

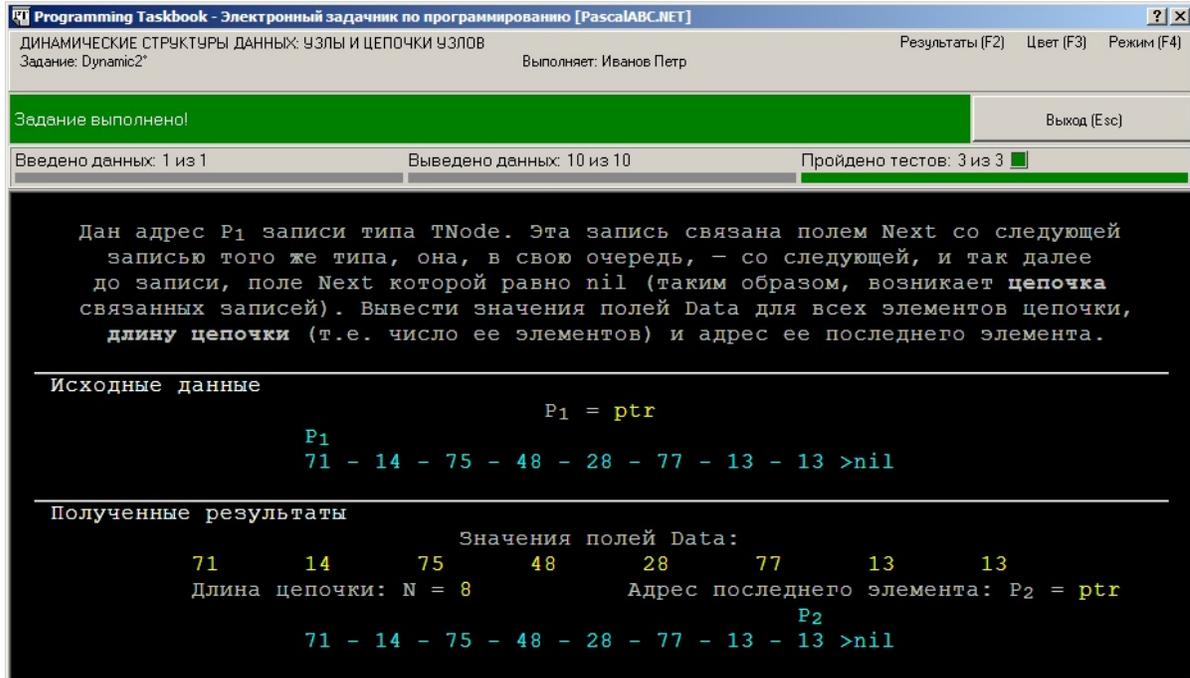
uses PT4;
var
  p1, p2: PNode;
  n: integer;
begin
  Task('Dynamic2');
  read(p1);
  n := 0;
  while p1 <> nil do
  begin
    write(p1^.Data);
    n := n + 1;
    p2 := p1;      { сохраняем адрес текущего элемента }
  
```

```

    p1 := p1^.Next; { и переходим к следующему элементу }
end;
write(n, p2);
end.

```

Запустив эту программу три раза, мы получим сообщение «Задание выполнено!»:



Пример 2. Добавление элемента к динамической структуре

Знакомство с заданием

Рассмотрим простейшее задание, связанное с добавлением элемента к динамической структуре-стеку: Dynamic3.

При ознакомительном запуске этого задания мы обнаружим новое обозначение в тексте, описывающем результирующий стек, а именно, точки, обрамляющие первый элемент стека:

```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ: СТЕК
Задание: Dynamic3*
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)
Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода.
Выход (Esc)

Дано число D и указатель P1 на вершину непустого стека.
Добавить элемент со значением D в стек
и вывести адрес P2 новой вершины стека.

Исходные данные
D = 96
P1 = ptr
P1
38 - 93 - 72 - 69 >nil

Пример верного решения
P2 = ptr
P2
.96.- 38 - 93 - 72 - 69 >nil
```

Точки обозначают элементы динамической структуры, память для которых *должна быть выделена программой учащегося* (в отличие от тех элементов, которые размещаются в памяти самим задачиком).

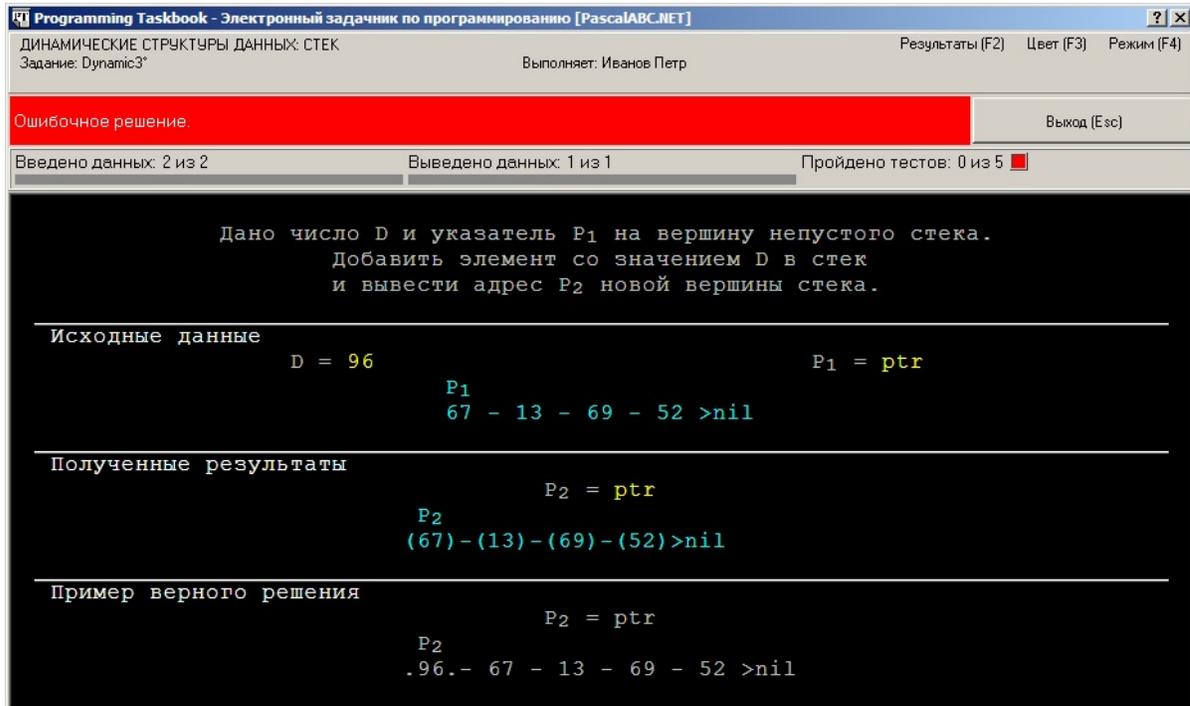
Приступаем к решению

Что произойдет, если динамическая структура будет создана с ошибками? Для того чтобы это выяснить, вернем в программе, решающей задание Dynamic3, указатель на прежнюю вершину стека, не добавляя к ней новый элемент:

```
uses PT4;
var
  d: integer;
  p1: PNode;
begin
  Task('Dynamic3');
```

```
read(d, p1);
write(p1);
end.
```

После запуска данной программы окно задачника примет вид:



Скобки вокруг каждого элемента результирующего стека означают, что эти элементы созданы самим задачиком, но *располагаются не на тех позициях, на которых они должны находиться при правильном решении*. Действительно, тот элемент, который в решении является первым, должен (после добавления нового элемента) оказаться вторым и т. д. Итак, наличие скобок в тексте результирующей динамической структуры означает, что ее элементы располагаются не в том порядке, который требуется.

Правильное решение

Для получения правильного решения задания Dynamic3 необходимо явно выделить память для нового элемента, используя процедуру [New](#), и заполнить поля этого элемента, связав его с текущей вершиной стека (в результате сам этот элемент станет новой вершиной, адрес которой и следует вывести):

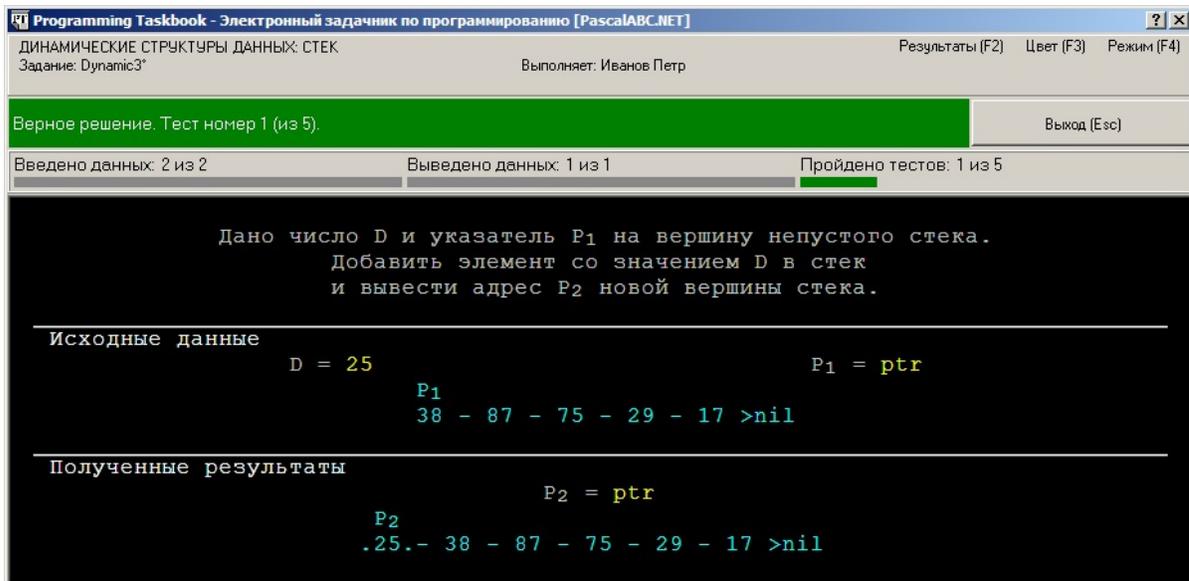
```
uses PT4;
```

```

var
  d: integer;
  p1, p2: PNode;
begin
  Task('Dynamic3');
  read(d, p1);
  New(p2);
  p2^.Data := D;
  p2^.Next := p1;
  write(p2);
end.

```

Приведем вид окна задачника при первом запуске этой программы:



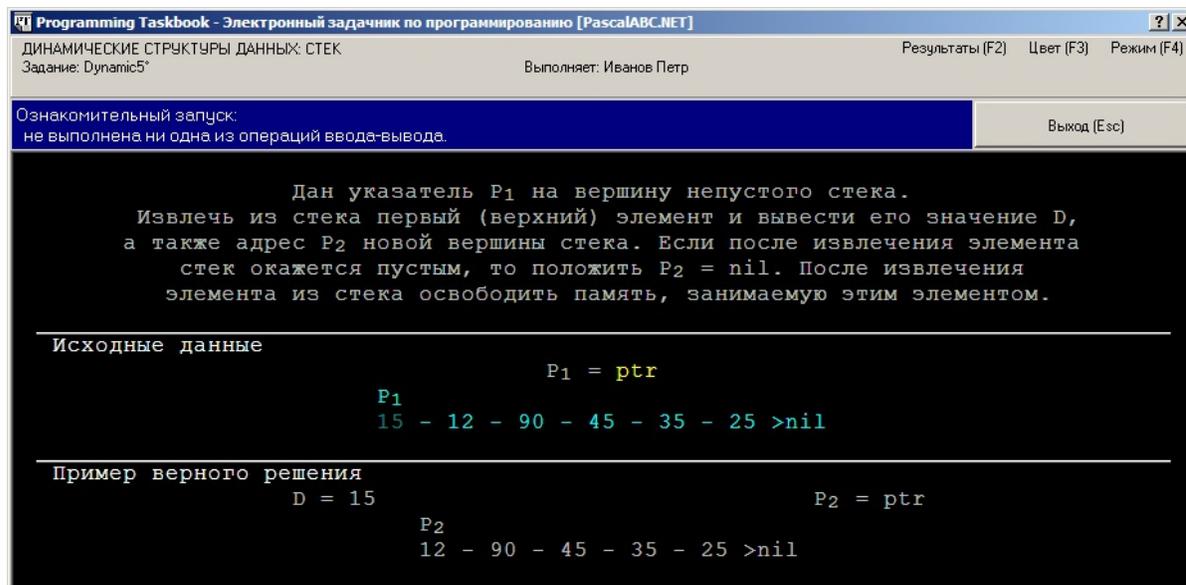
Пример 3. Удаление элемента из динамической структуры

Знакомство с заданием

Рассмотрим простейшее задание на удаление элемента из динамической структуры — Dynamic5. В нем требуется удалить из стека вершину и вернуть указатель на новую вершину, то есть на элемент, расположенный непосредственно за удаленным.

Особенность заданий на удаление элементов из динамических структур заключается в том, что удаляемый элемент необходимо не только «отсоединить» от исходной динамической структуры, но и полностью «уничтожить», то есть *освободить память, занимаемую этим элементом*.

Для того чтобы напомнить учащемуся о необходимости уничтожения некоторых элементов исходной динамической структуры, эти элементы выделяются на экране синим цветом меньшей яркости, чем обычные элементы (на рисунке таким способом выделен элемент 15):



```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ: СТЕК
Задание: Dynamic5
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)
Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода.
Выход (Esc)

Дан указатель P1 на вершину непустого стека.
Извлечь из стека первый (верхний) элемент и вывести его значение D,
а также адрес P2 новой вершины стека. Если после извлечения элемента
стек окажется пустым, то положить P2 = nil. После извлечения
элемента из стека освободить память, занимаемую этим элементом.

Исходные данные
P1 = ptr
P1
15 - 12 - 90 - 45 - 35 - 25 >nil

Пример верного решения
D = 15
P2 = ptr
P2
12 - 90 - 45 - 35 - 25 >nil
```

Приступаем к решению

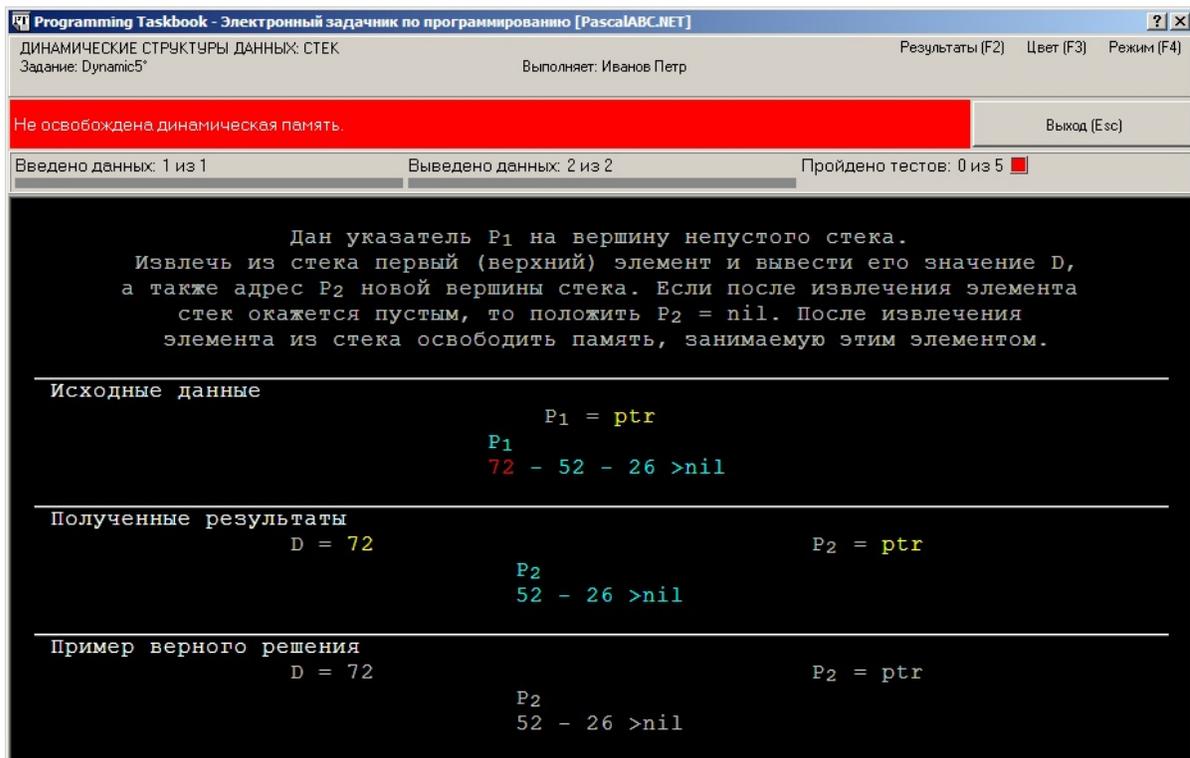
Вначале приведем неправильный вариант решения, в котором не освобождается память, занимаемая удаленным из стека элементом:

```

uses PT4;
var p1: PNode;
begin
    Task('Dynamic5');
    read(p1);
    write(p1^.Data, p1^.Next);
end.

```

Несмотря на то что все результирующие данные будут совпадать с контрольными (то есть текст в разделах «Полученные результаты» и «Пример верного решения» будет одинаковым), на информационной панели появится сообщение об ошибке «*Не освобождена динамическая память*», а в разделе исходных данных будет выделен красным цветом тот элемент, который требовалось удалить:



Правильное решение

Для получения правильного решения достаточно добавить в конец программы оператор вызова процедуры `Dispose`, освобождающий память, на которую указывает указатель `p1`:

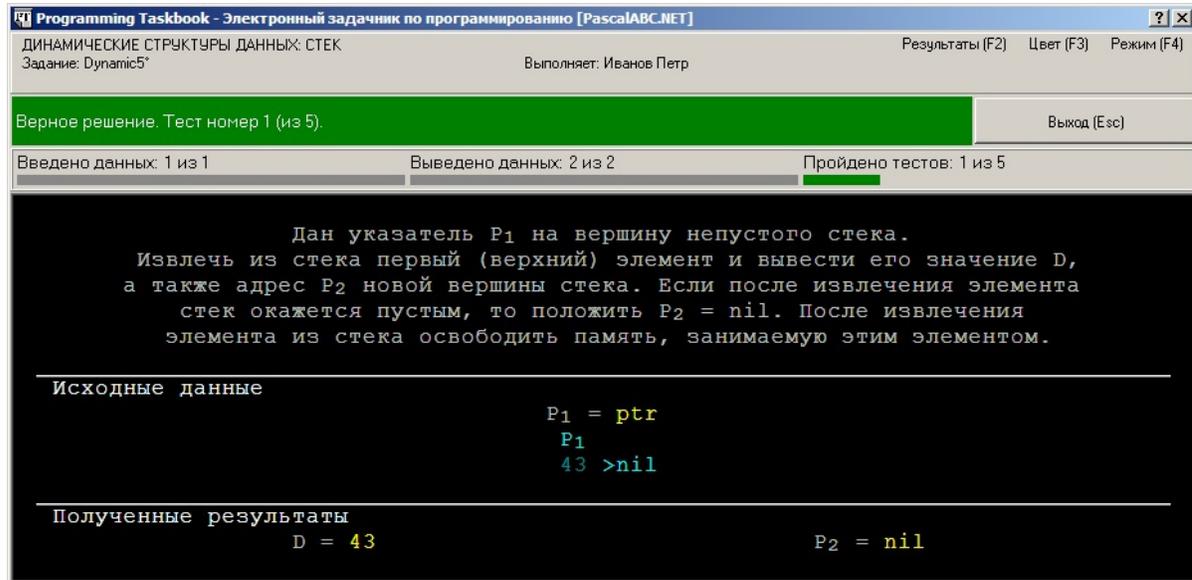
```

uses PT4;

```

```
var p1: PNode;  
begin  
  Task('Dynamic5');  
  read(p1);  
  write(p1^.Data, p1^.Next);  
  Dispose(p1);  
end.
```

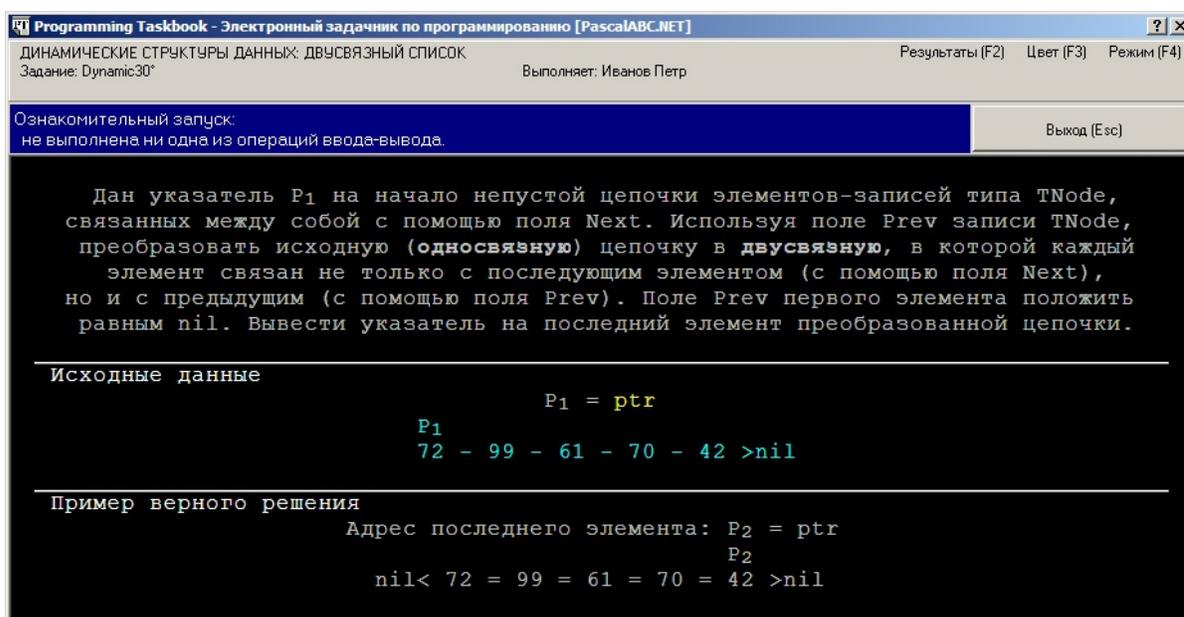
Приведем вид окна задачника при первом запуске этой программы:



Пример 4. Двусвязные динамические структуры

Знакомство с заданием

Особенности работы с двусвязными динамическими структурами рассмотрим на примере задания Dynamic30, в котором требуется преобразовать исходную односвязную структуру в двусвязную. Запустив программу-заготовку, созданную для этого задания, мы увидим в области исходных данных информацию об «обычной» односвязной структуре, подобной рассмотренным в предыдущих примерах:



```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ: ДВУСВЯЗНЫЙ СПИСОК
Задание: Dynamic30
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода. Выход (Esc)

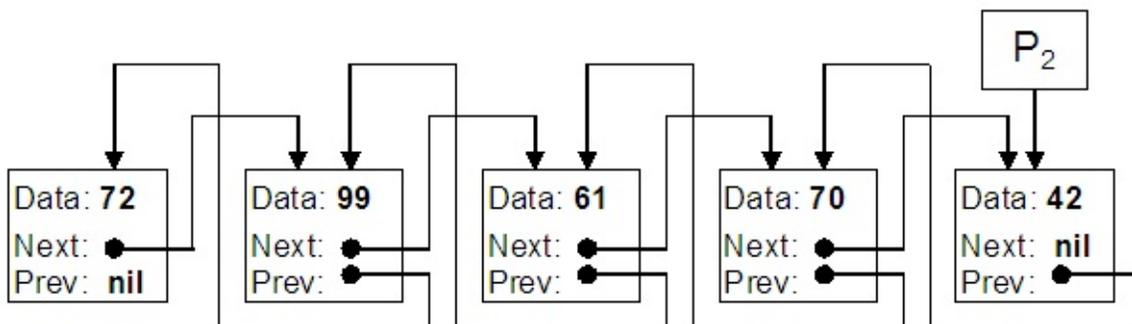
Дан указатель P1 на начало непустой цепочки элементов-записей типа TNode,
связанных между собой с помощью поля Next. Используя поле Prev записи TNode,
преобразовать исходную (односвязную) цепочку в двусвязную, в которой каждый
элемент связан не только с последующим элементом (с помощью поля Next),
но и с предыдущим (с помощью поля Prev). Поле Prev первого элемента положить
равным nil. Вывести указатель на последний элемент преобразованной цепочки.

Исходные данные
P1 = ptr
P1
72 - 99 - 61 - 70 - 42 >nil

Пример верного решения
Адрес последнего элемента: P2 = ptr
P2
nil < 72 = 99 = 61 = 70 = 42 >nil
```

Динамическая структура, приведенная в разделе результатов, имеет две особенности: во-первых, ее элементы связаны символом =, а во-вторых, перед первым элементом присутствует текст nil<.

Это означает, что результирующая структура является двусвязной, то есть каждый ее элемент связан не только с последующим элементом (с помощью поля Next, как в односвязной структуре), но и с предыдущим элементом (с помощью нового поля Prev), а поле Prev первого элемента имеет значение nil:



Приступаем к решению

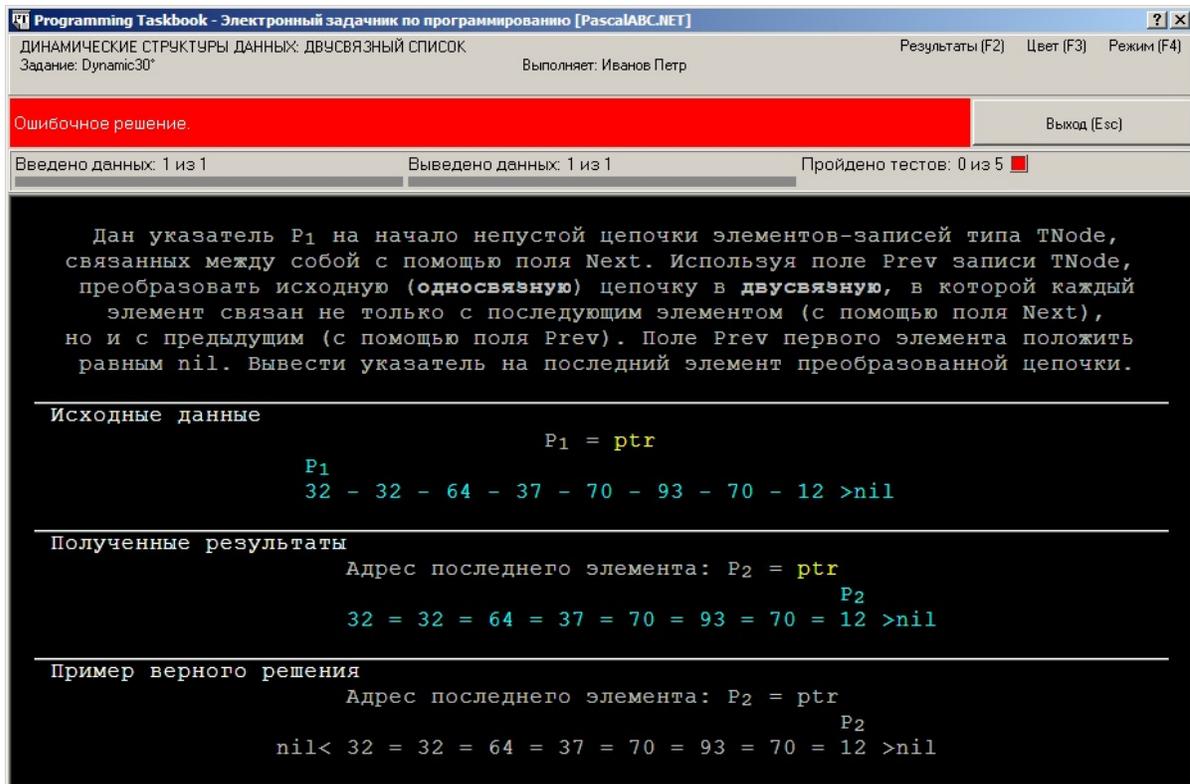
Для преобразования исходной односвязной структуры в двусвязную необходимо задать правильные значения для полей `Prev` всех элементов структуры, перебирая в цикле пары соседних элементов:

```

uses PT4;
var
    p1, p: PNode;
begin
    Task('Dynamic30');
    read(p1);
    p := p1^.Next;
    while p <> nil do
        begin
            p^.Prev := p1;
            p1 := p1^.Next;
            p := p^.Next;
        end;
    write(p1); { вывод указателя на последний элемент }
end.

```

В этой программе мы определили поля `Prev` для всех элементов, кроме первого. Поэтому решение будет считаться ошибочным (обратите внимание на то, что перед первым элементом полученного списка отсутствует текст `nil`):



Замечание. При анализе ошибочного решения часто оказывается полезным и специальное обозначение « \Leftarrow » для двойной связи. Предположим, например, что информация о результирующей двусвязной структуре, созданной программой, имеет вид:

$$\text{nil} \leftarrow 33 = 64 - 78 = 12 = 51 \rightarrow \text{nil}$$

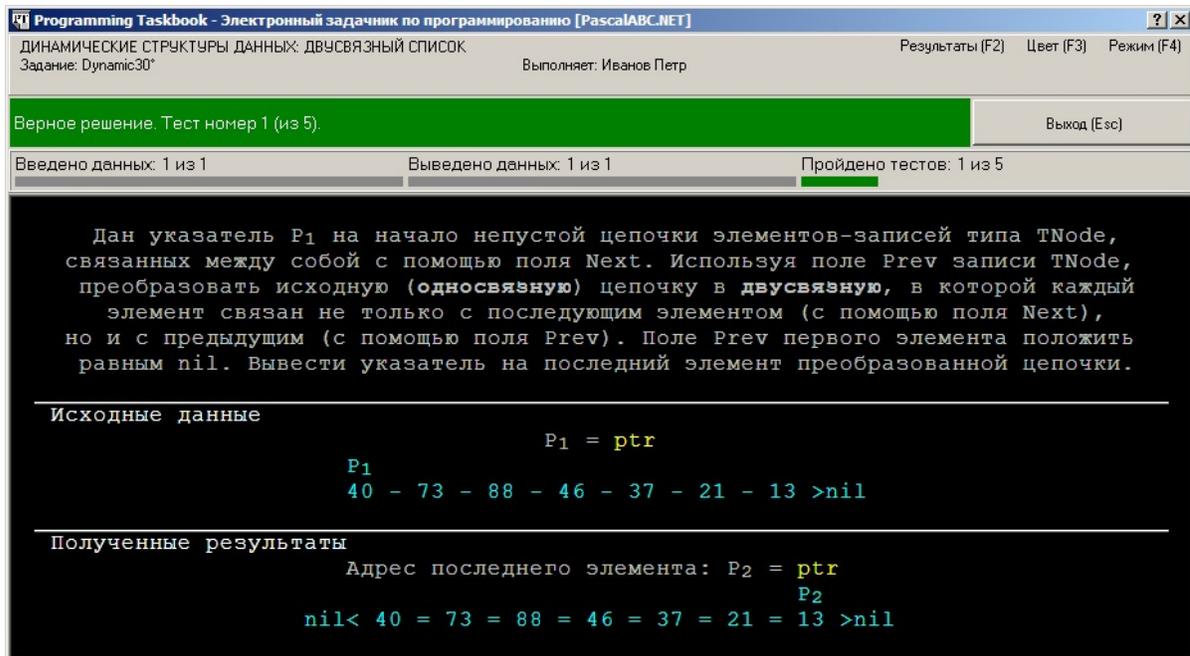
Это означает, что между вторым и третьим элементом структуры имеется не двойная, а *одинарная* связь (поле `Next` второго элемента содержит адрес третьего элемента, а поле `Prev` третьего элемента *не содержит* адрес второго).

Правильное решение

Для получения правильного решения достаточно добавить в программу перед циклом `while` следующий оператор:

```
p1^.Prev := nil;
```

Приведем вид окна задачника при первом запуске исправленной программы:



Замечание. Для задания Dynamic30 возможен более короткий вариант решения, в котором не требуется особо обрабатывать первый элемент списка:

```

uses PT4;
var
  p1, p: PNode;
begin
  Task('Dynamic30');
  p := nil;
  read(p1);
  while p1 <> nil do
  begin
    p1^.Prev := p;
    p := p1;
    p1 := p1^.Next;
  end;
  write(p);
end.

```

Пример 5. Циклические динамические структуры

Знакомство с заданием

Динамическая структура называется *циклической*, если она замкнута в «кольцо», то есть ее последний элемент связан полем `Next` с первым (в случае двусвязной структуры требуется также, чтобы ее первый элемент был связан полем `Prev` с последним элементом).

Простейшим заданием на циклические структуры является `Dynamic55`, в котором требуется преобразовать обычный двусвязный список в циклический.

Запустив программу-заготовку для этого задания, мы увидим на экране изображение двух динамических структур, причем исходная структура является «обычным» двусвязным списком, а результирующая структура — циклическим двусвязным списком:

```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ: ДВУСВЯЗНЫЙ СПИСОК
Задание: Dynamic55*
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода. Выход (Esc)

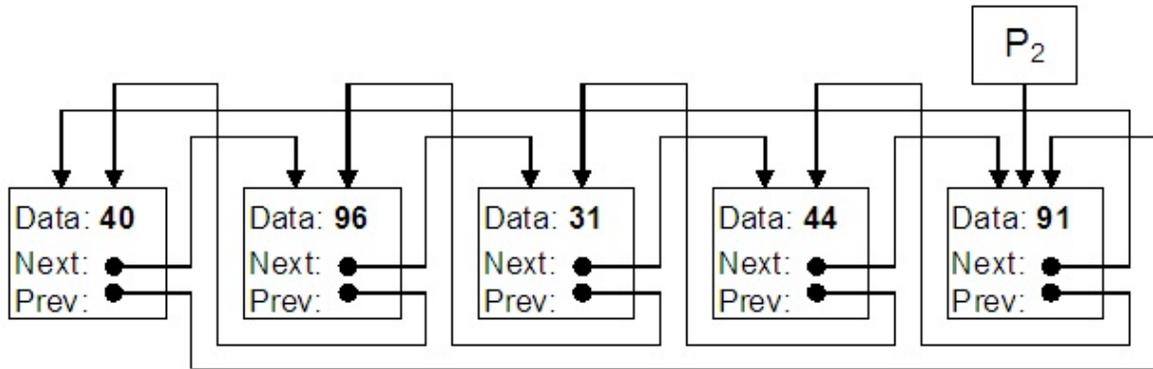
Дан указатель P1 на первый элемент непустого двусвязного списка.
Преобразовать список в циклический, записав в поле Next последнего
элемента списка адрес его первого элемента, а в поле Prev первого
элемента — адрес последнего элемента. Вывести указатель на элемент,
который был последним элементом исходного списка.

Исходные данные
P1 = ptr
nil < 40 = 96 = 31 = 44 = 91 > nil

Пример верного решения
Адрес последнего элемента: P2 = ptr
P2
<< = 40 = 96 = 31 = 44 = 91 = >>
```

Обозначения `<< =` и `= >>` позволяют отличить циклический список от обычного (напомним, что у обычного двусвязного списка поле `Prev` первого элемента и поле `Next` последнего элемента равны `nil`).

Таким образом, экранный текст, описывающий циклический двусвязный список, является упрощенным вариантом следующей схемы:



Приступаем к решению

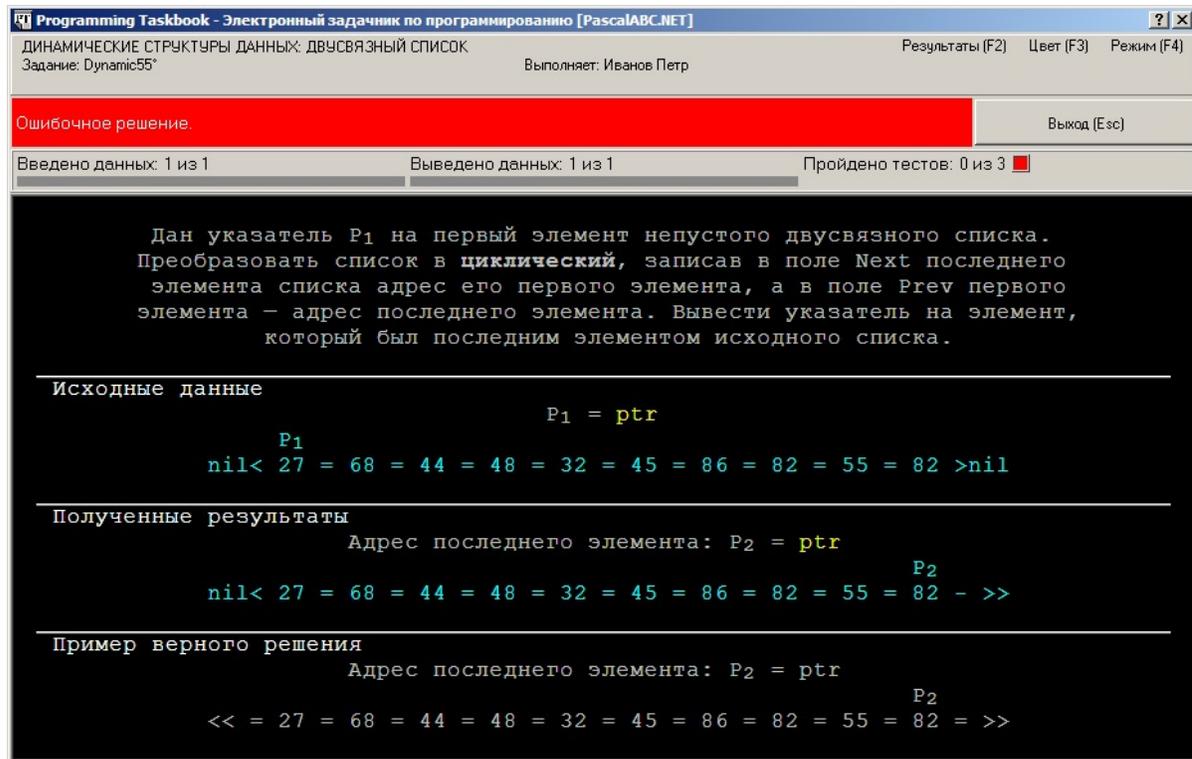
Для решения задания Dynamic55 достаточно найти последний элемент исходного списка и связать его с первым элементом:

```

uses PT4;
var
    p1, p2: PNode;
begin
    Task('Dynamic55');
    read(p1);
    p2 := p1;
    while p2^.Next <> nil do
        p2 := p2^.Next;
    p2^.Next := p1;
    write(p2);
end.

```

В данном варианте решения мы «забыли» о том, что надо связать не только последний элемент с первым, но и первый с последним (поскольку наш список — двусвязный). Поэтому решение оказалось ошибочным (обратите внимание на то, что после последнего элемента полученного списка изображена одинарная, а не двойная черта):

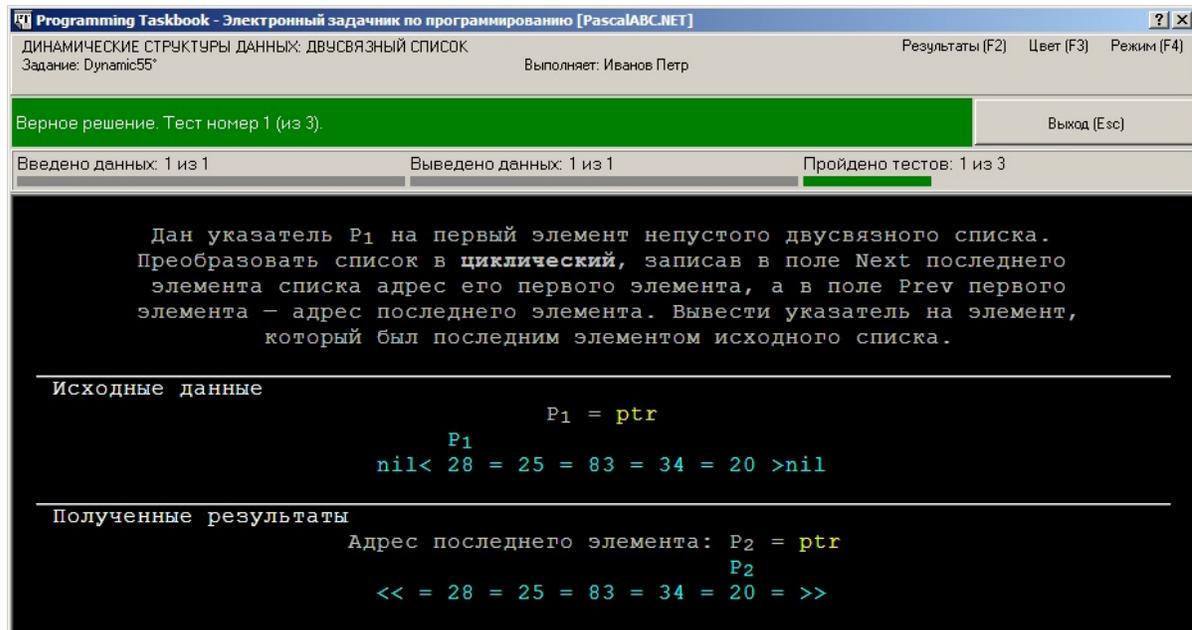


Правильное решение

Для получения правильного решения достаточно добавить в программу перед процедурой вывода `write` следующий оператор:

```
p1^.Prev := p2;
```

Приведем вид окна задачника при первом запуске исправленной программы:



Просмотр результатов выполнения заданий

Щелкнув мышью на метке «Результаты (F2)», расположенной в правом верхнем углу окна задачника, или нажав клавишу **F2**, мы можем вывести на экран *окно результатов*, в котором будет перечислены все наши попытки решения задачи:

```

Dynamic2    a08/09 13:11 Ознакомительный запуск.
Dynamic2    a08/09 13:15 Выведены не все результирующие данные.
Dynamic2    a08/09 13:17 Ошибочное решение.
Dynamic2    a08/09 13:20 Задание выполнено!
Dynamic3    a08/09 13:21 Ознакомительный запуск.
Dynamic3    a08/09 13:24 Ошибочное решение.
Dynamic3    a08/09 13:28 Задание выполнено!
Dynamic5    a08/09 13:29 Ознакомительный запуск.
Dynamic5    a08/09 13:30 Не освобождена динамическая память.
Dynamic5    a08/09 13:31 Задание выполнено!
Dynamic30   a08/09 13:34 Ознакомительный запуск.
Dynamic30   a08/09 13:42 Ошибочное решение.
Dynamic30   a08/09 13:43 Задание выполнено!
Dynamic55   a08/09 13:54 Ознакомительный запуск.
Dynamic55   a08/09 13:57 Ошибочное решение.
Dynamic55   a08/09 13:58 Задание выполнено!

```

Для закрытия окна результатов достаточно нажать клавишу **Esc**. Окно результатов можно отобразить на экране и после закрытия окна задачника и возврата в среду PascalABC.NET. Для этого надо

использовать команду меню «Модули | Просмотреть результаты»,
кнопку  или клавиатурную комбинацию **Shift+Ctrl+R**.

Задания на обработку деревьев

Пример 1. Анализ бинарного дерева

В заданиях группы Tree, как и в заданиях группы Dynamic, мы встречаемся с двумя новыми видами данных: это *древовидные динамические структуры*, реализованные в виде наборов связанных друг с другом записей типа `TNode`, и *указатели* типа `PNode` на записи `TNode`: `PNode = ^TNode`. [Типы TNode и PNode](#) не являются стандартными типами языка Паскаль; они определены в задачнике Programming Taskbook.

Особенности, связанные с использованием новых типов данных, рассмотрим на примере задания Tree2.

Tree2°. Дан адрес P_1 записи типа `TNode` — корня дерева. Эта запись связана полями `Left` и `Right` с другими записями того же типа (дочерними вершинами), они, в свою очередь, — со своими дочерними вершинами, и так далее до записей, поля `Left` и `Right` которых равны `nil` (у некоторых вершин может быть равно `nil` одно из полей `Left` или `Right`). Вывести количество вершин дерева.

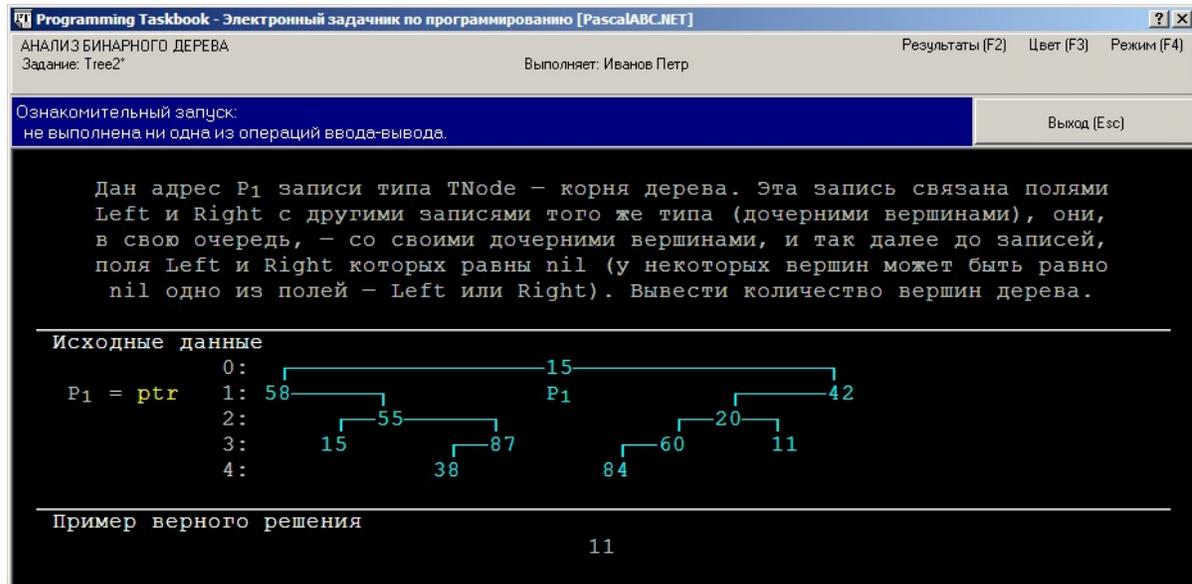
Создание программы-заготовки и знакомство с заданием

Напомним, что программу-заготовку для решения этого задания можно создать с помощью команды меню «Модули | Создать шаблон программы», кнопки  или клавиатурной комбинации **Shift+Ctrl+L**. Приведем текст созданной заготовки:

```
uses PT4;

begin
  Task('Tree2');
end.
```

После запуска программы на экране появится [окно задачника](#):



Это окно содержит в качестве исходных и результирующих данных новые элементы: бинарные деревья и указатели.

Начнем с описания того, как отображается на экране *дерево*. Для его вывода используется несколько экранных строк. На каждой строке изображаются вершины дерева, находящиеся на определенном уровне (номер уровня указывается слева от изображения дерева). Для каждой вершины выводится ее значение, т. е. значение поля Data соответствующей записи типа TNode. Любая вершина соединяется линиями со своими дочерними вершинами, расположенными на следующем уровне дерева; левая дочерняя вершина изображается слева от родительской вершины, а правая — справа. Отсутствие у вершины одной или обеих дочерних вершин означает, что ее поля `Left` и/или `Right` равны `nil`.

Рассмотрим в качестве примера дерево, приведенное на рисунке. Корень этого дерева имеет значение 15, левая дочерняя вершина корня равна 58, правая дочерняя вершина равна 42, глубина дерева равна 4. Все листья дерева находятся на уровнях 3 и 4; листья на уровне 3 имеют значения 15 и 11, листья на уровне 4 — значения 38 и 84. Некоторые из внутренних вершин дерева имеют по две дочерние вершины (это корень и вершины со значениями 55 и 20), некоторые по одной: левой (вершины 42, 87 и 60) или правой (вершина 58).

Поскольку это дерево указано в разделе исходных данных,

следовательно, после инициализации задания оно уже существует и размещается в некоторой области динамической памяти. Для доступа к данным, размещенным в динамической памяти, необходимо знать их адрес, поэтому в любом задании на обработку деревьев в набор исходных данных входят *указатели*, содержащие адреса каких-либо вершин этих деревьев (как правило, указывается адрес корня дерева).

Работа с исходными и результирующими данными типа указателя подробно обсуждается в разделе, посвященном [линейным динамическим структурам](#).

Решение задачи

В задании Tree2 не требуется ни создавать, ни преобразовывать исходное дерево; его необходимо лишь проанализировать, а именно определить количество его вершин.

Для выполнения этого задания, как и для подавляющего большинства других заданий на обработку деревьев, следует воспользоваться вспомогательной *рекурсивной* подпрограммой (функцией или процедурой). Рекурсивная природа алгоритмов, связанных с обработкой деревьев (в частности, бинарных деревьев), объясняется тем, что сами определения деревьев общего вида и бинарных деревьев являются рекурсивными. Так, дать словесное описание функции `NodeCount(P)`, подсчитывающей число вершин дерева с корнем, с которым связан указатель `P`, можно следующим образом: если указатель `P` равен `nil`, то следует вернуть значение `0`; в противном случае следует вернуть значение `1 + NodeCount(P.Left) + NodeCount(P.Right)` (в этом выражении первое слагаемое соответствует корню дерева, второе — его левому поддереву, а третье — его правому поддереву; при этом не требуется проверять, что указанные поддеревья существуют, так как при их отсутствии соответствующее слагаемое просто будет равно нулю).

Таким образом, решение задачи будет иметь следующий вид:

```
uses PT4;  
function NodeCount(P: PNode): integer;
```

```
begin
  if P = nil then
    result := 0
  else
    result := 1 + NodeCount(P^.Left) + NodeCount(P^.Right);
  end;
var P1: PNode;
begin
  Task('Tree2');
  read(P1);
  write(NodeCount(P1));
end.
```

Цепочка рекурсивных вызовов функции `NodeCount` завершается при достижении терминальной вершины (листа), у которой поля `Left` и `Right` равны `nil`. Благодаря наличию функции `NodeCount`, раздел операторов программы является очень кратким: в нем считывается адрес `P1` корня исходного дерева, после чего вызывается функция `NodeCount(P1)`, возвращаемое значение которой сразу выводится процедурой `write`.

Запустив эту программу пять раз, мы получим сообщение «*Задание выполнено!*».

Пример 2. Бинарные деревья с обратной связью

Рассмотренная выше реализация бинарных деревьев позволяет легко переходить от родительских вершин к их дочерним вершинам, но не допускает обратного перехода. В то же время, для некоторых задач, связанных с обработкой деревьев, возможность обратного перехода от потомков к их предку позволяет получить более простое решение. Ясно, что для обеспечения возможности обратного перехода каждую вершину дерева надо снабдить еще одним полем связи, в котором должна храниться ссылка на ее родительскую вершину. Это поле связи естественно назвать `Parent`. Поскольку корень дерева предка не имеет, его поле `Parent` должно быть равно `nil`.

Деревья, вершины которых содержат информацию о своих родителях, будем называть *деревьями с обратной связью*. Особенности работы с подобными деревьями рассмотрим на примере задания `Tree49`.

`Tree49`. Дан указатель P_1 на корень дерева, вершинами которого являются записи типа `TNode`, связанные между собой с помощью полей `Left` и `Right`. Используя поле `Parent` записи `TNode`, преобразовать исходное дерево в *дерево с обратной связью*, в котором каждая вершина связана не только со своими дочерними вершинами (полями `Left` и `Right`), но и с родительской вершиной (полем `Parent`). Поле `Parent` корня дерева положить равным `nil`.

Запустив программу-заготовку, созданную для задания `Tree49`, мы увидим в области исходных данных изображение «обычного» бинарного дерева, в то время как в области результатов будет изображено дерево с обратной связью, вершины которого связаны не одинарными, а двойными линиями.

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

БИНАРНЫЕ ДЕРЕВЬЯ С ОБРАТНОЙ СВЯЗЬЮ
 Выполняет: Иванов Петр
 Результаты (F2) Цвет (F3) Режим (F4)

Задание: Tree49*

Ознакомительный запуск:
 не выполнена ни одна из операций ввода-вывода. Выход (Esc)

Дан указатель P_1 на корень дерева, вершинами которого являются записи типа TNode, связанные между собой с помощью полей Left и Right. Используя поле Parent записи TNode, преобразовать исходное дерево в **дерево с обратной связью**, в котором каждая вершина связана не только со своими дочерними вершинами (полями Left и Right), но и с родительской вершиной (полем Parent). Поле Parent корня дерева положить равным nil.

Исходные данные

```

0:
1: 46
2:   11
3:  43  47
4:    62

```

```

0:
1: 91
2:  63  81
3:  92  96
4:  19  17

```

$P_1 = ptr$

Пример верного решения

```

0:
1: 46
2:   11
3:  43  47
4:    62

```

```

0:
1: 91
2:  63  81
3:  92  96
4:  19  17

```

Обратите также внимание на то, что в области результатов отсутствуют какие-либо данные, кроме измененного дерева. Это означает, что в программе, решающей задачу, не требуется использовать процедуры вывода; достаточно лишь преобразовать исходное дерево требуемым образом. Поскольку при таком преобразовании адрес корня дерева P_1 не изменится, задачник сможет получить доступ к этому дереву и проверить его правильность.

Для преобразования исходного дерева в дерево с обратной связью необходимо задать правильные значения для полей Parent всех вершин дерева, перебирая эти вершины с помощью подходящей рекурсивной процедуры. В эту процедуру удобно передавать в качестве параметров не только указатель P на текущую вершину, но и указатель Par на предка этой вершины:

```

uses PT4;
procedure SetParent(P, Par: PNode);
begin
  if P = nil then
    exit;
  P^.Parent := Par;
  SetParent(P^.Left, P);

```

```
    SetParent(P^.Right, P);  
end;  
var P1: PNode;  
begin  
    Task('Tree49');  
    read(P1);  
    SetParent(P1, nil);  
end.
```

При стартовом запуске рекурсивной процедуры `SetParent` в качестве второго параметра указывается `nil`.

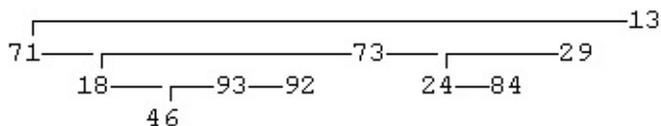
Примечание. Обозначение для двойной связи может оказаться полезным при анализе ошибочного решения. Так, если в изображении дерева с обратной связью имеется вершина, соединенная со своей родительским вершиной не двойной, а одинарной линией, значит, у этой вершины поле `Parent` содержит ошибочное значение (например, равно `nil`).

Пример 3. Деревья общего вида

С помощью связанных записей типа `TNode` можно моделировать не только бинарные деревья, но и произвольные упорядоченные деревья, вершины которых имеют любое число непосредственных потомков (будем называть такие деревья *деревьями общего вида*; для них также используется название «деревья с множественным ветвлением»). Рассмотрим задание `Tree86` — первое из заданий, связанных с деревьями общего вида, в котором описываются особенности подобных деревьев.

`Tree86`. *Дерево общего вида* (каждая вершина которого может иметь произвольное число дочерних вершин, расположенных в фиксированном порядке в направлении слева направо) реализуется с помощью набора связанных записей типа `TNode` следующим образом: для каждой внутренней вершины ее поле `Left` содержит указатель на ее первую (т. е. левую) дочернюю вершину, а поле `Right` — указатель на ее правую *сестру*, т. е. вершину, имеющую в дереве общего вида того же родителя. Поле `Right` корня дерева общего вида всегда равно `nil`, так как корень сестер не имеет. Дан указатель P_1 на корень непустого бинарного дерева. Создать дерево общего вида, соответствующее исходному бинарному дереву, и вывести указатель P_2 на его корень.

Приведем пример дерева общего вида, которое реализовано с помощью связанных записей типа `TNode` (аналогичным образом деревья общего вида изображаются в окне задачника):



Корень этого дерева (со значением 13) имеет три дочерние вершины (71, 73 и 29), причем вершина 71 не имеет потомков, вершина 73 имеет три непосредственных потомка (18, 93 и 92), а вершина 29 — два (24 и 84). На последнем уровне располагается вершина 46, являющаяся единственной дочерней вершиной вершины 93.

При ознакомительном запуске задания `Tree86` на экране появится окно, подобное следующему.

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]

ДЕРЕВЬЯ ОБЩЕГО ВИДА
Задание: Tree86*

Выполняет: Иванов Петр

Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода.

Выход (Esc)

Дерево общего вида (каждая вершина которого может иметь произвольное число дочерних вершин, расположенных в фиксированном порядке в направлении слева направо) реализуется с помощью набора связанных записей типа TNode следующим образом: для каждой внутренней вершины ее поле Left содержит указатель на ее первую (т.е. левую) дочернюю вершину, а поле Right – указатель на ее правую сестру, т.е. вершину, имеющую в дереве общего вида того же родителя. Поле Right корня дерева общего вида всегда равно nil, так как корень сестер не имеет. Дан указатель P1 на корень непустого бинарного дерева. Создать дерево общего вида, соответствующее исходному бинарному дереву, и вывести указатель P2 на его корень.

Исходные данные

```

0:
P1 = ptr 1:
2: 15
3:
4:

```

```

0:
P2 = ptr 1:
2:
3:
4:

```

Пример верного решения

```

0:
P2 = ptr 1:
2:
3:
4:

```

Обратите внимание на то, как выглядит одно и то же дерево в двух различных представлениях: вариант, соответствующий обычному бинарному дереву, приводится в разделе исходных данных, а вариант, соответствующий дереву общего вида, — в разделе результатов. При переходе от бинарного дерева к дереву общего вида часть информации о структуре бинарного дерева теряется, поскольку в случае, если некоторая вершина дерева общего вида имеет только одного непосредственного потомка, нельзя определить, каким был этот потомок в исходном бинарном дереве — левым или правым.

Напомним, что точки, обрамляющие значения вершин в разделе результатов, означают, что все эти вершины должны быть созданы программой учащегося (в отличие от вершин исходного дерева, созданных самим задачиком при инициализации задания).

При формировании нового дерева будем использовать рекурсивную функцию `CreateNode(P)`. Параметр `P` содержит указатель на вершину исходного дерева, копия которой создается при вызове функции. Возвращаемым значением функции является указатель на

созданную вершину (как обычно, если $P = nil$, то функция не выполняет никаких действий и возвращает nil). Для создания дочерних вершин выполняется рекурсивный вызов этой функции. Заметим, что цепочка дочерних вершин может быть пустой (если вершина P является листом), содержать один элемент (если вершина P имеет только одного непосредственного потомка) или два элемента. Перед формированием цепочки дочерних вершин удобно занести адреса дочерних вершин вершины P во вспомогательные переменные $P1$ и $P2$. При этом в случае, если вершина P имеет только одного потомка (неважно, левого или правого), адрес этого потомка заносится в переменную $P1$, а переменная $P2$ остается равной nil . Благодаря использованию переменных $P1$ и $P2$, фрагмент кода, отвечающий за формирование списка дочерних вершин, удастся сделать более кратким. Приведем текст программы, решающей задачу Tree86.

```

uses PT4;
function CreateNode(P: PNode): PNode;
var P1, P2: PNode;
begin
  if P = nil then
    begin
      result := nil;
      exit;
    end;
  New(result);
  result^.Data := P^.Data;
  result^.Right := nil;
  P1 := P^.Left;
  P2 := P^.Right;
  if P1 = nil then
    begin
      P1 := P2;
      P2 := nil;
    end;
  { формирование списка дочерних вершин }
  result^.Left := CreateNode(P1);
  if P1 <> nil then
    result^.Left^.Right := CreateNode(P2);
end;
var P1: PNode;
begin
  Task('Tree86');

```

```
read(P1);  
write(CreateNode(P1));  
end.
```

Примечание. Фрагмент дерева общего вида, содержащий все дочерние вершины некоторой вершины, можно рассматривать как *односвязный список*, элементы которого связаны между собой с помощью поля *Right* (у последнего элемента списка поле *Right* равно *nil*). Каждый элемент подобного списка может содержать «подсписок» своих дочерних элементов; адрес начала этого подсписка хранится в поле *Left* данного элемента. Поэтому в алгоритмах, связанных с обработкой вершин деревьев общего вида, для перебора непосредственных потомков некоторой вершины удобно использовать *цикл* (как при переборе элементов списка), в то время как для обработки каждой дочерней вершины следует, как обычно, использовать *рекурсию*.

Задания, связанные с ЕГЭ по информатике

Пример 1. Простая задача на реализацию базовых алгоритмов

Группа заданий ExamBegin посвящена базовым алгоритмическим задачам, включенным в кодификатор ЕГЭ по информатике. Процесс выполнения подобных заданий мы рассмотрим на примере одной из простых задач, связанных с нахождением максимумов и минимумов из двух, трех или четырех чисел без использования массивов и циклов.

ExamBegin2°. На вход подаются три вещественных числа; числа расположены в одной строке. Вывести вначале минимальное, а затем максимальное из них. Каждое число должно выводиться на новой строке и снабжаться комментарием: «MIN=» для минимального, «MAX=» для максимального.

Создание программы-заготовки и знакомство с заданием

Напомним, что программу-заготовку для решения этого задания можно создать с помощью команды меню «Модули | Создать шаблон программы», кнопки  или клавиатурной комбинации **Shift+Ctrl+L**. Приведем текст созданной заготовки:

```
uses PT4Exam;  
  
begin  
    Task('ExamBegin2');  
  
end.
```

После запуска программы на экране появится [окно задачника](#):

```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
УСЛОВНЫЕ ОПЕРАТОРЫ И ЦИКЛЫ
Задание: ExamBegin2*
Выполняет: Иванов Петр
Результаты (F2)  Цвет (F3)  Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций вывода.

На вход подаются три вещественных числа; числа расположены в одной строке.
Вывести вначале минимальное, а затем максимальное из них. Каждое число
должно выводиться на новой строке и снабжаться комментарием:
«MIN=» для минимального, «MAX=» для максимального.

Исходные данные
1: '-28.21 -161.53 34.28'

Пример верного решения
1: 'MIN=-161.53'
'MAX=34.28'
```

Обсудим особенности программы-заготовки и окна задачника.

В программе-заготовке вместо модуля **PT4** подключается модуль **PT4Exam**, специально предназначенный для использования при выполнении заданий групп Exam. Данный модуль содержит реализацию единственной процедуры **Task**, инициализирующей задание. Никакие дополнительные процедуры, связанные с вводом-выводом, в него не включены. Это обусловлено тем, что ввод-вывод при выполнении заданий групп Exam надо выполнять, используя *стандартные процедуры языка Pascal*.

Основной особенностью окна задачника является то, что в разделе исходных данных отсутствуют данные, выделенные желтым цветом (напомним, что желтый цвет используется для выделения данных, которые необходимо вводить с помощью специальных процедур ввода задачника). Вместо этого в окне отображается строка бирюзового цвета, содержащая числовые данные. Вид строки подчеркивает то обстоятельство, что вводить данные требуется не с помощью специальных процедур ввода, имеющих в задачнике, а с помощью стандартных процедур языка Pascal. Отметим, что бирюзовый цвет используется в окне задачника для отображения «внешних» данных (содержащихся в файлах или динамических структурах), доступ к которым должен осуществляться с помощью стандартных средств используемого языка программирования.

Пример верного решения выделяется серым цветом (в отличие от «настоящих» результатов, выведенных программой учащегося, которые, как и входные данные, выделяются бирюзовым цветом),

однако *представление* выходных данных совпадает с представлением входных: это набор строк, содержащих числовые данные (дополненные комментариями). Вид данных в разделе результатов показывает, что для их вывода, как и для ввода исходных данных, необходимо использовать стандартные процедуры языка Pascal.

Примечание. Если вы уже выполняли задания, связанные с обработкой файлов, то можете заметить, что отображение данных в заданиях групп Exam в точности соответствует способу отображения содержимого *текстовых файлов*. Это совпадение не случайно. На самом деле во всех заданиях групп Exam все исходные данные хранятся в специальном входном текстовом файле, а все результаты должны записываться в специальный выходной текстовый файл. Однако при этом не требуется выполнять особых действий, связанных с определением имен этих файлов, связыванием файлов с файловыми переменными, открытием и закрытием файлов (все эти действия выполняются задачиком автоматически). Для программы, выполняющей задание, эти файлы играют роль *стандартных потоков ввода-вывода*, поэтому для доступа к ним достаточно использовать обычные процедуры ввода-вывода языка Pascal.

Ввод исходных данных и их обработка

Приступим к выполнению задания. В данном случае следует использовать алгоритм, не требующий применения массивов, поэтому опишем три простые переменные вещественного типа и введем в них исходные данные:

```
uses PT4Exam;
var
  a, b, c: real;
begin
  Task('ExamBegin2');
  read(a, b, c);
end.
```

Мы воспользовались стандартной процедурой ввода `read`, введя все три исходных числа за один ее вызов. Этого же результата мы могли

бы добиться и с помощью процедуры `readln`:

```
readln(a, b, c);
```

Заметим, что использование *отдельных* процедур `readln` для ввода каждого числа приведет к ошибочному результату:

```
readln(a);  
readln(b);  
readln(c);
```

В этом случае уже после ввода первого числа произойдет автоматический переход на следующую строку с исходными данными. Поэтому оставшиеся в первой строке числа будут пропущены, а поскольку во второй строке «ничего нет» (входной поток состоит из единственной строки), будет выведено сообщение об ошибке «*Input string was not in a correct format*» («*Входная строка имела неверный формат*»).

Приведенный пример показывает, что при организации ввода данных в заданиях групп Exam необходимо учитывать особенности стандартных процедур `read` и `readln`.

При запуске приведенного выше варианта программы вид окна не изменится, поскольку мы не вывели никаких данных. В заданиях групп Exam запуск программы считается ознакомительным до тех пор, пока программа не выведет хотя бы один элемент результирующих данных. Кроме того, задачник не контролирует, каким образом программа читает исходные данные (например, мы могли бы ввести всю исходную строку в переменную типа `string`, а затем «разобрать» эту строку, выделив из нее три числа и преобразовав их к типу `real`). Отмеченные особенности характерны именно для заданий групп Exam, в которых для ввода данных не используются специальные средства задачника.

Реализуем алгоритм нахождения минимального и максимального элемента. Для этого опишем еще две переменные `min` и `max` типа `real` и добавим в конец программы следующие операторы:

```
if a < b then  
begin  
  min := a;
```

```
    max := b;
end
else
begin
    min := b;
    max := a;
end;
if c < min then
    min := c
else
    if c > max then
        max := c;
```

Таким образом, для одновременного нахождения минимального и максимального из трех чисел требуется не более трех операций сравнения и не более трех операций присваивания.

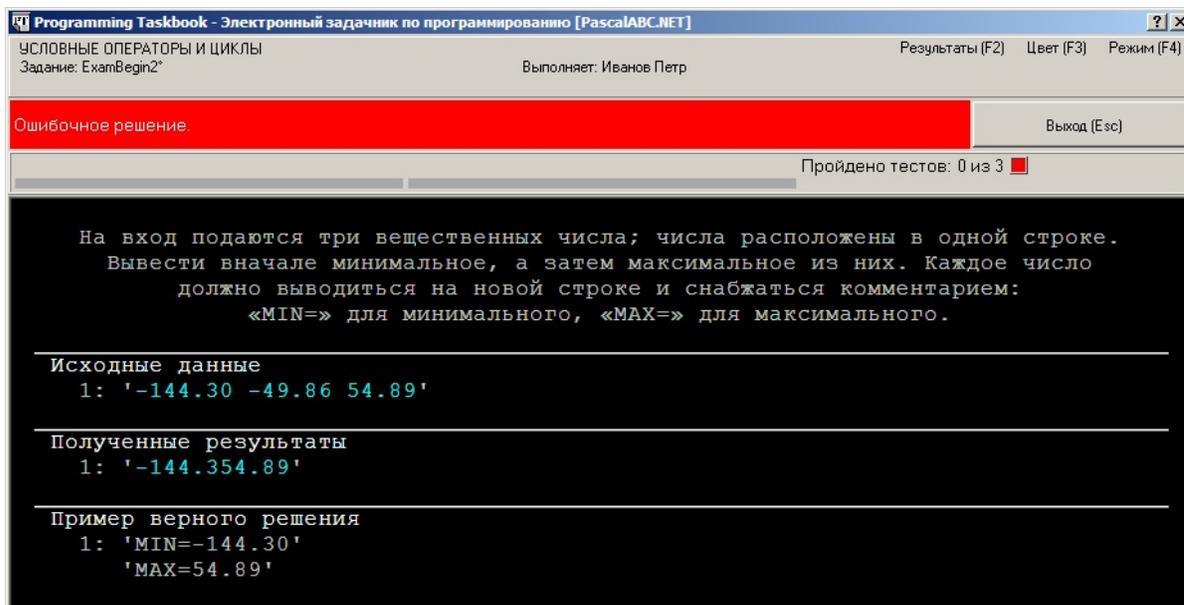
Вывод результатов и их форматирование

Осталось вывести полученные результаты. Вывод, как и ввод, следует выполнять с помощью стандартных процедур языка Pascal, учитывая их особенности.

Вначале, в качестве примера, организуем вывод, не соответствующий условиям задачи. Для этого добавим в конец программы следующий оператор:

```
writeln(min, max);
```

Приведем вид окна задачника при запуске полученной программы:



Мы видим, что минимум и максимум найдены правильно, однако выведены не в том формате, который требовался (требуемый формат приводится в разделе с примером верного решения). Мы допустили при выводе три ошибки: во-первых, вывели оба числа в одной строке (при этом они «слились», поскольку мы не предусмотрели вывод пробела-разделителя), во-вторых, не снабдили числа комментариями и, в-третьих, не настроили их отображение в формате с двумя дробными знаками (первое число было выведено с единственным знаком после точки).

Примечание. Следует обратить внимание на панель индикаторов, которая отображается между информационной панелью и разделом с формулировкой задания в случае, если запуск программы не является ни демонстрационным, ни ознакомительным. Обычно на этой панели выводятся три индикатора: первый указывает количество введенных исходных данных, второй — количество выведенных результатов, а третий — количество успешно пройденных тестовых испытаний. При выполнении заданий, связанных с ЕГЭ, первые два индикатора являются неактивными, поскольку, как было отмечено выше, для получения исходных данных и записи результатов не используются средства задачника, и поэтому он не в состоянии проконтролировать каждую операцию ввода-вывода.

Для исправления первых двух ошибок достаточно изменить вывод

следующим образом:

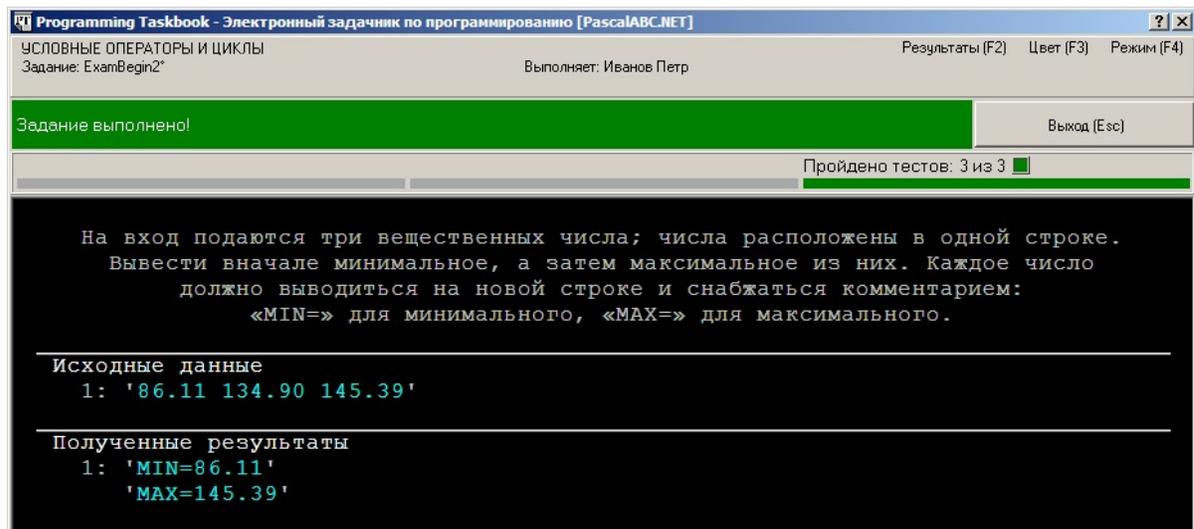
```
writeln('MIN=', min);  
writeln('MAX=', max);
```

Однако в этом случае числа по-прежнему могут содержать неверное число дробных знаков. Для исправления этой последней ошибки проще всего использовать *атрибуты форматирования*, начинающиеся с символа «:» (двоеточие):

```
writeln('MIN=', min:0:2);  
writeln('MAX=', max:0:2);
```

Первый атрибут определяет *ширину поля вывода* (если ширина равна 0, то используется минимально необходимое поле вывода). Наличие второго атрибута (допустимого только для вещественных чисел) означает, что число надо вывести в формате с фиксированной точкой, причем его значение равно количеству дробных знаков.

При запуске исправленной программы будет выведено сообщение о верном решении, а после трех запусков — сообщение о том, что задание выполнено:



В случае успешного прохождения тестового испытания в окно задачника не включается раздел с примером верного решения, поскольку данные этого раздела совпадают с результатами, полученными программой.

Примечание. В системе PascalABC.NET, благодаря специальному механизму перенаправления данных, стандартные процедуры `read-write` можно использовать и при выполнении заданий из других групп, однако важно учитывать ряд особенностей использования этих процедур в заданиях групп Exam. Во-первых, только в заданиях групп Exam будут проявляться различия в использовании процедур `read-write` и `readln-writeln` (в остальных группах для ввода-вывода можно использовать как вариант процедуры с суффиксом «ln», так и вариант без этого суффикса). Во-вторых, только в заданиях групп Exam можно при необходимости *использовать атрибуты форматирования* при выводе результатов, а также *выводить дополнительные комментарии*, если этого требует условие задачи. В-третьих, только в заданиях групп Exam можно вводить и выводить элементы данных несколькими способами, с использованием переменных различных типов; например, исходную строку можно либо сразу прочесть в строковую переменную, либо вводить посимвольно в цикле (в других группах заданий проводится более строгая проверка на соответствие типа переменной типу элемента исходных или результирующих данных).

Пример 2. Ввод и вывод массивов

Рассмотрим еще одно задание группы ExamBegin, особенностью которого является вывод в качестве результата элементов двумерного массива.

ExamBegin28°. На вход в первой строке подаются два целых положительных числа M и N , во второй строке — вещественное число D , а в третьей строке — набор из M вещественных чисел. Сформировать и вывести двумерный вещественный массив размера $M \times N$, у которого первый столбец совпадает с исходным набором чисел, а элементы каждого следующего столбца равны сумме соответствующего элемента предыдущего столбца и числа D (в результате каждая строка массива будет содержать элементы *арифметической прогрессии*). Каждую строку элементов массива выводить на новой экранной строке, для каждого числа отводить 7 экранных позиций.

При запуске программы-заготовки, созданной для этого задания, окно задачника примет следующий вид:

```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ФОРМИРОВАНИЕ МАССИВОВ
Задание: ExamBegin28°
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций вывода. Выход (Esc)

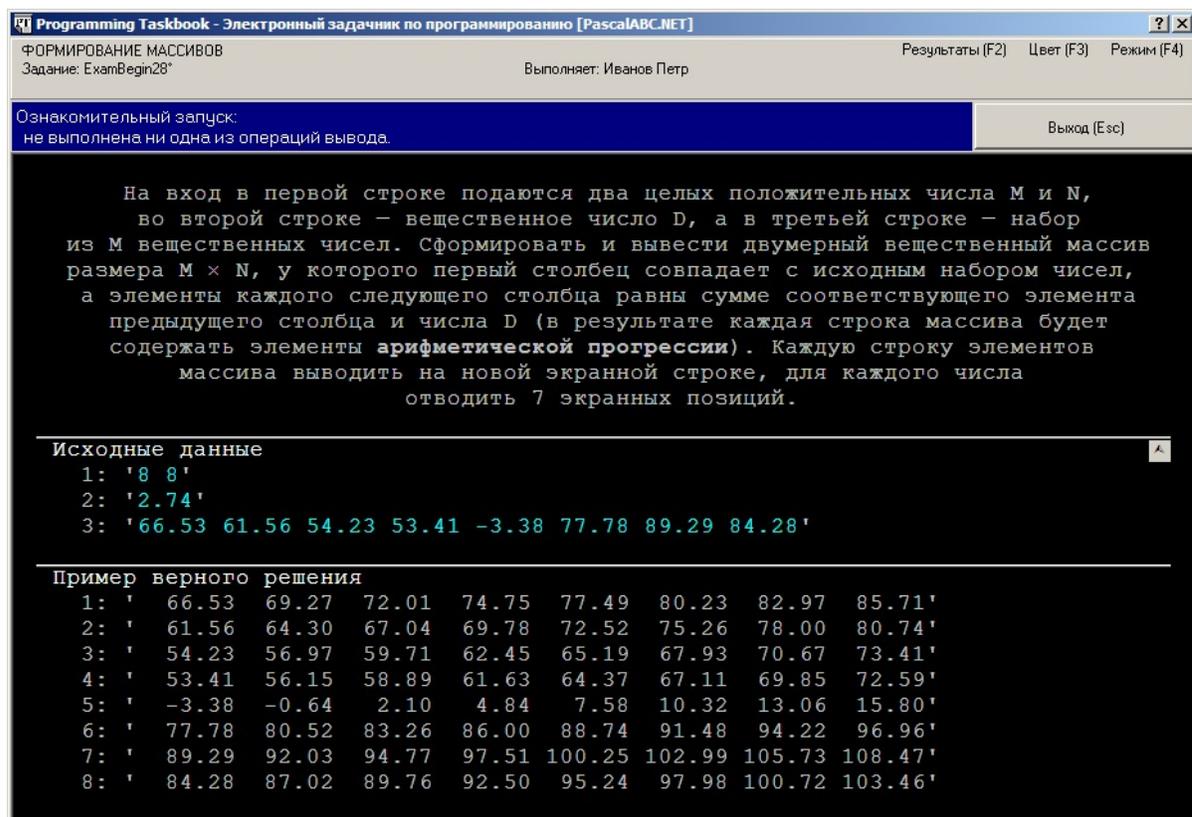
На вход в первой строке подаются два целых положительных числа M и N,
во второй строке - вещественное число D, а в третьей строке - набор
из M вещественных чисел. Сформировать и вывести двумерный вещественный массив
размера M x N, у которого первый столбец совпадает с исходным набором чисел,
а элементы каждого следующего столбца равны сумме соответствующего элемента
предыдущего столбца и числа D (в результате каждая строка массива будет
содержать элементы арифметической прогрессии). Каждую строку элементов
массива выводить на новой экранной строке, для каждого числа
отводить 7 экранных позиций.

Исходные данные
1: '8 8'
   '2.74'
   '66.53 61.56 54.23 53.41 -3.38 77.78 89.29 84.28'

Пример верного решения
1: ' 66.53 69.27 72.01 74.75 77.49 80.23 82.97 85.71'
   ' 61.56 64.30 67.04 69.78 72.52 75.26 78.00 80.74'
   ' 54.23 56.97 59.71 62.45 65.19 67.93 70.67 73.41'
   ' 53.41 56.15 58.89 61.63 64.37 67.11 69.85 72.59'
   ' -3.38 -0.64 2.10 4.84 7.58 10.32 13.06 15.80'
```

Анализируя исходные данные, можно заметить, что полученная

матрица должна иметь 8 строк, тогда как на экране отображаются только первые пять. Это связано с тем, что по умолчанию используется режим «свернутого» отображения данных, при котором на экране выводится только несколько начальных строк. Признаком того, что имеются данные, не выведенные на экране, является кнопка, которая отображается в правом верхнем углу раздела исходных данных (на этой кнопке изображается стилизованная стрелка, направленная вниз). Для вывода всех данных достаточно нажать эту кнопку; можно также нажать клавишу **Ins** или выполнить щелчок мышью в любом месте раздела с данными задания (кроме раздела, содержащего формулировку). Если выполнить эти действия для нашего окна, то оно изменится следующим образом:



Повторный щелчок мышью, нажатие клавиши **Ins** или кнопки (на которой в данной ситуации будет отображаться стрелка, направленная вверх — см. рисунок) восстанавливает «сокращенное» отображение данных. Заметим, что в режиме «сокращенного» отображения нумеруется только первая строка данных, а в режиме полного отображения нумерацией снабжаются все строки.

Дополнительные возможности, связанные с просмотром данных большого размера, будут описаны далее, в пункте, посвященном задачам повышенной сложности.

Если закрыть окно задачника, находясь в режиме отображения всех данных, то при последующих запусках программы окно будет сразу отображаться в этом режиме.

Приведем вариант правильного решения данной задачи (в этом варианте учитывается, что результирующий двумерный массив имеет не более 10 строк и столбцов; соответствующее условие приведено в преамбуле к описанию группы ExamBegin):

```
uses PT4Exam;
var
  m, n, i, j: integer;
  d: real;
  a: array[1..10, 1..10] of real;
begin
  Task('ExamBegin28');
  readln(m, n, d);
  for i := 1 to m do
    read(a[i, 1]);
  for j := 2 to n do
    for i := 1 to m do
      a[i, j] := a[i, j - 1] + d;
  for i := 1 to m do
  begin
    for j := 1 to n do
      write(a[i, j]:7:2);
    writeln;
  end;
end.
```

В приведенном решении следует обратить особое внимание на организацию ввода-вывода. Укажем две особенности, связанные с вводом. Во-первых, несмотря на то что по условию число d находится во второй строке, мы смогли включить его в один список с предшествующими числами m и n (поскольку при чтении числовых данных переход на новую строку выполняется автоматически). Во-вторых, при чтении элементов исходного набора *необходимо* использовать вариант процедуры `read` без суффикса «ln», чтобы не пропустить оставшиеся в этой строке числа.

При выводе полученного двумерного массива необходимо обеспечить его правильное *форматирование*: каждый элемент должен выводиться на семи экранных позициях с двумя дробными знаками и, кроме того, каждая строка массива должна выводиться на новой экранной строке. Это достигается за счет использования соответствующих атрибутов форматирования и явного перехода на новую строку с помощью процедуры `writeln` без параметров.

Примечание. Заметим, что в заданиях группы Matrix, также посвященной обработке двумерных массивов, специальные действия по форматированию полученных массивов выполнять не требуется, так как задачник автоматически форматирует все полученные результаты. Таким образом, задания группы ExamBegin «более приближены» к реальной экзаменационной ситуации, в которой программа учащегося должна не только обрабатывать исходные данные, но и обеспечивать наглядное отображение результатов.

Пример 3. Обработка сложных наборов данных

Группа ExamTaskC содержит 100 типовых заданий, аналогичных заданиям, которые предлагаются на ЕГЭ по информатике в качестве задач повышенной сложности (задача C4). Основную часть данной группы составляют задания на обработку сложных наборов данных (записей) с элементами-полями различных типов. В подобных заданиях требуется правильно выбрать способ хранения данных и организовать их эффективную обработку; при этом обычно требуется применить *несколько* базовых алгоритмов, например, алгоритм суммирования или нахождения минимума/максимума и алгоритм поиска нужного элемента или сортировки набора данных по требуемому ключу. В группу ExamTaskC включены также задания повышенной сложности на обработку текстовых данных (подобные задания содержатся в завершающем разделе данной группы).

Следует заметить, что возможность автоматической генерации больших наборов исходных данных, предоставляемая задачиком Programming Taskbook, позволяет существенно ускорить тестирование учебных программ и сделать его более надежным, что, в свою очередь, повышает эффективность изучения типовых приемов решения задач группы C.

В заданиях группы ExamTaskC ввод и вывод имеет те же особенности, что и в заданиях группы ExamBegin.

Рассмотрим следующее задание.

ExamTaskC25°. На вход подаются сведения об абитуриентах. В первой строке указывается количество абитуриентов N , каждая из последующих N строк имеет формат

<Номер школы> <Год поступления> <Фамилия>

Номер школы содержит не более двух цифр, годы лежат в диапазоне от 1990 до 2010. Для каждого года, присутствующего в исходных данных, вывести общее число абитуриентов, поступивших в этом году (вначале выводить год, затем число абитуриентов).

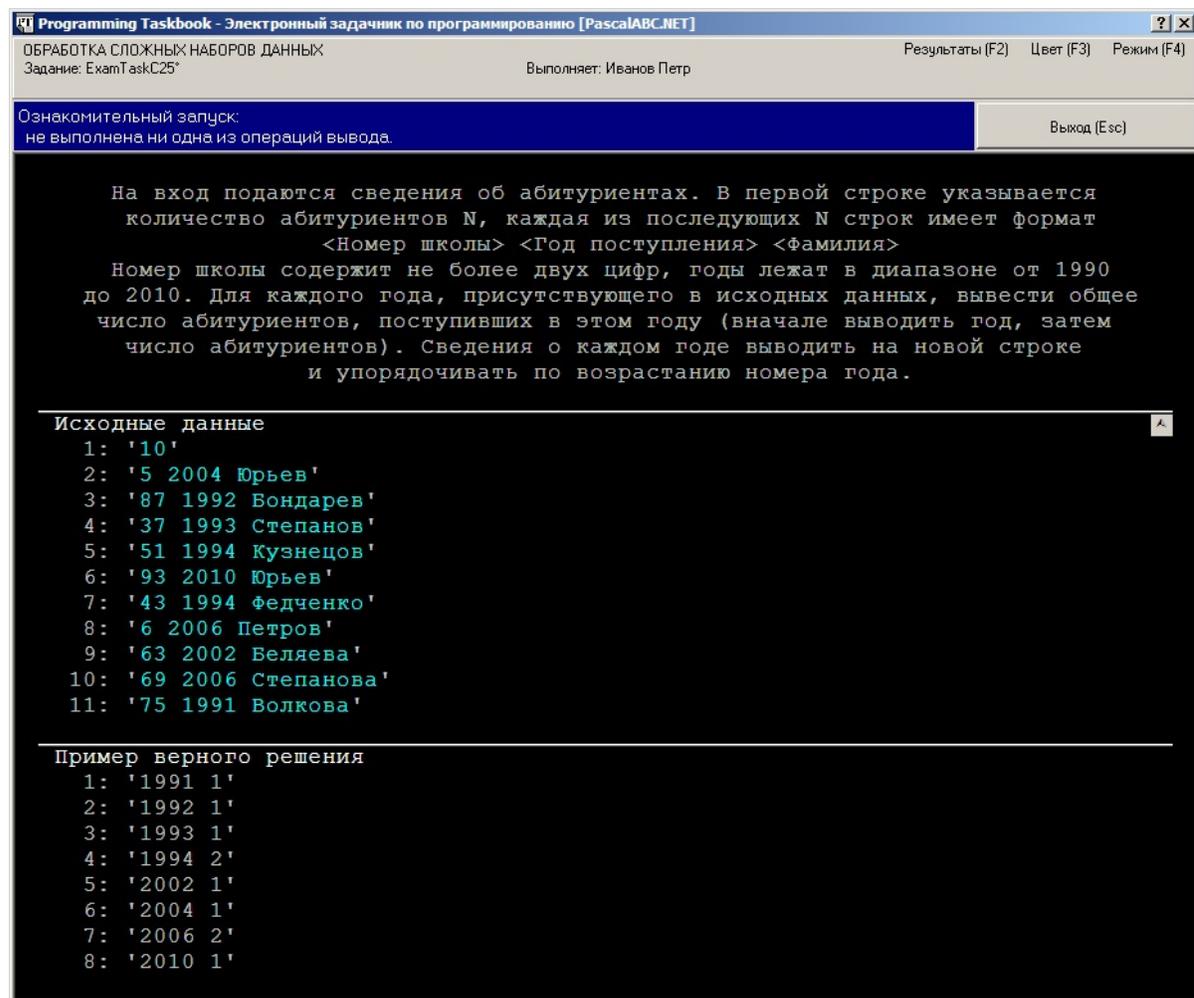
Сведения о каждом годе выводить на новой строке и упорядочивать по возрастанию номера года.

Программа-заготовка, созданная для этого задания, подобно

заготовкам для заданий группы ExamBegin, будет использовать специальный модуль PT4Exam:

```
uses PT4Exam;  
  
begin  
  Task('ExamTaskC25');  
  
end.
```

При запуске этой программы на экране появится окно задачника, содержащее следующие данные:



Окно будет иметь такой вид, если при его предшествующем закрытии оно находилось в режиме отображения всех данных. Для отображения всех данных на экране может потребоваться увеличить высоту окна; для этого достаточно зацепить мышью заголовок окна и переместить его вверх (для перемещения заголовка окна задачника

вверх и вниз можно также воспользоваться клавиатурными комбинациями **Ctrl+Up** и **Ctrl+Down**).

При первом тестовом испытании программы ей будет предложен для обработки набор данных не слишком большого размера (порядка 10–20 элементов).

Вначале следует определиться со структурами данных, которые будут использоваться в программе. Поскольку требуется найти одну характеристику для каждого года, а число лет невелико, можно использовать числовой массив `year`, каждый элемент которого соответствует определенному году. Так как в языке Pascal можно использовать произвольные границы индексов, удобно в качестве диапазона индексов указать диапазон лет, который требуется проанализировать: `1990..2010`. В начале программы выполним инициализацию элементов массива, положив их значения равными 0 (заметим, что если после обработки исходных данных некоторые элементы массива `year` останутся нулевыми, то это будет означать, что соответствующие годы не были представлены в наборе исходных данных, и выводить информацию о них не следует).

После инициализации массива следует прочесть информацию о количестве абитуриентов и организовать цикл, в котором будут обрабатываться данные о каждом абитуриенте и соответствующим образом корректироваться элементы массива `year`. В дальнейшем сведения об уже обработанном абитуриенте нам не будут нужны, поэтому сохранять их в специальном наборе данных (например, массиве) не требуется. Заметим также, что фамилия абитуриента для решения задачи не требуется, поэтому после чтения двух числовых данных можно сразу переходить к новой строке, пропуская строковый элемент данных (фамилию). В задаче не нужно использовать и номера школ, однако их придется считывать, так как только после номера школы указывается интересующий нас год поступления абитуриента.

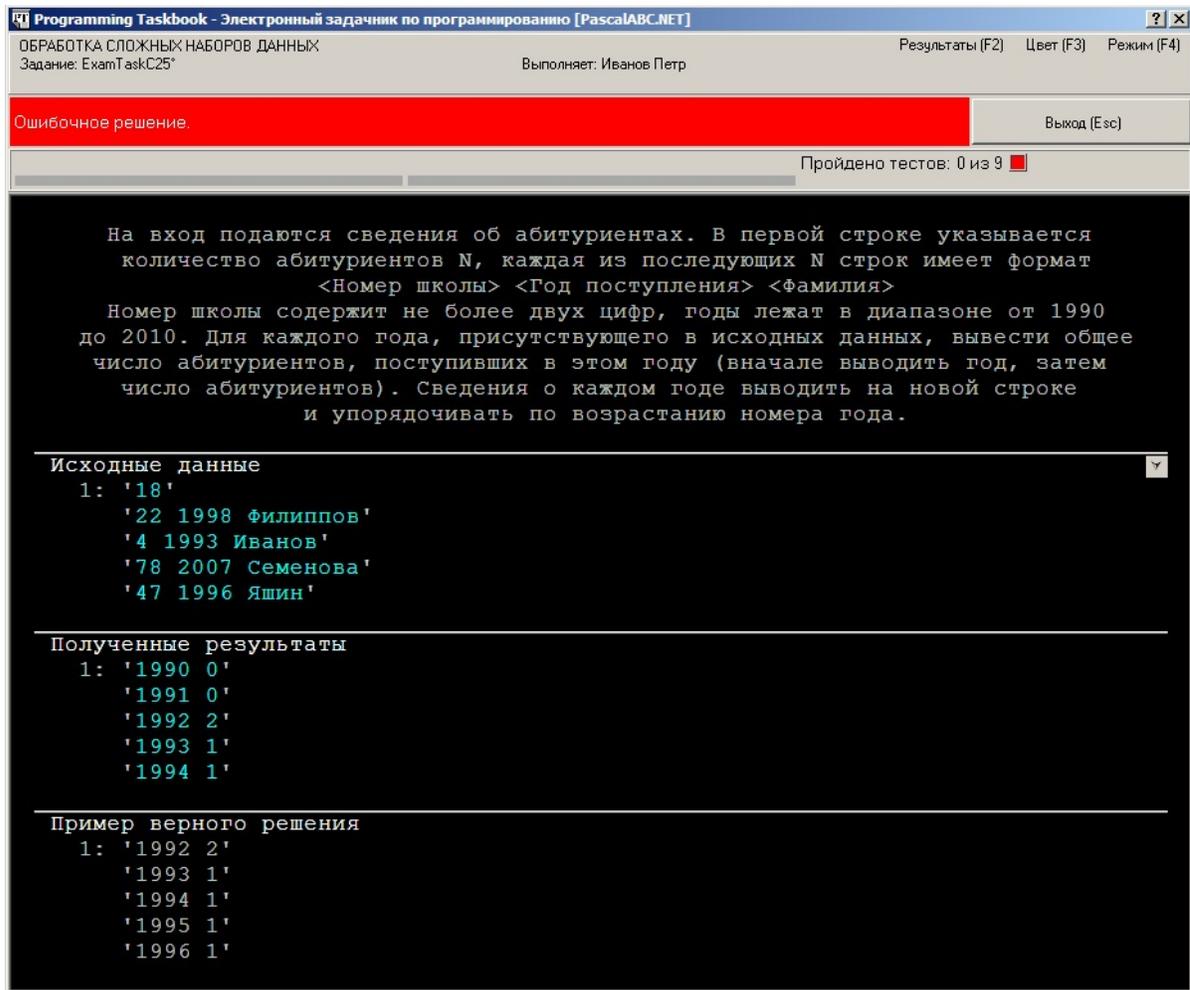
Когда данные обо всех абитуриентах будут обработаны, в массиве `year` будет содержаться вся необходимая информация, которую останется вывести в формате, указанном в условии задачи.

Приведем первый вариант решения (этот вариант содержит одну

ошибку):

```
uses PT4Exam;
var
  n, i, k, m: integer;
  year: array[1990..2010] of integer;
begin
  Task('ExamTaskC25');
  for i := 1990 to 2010 do
    year[i] := 0;
  readln(n);
  for i := 1 to n do
    begin
      readln(k, m); { k - номер школы, m - год поступления }
      Inc(year[m]);
    end;
  for i := 1990 to 2010 do
    writeln(i, ' ', year[i]);
  end.
```

Ошибка связана с тем, что на экран выводится информация о годах, отсутствующих в наборе исходных данных. Поэтому она сразу будет выявлена при обработке наборов данных небольшого размера, предлагаемых программе при первом тестовом запуске (для большей наглядности приведем окно задачника в режиме сокращенного отображения данных, при котором выводятся только пять первых элементов из каждого набора данных):



Для исправления ошибки достаточно добавить в последний цикл условный оператор:

```
for i := 1990 to 2010 do  
  if year[i] > 0 then  
    writeln(i, ' ', year[i]);
```

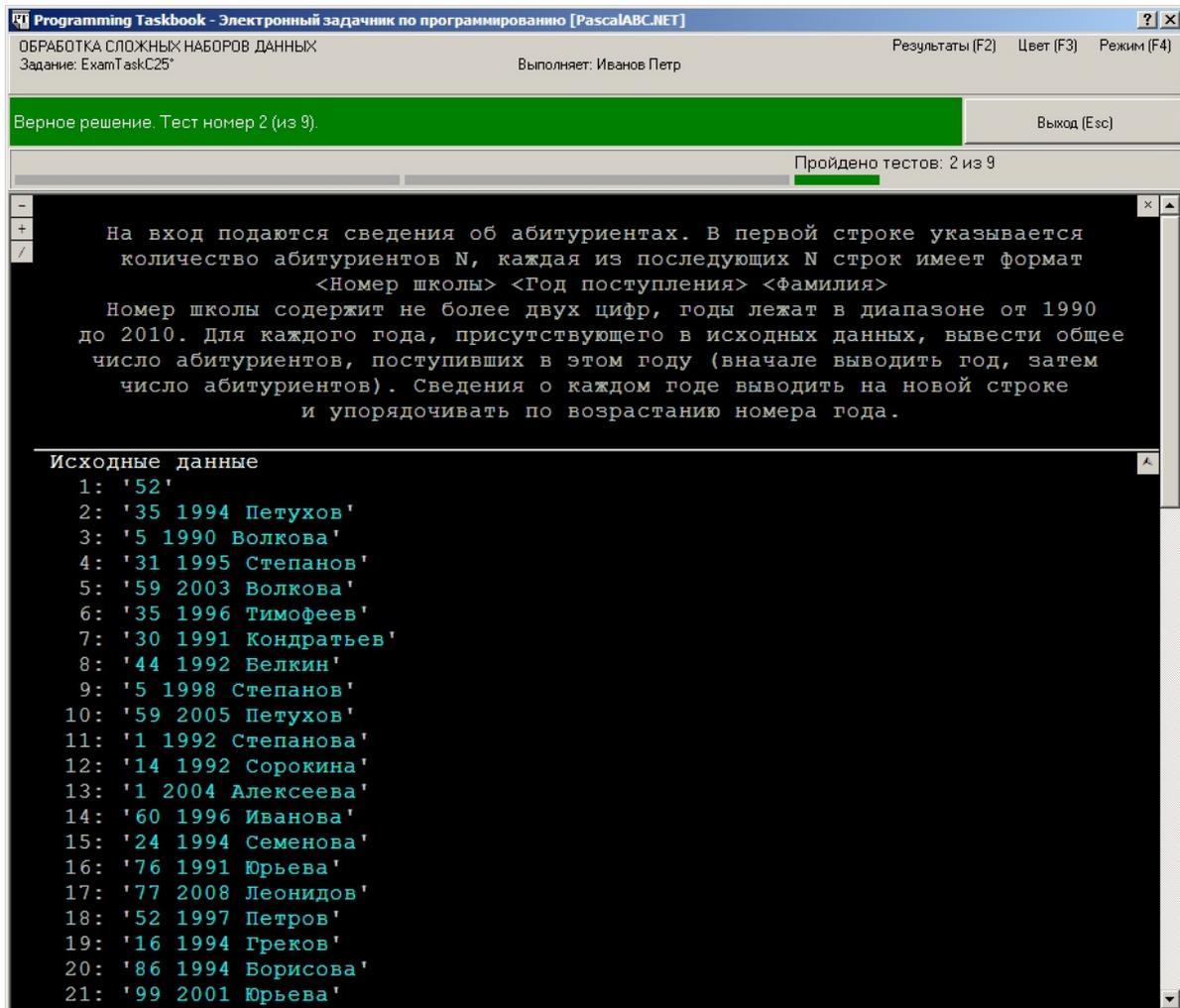
Теперь все 9 тестовых испытаний программы, требуемых для того, чтобы решение было зачтено как выполненное, будут пройдены успешно.

Завершая рассмотрение этого задания, опишем некоторые дополнительные возможности, связанные с просмотром больших наборов данных.

Начиная со второго испытания, программе может быть предложен для обработки набор исходных данных большего размера (порядка 50–100 элементов). При этом уже не удастся отобразить на экране

все данные, связанные с заданием. В подобной ситуации у правой границы окна задачника появится *полоса прокрутки*, позволяющая перемещаться к той части данных, которая первоначально не отображается на экране. Прокрутку данных можно выполнять не только с помощью полосы прокрутки, но и используя клавиши со стрелками, **PgUp**, **PgDn**, **Home**, **End**, а также колесико мыши.

Помимо стандартных действий по прокрутке данных, в окне задачника предусмотрены возможности «интеллектуальной» прокрутки, позволяющие быстро перейти к началу каждого раздела задания, а также сравнить соответствующие фрагменты полученных результатов и примера верного решения. Для циклического перебора разделов сверху вниз предназначена клавиша **[+]** (а также комбинация **Ctrl+PgDn**), для циклического перебора разделов снизу вверх — клавиша **[-]** (а также комбинация **Ctrl+PgUp**). Для быстрого переключения между соответствующими фрагментами разделов с результатами и с примером верного решения предназначена клавиша **[/]** (а также комбинация **Ctrl+Tab**). Все эти действия можно выполнить и с помощью мыши; для этого предусмотрены кнопки в левом верхнем углу прокручиваемой области окна, отведенной под отображение разделов задания (эти кнопки отображаются на экране, если размер данных, связанных с заданием, превышает размеры окна). Приведем вид окна задачника с полосой прокрутки и дополнительными кнопками:



Обозначения на кнопках совпадают с клавишами, выполняющими те же действия; при наведении мышью на кнопку рядом с ней появляется всплывающая подсказка.

Кроме трех кнопок, связанных с «интеллектуальной» прокруткой, в приведенном на рисунке окне отображаются еще две дополнительные кнопки. Первая из них располагается в правом верхнем углу раздела с формулировкой и позволяет временно скрыть (а в дальнейшем опять отобразить) раздел с формулировкой (эти же действия можно выполнить с помощью клавиши **Del** или щелчка мышью на разделе с формулировкой). Вторая дополнительная кнопка расположена в правом верхнем углу раздела с исходными данными. Как уже отмечалось ранее, эта кнопка позволяет переключаться между полным и сокращенным отображением наборов данных. Напомним, что изображение на этой

кнопке показывает текущий режим отображения данных. Например, если на кнопке изображена стилизованная стрелка, направленная вверх (как на приведенном выше рисунке), значит, в данный момент в окне отображаются все данные, а нажатие на эту кнопку переведет окно в режим отображения нескольких начальных (как правило, пяти) элементов каждого набора данных.

Дополнительная информация о возможностях режима окна с динамической компоновкой приведена в [соответствующем разделе](#) страницы, посвященной описанию окна задачника.

Пример 4. Более сложное задание на обработку наборов данных

ExamTaskC53°. На вход подаются сведения о ценах на бензин на автозаправочных станциях (АЗС). В первой строке содержится значение M одной из марок бензина, во второй строке указывается целое число N , а каждая из последующих N строк имеет формат <Марка бензина> <Улица> <Компания> <Цена 1 литра (в копейках)>. Имеется не более 20 различных компаний и не более 30 различных улиц; названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Цена задается целым числом в диапазоне от 2000 до 3000. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы, на которой имеются АЗС с бензином марки M , определить максимальную цену бензина этой марки (вначале выводить максимальную цену, затем название улицы). Сведения о каждой улице выводить на новой строке и упорядочивать по возрастанию максимальной цены, а для одинаковой цены — по названиям улиц в алфавитном порядке. Если ни одной АЗС с бензином марки M не найдено, то вывести текст «Нет».

Приведем окно задачника, которое появится на экране при запуске программы-заготовки для данного задания (в данном окне скрыт раздел с формулировкой; в результате оказались скрытыми и кнопки, отвечающие за «интеллектуальную» прокрутку, поскольку в окне полностью отображается содержимое оставшихся разделов):

```
Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]
ОБРАБОТКА СЛОЖНЫХ НАБОРОВ ДАННЫХ
Задание: ExamTaskC53
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)
Ознакомительный запуск:
не выполнена ни одна из операций вывода.
Выход (Esc)
Исходные данные
1: '92'
2: '18'
3: '95 ул.Садовая Нефть-плюс 2540'
4: '98 ул.Первомайская Лидер 2780'
5: '92 ул.Садовая Нефть-плюс 2120'
6: '95 ул.Почтовая Нефть-супер 2570'
7: '92 ул.Айвазовского Альфа-нефть 2210'
8: '95 ул.Репина Ойл-бренд 2330'
9: '98 ул.Первомайская Оптима 2750'
10: '95 ул.фабричная Стандарт-ойл 2570'
11: '95 ул.Почтовая Альфа-нефть 2330'
12: '92 ул.Чайковского Стандарт-ойл 2180'
13: '98 ул.Садовая Юг-нефть 2840'
14: '98 ул.Айвазовского Люкс-ойл 2600'
15: '98 ул.Айвазовского Ойл-вест 2780'
16: '92 ул.Планерная Нефть-плюс 2030'
17: '95 ул.фабричная Ойл-бренд 2570'
18: '95 ул.Планерная Стандарт-ойл 2450'
19: '92 ул.Репина Ойл-вест 2150'
20: '92 ул.Садовая Авангард 2240'
Пример верного решения
1: '2030 ул.Планерная'
2: '2150 ул.Репина'
3: '2180 ул.Чайковского'
4: '2210 ул.Айвазовского'
5: '2240 ул.Садовая'
```

Выясним, какая структура является наиболее подходящей для хранения информации, необходимой для решения задачи. Нам требуется информация, связанная с различными улицами, которых по условию не более 30, причем для каждой улицы надо хранить сведения двух видов: ее название и максимальную цену бензина марки M . Поэтому мы можем либо завести массив из 30 элементов-записей с двумя полями, либо два массива: один содержащий названия улиц, а другой — максимальные цены. Учитывая, что в конце программы нам потребуется выполнять сортировку полученных данных, целесообразнее использовать массив записей, поскольку это позволит записать алгоритм сортировки в более компактной форме.

Определим запись `Street` с двумя полями `name` и `max` и опишем массив `s` из 30 элементов типа `Street`. Следует также завести переменную `ns`, в которой будет храниться количество заполненных элементов массива `s`.

При обработке каждой строки с исходными данными нам будут нужны прежде всего сведения о марке бензина. Если марка бензина не равна M , то оставшуюся часть строки обрабатывать не требуется, и можно сразу перейти к разбору следующей строки. Если марка бензина равна M , то необходимо узнать название улицы s_0 и цену бензина p . Заметим, что название компании для решения задачи не требуется, однако его необходимо прочесть, чтобы определить следующий элемент данных — цену бензина.

Если улица с названием s_0 еще не была включена в массив s , то ее необходимо включить в массив, присвоив полю max значение p . Если же улица уже присутствует в массиве, то необходимо сравнить поле max для данной улицы и значение p , изменив при необходимости поле max (здесь мы используем базовый алгоритм нахождения максимального значения).

Для ввода названий улиц и компаний в нашем случае удобно организовать *посимвольное* чтение строковых данных; признаком завершения такого чтения будет обнаружение пробельного символа.

После обработки набора исходных данных необходимо проверить, найдена ли хотя бы одна улица с АЗС, предлагающей марку бензина M (для этого достаточно сравнить значение ns с нулем). Если ни одна улица не найдена, то надо вывести строку «Нет»; в противном случае требуется выполнить сортировку массива s по указанному набору ключей и вывести полученные данные в требуемом порядке. Поскольку размер массива невелик, для его сортировки вполне допустимо использовать один из простых алгоритмов, например, алгоритм *пузырьковой сортировки*.

Приведем один из вариантов правильного решения задачи:

```
uses PT4Exam;
type
  Street = record
    name: string;
    max: integer;
  end;
var
  m, n, ns, i, j, k, p: integer;
  s: array[1..30] of Street;
  s0: string;
```

```

x: Street;
c: char;
begin
  Task('ExamTaskC53');
  readln(m); { m - марка бензина }
  readln(n);
  ns := 0;
  for i := 1 to n do
    begin
      read(k);
      if k <> m then
        readln { пропускаем оставшуюся часть строки }
      else
        begin
          s0 := '';
          read(c); { пропускаем пробел после первого числа }
          read(c); { читаем первый символ названия улицы }
          while c <> ' ' do
            begin
              s0 := s0 + c;
              read(c);
            end;
          read(c); { читаем первый символ названия компании }
          while c <> ' ' do
            read(c); { название компании не сохраняем }
          readln(p); { читаем цену бензина и переходим на новую строку }
          { Обработка прочитанной информации }
          k := 0;
          for j := 1 to ns do
            if s[j].name = s0 then { улица уже содержится в массиве s }
              begin
                k := 1;
                if s[j].max < p then
                  s[j].max := p;
                  break;
                end;
            if k = 0 then { улица еще не содержится в массиве s }
              begin
                Inc(ns);
                s[ns].name := s0;
                s[ns].max := p;
              end;
            end;
          end;
          if ns = 0 then { ни одной улицы не найдено }
            writeln('Нет')
          else

```

```
begin
  { Сортировка по возрастанию максимальной цены,
    а для одинаковых цен - по названиям улиц }
  for k := 1 to ns - 1 do
    for i := 1 to ns - k do
      if (s[i].max > s[i + 1].max) or
        (s[i].max = s[i + 1].max) and
        (s[i].name > s[i + 1].name) then
        begin
          x := s[i];
          s[i] := s[i + 1];
          s[i + 1] := x;
        end;
      { Вывод результатов в требуемом порядке }
      for i := 1 to ns do
        writeln(s[i].max, ' ', s[i].name);
    end;
end.
```

Электронный задачник Programming Taskbook

Общее описание

Электронный задачник **Programming Taskbook** предназначен для обучения программированию на языках Pascal, Visual Basic, C++, C#, Visual Basic .NET, Python и Java. Он содержит 1300 учебных заданий, охватывающих все основные разделы базового курса программирования: от скалярных типов и управляющих операторов до сложных структур данных и рекурсивных алгоритмов. Начиная с версии 4.10, в базовый набор заданий входят группы, связанные с [ЕГЭ по информатике и ИКТ](#).

Автором задачника **Programming Taskbook** является доцент факультета математики, механики и компьютерных наук Южного федерального университета Михаил Эдуардович Абрамян (mabr@math.sfedu.ru).

Версия 4.11 задачника Programming Taskbook реализована для следующих сред:

- Borland Delphi 7.0 и 2006, в частности, Turbo Delphi 2006 for Windows,
- Free Pascal Lazarus 1.0,
- Microsoft Visual Basic 5.0 и 6.0,
- Microsoft Visual C++ 6.0,
- Microsoft Visual Studio .NET 2003, 2005, 2008, 2010 и 2012 (языки Visual C++, Visual Basic .NET и Visual C# .NET),
- Python 2.5, 2.6, 2.7, 3.2,
- NetBeans IDE 6.x и 7.x (язык Java).

Задачник включен в учебную систему программирования Pascal ABC (<http://sunschool.mmcs.sfedu.ru>, автор доц. С. С. Михалкович), образуя единый программный комплекс «Pascal ABC & Programming Taskbook». Задачник может также использоваться совместно с системой программирования PascalABC.NET и веб-средой разработки ProgrammingABC.NET WDE (<http://www.pascalabc.net/WDE>).

Начиная с версии 4.9, к задачнику в качестве дополнения можно подключать комплекс **Programming Taskbook for MPI** — *электронный задачник по параллельному программированию с использованием технологии MPI*.

Начиная с версии 4.10, к задачку в качестве дополнения можно подключать комплекс **Programming Taskbook for Bioinformatics** — *электронный задачник по строковым алгоритмам биоинформатики*.

В версии 4.11 добавлена поддержка языков Python и Java и дополнен интерфейс [окна задачника](#) (в частности, реализован новый [режим окна с динамической компоновкой](#)).

Задачник Programming Taskbook предоставляет учащимся следующие возможности:

- отображение на экране текста задания и связанных с ним данных;
- демонстрация правильных результатов для каждого задания;
- предоставление исходных данных программе учащегося;
- выявление стандартных ошибок ввода-вывода, связанных с неверным количеством или неверным типом исходных или результирующих данных;
- проверка правильности результатов, полученных программой;
- запись в особый *файл результатов* информации о каждом тестовом испытании программы;
- регистрация задания как выполненного после проведения серии успешных тестовых испытаний программы.

Использование электронного задачника существенно ускоряет процесс выполнения заданий, так как избавляет учащегося от дополнительных усилий по организации ввода-вывода, что особенно удобно при обработке массивов, строк, файлов и динамических структур. Предлагая учащемуся готовые исходные данные, задачник акцентирует его внимание на разработке и программной реализации *алгоритма* решения заданий, причем разнообразие исходных данных обеспечивает надежное *тестирование* предложенного алгоритма.

Программный комплекс «Электронный задачник по программированию Programming Taskbook 4» зарегистрирован в Реестре программ для ЭВМ 28 апреля 2007 г. (свидетельство об официальной регистрации программы для ЭВМ номер 2007611815).

Подробная информация о задачнике, включающая его общее

описание, формулировки всех учебных заданий, примеры выполнения типовых заданий для различных языков программирования, содержится на сайте ptaskbook.com.

Программные модули

В состав задачника входят следующие программные модули:

- **PT4Demo** — позволяет просмотреть в [демонстрационном режиме](#) все задания, включенные в задачник;
- **PT4Load** — обеспечивает [генерацию программы-шаблона](#) для требуемого учебного задания и ее немедленную загрузку в выбранную среду программирования;
- **PT4Results** — предназначен для расшифровки, анализа и отображения на экране содержимого [файла результатов](#), в который заносятся сведения о ходе выполнения заданий.

В варианте задачника для PascalABC.NET эти модули могут быть вызваны непосредственно из среды **PascalABC.NET** с помощью соответствующих команд меню «Модули», а также кнопок и горячих клавиш:

-  или **Shift+Ctrl+D** для **PT4Demo**,
-  или **Shift+Ctrl+L** для **PT4Load**,
-  или **Shift+Ctrl+R** для **PT4Results**.

Начиная с версии 4.11, программный модуль **PT4Results** можно вызывать непосредственно из окна задачника, используя клавишу **F2**.

Мини-вариант задачника

Часть заданий можно выполнять в *мини-варианте* задачника, не требующем приобретения лицензии и регистрации лицензированной копии задачника с помощью программы настройки PT4Setup).

Задания, доступные для выполнения в мини-варианте, помечаются в [окне задачника](#) символом «°». В мини-вариант включены 310 заданий, в том числе все задания групп Begin, Integer, Boolean, а также 200 избранных заданий из других разделов задачника. Ниже приводится список всех заданий, включенных в мини-вариант задачника:

Begin1–Begin40, Integer1–Integer30, Boolean1–Boolean40, If4, If6, If8, If12, If22, If26, Case2, Case4, Case9–Case10, Case18, For5, For12–For13, For15–For16, For19–For20, For33, For36, While1–While2, While4, While7, While11–While12, While22–While23, Series1, Series15–Series17, Series19, Series21, Series30, Proc4, Proc8, Proc10, Proc20–Proc21, Proc25, Proc40, Minmax1, Minmax6, Minmax12, Minmax19, Minmax22, Array4, Array7, Array16, Array32, Array47, Array54, Array63, Array71, Array79, Array89, Array92, Array108, Array112, Array116, Array134, Matrix7, Matrix24, Matrix36, Matrix53, Matrix74, Matrix82, Matrix88, Matrix100, String9–String10, String19, String29, String41, String44, String63, String70, File2, File10, File25, File27, File41, File43, File48, File50, File58, File61, File63, File67, File74, Text1, Text4, Text16, Text21, Text24, Text34, Text38, Text42, Text44, Text57, Param1, Param17, Param30, Param40, Param49, Param53, Param59–Param61, Recur1, Recur4–Recur5, Recur10, Recur14–Recur18, Recur21, Recur25, Recur27, Dynamic2–Dynamic3, Dynamic5, Dynamic8–Dynamic12, Dynamic25, Dynamic30, Dynamic49, Dynamic55, Dynamic59, Dynamic63, Dynamic70, Dynamic74, Dynamic78, Tree2, Tree6, Tree9, Tree12–Tree13, Tree32, Tree34, Tree40, Tree47, Tree49, Tree53, Tree59, Tree65, Tree70, Tree74–Tree76, Tree79, Tree86, Tree92, ExamBegin2, ExamBegin5, ExamBegin7, ExamBegin12, ExamBegin21, ExamBegin28, ExamBegin33, ExamBegin38, ExamBegin42, ExamBegin45, ExamBegin49, ExamBegin51, ExamBegin53, ExamBegin61, ExamBegin65, ExamBegin71, ExamBegin83–ExamBegin84, ExamBegin87, ExamBegin95, ExamTaskC1, ExamTaskC4, ExamTaskC13, ExamTaskC19,

ExamTaskC25, ExamTaskC34, ExamTaskC37, ExamTaskC44,
ExamTaskC49, ExamTaskC53, ExamTaskC62, ExamTaskC68,
ExamTaskC73, ExamTaskC81, ExamTaskC83, ExamTaskC86,
ExamTaskC88, ExamTaskC92, ExamTaskC97, ExamTaskC100.

Мини-вариант задачника можно рекомендовать для использования при самостоятельном изучении программирования, так как он охватывает все основные темы и не содержит однотипных заданий. Полный вариант задачника предназначен, прежде всего, для преподавателей программирования, поскольку он позволяет легко создавать наборы индивидуальных заданий и существенно повышает эффективность групповых практических занятий.

Группы заданий

Ниже перечислены все базовые группы заданий, включенные в электронный задачник **Programming Taskbook** версии 4.11 (в скобках указывается количество заданий в данной группе).

- **Begin** — ввод и вывод данных, оператор присваивания (40),
- **Integer** — целые числа (30),
- **For** — цикл с параметром (40),
- **Boolean** — логические выражения (40),
- **If** — условный оператор (30),
- **Case** — оператор выбора (20),
- **While** — цикл с условием (30),
- **Series** — последовательности (40),
- **Proc** — процедуры и функции (60),
- **Minmax** — минимумы и максимумы (30),
- **Array** — одномерные массивы (140),
- **Matrix** — двумерные массивы (матрицы) (100),
- **String** — символы и строки (70),
- **File** — двоичные (типизированные) файлы (90),
- **Text** — текстовые файлы (60),
- **Param** — составные типы данных в процедурах и функциях (70),
- **Recur** — рекурсия (30),
- **Dynamic** — динамические структуры данных (80),
- **Tree** — деревья (100),
- **ExamBegin** — ЕГЭ по информатике: базовые алгоритмы (100),
- **ExamTaskC** — ЕГЭ по информатике: задачи повышенной сложности (100).

В варианте задачника для системы PascalABC.NET имеются две дополнительные группы ObjDyn и ObjTree, в содержательном отношении идентичные группам Dynamic и Tree, однако использующие в формулировках объектную терминологию (группы Dynamic и Tree ориентированы на использование указателей).

Для выполнения заданий из задачника **Programming Taskbook** к программе необходимо подключить [модуль PT4](#).

Используя [конструктор учебных заданий PT4TaskMaker](#), можно создавать новые группы заданий, включая в них новые задания или импортируя имеющиеся задания из других групп.

Замечания о формулировках заданий и используемых в них данных

Если о типе исходных или результирующих числовых данных в задании ничего не сказано, то предполагаются *вещественные* данные. Исключение составляет группы заданий Dynamic и Tree (а также ObjDyn и ObjTree), в которой все числовые данные считаются *целыми*, и в формулировках заданий это особо не оговаривается.

При обработке наборов *вещественных* чисел следует предполагать, что все элементы набора являются *различными* (таким образом, любой набор вещественных чисел содержит единственный минимальный и единственный максимальный элемент). В наборах *целых* чисел могут присутствовать *одинаковые* элементы; в частности, наборы целых чисел могут содержать несколько минимальных и максимальных элементов. Аналогичные предположения справедливы для числовых массивов, а также для файлов, содержащих числовые данные.

Во всех заданиях на обработку *массивов* (как одномерных, так и двумерных) начальное значение любого индекса считается равным 1. Если в задании не указан максимальный размер исходных массивов, то его можно считать равным 10 для одномерных и 10×10 для двумерных массивов.

При описании элементов одномерных и двумерных массивов используется понятие *порядкового номера элемента*, причем начальный элемент массива A размера N всегда имеет порядковый номер 1 и обозначается в формулировках заданий как A_1 , а конечный элемент этого же массива имеет порядковый номер N и обозначается как A_N . Аналогично, начальный элемент двумерного массива B обозначается как $B_{1,1}$. Кроме того, понятие порядкового номера применяется к *строкам* и *столбцам* двумерных массивов (матриц): начальная строка и начальный столбец матрицы размера $M \times N$ имеют порядковый номер 1, конечная строка — номер M , а конечный столбец — номер N . Подобный подход не зависит от выбора языка программирования и соответствует традиционно используемой в математике нумерации элементов векторов и

матриц.

Максимальный размер исходных *файлов* не указывается, поэтому при решении заданий на файлы не следует использовать вспомогательные массивы, содержащие все элементы исходных файлов, однако допускается использование *вспомогательных файлов*. Все исходные файлы считаются существующими, за исключением специально оговоренных случаев, в которых существование исходных файлов требуется проверять в ходе выполнения задания.

Под *размером* двоичного типизированного файла всегда подразумевается количество содержащихся в нем *элементов* указанного типа (а не количество байтов, как это принято в операционной системе). В формулировках заданий предполагается, что элементы двоичных файлов, как и элементы массивов, нумеруются от 1.

Задания, связанные с ЕГЭ по информатике

Начиная с версии 4.10, задачник Programming Taskbook включает набор групп заданий, связанных с ЕГЭ по информатике и ИКТ. Эти группы начинаются с префикса Exam; они доступны для программных сред языков Pascal (в том числе PascalABC.NET) и C++.

В базовый набор заданий включены две группы Exam: группа ExamBegin, содержащая задания на освоение базовых алгоритмов, включенных в кодификатор ЕГЭ по информатике и ИКТ, и группа ExamTaskC, содержащая типовые задания повышенного уровня сложности, включенных в ЕГЭ в качестве заданий группы C. Каждая из групп состоит из 100 учебных заданий; 20 заданий каждой группы доступны для выполнения в [мини-варианте задачника](#).

Особенностью групп Exam является то, что при их выполнении не требуется использовать специальные средства ввода-вывода, входящие в задачник. Для того чтобы максимально приблизить вид программы, выполняющей задание, к виду, требуемому на экзамене, в задачнике реализован специальный механизм, позволяющий оформлять ввод-вывод данных с применением *стандартных средств используемого языка программирования*: процедур Read/Readln-Write/Writeln для языка Pascal и стандартных потоков ввода-вывода cin-cout для языка C++.

При использовании заданий групп Exam сохраняются основные особенности задачника: автоматическое предоставление программе учащегося исходных данных и автоматическая проверка правильности предложенного решения. Следует отметить, что эти особенности оказываются наиболее полезными при решении задач повышенной сложности (группа ExamTaskC), так как в них, как правило, должны использоваться наборы исходных данных большого размера.

При выполнении заданий групп Exam учащийся должен обеспечивать надлежащее *форматирование* выходных данных (в других группах заданий это не требуется, поскольку средства вывода электронного задачника выполняют форматирование автоматически).

Отказ от использовании специальных средств ввода-вывода приводит к тому, что любые ошибки ввода-вывода уже не обрабатываются задачиком и обычно приводят к сообщениям об ошибке времени выполнения. Это обстоятельство несколько затрудняет поиск ошибок, но в то же время позволяет приблизить его к реальному процессу отладки программы, не использующему «подсказки» задачника.

Модуль РТ4

Описанные в данном разделе типы и процедуры будут доступны в программе, если к ней с помощью оператора **uses** подключен модуль РТ4 (см. [пример](#) программы, использующей задачник **Programming Taskbook**).

Дополнительные типы данных PNode и TNode

```
type   PNode = ^TNode;  
       TNode = record  
         Data: integer;  
         Next: PNode;  
         Prev: PNode;  
         Left: PNode;  
         Right: PNode;  
         Parent: PNode;  
       end;
```

Типы `PNode` и `TNode` используются в заданиях групп `Dynamic` и `Tree`. В заданиях на стеки и очереди (`Dynamic1–Dynamic28`) при работе с записями типа `TNode` используются только поля `Data` и `Next`; в заданиях на двусвязные списки (`Dynamic29–Dynamic80`) используются поля `Data`, `Next` и `Prev`. В большинстве заданий на бинарные деревья (группа `Tree`) используются поля `Data`, `Left` и `Right`; в заданиях на обработку бинарных деревьев с обратной связью (`Tree48–Tree56` и `Tree70–Tree71`) дополнительно используется поле `Parent`.

Все исходные и результирующие данные-указатели в заданиях имеют тип `PNode`; их ввод и вывод должен осуществляться с помощью процедур `GetP` и `PutP` (в системе **PascalABC.NET** указатели, как и другие данные, можно получать из задачника и передавать ему с помощью стандартных процедур ввода-вывода `read` и `write`).

В программе учащегося не следует повторно описывать типы `PNode` и `TNode`.

Инициализация задания, ввод–вывод данных

```
procedure Task(Name: string);
```

Процедура инициализирует задание с именем `Name`. Она должна вызываться в начале программы, выполняющей это задание (до вызова процедур ввода-вывода). Если в программе, подключившей модуль `PT4`, не указана процедура `Task`, то при запуске программы будет выведено окно с сообщением «*Не вызвана процедура Task с именем задания*».

Имя задания `Name` должно включать имя темы и порядковый номер в пределах темы (например, `'Begin3'`). Регистр букв в имени темы может быть произвольным. Если указана неверная тема задания, то программа выведет сообщение об ошибке, в котором будут перечислены названия всех имеющихся тем. Если указан недопустимый номер задания, то программа выведет сообщение, в котором будет указан диапазон допустимых номеров для данной темы. Если после имени задания в параметре `Name` указан символ `?` (например, `'Begin1?'`), то программа будет работать в [демонстрационном режиме](#).

Начиная с версии 4.8, процедура `Task` может также использоваться для генерации и вывода на экран html-страницы с текстом задания или группы заданий. Для этого необходимо указать в качестве параметра `Name` имя конкретного задания или группы заданий и символ `#`, например, `'Begin3#'` или `'Begin#'`. Дополнительные сведения о генерации html-страниц с описаниями заданий приводятся в разделе, посвященном демонстрационному режиму задачника.

Если при первом вызове процедуры `Task` в параметре не указывается символ `#`, то все последующие вызовы процедуры `Task` игнорируются. Если при первом вызове процедуры `Task` в параметре указывается символ `#`, то игнорируются все последующие вызовы процедуры `Task`, не содержащие этот символ. С помощью нескольких вызовов процедуры `Task`, содержащей в параметре символ `#`, можно обеспечить генерацию html-страницы с описанием нескольких групп заданий, причем в каждой группе при этом можно

отображать только некоторые задания.

```
procedure GetB(var X: boolean);  
procedure GetC(var X: char);  
procedure GetN(var X: integer);  
procedure GetR(var X: real);  
procedure GetS(var X: string);  
procedure GetP(var X: PNode);
```

Процедуры обеспечивают ввод исходных данных в программу, выполняющую учебное задание. Они должны вызываться после вызова процедуры `Task`; в случае их вызова до вызова процедуры `Task` при запуске программы будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

Используемая процедура ввода должна соответствовать типу очередного элемента исходных данных; в противном случае выводится сообщение об ошибке «*Неверно указан тип при вводе исходных данных*» (такое сообщение будет выведено, например, если очередной элемент данных является символом, а для его ввода используется процедура `GetN`).

При попытке ввести больше исходных данных, чем это предусмотрено в задании, выводится сообщение об ошибке «*Попытка ввести лишние исходные данные*». Если исходные данные, необходимые для решения задания, введены не полностью, то выводится сообщение «*Введены не все требуемые исходные данные*».

При использовании задачника в системе **PascalABC.NET** вместо процедур группы `Get` можно использовать обычные процедуры ввода `read`.

```
procedure PutB(X: boolean);  
procedure PutC(X: char);  
procedure PutN(X: integer);  
procedure PutR(X: real);  
procedure PutS(X: string);  
procedure PutP(X: PNode);
```

Процедуры обеспечивают вывод на экран результирующих данных, найденных программой, и их сравнение с *контрольными данными* (то есть с правильным решением). Как и процедуры группы **Get**, эти процедуры должны вызываться после вызова процедуры **Task**; в противном случае при запуске программы будет выведено сообщение об ошибке *«В начале программы не вызвана процедура Task с именем задания»*.

В отличие от процедур группы **Get**, в качестве параметра процедур группы **Put** можно указывать не только переменные, но и *выражения* (в частности, *константы* соответствующего типа). Используемая процедура должна соответствовать типу очередного элемента результирующих данных, в противном случае выводится сообщение об ошибке *«Неверно указан тип при выводе результатов»*.

Как и в случае процедур группы **Get**, при вызовах процедур группы **Put** программа осуществляет контроль за соответствием количества требуемых и выведенных результирующих данных. Если программа выведет недостаточное или избыточное количество результирующих данных, то после проверки этих данных появится сообщение *«Выведены не все результирующие данные»* или, соответственно, *«Попытка вывести лишние результирующие данные»*.

При использовании задачника в системе **PascalABC.NET** вместо процедур группы **Put** можно использовать обычные процедуры вывода **write**.

```
procedure Dispose(var P: PNode);
```

Данная процедура переопределяет стандартную процедуру **Dispose** для того, чтобы контролировать действия учащегося по освобождению памяти при выполнении заданий групп **Dynamic** и **Tree**.

Класс Node и альтернативный ввод–вывод в стиле .NET

В варианте задачника **Programming Taskbook**, включенном в систему **PascalABC.NET**, предусмотрен альтернативный способ организации ввода-вывода, характерный не для традиционного Паскаля, а для языков платформы .NET. Наличие двух способов ввода-вывода обусловлено тем обстоятельством, что система **PascalABC.NET** позволяет разрабатывать программы как в стиле, характерном для традиционного Паскаля, так и в .NET-стиле, ориентированном на использование стандартных средств платформы .NET, в том числе ее библиотеки классов.

Кроме новых средств ввода-вывода в варианте задачника для системы **PascalABC.NET** предусмотрен класс **Node**, который следует использовать вместо типов **PNode** и **TNode** при выполнении заданий на динамические структуры «в объектном стиле». Заметим, что в этом случае необходимо пользоваться группами **ObjDyn** и **ObjTree**, в которых (в отличие от групп **Dynamic** и **Tree**) применяется «объектная» терминология, ориентированная на применение классов платформы .NET.

```
type
    Node = class(IDisposable)
    . . .
    public
        // Конструкторы:
        constructor Create;
        constructor Create(aData: integer);
        constructor Create(aData: integer; aNext: Node);
        constructor Create(aData: integer; aNext, aPrev:
Node);
        constructor Create(aLeft, aRight: Node; aData:
integer);
        constructor Create(aLeft, aRight: Node; aData:
integer; aParent: Node);
        // Свойства (доступны для чтения и записи):
        property Data: integer;
        property Next: Node;
        property Prev: Node;
        property Left: Node;
```

```
    property Right: Node;  
    property Parent: Node;  
    // Метод, освобождающий ресурсы, используемые  
    объектом Node:  
    procedure Dispose;  
end;
```

Класс `Node` используется в заданиях групп `ObjDyn` и `ObjTree`. В заданиях на стеки и очереди (`ObjDyn1–ObjDyn28`) при работе с объектами типа `Node` используются только свойства `Data` и `Next`; в заданиях на двусвязные списки (`ObjDyn29–ObjDyn80`) используются свойства `Data`, `Next` и `Prev`. В большинстве заданий на бинарные деревья (группа `ObjTree`) используются свойства `Data`, `Left` и `Right`; в заданиях на обработку бинарных деревьев с обратной связью (`ObjTree48–ObjTree56` и `ObjTree70–ObjTree71`) дополнительно используется свойство `Parent`.

Варианты конструктора класса `Node` позволяют задавать значения требуемых свойств при создании объекта; прочие свойства инициализируются нулевыми значениями (числом 0 для свойства `Data`, нулевой ссылкой `nil` для остальных свойств).

Следует обратить внимание на то, что данный класс реализует интерфейс `IDisposable`, поэтому при завершении работы с объектом типа `Node` требуется вызвать его метод `Dispose`, освобождающий *неуправляемые ресурсы*, выделенные для этого объекта (исключение делается только для тех объектов, которые передаются обратно задачику в качестве результирующих данных). Если в задании требуется вызвать метод `Dispose` для некоторых объектов, но этот вызов не выполняется, то при запуске программы выводится сообщение об ошибке «*Не вызван метод Dispose для объекта типа Node*».

Все исходные и результирующие данные-ссылки в заданиях группы `ObjDyn` и `ObjTree` имеют тип `Node`; их ввод и вывод должен осуществляться с помощью функции `GetNode` и процедуры `Put`, описанных ниже.

```
function GetBoolean: boolean;  
function GetChar: char;
```

```
function GetInt: integer;  
function GetNode: Node;  
function GetReal: real;  
function GetString: string;
```

Функции обеспечивают ввод исходных данных в программу, выполняющую учебное задание, причем ввод организуется в стиле, характерном для платформы .NET (поскольку в стандартной библиотеке .NET ввод данных *всегда* выполняется с помощью функций). Эти функции должны вызываться после вызова процедуры **Task**; в случае их вызова до вызова процедуры **Task** при запуске программы будет выведено сообщение об ошибке *«В начале программы не вызвана процедура Task с именем задания»*.

Используемая функция ввода должна соответствовать типу очередного элемента исходных данных; в противном случае выводится сообщение об ошибке *«Неверно указан тип при вводе исходных данных»* (такое сообщение будет выведено, например, если очередной элемент данных является символом, а для его ввода используется функция **GetInt**).

При попытке ввести больше исходных данных, чем это предусмотрено в задании, выводится сообщение об ошибке *«Попытка ввести лишние исходные данные»*. Если исходные данные, необходимые для решения задания, введены не полностью, то выводится сообщение *«Введены не все требуемые исходные данные»*.

```
procedure Put(params A: array of object);
```

Процедура **Put** обеспечивает вывод на экран результирующих данных, найденных программой, и их сравнение с *контрольными данными* (то есть с правильным решением). Как и описанные выше функции группы **Get**, процедура **Put** должна вызываться после вызова процедуры **Task**; в противном случае при запуске программы будет выведено сообщение об ошибке *«В начале программы не вызвана процедура Task с именем задания»*.

Благодаря использованию параметра-массива, снабженного атрибутом **params**, при вызове процедуры **Put** можно указывать *произвольное* число параметров. Параметры могут иметь тип

`boolean`, `integer`, `real`, `char`, `string`, `Node`. В качестве параметров процедуры `Put` можно указывать не только переменные, но и *выражения* (в частности, *константы* соответствующего типа, а также *нулевую ссылку* `nil`). Заметим, что нулевые ссылки, как и объекты типа `Node`, требуется выводить только в заданиях групп `ObjDyn` и `ObjTree`. Если в списке параметров указываются параметры недопустимого типа, то при выполнении программы выводится сообщение об ошибке «*В методе Put указан параметр недопустимого типа*».

Тип параметра должен не только быть допустимым, но и соответствовать типу очередного элемента результирующих данных; в противном случае выводится сообщение об ошибке «*Неверно указан тип при выводе результатов*».

Как и в случае функций группы `Get`, при вызовах процедуры `Put` программа осуществляет контроль за соответствием количества требуемых и выведенных результирующих данных. Если программа выведет недостаточное или избыточное количество результирующих данных, то после проверки этих данных появится сообщение «*Выведены не все результирующие данные*» или, соответственно, «*Попытка вывести лишние результирующие данные*».

Вывод отладочной информации

Описываемые далее отладочные средства появились в версии 4.9 задачника Programming Taskbook. С их помощью можно выводить отладочную информацию непосредственно в окно задачника (в специальный [раздел отладки](#)).

```
procedure Show(S: string);
```

Отображает текстовую строку *S* в разделе отладки окна задачника.

Если текущая экранная строка в разделе отладки уже содержит некоторый текст, то строка *S* снабжается начальным пробелом и приписывается к этому тексту, за исключением случая, когда при таком приписывании размер полученного текста превысит ширину области данных (равную 80 символам). В последнем случае вывод строки *S* осуществляется с начала следующей экранной строки; если же и в этой ситуации строка *S* превысит ширину области данных, то строка *S* будет выведена на нескольких экранных строках, причем разрывы текста будут выполняться по пробельным символам строки *S*, а при отсутствии пробелов — при достижении очередного фрагмента строки длины, равной 80.

Строка *S* может содержать явные команды перехода на новую экранную строку. В качестве таких команд можно использовать или символ с кодом 13 («возврат каретки»), или символ с кодом 10 («переход на новую строку»), или их комбинацию в указанном порядке (`#13#10`).

```
procedure Show([S: string;] A: integer[; W: integer]);  
procedure Show([S: string;] A: real[; W: integer]);
```

Перегруженные варианты процедуры Show, предназначенные для вывода числовых отладочных данных. Использование этих вариантов позволяет максимально упростить действия учащегося, связанные с выводом числовых данных, поскольку избавляет его от необходимости применять стандартные средства языка Pascal, предназначенные для преобразования чисел в их строковые представления.

При вызове приведенных вариантов можно не указывать один или

оба параметра, заключенные в квадратные скобки.

Строковый параметр *S* определяет необязательный комментарий, который указывается перед выводимым числом; если параметр *S* отсутствует, то комментарий полагается равным пустой строке.

Числовой параметр *A* определяет выводимое число.

Необязательный целочисленный параметр *W* определяет *ширину поля вывода* (т. е. количество экранных позиций, отводимое для вывода числа). Если указанной ширины *W* поля вывода недостаточно, то значение параметра *W* игнорируется; в этом случае (а также в случае, если параметр *W* отсутствует) используется ширина поля вывода, минимально необходимая для отображения данного числа. Если число не занимает всего поля вывода, то оно дополняется слева пробелами (т. е. выравнивается по *правой* границе поля вывода). В качестве десятичного разделителя для чисел с дробной частью используется *точка*.

Вещественные числа по умолчанию выводятся в формате с фиксированной точкой и двумя дробными знаками. Изменить формат вывода вещественных чисел можно с помощью вспомогательной процедуры `SetPrecision`, описываемой далее.

```
procedure ShowLine([S: string]);  
procedure ShowLine([S: string;] A: integer[; W:  
integer]);  
procedure ShowLine([S: string;] A: real[; W: integer]);
```

Модификации ранее описанных процедур `Show`; после вывода указанных данных в раздел отладки дополнительно осуществляют автоматический переход на следующую экранную строку. Смысл параметров — тот же, что и для соответствующих вариантов процедуры `Show`. Параметры, указанные в квадратных скобках, могут отсутствовать. Если процедура `ShowLine` вызывается без параметров, то она просто обеспечивает переход на новую экранную строку в разделе отладки.

```
procedure HideTask;
```

Вызов данной процедуры обеспечивает автоматическое скрытие

всех разделов окна задачника, кроме раздела отладки. Если раздел отладки в окне задачника не отображается (в частности, если программа запущена в демонстрационном режиме), то вызов процедуры `HideTask` игнорируется. Игнорируются также все повторные вызовы данной процедуры.

Скрыть/восстановить основные разделы окна задачника после его отображения на экране можно также с помощью клавиши пробела или соответствующей команды контекстного меню раздела отладки.

procedure `SetPrecision(N: integer);`

Процедура предназначена для настройки формата вывода *вещественных* отладочных данных. Если параметр `N` положителен, то он определяет количество выводимых дробных разрядов; при этом число выводится в формате с фиксированной точкой. Если параметр `N` равен нулю, то число выводится в формате с плавающей точкой (экспоненциальном формате); при этом число дробных знаков для экспоненциального формата определяется шириной поля вывода (т. е. параметром `w` процедуры `Show` или `ShowLine`). При отрицательных значениях параметра `N` выполняется та же настройка, что и при `N = 0`.

Действие текущей настройки числового формата, определенной процедурой `SetPrecision`, продолжается до очередного вызова этой процедуры. До первого вызова процедуры `SetPrecision` вещественные числа выводятся в формате с фиксированной точкой и двумя дробными знаками.

Особенности модуля PT4Exam

При выполнении заданий групп ExamBegin и ExamTaskC, связанных с ЕГЭ по информатике, к программам вместо модуля PT4 подключается модуль PT4Exam. В этом модуле реализована единственная дополнительная процедура [Task](#), обеспечивающая инициализацию задания с указанным именем. Все действия по вводу-выводу должны выполняться с использованием стандартных процедур языка Pascal. Отладочные средства модуля PT4 в модуле PT4Exam также недоступны, однако имеется возможность выводить дополнительные данные можно непосредственно в раздел результатов (хотя при этом решение будет считаться ошибочным).

Описанные выше ограничения модуля PT4Exam позволяют максимально приблизить полученное решение к виду, требуемому на экзамене (программа содержит всего две дополнительные конструкции: директиву подключения модуля PT4Exam и оператор вызова процедуры [Task](#), инициализирующей требуемое задание).

Создание шаблона программы

Входящий в состав задачника **Programming Taskbook** программный модуль **PT4Load** позволяет быстро создавать программы-заготовки для выполнения требуемого задания. Этот модуль можно вызвать непосредственно из среды **PascalABC.NET** командой меню «Модули | Создать шаблон программы» (с данной командой связана также кнопка и клавиатурная комбинация **Shift+Ctrl+L**).

При запуске данного модуля появляется окно, в котором требуется указать имя задания (в нижней части окна отображаются имена всех имеющихся групп заданий).



При вводе допустимого имени задания (то есть имени группы и порядкового номера, например, **Begin1**) кнопка «Загрузка» становится доступной, и после ее нажатия (или нажатия клавиши **Enter**) в рабочем каталоге создается файл, содержащий шаблон программы для выполнения выбранного задания (имя файла совпадает с именем задания). Созданная программа-шаблон сразу загружается в редактор **PascalABC.NET**; ее можно немедленно запустить, чтобы увидеть формулировку задания и пример исходных данных.

Приведем пример шаблона, созданного для задания **Begin3** (этот шаблон будет записан в файл **Begin3.pas**):

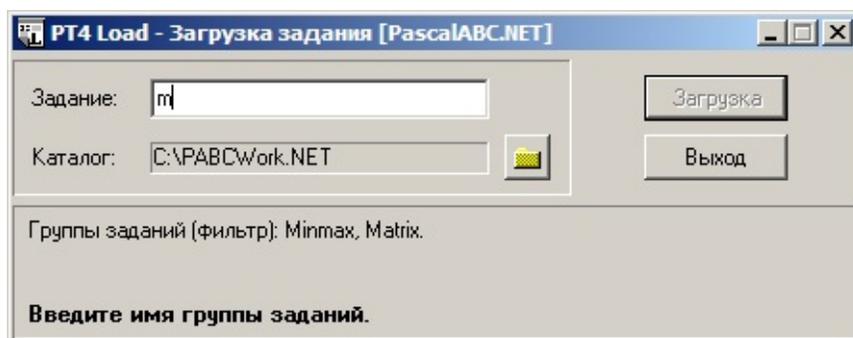
```
uses PT4;  
  
begin  
    Task( 'Begin3' );  
  
end.
```

Ввод имени задания для исполнителей Робот и Чертежник имеет следующую особенность: для них надо вначале ввести *префикс исполнителя* (**RB** для Робота или **DM** для Чертежника), после чего ввести имя набора заданий для соответствующего исполнителя и номер этого задания (например, **RBa1**).

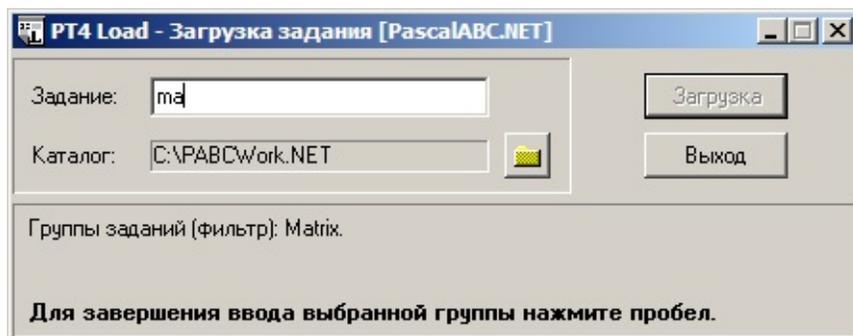
Если введено имя задания, для которого уже имеется файл с программой, то именно этот файл и будет загружен в редактор.

Особая группа **PAS** (последняя в списке групп заданий) позволяет загрузить в редактор любой файл с расширением **.pas** из рабочего каталога. При вводе имени этой группы и нажатии кнопки «Загрузка» появляется список всех pas-файлов; после выбора любого из этих файлов он немедленно загружается в редактор. Таким образом, использование группы **PAS** равносильно команде «Файл | Открыть».

В версии 4.11 задачника появилась возможность фильтрации групп в процессе ввода их названий. Фильтрация производится по уже введенным символам. Например, при вводе первого символа «m» в списке возможных групп останутся только группы, имена которых начинаются с этого символа; при этом перед списком отфильтрованных групп будет выведен текст «(фильтр)»:



Как только в списке останется единственный вариант, будет выведена подсказка о том, что для завершения ввода имени данной группы достаточно нажать пробел:



После нажатия пробела произойдет автоматическое дополнение введенной части имени группы:

PT4 Load - Загрузка задания [PascalABC.NET]

Задание:

Каталог: 

Группа Matrix: двумерные массивы (матрицы).
Количество заданий: 100.

Введите номер задания и нажмите кнопку "Загрузка" или [Enter].

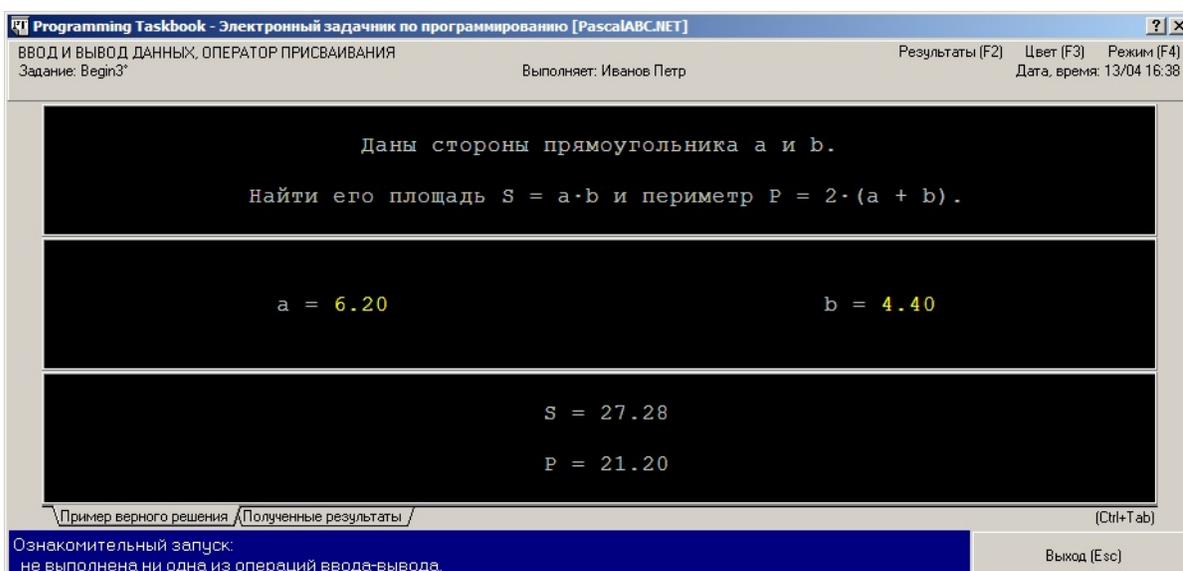
Окно задачника

Начиная с версии 4.11, наряду с традиционным режимом с *фиксированной компоновкой*, в котором для каждого раздела окна выделяется область из 5 экранных строк, можно использовать новый режим с *динамической компоновкой*, в котором размеры разделов определяются их фактическим содержанием, а размеры окна задачника «подстраиваются» под размеры разделов. Режим с динамической компоновкой более удобен при выполнении заданий, содержащих большие наборы исходных и результирующих данных, а также имеющих большие формулировки. В данном разделе вначале дается [общее описание окна задачника на примере режима с фиксированной компоновкой](#), затем описываются [дополнения, реализованные в версии 4.11](#), после чего приводится [описание режима с динамической компоновкой](#).

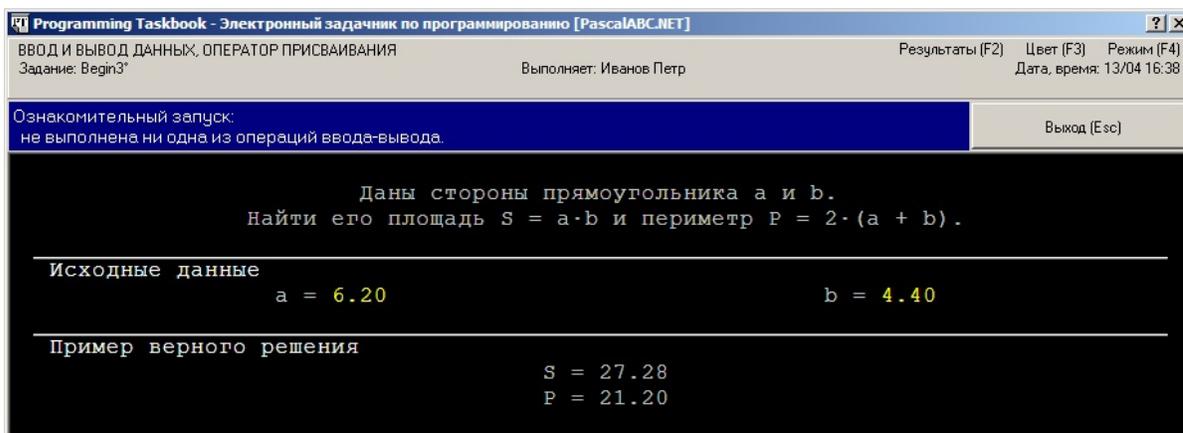
Общее описание

При запуске программы, использующей электронный задачник Programming Taskbook, на экране возникает *окно задачника*. В зависимости от текущей настройки, окно может отображаться либо в режиме с фиксированной компоновкой, либо в режиме с динамической компоновкой. Приведем вид этого окна для задания Begin3 в случае использования системы программирования PascalABC.NET (имя используемого языка программирования указывается в заголовке окна).

Режим с фиксированной компоновкой:



Режим с динамической компоновкой:



Переключаться между режимами окна можно, нажимая клавишу **F4** или щелкая мышью на кнопке быстрого доступа «Режим (F4)»,

расположенной в правом верхнем углу окна.

В данном пункте мы опишем основные компоненты окна задачника, используя в качестве образца окно с фиксированной компоновкой. [Дополнительные возможности](#), появившиеся в версии 4.11, а также [особенности режима с динамической компоновкой](#) будут описаны в последующих пунктах.

В верхней части окна выводится информация о теме, к которой относится выполняемое задание («ВВОД И ВЫВОД ДАННЫХ, ОПЕРАТОР ПРИСВАИВАНИЯ»), название задания, сведения об учащемся, дата и время запуска программы и кнопки быстрого доступа, связанные с некоторыми командами (кнопки располагаются в правом верхнем углу окна). Символ «°», указанный после названия задания, означает, что данное задание можно выполнять в [мини-варианте задачника](#). Затем следует раздел с *формулировкой задания*.

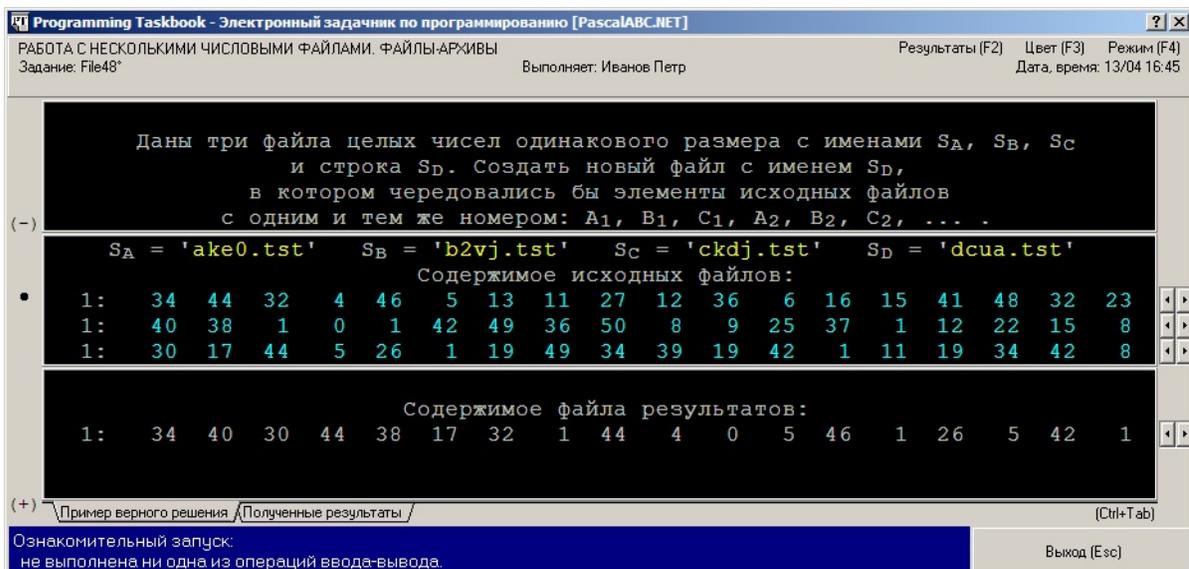
Далее располагается раздел с *исходными данными*. В случае задания Begin3 это вещественные числа a и b , определяющие длины сторон прямоугольника. При необходимости данные могут снабжаться комментариями; в нашем случае это строки « $a =$ » и « $b =$ », расположенные перед числами a и b .

Последним разделом в окне с фиксированной компоновкой является раздел *контрольных данных и результатов*, содержащий две вкладки: «Пример верного решения» и «Полученные результаты». Если в программе не выполняются действия по вводу-выводу данных, то при запуске программы активной становится вкладка «Пример верного решения» с *контрольными* (т. е. «правильными») результирующими данными. В нашем случае в ней отображаются правильные значения площади S и периметра P , снабженные комментариями. Для переключения между вкладками достаточно щелкнуть мышью на ярлычке нужной вкладки или ввести клавиатурную комбинацию **Ctrl+Tab**. В режиме с динамической компоновкой разделы с результатами и с контрольными данными отображаются одновременно; кроме того, в некоторых ситуациях один из этих разделов может отсутствовать (например, при ознакомительном запуске отсутствует раздел с результатами).

Чтобы отличить комментарии от данных (как исходных, так и результирующих), используется цветовое выделение: комментарии выводятся светло-серым цветом, данные — желтым. Контрольные данные отображаются тем же цветом, что и комментарии.

В нижней части окна располагается информационная панель и кнопка «Выход». Для выхода из программы надо нажать кнопку «Выход», клавишу **Esc** или ту клавишу, которая используется для запуска программы (для среды PascalABC.NET это клавиша **F9**). Таким образом, для проверки программы на нескольких наборах исходных данных достаточно несколько раз нажать клавишу **F9**.

Если формулировка задания или раздел с исходными или результирующими данными содержит более 5 строк, а также при наличии в задании исходных или результирующих данных, допускающих прокрутку, в окне с фиксированной компоновкой появляются дополнительные управляющие элементы, обеспечивающие прокрутку соответствующих компонентов задания. Ниже приводится пример окна, содержащего четыре набора файловых данных, каждый из которых допускает прокрутку.



В окне с динамической компоновкой формулировка задания и все данные отображаются полностью, поэтому для большинства заданий прокрутка не требуется (при необходимости окно с динамической компоновкой допускает общую прокрутку своего содержимого).

Элементы файлов отображаются бирюзовым цветом, чтобы подчеркнуть их отличие от обычных исходных данных (желтого цвета) и комментариев (светло-серого цвета).

Для прокрутки содержимого файла с помощью мыши достаточно щелкнуть на одной из двух кнопок , расположенных справа от требуемой экранной строки. С помощью клавиатуры можно выполнять поэлементную или «постраничную» прокрутку. Для поэлементной прокрутки используются клавиши со стрелками **Right** или **Down** (прокрутка вперед) и **Left** или **Up** (прокрутка назад); для «постраничной» прокрутки, обеспечивающей переход к новой «порции» элементов, уместяющейся на экранной строке, используются клавиши **Ctrl+Right** или **PgDn** (прокрутка вперед) и **Ctrl+Left** или **PgUp** (прокрутка назад). Кроме того, клавиша **Home** обеспечивает перемещение на начало, а клавиша **End** — на конец файла.

Начиная с версии 4.11, доступен еще один способ прокрутки: с помощью колесика мыши; при этом курсор мыши достаточно расположить в области прокручиваемого элемента.

Если окно программы содержит несколько элементов, допускающих прокрутку (например, несколько строк с файловыми данными — см. рисунок, приведенный выше), то прокрутка с помощью клавиатуры выполняется для *выделенного элемента*, помеченного в левой части окна задачника маркером в виде черного кружка. Для смены выделенного элемента предусмотрены клавиши **[+]** и **[-]**: первая из них обеспечивает циклический перебор всех отображенных на экране элементов, допускающих прокрутку, в направлении сверху вниз, а вторая — в направлении снизу вверх.

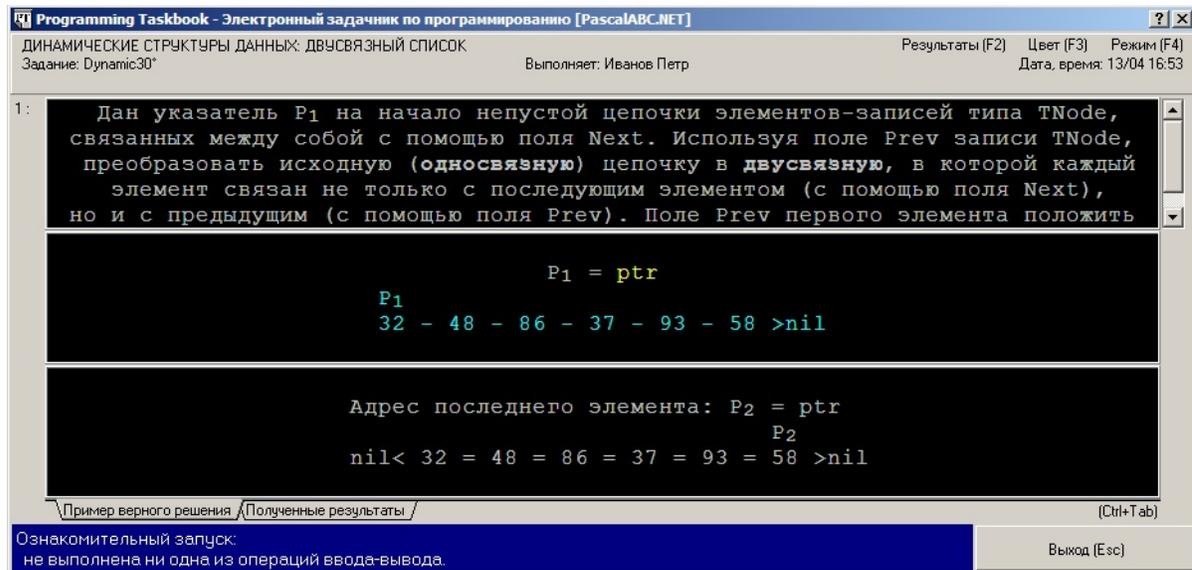
Для того чтобы определить, какие элементы файла отображаются в данный момент на экране, предусмотрен *указатель текущей позиции*, значение которого равно номеру первого из элементов, отображенных на экране. Данный указатель располагается в начале строки с элементами файла и отделяется от них двоеточием. Нумерация элементов файла начинается с единицы.

В отличие от двоичных файлов, содержимое которых всегда отображается на одной экранной строке, *текстовые файлы*

отображаются на нескольких (от двух до пяти) экранных строках, причем на каждой экранной строке размещается *одна* строка из текстового файла (это связано с тем, что строки, входящие в текстовый файл, могут иметь большую длину). Указатель текущей позиции в данном случае содержит номер первой отображаемой на экране файловой строки (нумерация строк, как и элементов двоичных файлов, ведется от единицы); этот указатель размещается в начале первой из экранных строк, отведенных для отображения текстового файла. Прокрутка строк текстового файла обеспечивается теми же клавишами, что и прокрутка элементов типизированного файла (в данном случае, поскольку элементы-строки расположены *по вертикали*, более естественно использовать клавиши **Up**, **Down** и **PgUp**, **PgDn**, в то время как для элементов двоичных файлов, расположенных *по горизонтали*, удобнее пользоваться клавишами **Left**, **Right** и их **Ctrl**-комбинациями). Текстовые файлы снабжаются вертикальными полосами прокрутки (вместо горизонтально расположенных парных кнопок, которыми снабжаются двоичные файлы).

Прокручивать можно не только элементы данных, но и текст с формулировкой задания, если он не уместится в выделенных для этих целей пяти экранных строках. Действия по прокрутке текста задания аналогичны действиям по прокрутке текстовых файлов. Для текста задания, допускающего прокрутку, также предусмотрен указатель текущей позиции. Этот указатель размещается слева от раздела с формулировкой задания и содержит номер первой из тех строк текста, которые в данный момент отображаются на экране. Если в результате прокрутки на экране выведены последние строки формулировки задания, то указатель текущей позиции подчеркивается (как на приведенном ниже рисунке). Это означает, что дальнейшая прокрутка вниз невозможна.

В приведенном ниже окне присутствуют также особые элементы исходных и результирующих данных: *цепочки узлов*. Эти элементы используются в заданиях группы Dynamic («Динамические структуры данных»). Особенности, связанные с отображением цепочек узлов, описываются в [примерах выполнения заданий на обработку динамических структур](#).



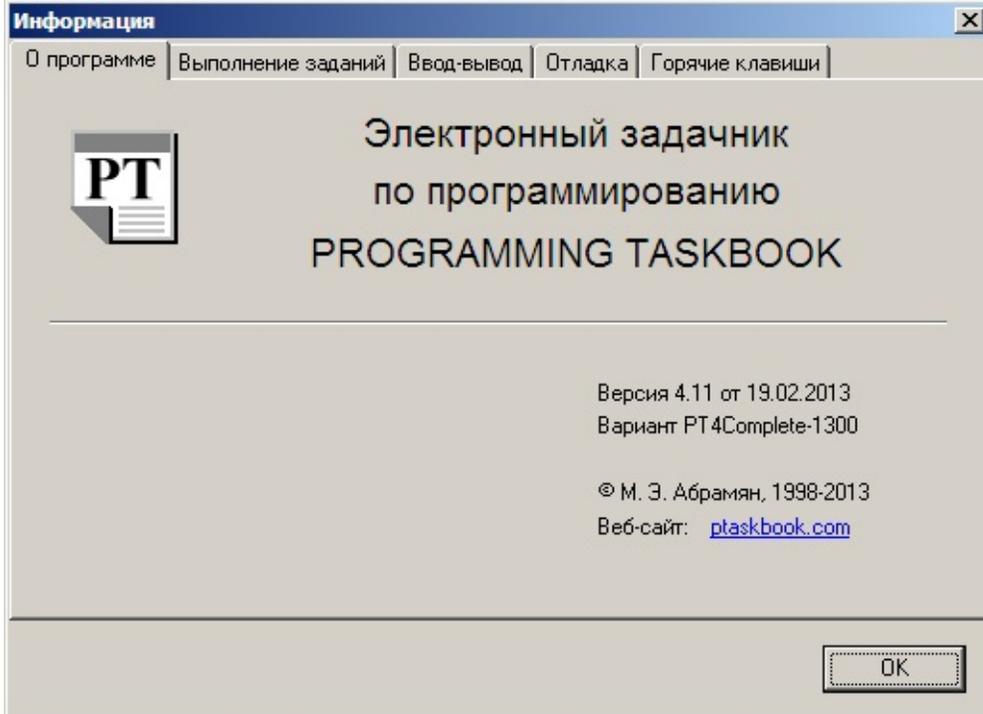
Имеется еще один особый вид исходных и результирующих данных: *бинарные деревья*; данные этого вида описываются в [примерах выполнения заданий группы Tree](#). Если глубина бинарного дерева превышает количество экранных строк, отведенных для его отображения, то для данного дерева также доступна прокрутка.

Если программа запущена в [демонстрационном режиме](#), то на месте информационной панели отображаются дополнительные кнопки, позволяющие быстро перейти к предыдущему или следующему заданию данной группы или просмотреть новый набор исходных и контрольных данных. При демонстрационном, как и при ознакомительном запуске, активной является вкладка «Пример верного решения».

В версии 4.9 окно задачника было дополнено [разделом отладки](#), который отображается на экране, если программа учащегося при выполнении задания выводит отладочную информацию.

Предусмотренные в задачнике отладочные процедуры подробно описываются в разделе, посвященном [типам и процедурам модуля РТ4](#).

Если окно задачника является активным, то нажатие клавиши **F1** (или кнопки «?» в правой части заголовка окна) приводит к отображению на экране *информационного окна*. Начиная с версии 4.9, данное окно содержит набор вкладок:



Дополнения, реализованные в версии 4.11

В версии 4.11 интерфейс окна задачника был дополнен рядом новых возможностей. В данном пункте описываются те из них, которые реализованы для любого режима окна (как с фиксированной, так и с динамической компоновкой).

Расширенное цветовое выделение ошибок

Наряду со стандартным красным цветом, свидетельствующим об ошибочном решении, для фона информационной панели используются три дополнительных оттенка красного цвета, которые связываются с тремя видами ошибок ввода-вывода.

1. Введено или выведено недостаточно данных (оранжевый фон):

Введены не все требуемые исходные данные.
Количество прочитанных данных: 0 (из 1).

2. Попытка ввести или вывести лишние данные (малиновый фон):

Попытка ввести лишние исходные данные.

3. Попытка ввести или вывести данные неверного типа (фиолетовый фон):

Неверно указан тип при вводе исходных данных.
Для ввода 1-го элемента (вещественного типа) использована переменная целого типа.

4. Прочие ошибки (красный фон):

Ошибочное решение.

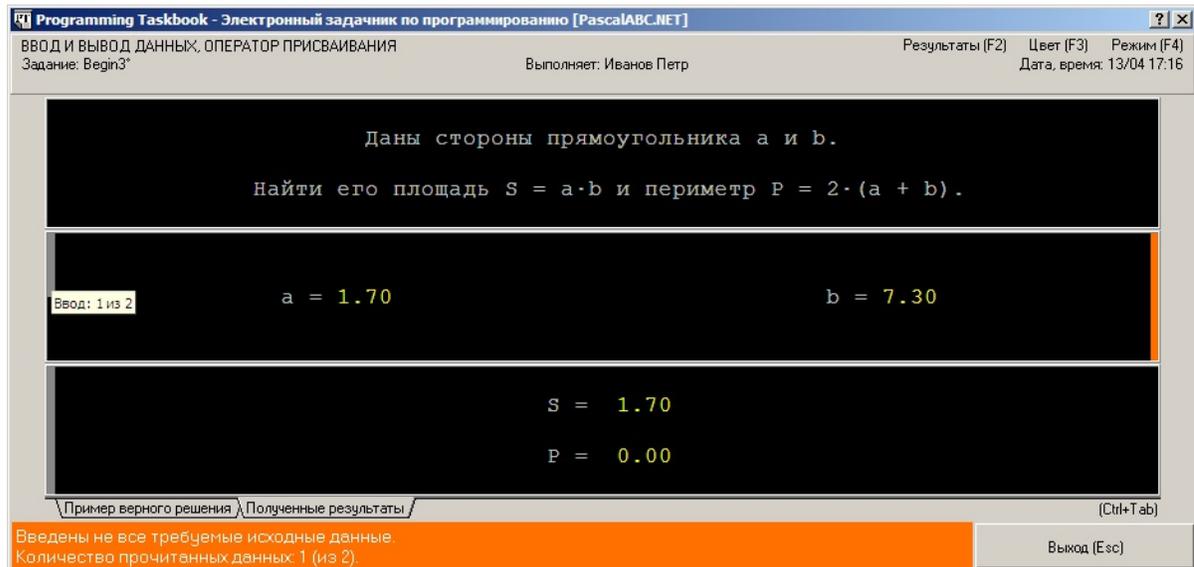
Индикаторы ввода-вывода и прогресса выполнения задания

В левой и правой части раздела исходных данных и раздела результатов окна с фиксированной компоновкой выделено место для отображения дополнительных *индикаторов* в виде вертикальных цветных полос (аналогичные индикаторы для окна с динамической компоновкой являются горизонтальными и располагаются на специальной *панели индикаторов* —

[см. следующий пункт](#)). Индикаторы в левой части разделов показывают долю введенных и выведенных данных (по отношению к общему количеству исходных и результирующих данных). При вводе и выводе всех требуемых данных полоса соответствующего индикатора занимает весь раздел по вертикали; при вводе/выводе части данных высота полосы

соответствует размеру этой части. При наведении мыши на индикатор во всплывающей подсказке отображается точная информация о числе введенных/выведенных данных, например, «Ввод: 2 из 5» или «Вывод: 4 из 8».

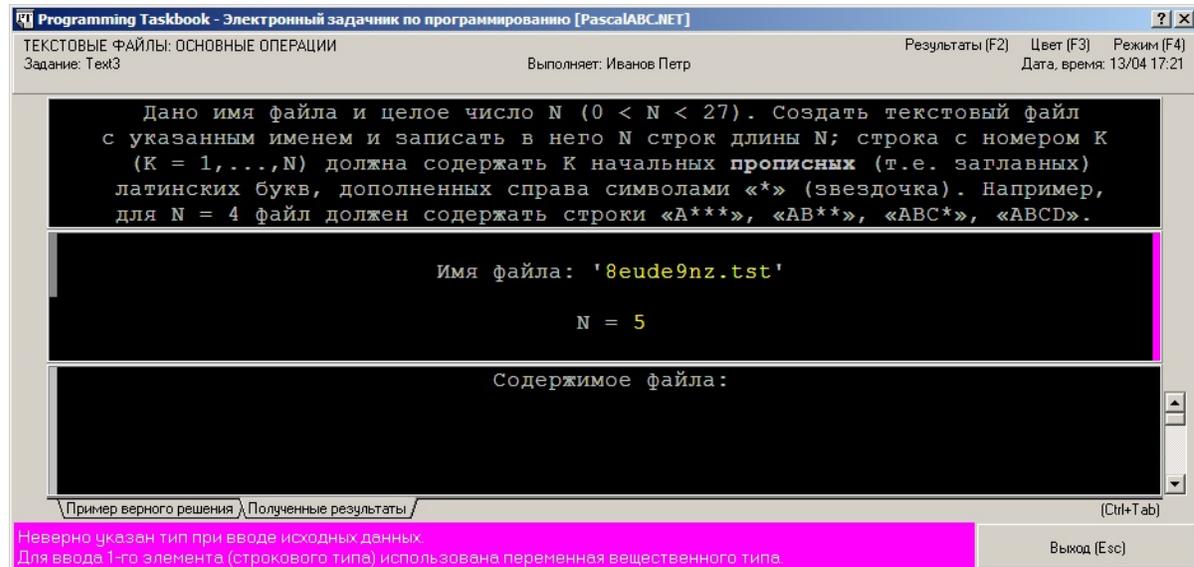
Индикаторы имеют темно-серый цвет.



В случае, когда в задании не требуется пересылать данные непосредственно задачику или не требуется получать от него исходные данные, индикаторы всегда занимают весь раздел по вертикали и при этом имеют светло-серый цвет. Для индикаторов такого типа подсказка не выводится. Примерами заданий, в которых не требуется пересылать данные задачику, являются многие задания на преобразование файлов и деревьев и все задания, связанные с ЕГЭ. Задания, связанные с ЕГЭ, являются также примерами заданий, в которых не требуется получать исходные данные от задачника (данные должны считываться из входного файла, который связывается со стандартным потоком ввода).

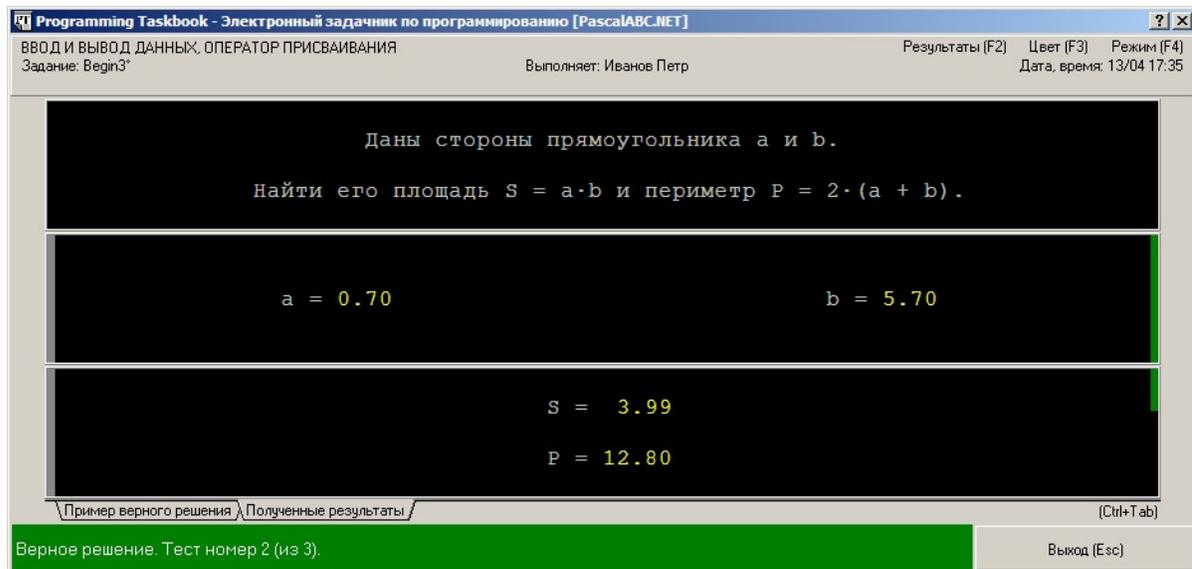
В правой части разделов с входными и выходными данными могут отображаться индикаторы ошибок ввода-вывода и прогресса выполнения задания. Индикатор ошибки, связанной с вводом, размещается в правой части раздела входных данных, а индикатор ошибки, связанной с выводом, — в правой части раздела выходных данных. Он всегда занимает весь раздел по вертикали; его цвет соответствует типу ошибки (этот же цвет используется и для фона

информационной панели). В то время как цвет определяет *характер ошибки* (недостаточно данных, избыточное число данных, данные неверного типа), расположение индикатора показывает, к какой *категории данных* (входных или выходных) эта ошибка относится. С индикаторами ошибок ввода-вывода не связываются подсказки, так как подробные сведения об обнаруженной ошибке выводятся на информационной панели.



В правой части разделов с входными и выходными данными отображается также индикатор прогресса выполнения задания. Он имеет зеленый цвет и связывается с обоими разделами. Высота индикатора прогресса зависит от количества успешных тестовых испытаний программы. В случае выполнения требуемого числа испытаний этот индикатор заполняет по высоте оба раздела. Таким образом, при успешном выполнении задания разделы исходных и результирующих данных полностью обрамляются и слева и справа: обрамление слева (серого цвета) означает, что были введены и выведены все требуемые данные, обрамление справа (зеленого цвета) означает, что были успешно пройдены все тестовые испытания. В подсказке к индикатору прогресса выводится текст «Тесты:», после которого указывается число успешно пройденных тестов и число тестов, которые необходимо пройти для того, чтобы задание было зачтено как выполненное.

Ниже приведен вид окна задачника при успешном выполнении второго теста (из трех требуемых).



При выполнении заданий по параллельному программированию индикаторы ввода-вывода отображают общее число данных, введенных и выведенных всеми процессами. В этом случае подсказки к индикаторам ввода-вывода дополнительно содержат информацию о том, сколько данных было введено (или, соответственно, выведено) каждым из процессов, использованных в задании.

Изменение режима окна, цветовой схемы и отключение отображения даты/времени

Клавиша **F4** и кнопка быстрого доступа «Режим», размещенная в правой верхней части окна задачника, позволяют переключаться между двумя режимами окна: с фиксированной и динамической компоновкой. Режим с динамической компоновкой появился в версии 4.11, он подробно описывается в [следующем пункте](#).

Клавиша **F3** и кнопка быстрого доступа «Цвет», размещенная рядом с кнопкой переключения режима окна, позволяют переключаться между двумя цветовыми схемами окна задачника: традиционной (в стиле консольного окна: светлые символы на черном фоне) и появившейся в версии 4.11 (темные символы на белом фоне). Элементы исходных и результирующих данных, которые в традиционной «черной» схеме отображаются желтым цветом, в новой, «белой» схеме отображаются синим цветом. Цвет «внешних» данных (элементов файлов и динамических структур) в обеих

схемах является одинаковым (бирюзовым); изменяется лишь его яркость.

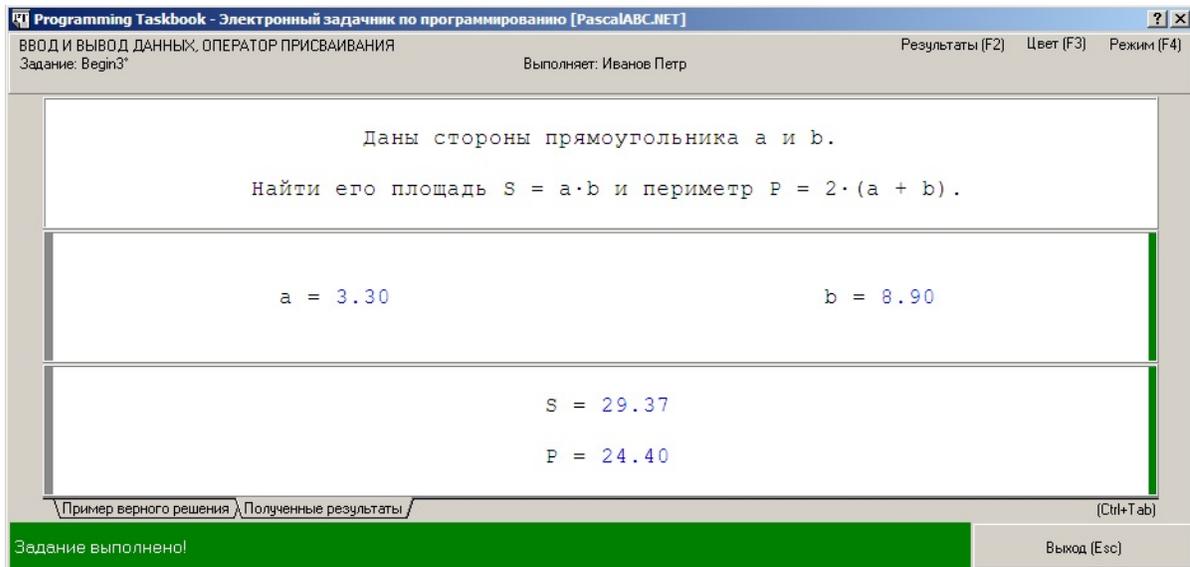
Цвета индикаторов подобраны таким образом, чтобы нормально восприниматься в любой цветовой схеме.

«Белая» схема является менее контрастной и поэтому проигрывает в наглядности по сравнению с «черной» схемой; в то же время «белая» схема позволяет создавать более наглядные скриншоты окон задачника в печатных пособиях.

Еще одной возможностью, связанной исключительно с удобством оформления учебных материалов, является скрытие даты/времени в верхней левой части окна задачника, которое может выполняться при нажатии комбинации **Ctrl+D** (клавиша работает как переключатель). Эта возможность предназначена, в основном, для преподавателя, готовящего презентации или пособия, связанные с задачником. Скрытие даты/времени позволяет не следить за согласованием даты/времени при подготовке серии скриншотов с окнами задачника и, кроме того, дает возможность скрыть информацию о дате и времени подготовки скриншотов.

В режиме с динамической компоновкой для скрытия/отображения даты и времени можно также использовать соответствующую команду контекстного меню окна задачника.

Все описанные настройки (выбранный режим окна, цветовая схема и режим отключения даты/времени) запоминаются в файле настроек pt4.ini в текущем каталоге и при последующих запусках программ с заданиями восстанавливаются автоматически.



Быстрый просмотр результатов

Для вызова [программного модуля PT4Results](#) предусмотрена кнопка быстрого доступа «Результаты» в правой верхней части окна задачника и связанная с ней клавиша **F2**. В демо-режиме эта возможность отключена (поскольку модуль PT4Demo может быть запущен не из рабочего каталога, а из системного каталога задачника, с которым не связывается информация о результатах какого-либо учащегося). Запустить можно лишь одну копию модуля PT4Results; на время работы с этим модулем кнопка «Результаты» делается недоступной.

Режим с динамической компоновкой

Особенности режима с динамической компоновкой

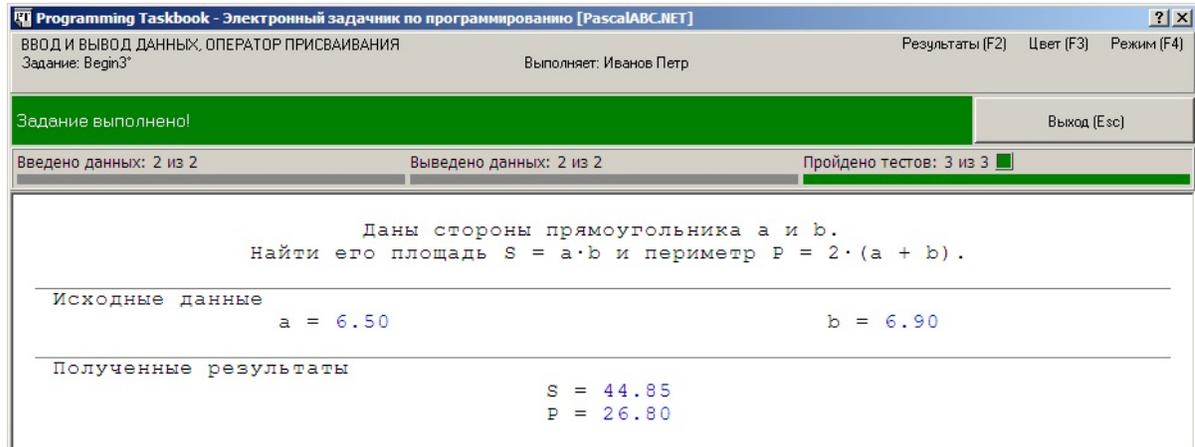
Как было отмечено выше, с помощью кнопки быстрого доступа «Режим» и связанной с ней клавиши **F4** можно осуществлять переключение между двумя режимами окна задачника: традиционного *режима с фиксированной компоновкой*, в котором разделы с формулировкой, исходными данными и результатами имеют фиксированный размер (5 экранных строк) и допускают независимую прокрутку, и появившегося в версии 4.11 *режима с динамической компоновкой* (в стиле окна задачника для веб-среды ProgrammingABC.NET WDE), в котором размеры разделов определяются содержащимися в них данными, состав разделов зависит от результата тестового запуска, и для всего содержимого окна выполняется, при необходимости, общая прокрутка.

Подобно цветовой схеме и другим настройкам окна, выбранный вариант режима запоминается в файле pt4.ini, и при последующих запусках программ автоматически восстанавливается. Начиная с версии 4.11, в качестве режима по умолчанию установлен режим с динамической компоновкой.

В режиме с динамической компоновкой управляющие элементы и информационная панель располагаются в верхней части окна, поскольку высота окна для различных заданий (и даже для различных запусков программы с одним и тем же заданием) может изменяться.

Окно в режиме с динамической компоновкой может содержать от двух до четырех разделов в различных сочетаниях: раздел формулировки (без заголовка), раздел исходных данных (с заголовком «Исходные данные»), раздел с результатами (с заголовком «Полученные результаты») и раздел с образцом правильного решения (с заголовком «Пример верного решения»). Разделы с формулировкой и исходными данными отображаются при любом варианте запуска. В демонстрационном режиме и при ознакомительном запуске раздел с образцом правильного решения отображается, а (пустой) раздел с результатами — нет. В случае успешного тестового запуска раздел с образцом правильного

решения не отображается, так как образец в этом случае совпадает с полученными результатами. Все четыре раздела отображаются в случае, когда при тестовом испытании была выявлена какая-либо ошибка.



В режиме с динамической компоновкой все [индикаторы](#) (ввода-вывода и прогресса выполнения задания) вынесены на специальную *панель индикаторов*, размещенную под информационной панелью. Панель индикаторов не отображается в демо-режиме, а также при ознакомительном запуске. Помимо собственно индикаторов, которые в режиме с динамической компоновкой располагаются горизонтально, на панели индикаторов размещаются дополнительные цветовые метки. Зеленая метка появляется над индикатором прогресса выполнения задания в случае, когда успешно пройдены все требуемые тесты. Метки с различными оттенками красного цвета отображаются при выявлении ошибок; их размещение зависит от характера ошибки: если это ошибка ввода, то метка отображается над индикатором ввода, если это ошибка вывода, то над индикатором вывода, если же это ошибка другого вида, то метка размещается над индикатором прогресса выполнения. Таким образом, при наличии любой ошибки в разделе индикаторов будет указана соответствующая метка. Поскольку, помимо индикаторов, панель содержит поясняющий текст, дополнительные подсказки к индикаторам не требуются. Исключение составляют задания по параллельному программированию, для которых предусмотрены подсказки, связанные с индикаторами ввода-вывода и поясняющим текстом: подсказки содержат информацию о том, сколько было введено или выведено данных в

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ГЛОБАЛЬНОЕ ВЫРАВНИВАНИЕ СТРОК
 Задача: Align31*
 Демо-запуск: Иванов Петр
 Цвет (F3) Режим (F4)

Новые данные (Space) Предыдущее задание (BS) Следующее задание (Enter) Выход (Esc)

Даны строки S_1 и S_2 . Кроме того, дано число m (<10) – размер алфавита, состоящего из первых строчных латинских букв, и матрица оценок (структура матрицы описана в Align29). Для эффективного вычисления сходства строк $V(|S_1|, |S_2|)$ по формулам из Align30 можно использовать **метод восходящей рекурсии**, основанный на последовательном вычислении элементов вспомогательной матрицы V размера $(|S_1|+1) \times (|S_2|+1)$ (действия при заполнении матрицы V аналогичны действиям при заполнении матрицы D , описанной в Align5). Элемент, расположенный в последней строчке и последнем столбце матрицы V , определяет значение сходства данных строк. Используя метод восходящей рекурсии, найти матрицу V и вывести элементы ее двух последних строчек: $V_{n-1, j}$, $j=0, \dots, |S_2|$, $V_{n, j}$, $j=0, \dots, |S_2|$, где $n=|S_1|$.

Исходные данные

```

S1 = 'beffbehigbbfcbd'
S2 = 'begffbggaibgehfcdb'
      |1 |5 |1|0 |1|5
Верхняя треугольная часть симметричной матрицы оценок:
      «a» «b» «c» «d» «e» «f» «g» «h» «i» « »
      «a»  2  -9  -4  -4  -9  -9  -8  -1  -2  -8
      «b»  7  -7  -1  -5  -7  -9  -8  -8  -7
      «c»          9  -6  -2  -1  -9  -2  -7  -4
      «d»                0  -1  -2  -7  -3  -6  -9
      «e»                      3  -2  -1  -8  -6  -4
      «f»                            9  -1  -9  -3  -7
      «g»                                  3  -3  -1  -7
      «h»                                        7  -5  -7
      «i»                                              2  -4
      « »                                                    0
    
```

Пример верного решения

```

      0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
      V_{n-1, j}, j = 0, ..., |S2|:
-90 -76 -69 -59 -43 -31 -17 -20 -14 -12 -12  2  5  7  6  5  16  30  21
      V_{n, j}, j = 0, ..., |S2|:
-99 -85 -77 -68 -52 -40 -26 -24 -23 -18 -18 -7 -4  4  4  4  7  21  30
    
```

В приведенном ниже окне с фиксированной компоновкой для этого же задания требуется прокрутка как для формулировки, так и для раздела исходных данных.

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ГЛОБАЛЬНОЕ ВЫРАВНИВАНИЕ СТРОК
 Задача: Align31*
 Демо-запуск: Иванов Петр
 Цвет (F3) Режим (F4)

1: (-) Даны строки S_1 и S_2 . Кроме того, дано число m (<10) – размер алфавита, состоящего из первых строчных латинских букв, и матрица оценок (структура матрицы описана в Align29). Для эффективного вычисления сходства строк $V(|S_1|, |S_2|)$ по формулам из Align30 можно использовать **метод восходящей рекурсии**, основанный на последовательном вычислении элементов

1: $S_1 = 'beffbehigbbfcbd'$
 $S_2 = 'begffbggaibgehfcdb'$
 $m = 9$
 $|1 |5 |1|0 |1|5$
 Верхняя треугольная часть симметричной матрицы оценок: $\begin{matrix} \text{«a»} & 2 & -9 & -4 & -4 & -9 & -9 & -8 & -1 & -2 & -8 \\ \text{«b»} & & 7 & -7 & -1 & -5 & -7 & -9 & -8 & -8 & -7 \end{matrix}$

$0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \ 17 \ 18$
 $V_{n-1, j}, j = 0, \dots, |S_2|:$
 $-90 \ -76 \ -69 \ -59 \ -43 \ -31 \ -17 \ -20 \ -14 \ -12 \ -12 \ 2 \ 5 \ 7 \ 6 \ 5 \ 16 \ 30 \ 21$
 $V_{n, j}, j = 0, \dots, |S_2|:$
 $-99 \ -85 \ -77 \ -68 \ -52 \ -40 \ -26 \ -24 \ -23 \ -18 \ -18 \ -7 \ -4 \ 4 \ 4 \ 4 \ 7 \ 21 \ 30$

(+) Пример верного решения / Полученные результаты / (Ctrl+Tab)

Новые данные (Space) Предыдущее задание (BS) Следующее задание (Enter) Выход (Esc)

Если часть разделов с заданием не умещается на экране, то у правой границы окна отображается полоса прокрутки, а нижняя граница окна автоматически подстраивается под размеры экрана и располагается выше панели задач. При перемещении окна вверх в данной ситуации высота окна автоматически увеличивается до тех пор, пока все его содержимое не будет отображаться в окне; при этом полоса прокрутки опять будет скрыта. Для перемещения верхней границы окна вверх и вниз можно использовать клавиатурные комбинации **Ctrl+Up** и **Ctrl+Down** соответственно.

Если в режиме с динамической компоновкой отображается полоса прокрутки, то прокручивать данные можно как с ее помощью, так и с помощью колесика мыши, а также клавишами **Home**, **End**, **PgUp**, **PgDn**, **Up** и **Down**.

Окно в режиме с динамической компоновкой (в отличие от окна в режиме с фиксированной компоновкой) можно масштабировать, изменяя размер шрифта в его основных разделах (с формулировкой задания, исходными данными, результатами и примером правильного решения). Для масштабирования можно использовать команды контекстного меню или клавиши **Ctrl+[+]** и **Ctrl+[-]**. Ниже приведен вид окна с минимальным размером шрифта (равным 8 пунктам). В режиме с динамической компоновкой сохраняется возможность изменения шрифта в разделе отладки (с помощью команд контекстного меню или клавиш **Alt+[+]** и **Alt+[-]**); подобное изменение не влияет на размеры окна.

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? x

ГЛОБАЛЬНОЕ ВЫРАВНИВАНИЕ СТРОК
 Задание: Align31* Демон-запуск: Иванов Петр Цвет (F3) Режим (F4)

Новые данные (Space) Предыдущее задание (BS) Следующее задание (Enter) Выход (Esc)

Даны строки S_1 и S_2 . Кроме того, дано число m (<10) – размер алфавита, состоящего из первых строчных латинских букв, и матрица оценок (структура матрицы описана в Align29). Для эффективного вычисления сходства строк $V(|S_1|, |S_2|)$ по формулам из Align30 можно использовать метод восходящей рекурсии, основанный на последовательном вычислении элементов вспомогательной матрицы V размера $(|S_1|+1) \times (|S_2|+1)$ (действия при заполнении матрицы V аналогичны действиям при заполнении матрицы D , описанным в Align5). Элемент, расположенный в последней строчке и последнем столбце матрицы V , определяет значение сходства данных строк. Используя метод восходящей рекурсии, найти матрицу V и вывести элементы ее двух последних строчек: $V_{n-1, j}, j=0, \dots, |S_2|, V_{n, j}, j=0, \dots, |S_2|$, где $n=|S_1|$.

Исходные данные

```

S1 = 'beffbehigbbficbd'
S2 = 'begffbggaibgehfcdbd'
m = 9
  |1| 5| 1|0| 1|5
Верхняя треугольная часть
симметричной матрицы оценок:

```

	«a»	«b»	«c»	«d»	«e»	«f»	«g»	«h»	«i»	« »
«a»	2	-9	-4	-4	-9	-9	-8	-1	-2	-8
«b»		7	-7	-1	-5	-7	-9	-8	-8	-7
«c»			9	-6	-2	-1	-9	-2	-7	-4
«d»				0	-1	-2	-7	-3	-6	-9
«e»					3	-2	-1	-8	-6	-4
«f»						9	-1	-9	-3	-7
«g»							3	-3	-1	-7
«h»								7	-5	-7
«i»									2	-4
« »										0

Пример верного решения

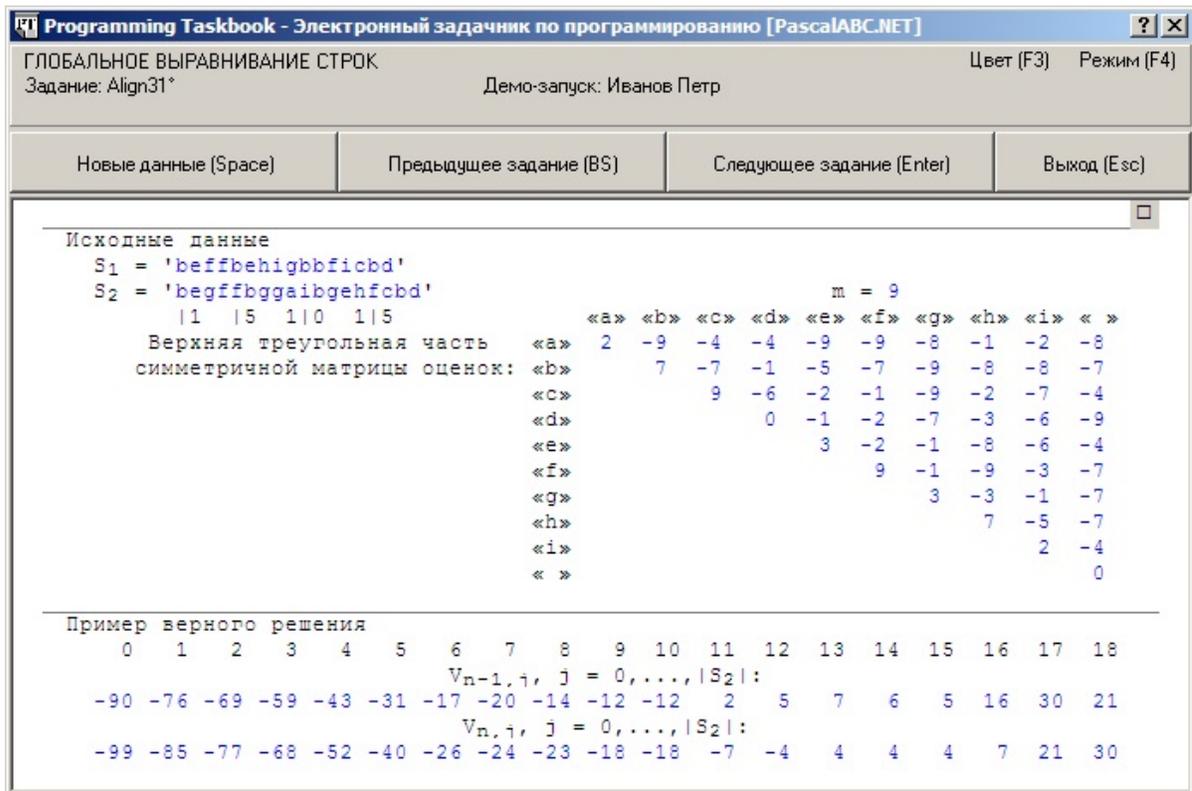
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
$V_{n-1, j}, j = 0, \dots, S_2 :$																				
	-90	-76	-69	-59	-43	-31	-17	-20	-14	-12	-12	2	5	7	6	5	16	30	21	
$V_{n, j}, j = 0, \dots, S_2 :$																				
	-99	-85	-77	-68	-52	-40	-26	-24	-23	-18	-18	-7	-4	4	4	4	7	21	30	

Текущая позиция верхней границы окна в режиме с динамической компоновкой и текущие размеры шрифта сохраняются в файле настроек pt4.ini и при последующем запуске автоматически восстанавливаются. Однако это сохранение производится только в случае, если при закрытии окна оно находилось в режиме с динамической компоновкой. В противном случае (если при запуске программы окно отображается в режиме с фиксированной компоновкой) для позиции верхней границы окна и размера шрифта в основных разделах устанавливаются стандартные значения, принятые для режима с фиксированной компоновкой.

Дополнительная настройка разделов с заданием

В режиме с динамической компоновкой можно скрыть формулировку

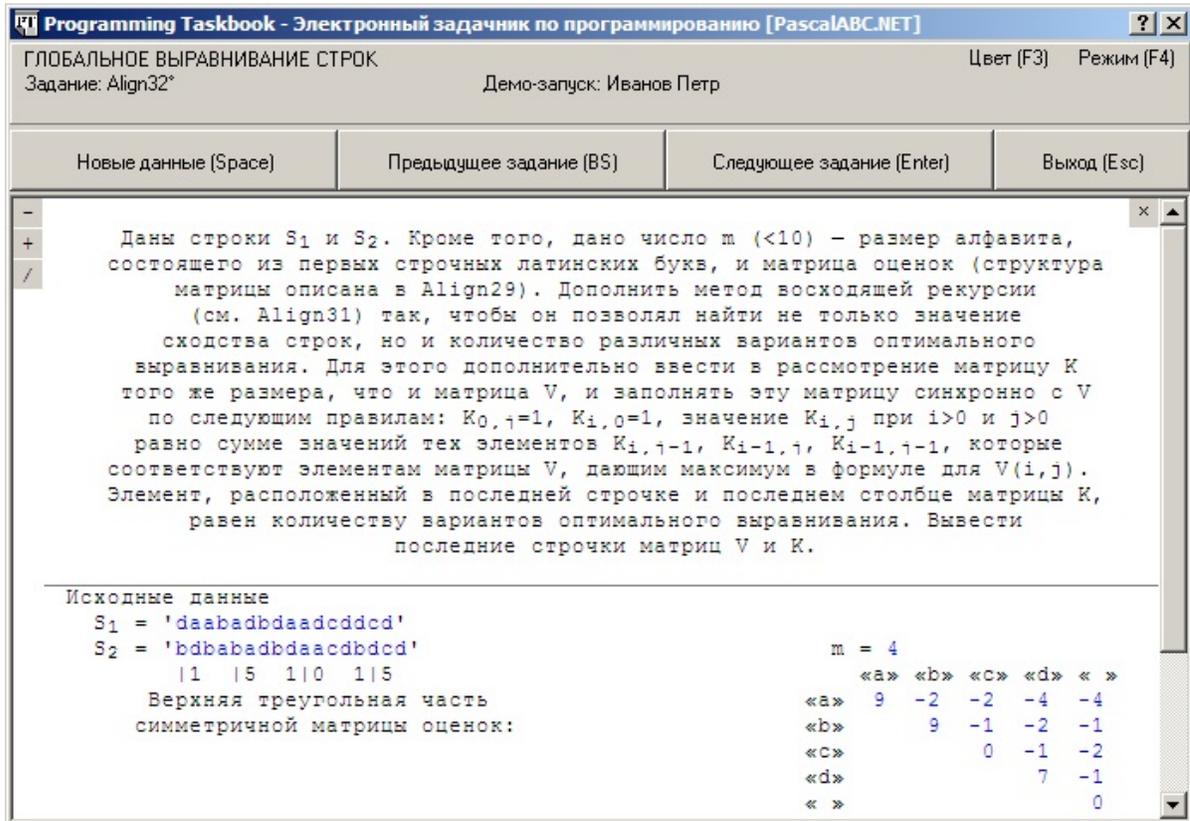
задания, чтобы обеспечить отображение других элементов задания в окне меньшей высоты. Для скрытия формулировки задания достаточно щелкнуть мышью в любом месте раздела с формулировкой задания или нажать клавишу **Del**. Если формулировка задания скрыта, то раздел с формулировкой уменьшается до одной (пустой) экранной строки, причем в правой части этого раздела отображается кнопка, позволяющая вновь отобразить формулировку задания (с данной кнопкой связана всплывающая подсказка «Показать формулировку задания (Del)»):



Для восстановления формулировки, помимо нажатия на кнопку, доступны те же действия, которые обеспечивают ее скрытие: щелчок в любом месте раздела с формулировкой и нажатие клавиши **Del**.

Как правило, необходимость в скрытии формулировки задания возникает в ситуации, когда часть данных, связанных с заданием, не отображается на экране (при этом окно снабжается полосой прокрутки). В подобной ситуации предусмотрен еще один способ скрытия раздела с формулировкой: с помощью кнопки, отображаемой в правом верхнем углу этого раздела (рядом с полосой прокрутки). На кнопке отображается символ «x», и с ней

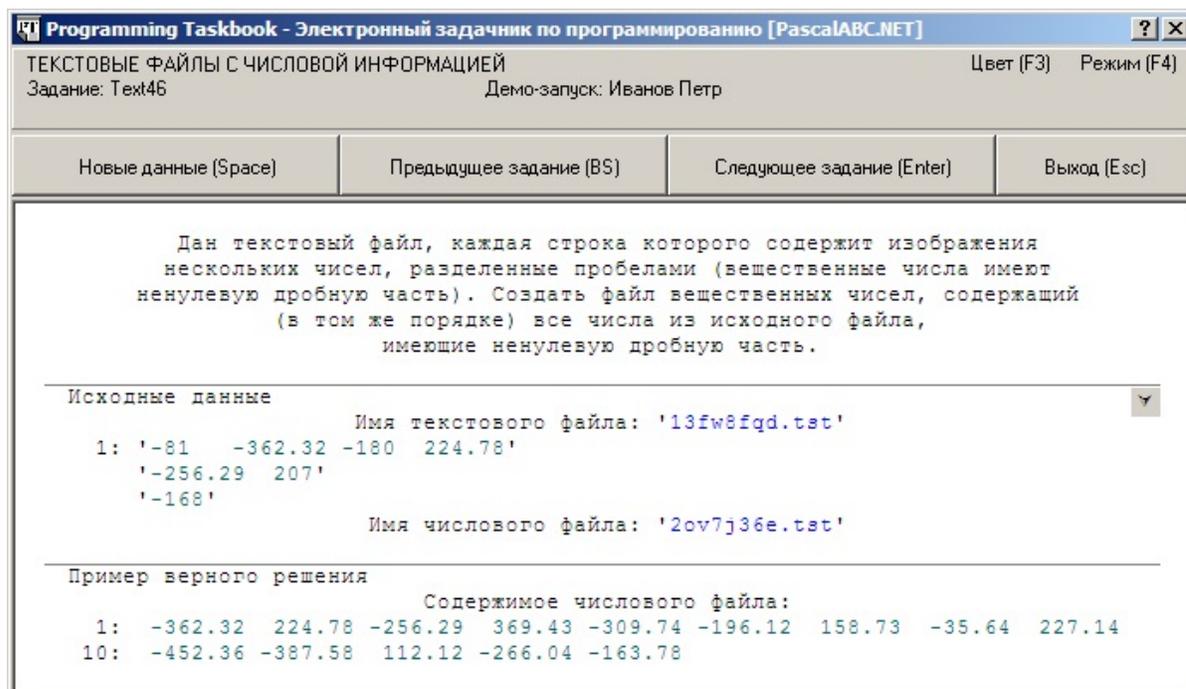
связывается всплывающая подсказка «Скрыть формулировку задания (Del)»:



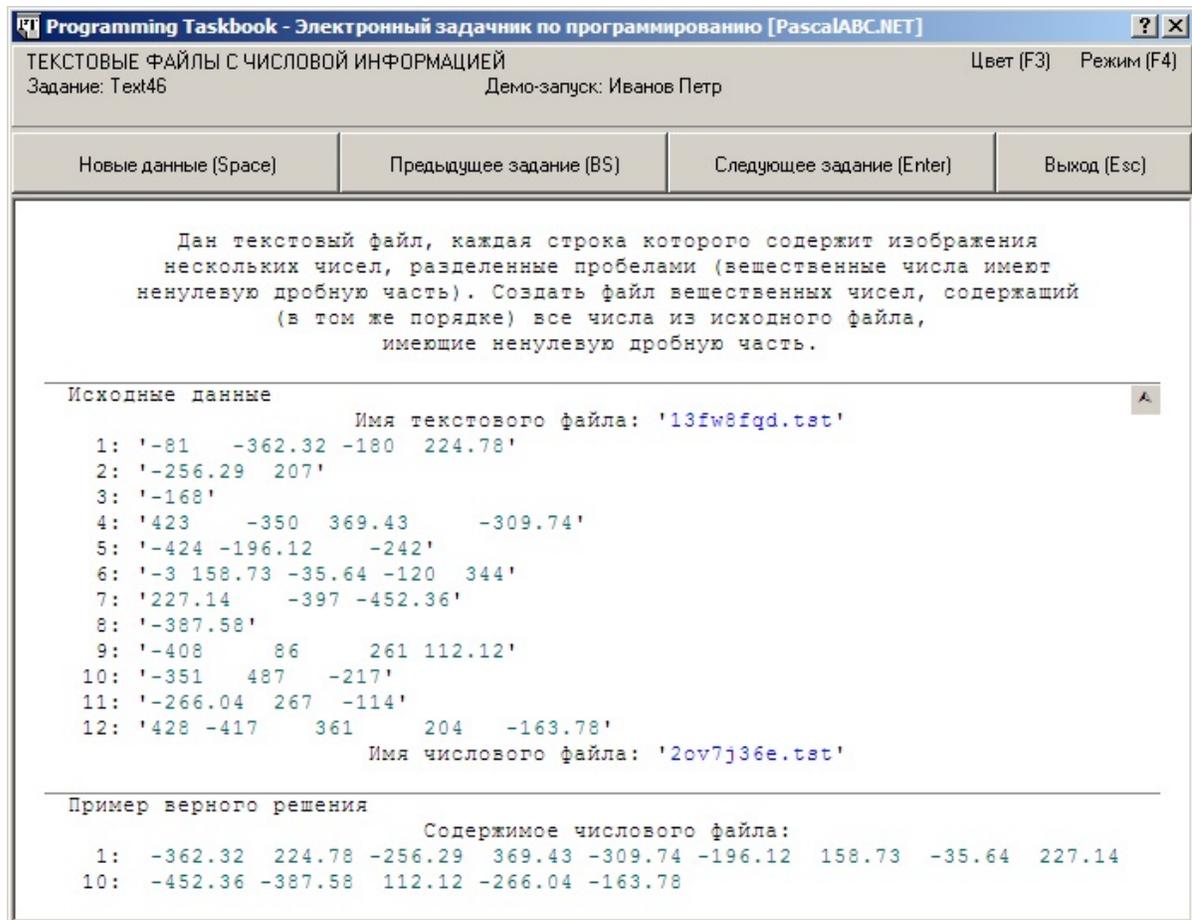
Если закрыть окно (не в демо-режиме) при скрытой формулировке, то при последующем запуске (также не в демо-режиме) формулировка останется скрытой. При начальном отображении каждого задания в демо-режиме формулировка всегда выводится; если ее скрыть, то она будет оставаться скрытой при последующих просмотрах других вариантов исходных и контрольных данных для этого задания (варианты перебираются кнопкой «Новые данные»).

В режиме с динамической компоновкой все прокручиваемые элементы данных (формулировка, элементы двоичных файлов, элементы бинарных деревьев, разделы исходных и результирующих данных, содержащие более 5 строк) отображаются полностью. Частично отображаются только элементы текстовых файлов. По умолчанию выводится столько же строк текстового файла, сколько и в режиме с фиксированной компоновкой; при этом первая строка файла снабжается номером, равным 1. Такой подход позволяет ознакомиться с заданием, отобразив его в окне сравнительно

небольшого размера:



Если в режиме с динамической компоновкой отображается только часть данных из текстовых файлов, то в правом верхнем углу раздела с исходными данными появляется кнопка со стилизованной стрелкой, направленной вниз. С ней связывается подсказка «Развернуть содержимое текстовых файлов (Ins)». Щелкнув на этой кнопке, нажав клавишу **Ins** или выполнив щелчок мышью на одном из разделов, связанных с заданием (кроме раздела с формулировкой), можно отобразить файловые данные в полном объеме; при этом около каждой файловой строки появится ее номер, а на кнопке изменится изображение: вместо стрелки, направленной вниз, будет изображена стрелка, направленная вверх:



Режим полного отображения текстовых файлов удобен для более тщательного изучения особенностей исходных и полученных файлов, а также для сравнения полученного ошибочного файла с файлом, указанным в примере правильного решения.

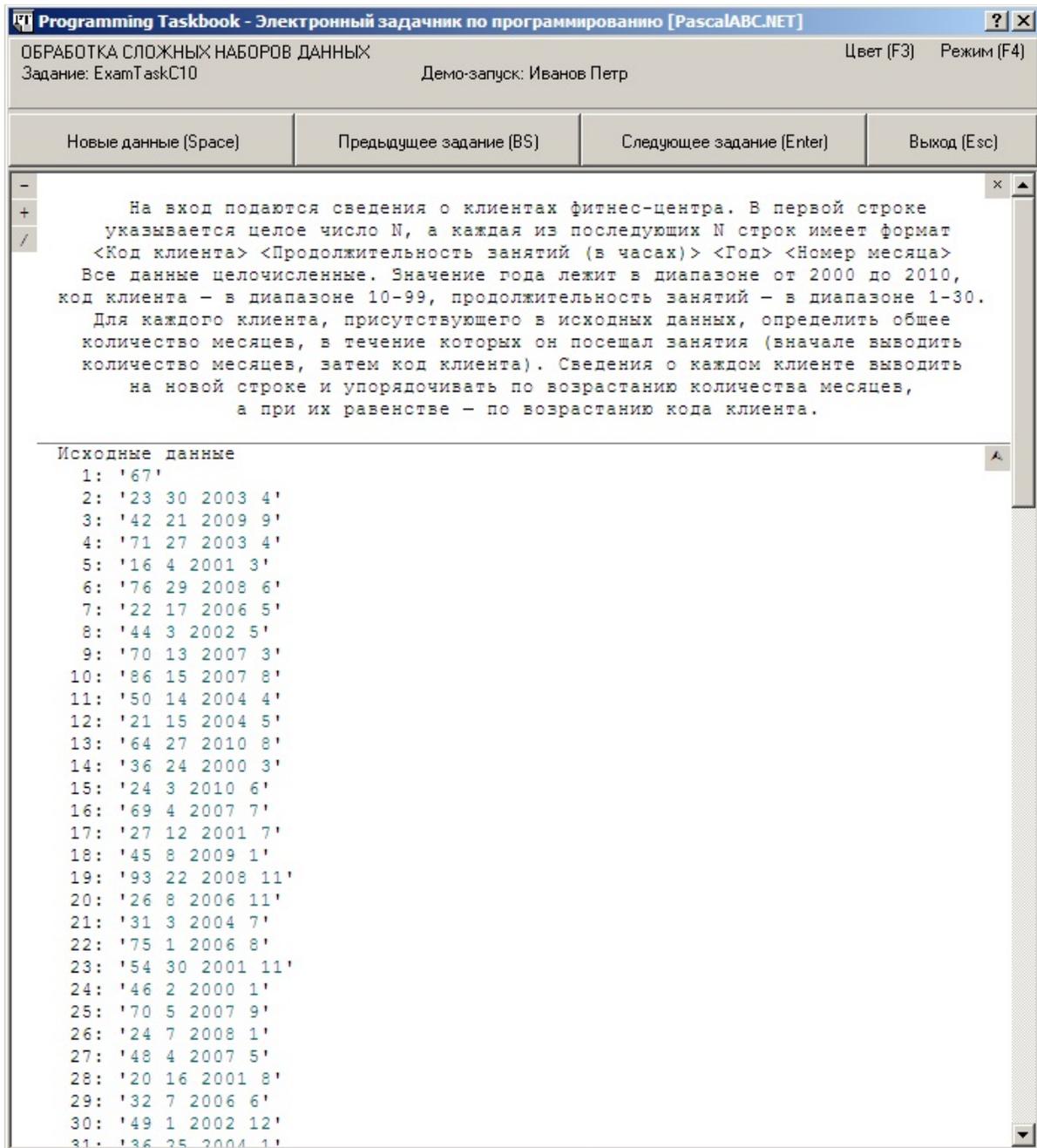
Если закрыть окно задачника (не в демо-режиме) при развернутом содержимом текстовых файлов, то при последующем запуске (также не в демо-режиме) это содержимое будет развернуто автоматически. При начальном отображении каждого задания в демо-режиме данные текстовых файлов всегда выводятся в свернутом виде; если их развернуть, то режим развернутого отображения будет сохраняться при последующих просмотрах других вариантов исходных и контрольных данных для этого задания (варианты перебираются кнопкой «Новые данные»).

Действия по разворачиванию/сворачиванию содержимого текстовых файлов можно выполнять и для файлов небольшого размера, когда даже в свернутом режиме на экране отображаются все файловые

строки. Однако в этом случае кнопка со стрелкой в окне задачника не появляется, и для переключения между режимами свернутого и развернутого отображения файловых данных следует нажимать клавишу **Ins** или выполнять щелчок мышью на разделе с заданием. Перейти в режим развернутого отображения содержимого для небольших файлов может потребоваться, например, для того, чтобы этот режим автоматически восстановился при следующем тестовом запуске программы.

«Интеллектуальная» прокрутка

В ситуации, когда разделы с исходными, результирующими или контрольными данными имеют большую высоту (например, при отображении в полном объеме содержимого текстовых файлов), с помощью стандартной прокрутки сложно обеспечить быстрый переход к началу требуемого раздела. Кроме того, при большом объеме результирующих данных затрудняется их сравнение с контрольными данными. Чтобы решить эти проблемы, в режиме с динамической компоновкой реализована возможность *«интеллектуальной» прокрутки*. Данная возможность доступна, если основные разделы окна (разделы с формулировкой, исходными данными, результатами и примером правильного решения) имеют суммарную высоту, превышающую размер окна. При этом в левом верхнем углу области окна, отводимой для отображения разделов задания, отображаются три дополнительные кнопки с символами «←», «+» и «/»:



Прокрутка содержимого окна не влияет на положение этих кнопок. Нажатие на кнопку «←» или нажатие клавиши [←] обеспечивает прокрутку к началу предыдущего раздела задания, нажатие на кнопку «→» или нажатие клавиши [→] обеспечивает прокрутку к началу следующего раздела задания; при этом перебор разделов выполняется циклически. Если раздел с формулировкой является скрытым, то он при переборе разделов не учитывается.

Нажатие на кнопку «↶» или нажатие клавиши [↶] обеспечивает

переход к началу раздела с результатами или раздела с примером правильного решения, если в окне присутствует только один из этих разделов. Если же окно содержит оба этих раздела, то данная кнопка и связанная с ней клавиша обеспечивают в дальнейшем переключение между этими разделами. При этом выполняется дополнительная синхронизация разделов: новый раздел отображается с той строки, которая соответствует верхней отображаемой строке прежнего раздела. Подобная синхронизация в еще большей степени упрощает сравнение полученных и правильных результатов.

Клавиши **[–]**, **[+]**, **[/]**, связанные с «интеллектуальной» прокруткой (как и клавиши **Ins** и **Del**, связанные с дополнительной настройкой внешнего вида разделов с заданием), располагаются в правой части цифровой клавиатуры и не зависят от режима **NumLock**; подобное расположение делает их удобными для использования в качестве горячих клавиш. Однако при отсутствии цифровой клавиатуры (например, на некоторых моделях ноутбуков) применение данных клавиш становится менее удобным. Чтобы и в этой ситуации упростить выполнение команд с помощью горячих клавиш, предусмотрены их альтернативные варианты: **Ctrl+PgDn** вместо **[+]**, **Ctrl+PgUp** вместо **[–]** и **Ctrl+Tab** вместо **[/]**. Заметим, что подобный способ использования комбинации **Ctrl+Tab** соответствует способу ее использования в режиме с фиксированной компоновкой, в котором она также обеспечивает переключение между полученными результатами и примером правильного решения. Все указанные клавиатурные комбинации приводятся во всплывающих подсказках, которые отображаются на экране при наведении курсора мыши на соответствующие кнопки:

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ОБРАБОТКА СЛОЖНЫХ НАБОРОВ ДАННЫХ Цвет (F3) Режим (F4)
Задание: ExamTaskC10 Демо-запуск: Иванов Петр

Новые данные (Space) Предыдущее задание (BS) Следующее задание (Enter) Выход (Esc)

- Пример верного решения
+ 1: '1 12'
/ 2: '1 16'

Следующий раздел (Ctrl+PgDn или [+])

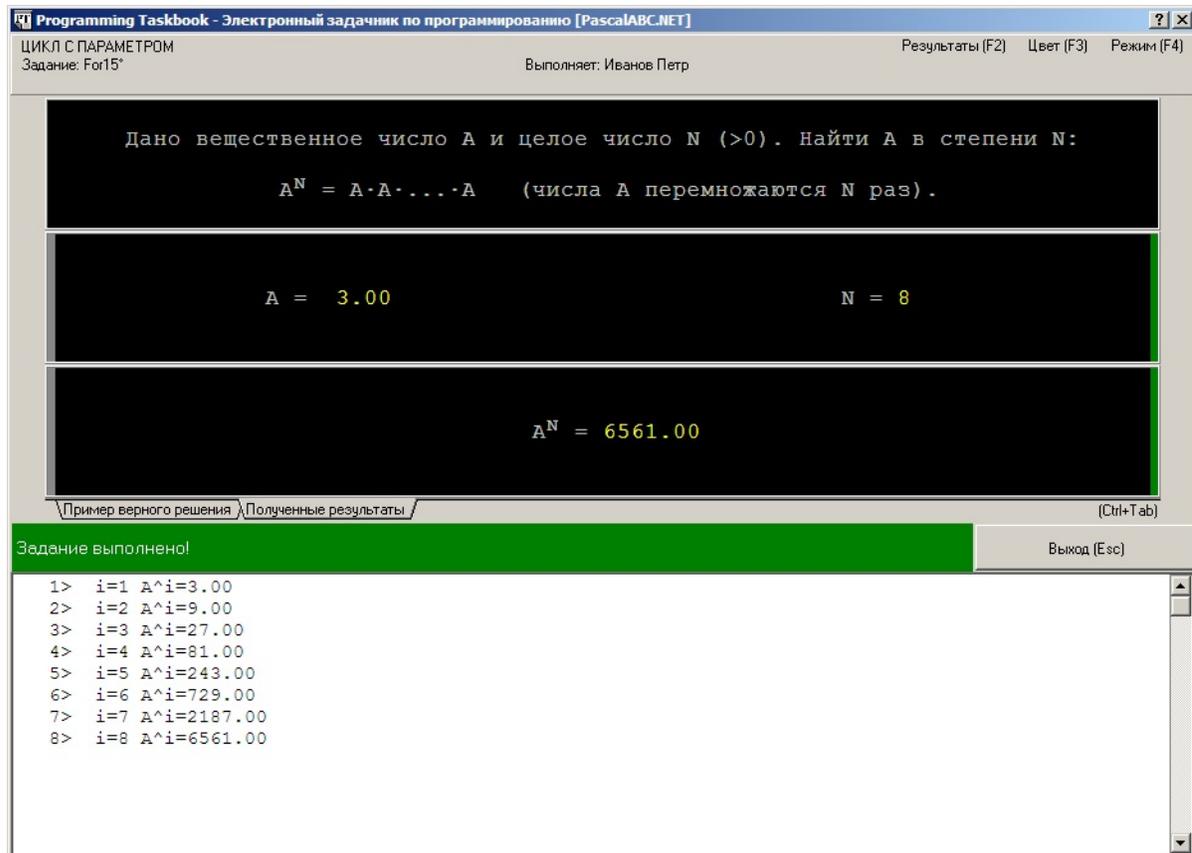
4: '1 21'
5: '1 22'
6: '1 26'
7: '1 29'
8: '1 31'
9: '1 32'
10: '1 35'
11: '1 37'
12: '1 38'
13: '1 42'
14: '1 44'
15: '1 45'
16: '1 46'
17: '1 48'
18: '1 51'
19: '1 58'
20: '1 61'
21: '1 62'
22: '1 69'
23: '1 71'
24: '1 75'
25: '1 78'
26: '1 82'
27: '1 86'
28: '1 88'
29: '1 96'
30: '1 98'
31: '2 20'
32: '2 23'
33: '2 27'
34: '2 36'
35: '2 49'
36: '2 50'
37: '2 54'
38: '2 59'
39: '2 64'
40: '2 70'
41: '2 76'
42: '2 87'
43: '2 88'

Раздел отладки

В версии 4.9 задачника Programming Taskbook появились средства, позволяющие выводить отладочную информацию непосредственно в окно задачника (в специальный *раздел отладки*). Необходимость в подобных дополнительных средствах возникает, прежде всего, при работе с комплексом Programming Taskbook for MPI при отладке *параллельных программ*, поскольку для них нельзя использовать такие стандартные средства отладки, как точки останова, пошаговое выполнение программы и окна просмотра значений переменных. Следует также отметить, что возможность вывода информации в раздел отладки позволяет использовать задачник для написания и отладки параллельных программ, не связанных с выполнением конкретных учебных заданий.

Отладочные средства задачника могут оказаться полезными и для обычных, непараллельных программ. В этом случае их можно применять в качестве дополнения к средствам встроенного отладчика.

Раздел отладки представляет собой одну или несколько многострочных текстовых областей вывода. Он располагается под основными разделами задачника и выводится на экран только в случае, если в нем содержится какой-либо текст:



При демонстрационном запуске программы все процедуры, связанные с разделом отладки, игнорируются, поэтому раздел отладки на экране не отображается.

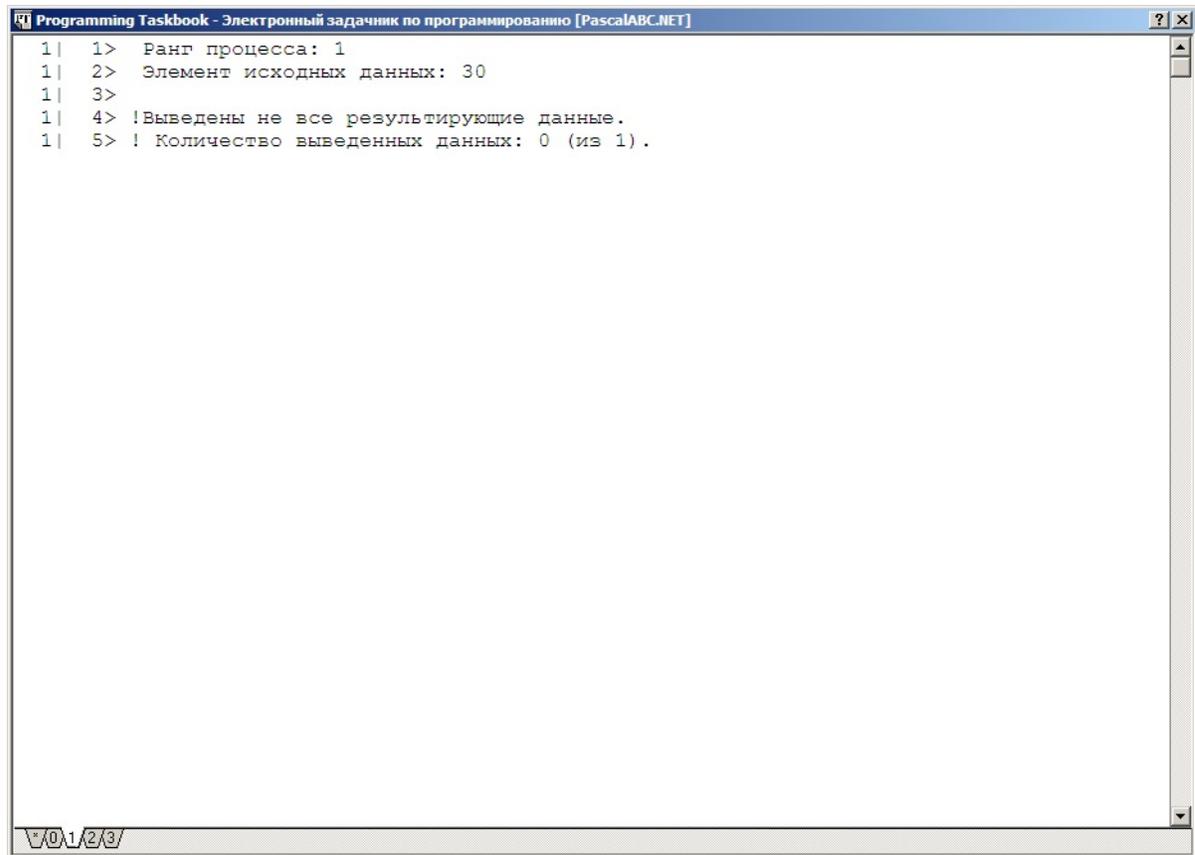
Имеется возможность скрыть в окне задачника все его разделы, кроме раздела отладки; для этого достаточно нажать клавишу пробела. Повторное нажатие пробела восстанавливает в окне задачника ранее скрытые разделы. Для скрытия/отображения основных разделов окна задачника можно также использовать соответствующую команду контекстного меню, связанного с разделом отладки. Скрыть все разделы окна задачника, кроме раздела отладки, можно также программным способом, вызвав процедуру HideTask.

Прокрутка содержимого раздела отладки может осуществляться с помощью вертикальной полосы прокрутки, расположенной у его правой границы. Можно также использовать колесико мыши и клавиатурные комбинации **Alt+Up**, **Alt+Down** (прокрутка на одну экранную строку), **Alt+PgUp**, **Alt+PgDn** (прокрутка на 10 экранных

строк), **Alt+Home**, **Alt+End** (прокрутка к первой или последней строке области вывода).

Предусмотрена возможность изменения размера шрифта, используемого в разделе отладки. Шрифт может изменяться от 7 до 14 пунктов с шагом 1. Для увеличения шрифта предназначена клавиатурная комбинация **Alt+[+]**, для уменьшения — комбинация **Alt+[-]**. Соответствующие команды имеются также в контекстном меню раздела отладки. Информация о текущем размере шрифта сохраняется в файле результатов и учитывается при последующих запусках программы.

Для «непараллельных» заданий раздел отладки содержит единственную область вывода. Для заданий по параллельному программированию число областей вывода равно числу параллельных процессов плюс 1; при этом в каждый момент времени в разделе отладки отображается одна из областей. Если областей вывода больше одной, то в нижней части раздела отладки выводится набор ярлычков, позволяющих переключиться на любую из имеющихся областей вывода:



```
1| 1> Ранг процесса: 1
1| 2> Элемент исходных данных: 30
1| 3>
1| 4> !Выведены не все результирующие данные.
1| 5> ! Количество выведенных данных: 0 (из 1).
```

0/1/2/3

Ярлычки с номерами (от 0 до N–1, где N — количество процессов) позволяют просмотреть содержимое области вывода, связанной с процессом соответствующего ранга; ярлычок с символом «*» позволяет просмотреть область вывода, содержащую объединенный текст всех других областей:

```
0| 1> Ранг процесса: 0
0| 2> Элемент исходных данных: 39
1| 1> Ранг процесса: 1
1| 2> Элемент исходных данных: 30
1| 3>
1| 4> !Выведены не все результирующие данные.
1| 5> ! Количество выведенных данных: 0 (из 1).
2| 1> Ранг процесса: 2
2| 2> Элемент исходных данных: 40
2| 3>
2| 4> !Выведены не все результирующие данные.
2| 5> ! Количество выведенных данных: 0 (из 1).
3| 1> Ранг процесса: 3
3| 2> Элемент исходных данных: 40
3| 3>
3| 4> !Выведены не все результирующие данные.
3| 5> ! Количество выведенных данных: 0 (из 1).
```

Для переключения на нужную область вывода достаточно щелкнуть мышью на соответствующем ярлычке. Кроме того, для последовательного перебора ярлычков слева направо или справа налево можно использовать комбинации **Alt+Right** и **Alt+Left** соответственно (перебор осуществляется циклически). Можно также сразу перейти к нужной области вывода, нажав соответствующую клавишу: для области «*» — клавишу **[*]**, для областей «0»–«9» — цифровые клавиши **0–9**, а для областей «10»–«35» — буквенные клавиши от **A** до **Z** (при выполнении заданий по параллельному программированию максимально возможное число процессов равно 36).

Если в основных разделах окна задачника отсутствуют прокручиваемые элементы или основные разделы являются скрытыми, то дополнительную клавишу **Alt** в перечисленных выше клавиатурных **Alt**-комбинациях можно не использовать.

Отладочная информация, получаемая из подчиненных процессов параллельной программы, предварительно сохраняется в

специальных временных файлов в каталоге учащегося, поэтому она будет доступна для просмотра, даже если на каком-либо этапе выполнения программы произойдет зависание некоторых ее подчиненных процессов. Отладочная информация, получаемая из главного процесса, выводится непосредственно в раздел отладки.

Количество отладочных строк для каждого процесса не должно превышать 999; если некоторый процесс пытается вывести данные в строку с номером, превышающим 999, то в связанной с этим процессом области отладки выводится сообщение об ошибке, и последующий вывод отладочных данных для этого процесса блокируется. Указанное ограничение позволяет, в частности, избежать проблем, возникающих при «бесконечном» выводе отладочной информации из какого-либо зациклившегося подчиненного процесса во временный файл.

Каждая экранная строка, отображаемая в разделе отладки, состоит из служебной области и области данных. Ширина служебной области равна 6 экранным позициям для «непараллельных» заданий и 9 позициям для заданий по параллельному программированию. Ширина области данных равна 80 позициям.

Служебная область состоит из следующих частей (см. рисунки, приведенные выше):

- *область нумерации процессов* (только для заданий по параллельному программированию): 2 экранные позиции, отводимые для ранга процесса, и символ «|»;
- *область нумерации строк*: 3 экранные позиции, отводимые для номера строки данных, после которых следует символ «>» и символ пробела;
- *область признака сообщения об ошибке*: одна экранная позиция, в которой может содержаться либо пробел (признак обычного отладочного текста), либо символ «!» (признак сообщения об ошибке).

Если в разделе отладки выводится текст, связанный со всеми процессами параллельного приложения (этот текст связан с ярлычком «*»), то нумерация строк для каждого процесса производится независимо.

При переключении между областями вывода, связанными с различными процессами параллельного приложения, сохраняется номер первой отображаемой строки (за исключением ситуации, когда в новой области вывода отсутствует строка с требуемым номером; в этом случае в новой области вывод осуществляется, начиная с первой строки данных). Отмеченная особенность позволяет быстро просмотреть (и сравнить) один и тот же фрагмент отладочных данных для различных процессов.

Содержимое области вывода, отображаемой в разделе отладки, можно копировать в буфер Windows; для этого предназначена стандартная клавиатурная комбинация **Ctrl+C** и соответствующая команда контекстного меню раздела отладки.

Для вывода данных в раздел отладки предназначены процедуры Show и ShowLine. Описания этих процедур приводятся в разделе, посвященном [типам и процедурам модуля PT4](#).

Возможность использования раздела отладки сохранена и в появившемся в версии 4.11 [режиме окна с динамической компоновкой](#). Как и в окне с фиксированной компоновкой, раздел отладки отображается ниже разделов с заданием, однако в данном случае для всех разделов окна используется общая полоса прокрутки. В окне с динамической компоновкой доступны почти все описанные выше действия по управлению разделом отладки, в частности, настройка размера шрифта в разделе отладки, копирование содержимого раздела отладки в буфер Windows, скрытие разделов окна с заданием.

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET]

ЦИКЛ С ПАРАМЕТРОМ
Задание: For15

Выполняет: Иванов Петр

Результаты (F2) Цвет (F3) Режим (F4)

Задание выполнено! Выход (Esc)

Введено данных: 2 из 2 Выведено данных: 1 из 1 Пройдено тестов: 5 из 5

Дано вещественное число A и целое число $N (>0)$. Найти A в степени N :
 $A^N = A \cdot A \cdot \dots \cdot A$ (числа A перемножаются N раз).

Исходные данные

$A = 3.00$ $N = 8$

Полученные результаты

$A^N = 6561.00$

```
1> i=1 A^i=3.00
2> i=2 A^i=9.00
3> i=3 A^i=27.00
4> i=4 A^i=81.00
5> i=5 A^i=243.00
6> i=6 A^i=729.00
7> i=7 A^i=2187.00
8> i=8 A^i=6561.00
```

В режиме с динамической компоновкой отсутствует возможность просмотра областей вывода, связанных с отдельными процессами параллельной программы: раздел отладки всегда содержит объединенный текст, полученный из всех областей. Для просмотра содержимого отдельных областей вывода следует переключиться в режим окна с фиксированной компоновкой, нажав клавишу **F4**.

Просмотр результатов

Результаты выполнения всех заданий из задачника **Programming Taskbook** заносятся в специальный *файл результатов results.abc*, который должен находиться в том каталоге, из которого запускаются программы с заданиями.

Данный файл автоматически создается в рабочем каталоге системы **PascalABC.NET** (по умолчанию рабочим каталогом является каталог PABCWork.NET, находящийся на диске C). При смене рабочего каталога (это можно сделать с помощью программы настройки задачника PT4Setup) в новом рабочем каталоге также создается файл результатов.

Данные хранятся в файле результатов в зашифрованном виде, поэтому его непосредственный просмотр и корректировка невозможны. Для просмотра содержимого файла результатов предназначен программный модуль **PT4Results**, который вызывается непосредственно из среды **PascalABC.NET** командой меню «Модули | Просмотреть результаты» (с данной командой связана также кнопка и клавиатурная комбинация **Shift+Ctrl+R**).

При просмотре файла результатов с помощью модуля **PT4Results** содержащаяся в нем информация отображается в двух вариантах.

Вариант с полной информацией представляет собой перечень всех запусков программ с учебными заданиями; для каждого запуска указывается имя задания, дата и время запуска, а также результат запуска. Например:

```
= Иванов Петр      (C:\PABCWork)
Begin1 A27/09 19:07 Выведены не все результирующие
данные.
Begin1 A27/09 19:07 Ошибочное решение.--2
Begin1 A27/09 19:07 Задание выполнено!
For1   A27/09 19:07 Ознакомительный запуск.
For1   A27/09 19:08 Выведены не все результирующие
данные.
For1   A27/09 19:10 Ошибочное решение.--3
For1   A27/09 19:13 Задание выполнено!
For5   A27/09 19:15 Неверно указан тип при вводе
```

```
исходных данных.--2
For5 A27/09 19:16 Выведены не все результирующие
данные.
For5 A27/09 19:19 Задание выполнено!
For10 A27/09 19:21 Неверно указан тип при вводе
исходных данных.
For10 A27/09 19:22 Ошибочное решение.
```

Буква **A** перед датой означает, что задание выполнялось в системе **PascalABC.NET**, а числа в конце некоторых строк указывают на то, что было проведено подряд несколько запусков программы с одинаковым результатом.

Вариант со сводной информацией позволяет быстро узнать время выполнения заданий, количество выполненных и незавершенных заданий, а также получить сводку о выполненных заданиях по каждой группе. Например:

```
= Иванов Петр (C:\PABCWork.NET)
-1- ВРЕМЯ ЗАНЯТИЙ:
27/09(19:07-19:22)[3/16] ВСЕГО: 0:16 [3/16]
-2- ВЫПОЛНЕННЫЕ ЗАДАНИЯ:
Begin1 A27/09 [4] For1 A27/09 [6] For5 A27/09
[4]
-3- НЕЗАВЕРШЕННЫЕ ЗАДАНИЯ:
For10 27/09 [2]
-5- СВОДКА ПО ГРУППАМ ЗАДАНИЙ:
Begin 1 For 2 ВСЕГО: 3
```

При использовании задачника в системе **PascalABC.NET** в файл результатов заносится также информация о выполненных заданиях для исполнителей Робот и Чертежник.

Начиная с версии 4.11, программный модуль **PT4Results** можно вызвать непосредственно из [окна задачника](#), нажав клавишу **F2**.

Демонстрационный режим

Для запуска задачника в *демонстрационном режиме* следует при указании имени задания в процедуре `Task` дополнить это имя символом `?`, например:

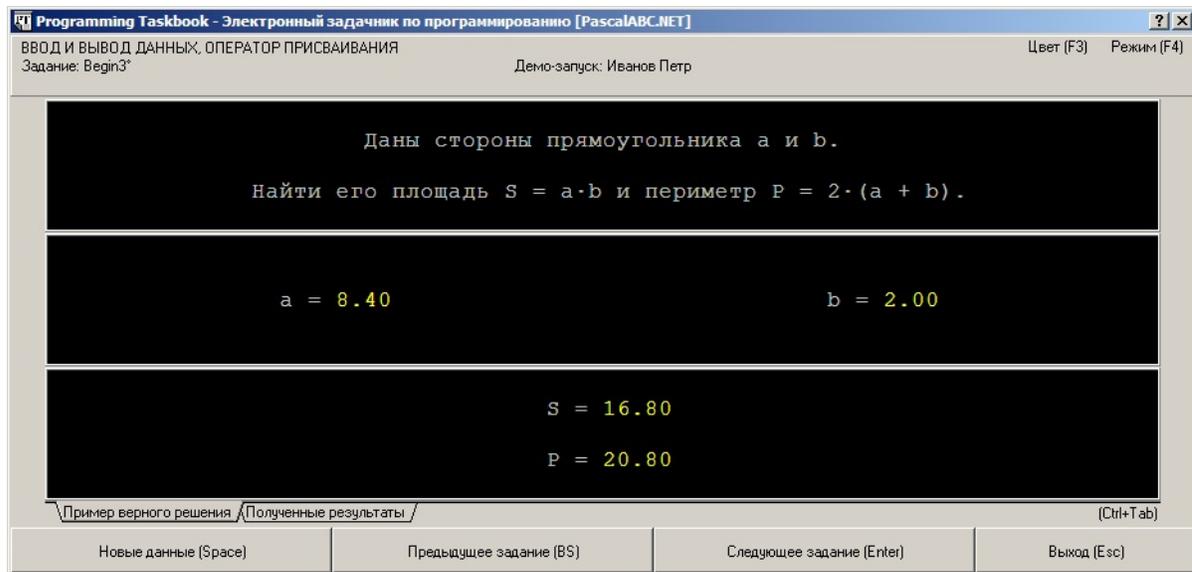
```
Task('Begin12?');
```

Можно также указать символ `?` сразу после имени темы, например, `'Begin?'`. В этом случае в окне задачника сразу будет отображено *последнее* задание указанной группы.

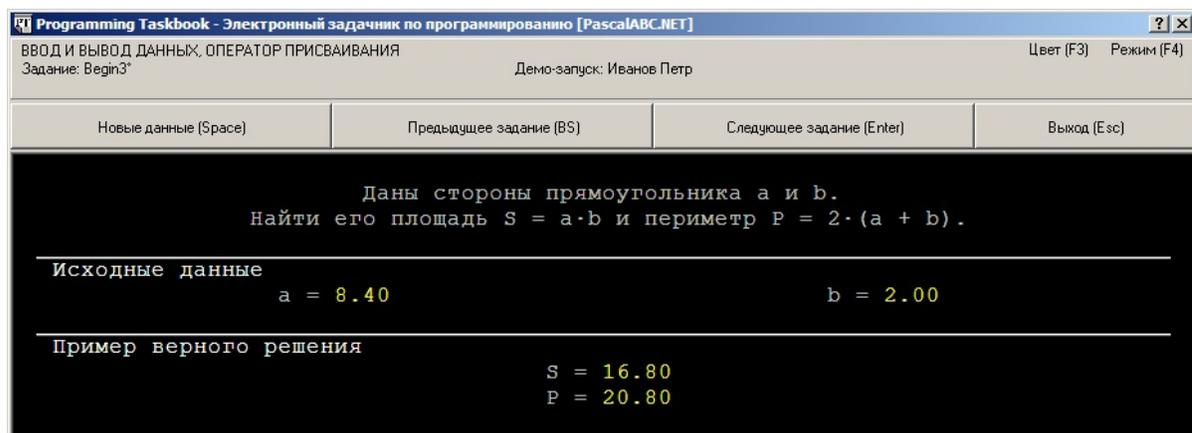
Демонстрационный режим задачника имеет следующие особенности:

- даже если программа содержит решение задания, это решение не анализируется и информация в файл результатов не заносится;
- после отображения на экране окна задачника в разделе результатов сразу будет выбрана вкладка «Пример верного решения»;
- при одном запуске программы можно просмотреть несколько вариантов исходных и контрольных данных; для смены набора данных требуется нажать кнопку «Новые данные» или клавишу пробела;
- при одном запуске программы можно последовательно просмотреть все задания данной группы; для перехода к заданию с большим номером требуется нажать кнопку «Следующее задание» или клавишу **Enter**, а для перехода к заданию с меньшим номером требуется нажать кнопку «Предыдущее задание» или клавишу **Backspace**. Задания перебираются циклически.

На рисунке приведен вид окна задачника в демонстрационном режиме.



При использовании [динамической компоновки](#), появившейся в версии 4.11, окно в демонстрационном режиме выглядит следующим образом:

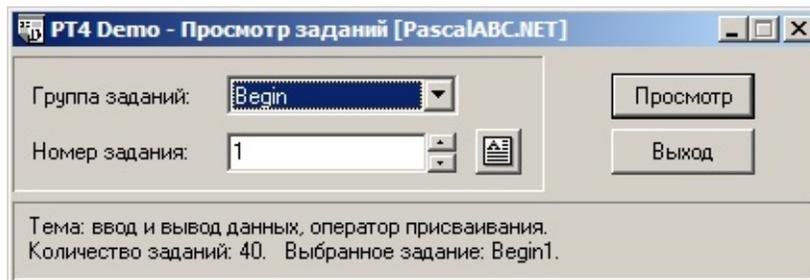


Демонстрационный режим удобно использовать для быстрого просмотра всех заданий требуемой группы, а также различных вариантов исходных данных для требуемого задания.

Для демо-просмотра всех групп заданий, включенных в базовый набор, предназначен программный модуль **PT4Demo**, который вызывается непосредственно из среды **PascalABC.NET** командой меню «Модули | Просмотреть задания» (с данной командой связана также кнопка  и клавиатурная комбинация **Shift+Ctrl+D**).

Ниже приводится вид окна модуля PT4Demo. В данном окне можно выбрать группу заданий (для быстрого перебора групп предназначены горячие клавиши **Ctrl+[<]** и **Ctrl+[>]**) и номер задания

в пределах группы (номера можно перебирать с помощью комбинаций **Ctrl+Shift+[<]** и **Ctrl+Shift+[>]**).



Если задачник не зарегистрирован, то он всегда запускается в демонстрационном режиме (исключение составляют задания, доступные для выполнения в мини-варианте). Переключение в демонстрационный режим автоматически происходит также в случае, если в каталоге с выполняемой программой отсутствует файл *результатов* results.abc.

В версии 4.8 задачника добавлена возможность генерации текста формулировок учебных заданий и дополнительных пояснений к заданиям в виде html-страницы. Для создания подобной страницы и ее немедленного отображения на экране (в html-браузере, установленном по умолчанию) достаточно вызвать процедуру `Task`, указав в качестве ее параметра имя группы заданий или имя конкретного задания, дополненное символом «#», например, `Begin#` или `Begin3#`. При указании группы генерируется текст всех заданий, включенных в эту группу. Процедуру `Task` с параметром, оканчивающимся символом #, можно вызывать несколько раз, указывая различные имена групп или конкретных заданий; в результате созданная html-страница будет содержать тексты всех заданий, указанных при различных запусках процедуры `Task` (в том же порядке).

Если при каком-либо вызове будет указано неверное имя группы или неверный номер задания в пределах группы, то выведется сообщение об ошибке, и html-страница создана не будет.

В создаваемую html-страницу включаются стилевые настройки, которые берутся из файла `PT4Tasks.css`, который ищется в текущем каталоге, а при его отсутствии в этом каталоге — в подкаталоге `PT4` системного каталога `PascalABC.NET`. Если данный стилиевой файл не

найден, то стилевые настройки в html-страницу не добавляются.

При успешной генерации html-страницы она сохраняется в файле со стандартным именем PT4Tasks.html в рабочем каталоге приложения. Если в этом каталоге нельзя создать файл, то выводится сообщение об ошибке.

Возможность просмотра html-страниц с описанием текущей группы заданий добавлена и в программный модуль **PT4Demo**. Для этого предусмотрена кнопка  (см. приведенный выше рисунок) и клавиша **F2**.

Конструкторы проверяемых заданий: обзор

В системе **PascalABC.NET** можно создавать проверяемые задания для исполнителей Робот и Чертежник, а также для электронного задачника **Programming Taskbook**. Задания разрабатываются с помощью конструкторов **RobotTaskMaker**, **DMTaskMaker** и **PT4TaskMaker**; конструкторы **RobotTaskMaker** и **DMTaskMaker** реализованы в виде одноименных модулей, конструктор **PT4TaskMaker** реализован в виде модуля **PT4TaskMakerNET**. В данном разделе приводятся подробные описания каждого из конструкторов и примеры их использования для создания новых заданий:

- [Модуль RobotTaskMaker](#)
- [Создание заданий для исполнителя Робот](#)
- [Модуль DMTaskMaker](#)
- [Создание заданий для исполнителя Чертежник](#)
- [Модуль PT4TaskMakerNET: общее описание](#)
- [Модуль PT4TaskMakerNET: основные компоненты](#)
- [Модуль PT4TaskMakerNET: дополнительные компоненты](#)
- [Модуль PT4TaskMakerNET: форматирование текста заданий](#)
- [Модуль PT4TaskMakerNET: примеры разработки учебных заданий](#)
- [Модуль PT4TaskMakerNET: разработка заданий, связанных с ЕГЭ по информатике](#)

Модуль RobotTaskMaker

Типы модуля RobotTaskMaker

```
type TaskProcType = procedure;
```

Тип процедуры, генерирующей конкретное задание. Каждое задание реализуется в виде отдельной процедуры; для связывания этой процедуры с именем задания необходимо использовать процедуру RegisterTask, описываемую ниже.

Процедуры модуля RobotTaskMaker

```
procedure Field(szx, szy: integer);
```

Задаёт поле Робота размера *szx* на *szy* клеток.

```
procedure HorizontalWall(x, y, len: integer);
```

Создаёт горизонтальную стену длины *len* и координатами левого верхнего угла (*x*, *y*).

```
procedure VerticalWall(x, y, len: integer);
```

Создаёт вертикальную стену длины *len* и координатами левого верхнего угла (*x*, *y*).

```
procedure RobotBegin(x, y: integer);
```

Задаёт начальное положение Робота в клетке с координатами (*x*, *y*).

```
procedure RobotEnd(x, y: integer);
```

Задаёт конечное положение Робота в клетке с координатами (*x*, *y*).

```
procedure RobotBeginEnd(x, y, x1, y1: integer);
```

Задаёт начальное положение Робота в клетке с координатами (*x*, *y*) и конечное в клетке с координатами (*x1*, *y1*).

```
procedure Tag(x, y: integer);
```

Помечает клетку (*x*, *y*) для закрашивания.

```
procedure TagRect(x, y, x1, y1: integer);
```

Помечает прямоугольник из клеток, задаваемый координатами противоположных вершин прямоугольника (*x*, *y*) и (*x1*, *y1*), для закрашивания.

```
procedure MarkPainted(x,y: integer);
```

Закрашивает клетку (x, y) (в задании некоторые клетки могут быть уже закрашены).

```
procedure TaskText(s: string);
```

Задаёт формулировку текста задания в строке s.

```
procedure RegisterGroup(name,description,unitname:  
string; count: integer);
```

Обеспечивает автоматическую регистрацию новой группы заданий в программном модуле **PT4Load**. В результате имя данной группы будет отображаться в окне модуля **PT4Load** в списке групп, связанных с исполнителем Робот, что позволит создать программу-заготовку для выполнения любого задания этой группы. В качестве параметров процедуры указывается имя группы `name`, краткое описание группы `description`, имя модуля `unitname`, в котором описана группа, и количество заданий `count`. Имя группы заданий должно содержать не более 7 символов (цифр и латинских букв) и не должно оканчиваться цифрой, количество заданий не должно превышать 999. Процедура `RegisterGroup` должна вызываться в секции инициализации модуля, содержащего реализацию новой группы заданий для Робота.

```
procedure RegisterTask(name: string; p: TaskProcType);
```

Связывает имя задания `name` с процедурой `p`, в которой реализовано данное задание. Данную процедуру следует вызывать для *каждого* задания. Подобно описанной выше процедуре `RegisterGroup`, процедура `RegisterTask` должна вызываться в секции инициализации модуля, содержащего реализацию новой группы заданий для Робота. Порядок вызова этих процедур может быть произвольным.

Создание заданий для исполнителя Робот

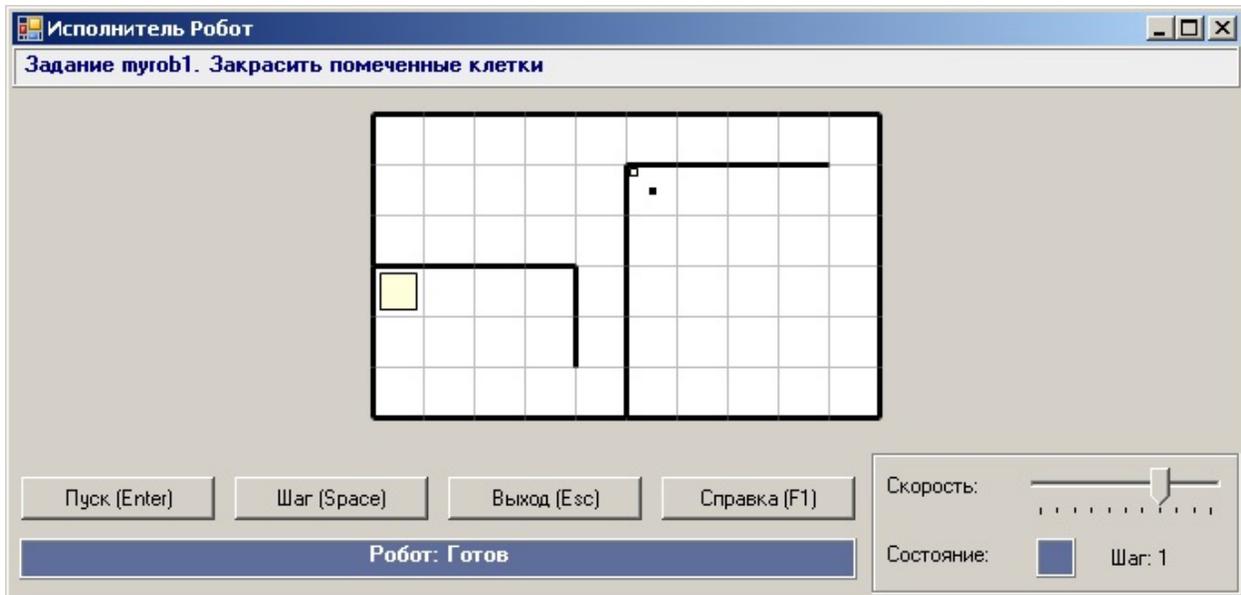
Опишем последовательность создания группы заданий для исполнителя Робот. Создадим модуль RobTasks.pas со следующим текстом:

```
unit RobTasks;  
interface  
uses RobotTaskMaker;  
implementation  
procedure FirstRob;  
begin  
    TaskText('Задание myrob1. Закрасить помеченные  
клетки');  
    Field(10,6);  
    HorizontalWall(0,3,4);  
    VerticalWall(4,3,2);  
    RobotBegin(1,4);  
    VerticalWall(5,1,5);  
    HorizontalWall(5,1,4);  
    RobotEnd(6,2);  
    Tag(6,2);  
end;  
begin  
    RegisterGroup('myrob','Мои задания для  
Робота','RobTasks',2);  
    RegisterTask('myrob1',FirstRob);  
end.
```

Наберем и запустим основную программу (сохранять ее в каком-либо файле не требуется):

```
uses Robot, RobTasks;  
begin  
    Task('myrob1');  
end.
```

Будет выведено следующее задание для Робота:



Добавим задание, в котором конфигурация поля случайна:

```

procedure SecondRob;
var n,i: integer;
begin
  TaskText('Задание myrob2. Закрасить клетки под
  закрасенными');
  n:=Random(4)+7;
  Field(n,4);
  RobotBeginEnd(1,3,n,3);
  MarkPainted(n,2);
  Tag(n,3);
  for i:=2 to n-1 do
    if Random(3)=1 then
      begin
        MarkPainted(i,2);
        Tag(i,3);
      end;
  end;

```

Добавим вызов процедуры регистрации для данного задания; в результате секция инициализации примет следующий вид:

```

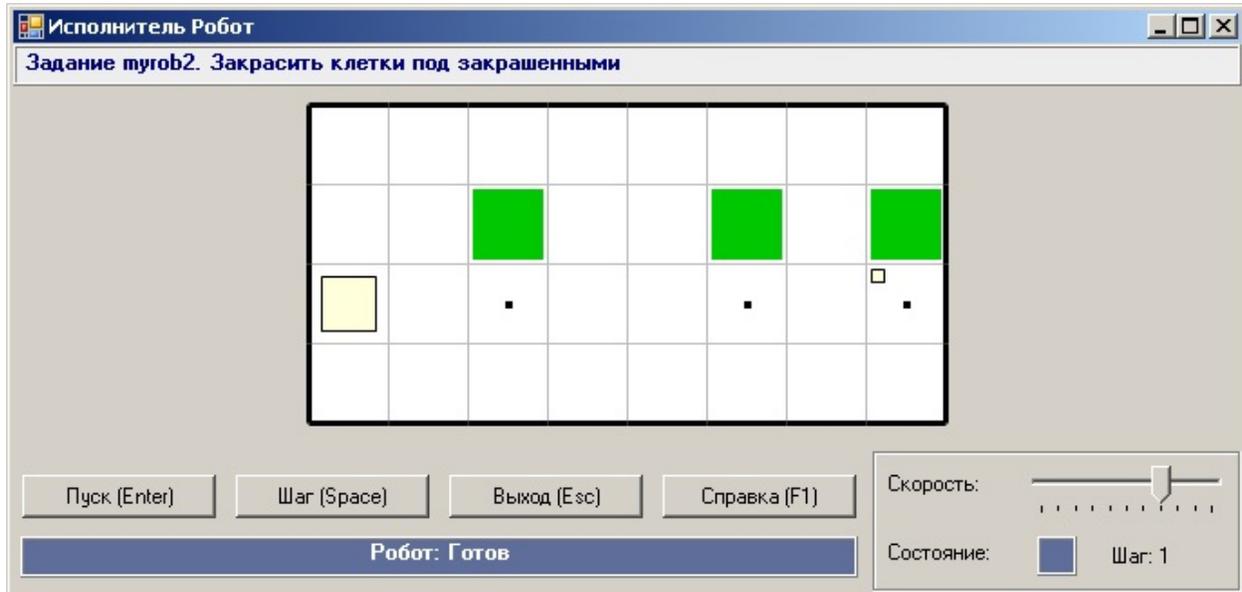
begin
  RegisterGroup('myrob','Мои задания для
  Робота','RobTasks',2);
  RegisterTask('myrob1',FirstRob);
  RegisterTask('myrob2',SecondRob);
end.

```

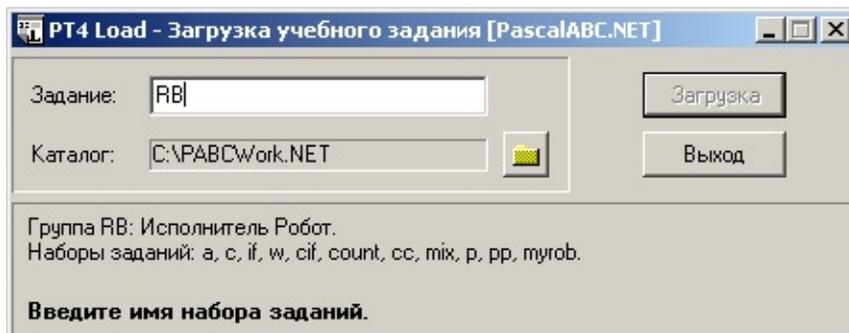
Изменим основную программу:

```
uses Robot, RobTasks;  
begin  
  Task('myrob2');  
end.
```

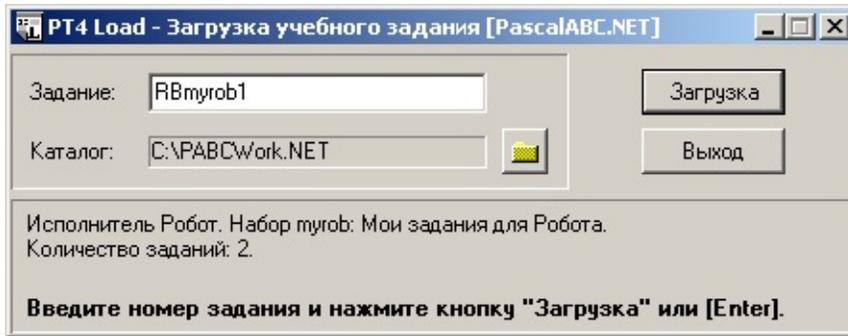
При запуске этой программы в окне исполнителя Робот будет выведено новое задание:



При первом запуске программы с подключенным модулем **DMTasks** созданная нами группа была автоматически зарегистрирована в мастере по созданию программ-заготовок **PT4Load**. Если теперь нажать кнопку  и в появившемся окне **PT4Load** ввести префикс **RB** в поле «Задание», то окно примет следующий вид:



Мы видим, что группа заданий **myrob** появилась в списке доступных групп для исполнителя Робот. Наберем имя задания **myrob1**:



После нажатия Enter в рабочем каталоге будет создан новый файл RBmyrob1.pas со следующим содержимым:

```
uses Robot, RobTasks;
```

```
begin  
    Task('myrob1');
```

```
end.
```

Можно приступать к решению собственноручно разработанной задачи :)

Модуль DMTaskMaker

Типы модуля DMTaskMaker

```
type TaskProcType = procedure;
```

Тип процедуры, генерирующей конкретное задание. Каждое задание реализуется в виде отдельной процедуры; для связывания этой процедуры с именем задания необходимо использовать процедуру RegisterTask, описываемую ниже.

Процедуры модуля DMTaskMaker

```
procedure Field(szx, szy: integer);
```

Задаёт поле Чертежника размера *szx* на *szy* клеток.

```
procedure DoToPoint(x, y: integer);
```

Перемещает перо Чертежника-постановщика заданий в точку с координатами (*x*, *y*).

```
procedure DoOnVector(dx, dy: integer);
```

Перемещает перо Чертежника-постановщика заданий в точку с координатами (*x*, *y*).

```
procedure DoPenUp;
```

Поднимает перо Чертежника-постановщика заданий.

```
procedure DoPenDown;
```

Опускает перо Чертежника-постановщика заданий.

```
procedure TaskText(s: string);
```

Задаёт имя и формулировку задания в строке *s*.

```
procedure RegisterGroup(name, description, unitname: string; count: integer);
```

Обеспечивает автоматическую регистрацию новой группы заданий в программном модуле **PT4Load**. В результате имя данной группы будет отображаться в окне модуля **PT4Load** в списке групп, связанных с исполнителем Чертежник, что позволит создать программу-заготовку для выполнения любого задания этой группы. В качестве параметров процедуры указывается имя группы *name*,

краткое описание группы `description`, имя модуля `unitname`, в котором описана группа, и количество заданий `count`. Имя группы заданий должно содержать не более 7 символов (цифр и латинских букв) и не должно оканчиваться цифрой, количество заданий не должно превышать 999. Процедура `RegisterGroup` должна вызываться в секции инициализации модуля, содержащего реализацию новой группы заданий для Чертежника.

```
procedure RegisterTask(name: string; p: TaskProcType);
```

Связывает имя задания `name` с процедурой `p`, в которой реализовано данное задание. Данную процедуру следует вызывать для *каждого* задания. Подобно описанной выше процедуре `RegisterGroup`, процедура `RegisterTask` должна вызываться в секции инициализации модуля, содержащего реализацию новой группы заданий для Чертежника. Порядок вызова этих процедур может быть произвольным.

Создание заданий для исполнителя Чертежник

Опишем последовательность создания группы заданий для исполнителя Чертежник. Создадим модуль DMTasks.pas со следующим текстом:

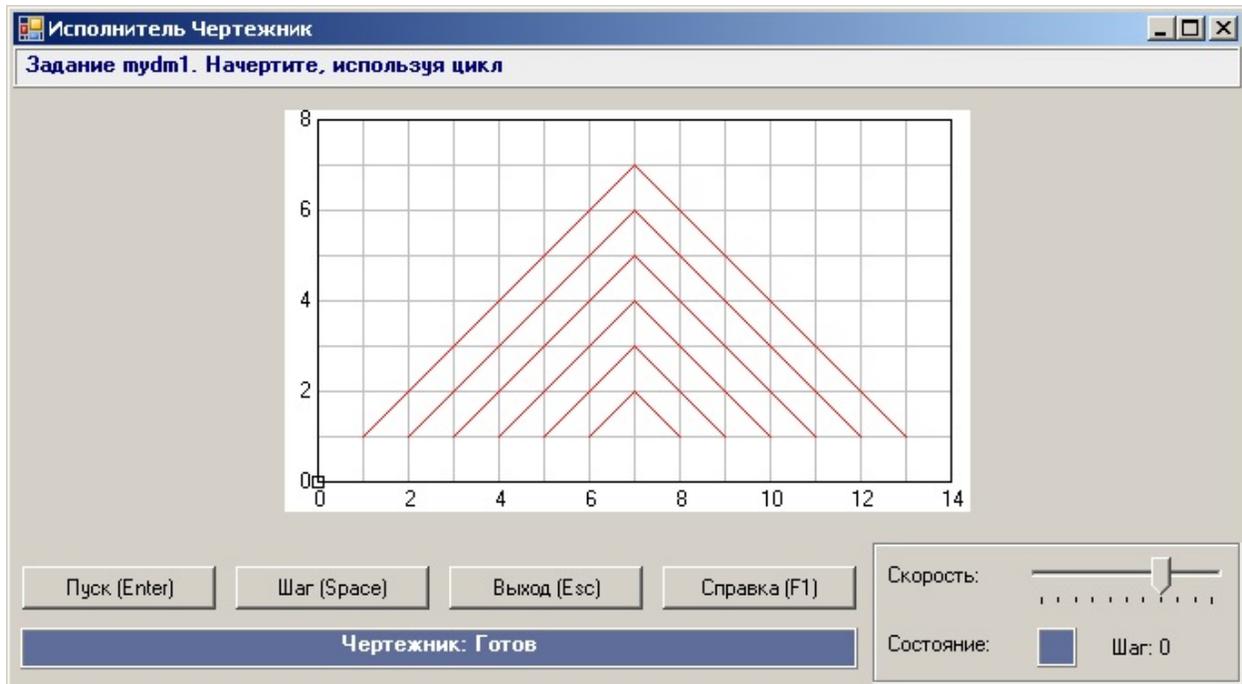
```
unit DMTasks;  
interface  
uses DMTaskMaker;  
implementation  
procedure FirstDM;  
var i,a: integer;  
begin  
    TaskText('Задание mydm1. Начертите, используя цикл');  
    Field(14,8);  
    DoToPoint(7,7);  
    a:=6;  
    for i:=1 to 6 do  
        begin  
            DoPenDown;  
            DoOnVector(a, -a);  
            DoOnVector(-a, a);  
            DoOnVector(-a, -a);  
            DoOnVector(a, a);  
            Dec(a);  
            DoPenUp;  
            DoOnVector(0, -1);  
        end;  
    end;  
  
    begin  
        RegisterGroup('mydm', 'Мои задания для  
Чертежника', 'DMTasks', 2);  
        RegisterTask('mydm1', FirstDM);  
    end.
```

Наберем и запустим основную программу (сохранять ее в каком-либо файле не требуется):

```
uses Drawman, DMTasks;  
begin
```

```
Task('mydm1');  
end.
```

Будет выведено следующее задание для Чертежника:



Добавим задание на разработку процедуры:

```
procedure DoCross;  
begin  
  DoPenDown;  
  DoOnVector(1,0); DoOnVector(0,-1);  
  DoOnVector(1,0); DoOnVector(0,-1);  
  DoOnVector(-1,0); DoOnVector(0,-1);  
  DoOnVector(-1,0); DoOnVector(0,1);  
  DoOnVector(-1,0); DoOnVector(0,1);  
  DoOnVector(1,0); DoOnVector(0,1);  
  DoPenUp;  
end;  
procedure SecondDM;  
begin  
  TaskText('Задание mydm2. Начертите, используя  
процедуру Cross');  
  Field(18,12);  
  DoToPoint(3,8);  
  DoCross;  
  DoToPoint(8,4);
```

```

DoCross;
DoToPoint(12,11);
DoCross;
DoToPoint(15,6);
DoCross;
end;

```

Добавим вызов процедуры регистрации для данного задания; в результате секция инициализации примет следующий вид:

```

begin
  RegisterGroup('mydm', 'Мои задания для
Чертежника', 'DMTasks', 2);
  RegisterTask('mydm1', FirstDM);
  RegisterTask('mydm2', SecondDM);
end.

```

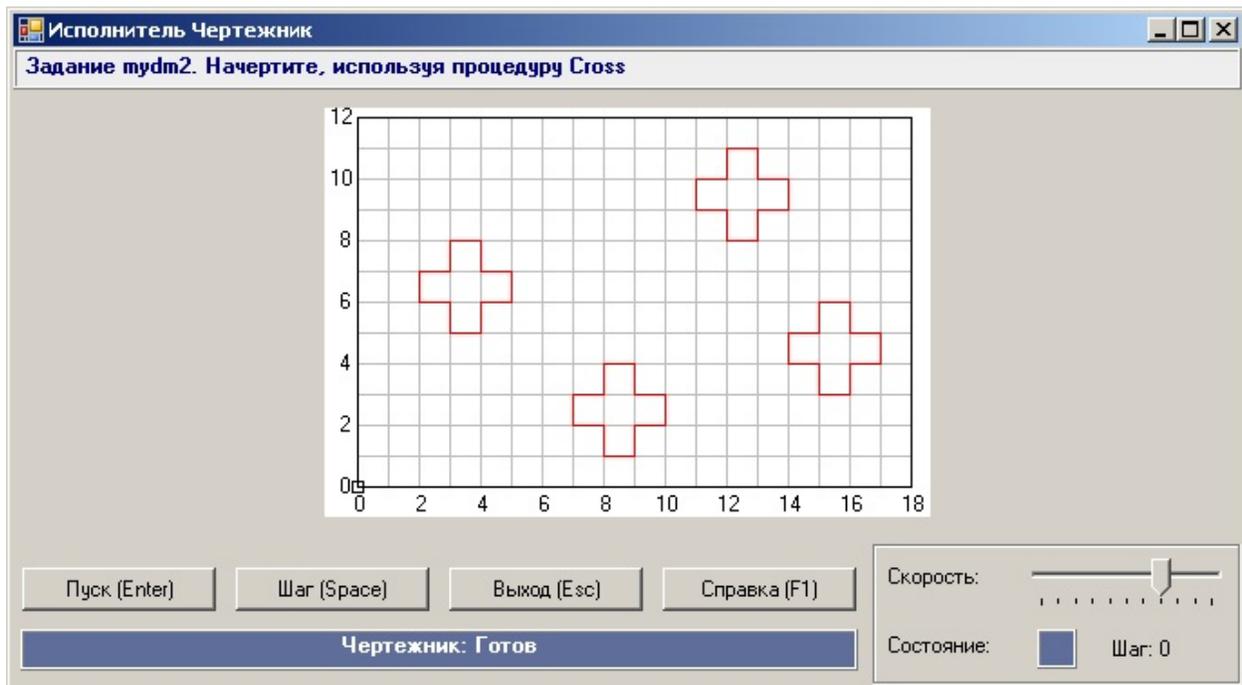
Изменим основную программу:

```

uses Drawman, DMTasks;
begin
  Task('mydm2');
end.

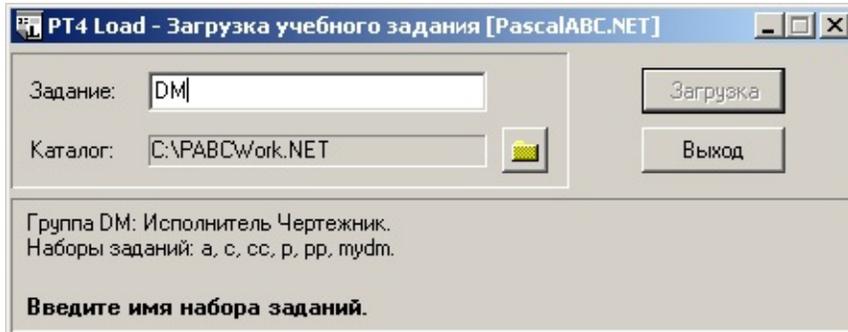
```

При запуске этой программы в окне исполнителя Чертежник будет выведено новое задание:

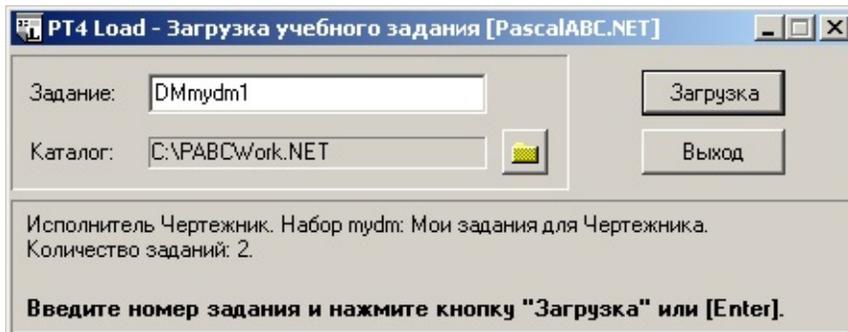


При первом запуске программы с подключенным модулем `DMTasks`

созданная нами группа была автоматически зарегистрирована в мастере по созданию программ-заготовок **PT4Load**. Если теперь нажать кнопку  и в появившемся окне **PT4Load** ввести префикс **DM** в поле «Задание», то окно примет следующий вид:



Мы видим, что группа заданий **mydm** появилась в списке доступных групп для исполнителя Чертежник. Наберем имя задания **mydm1**:



После нажатия клавиши **Enter** в рабочем каталоге будет создан новый файл **DMmydm1.pas** со следующим содержимым:

```
uses Drawman, DMTasks;
```

```
begin
```

```
  Task('mydm1');
```

```
end.
```

Можно попытаться решить :)

Модуль PT4TaskMakerNET: общее описание

Назначение и состав конструктора учебных заданий

Конструктор учебных заданий PT4TaskMaker позволяет разрабатывать новые группы заданий для электронного задачника Programming Taskbook.

Начиная с версии 4.11, новые группы заданий, доступные для всех поддерживаемых задачником сред, можно разрабатывать не только на языке Pascal, но и на языках C++ и C#, причем конструктор для языка Pascal можно использовать и в среде PascalABC.NET.

Конструктор для среды PascalABC.NET реализован в виде модуля PT4TaskMakerNET.

Каждая группа учебных заданий оформляется в виде отдельной динамической библиотеки (dll-файла). Dll-файлы с новыми группами могут находиться либо в рабочем каталоге учащегося, либо в подкаталоге Lib системного каталога задачника. Подключение новых групп происходит автоматически при инициализации задания, поэтому вид проекта-заготовки при работе с новыми группами не отличается от стандартного вида проекта, ориентированного на базовые группы заданий. Кроме того, новые группы автоматически добавляются в список доступных групп в программных модулях PT4Demo и PT4Load. Созданные в виде dll-файлов новые группы заданий могут использоваться не только в любых средах программирования, поддерживаемых универсальным вариантом задачника, но и в среде PascalABC.NET; для этого достаточно разместить dll-файл в подкаталоге PT4\Lib системного каталога PascalABC.NET или в рабочем каталоге учащегося.

Обзор элементов конструктора учебных заданий

Конструктор учебных заданий представляет собой набор констант, функций и процедур, предназначенных для определения каждого из входящих в группу заданий, а также для настройки свойств группы в целом.

Ниже перечислены элементы, входящие в конструктор PT4TaskMaker:

- процедурный тип [TInitTaskProc](#); в библиотеке с группой заданий должна быть определена *основная процедура группы* типа TInitTaskProc, позволяющая генерировать задание с требуемым номером;
- процедура CreateGroup, определяющая [общие характеристики](#) группы заданий;
- процедуры и константы, используемые для [создания нового задания](#) и добавления в него формулировки, а также исходных и контрольных данных базовых типов (логического, целочисленного, вещественного, символьного строкового); обычно каждое задание оформляется в виде отдельной процедуры, которая вызывается из основной процедуры группы;
- процедура [UseTask](#), позволяющая импортировать в создаваемую группу задания из других групп; эта процедура обычно вызывается в основной процедуре группы;
- процедуры, обеспечивающие [добавление комментариев](#), оформляемых в виде *преамбулы* к группе и ее подгруппам;
- функции и константы, позволяющие определить [текущее состояние задачника](#) (используемый язык программирования, текущую *локаль* — русскую или английскую —, текущую версию задачника, а также номер текущего тестового испытания программы, выполняющей задание);
- функции, предоставляющие разработчику заданий [образцы слов, предложений и многострочных текстов](#);
- процедуры, позволяющие включать в задание [файловые данные](#);
- процедуры, позволяющие включать в задание [указатели и динамические структуры данных](#) — линейные списки и деревья;
- процедуры для разработки заданий по [параллельному MPI-](#)

программированию.

Библиотечные и сводные группы

При разработке новых заданий целесообразно объединять их в группы с именами, содержащими, кроме названия темы, дополнительные сведения, например, версию созданной группы и данные об авторе. Однако допустимые имена групп могут содержать не более 9 символов, что является недостаточным для указания дополнительных сведений. С другой стороны, группы, содержащие только новые задания (*библиотечные группы*), вряд ли будут использоваться непосредственно в учебном процессе; более целесообразной будет компоновка этих новых заданий с заданиями из соответствующей базовой группы задачника (и, возможно, с заданиями из других библиотечных групп).

Поэтому при разработке группы, содержащей исключительно новые задания (*библиотечной группы, или библиотеки заданий*) разрешено указывать имя длины более 9 символов (но не более 25).

Библиотечные группы не предназначены для непосредственного использования в учебном процессе. Задания из этих групп можно запускать с помощью процедуры Task, однако они будут отображаться только в *демонстрационном режиме* (точнее, в режиме «просмотра библиотеки заданий», о чем будет свидетельствовать соответствующий текст, указываемый в окне задачника вместо сведений об учащемся). Библиотечные группы не включаются в список групп, отображаемый в модуле PT4Load, однако доступны для просмотра с помощью модуля PT4Demo (в этом модуле библиотечные группы указываются в конце списка групп).

Для применения на занятиях удобно использовать *сводные группы*, не содержащие реализации новых заданий, а лишь импортирующие наборы заданий из базовых и дополнительных библиотечных групп и компоновующие их в порядке, который требуется преподавателю. При импортировании задания из другой группы у него изменяется название и ключ (которые берутся из характеристик сводной группы), однако сохраняются такие характеристики импортируемой группы, как [заголовки подгрупп](#) и сведения об авторе. Имена сводных групп должны содержать не более 9 символов.

Структура проекта с описанием группы заданий

В данном разделе описываются правила, которым должен удовлетворять проект (dll-библиотека), содержащий описание новой группы заданий (см. также раздел «[Примеры](#)»).

Каждый проект, реализуемый в виде dll-библиотеки, должен содержать определение *единственной* группы заданий. Имя библиотеки с группой заданий должно иметь вид `PT4<имя группы><маркер локали>`, где *<маркер локали>* является либо пустой строкой, либо имеет вид `_ru` или `_en`. Например, группа TMDemoPas может быть реализована в виде библиотек `PT4TMDemoPas.dll`, `PT4TMDemoPas_ru.dll` и `PT4TMDemoPas_en.dll`. Библиотеки с явно указанным маркером локали используются только в варианте задачника, соответствующем данной локали (`ru` — в русском варианте задачника, `en` — в английском). Библиотеки, в которых не указан маркер локали, используются в любом варианте задачника (если они не перекрываются библиотекой с явно указанной локалью). Порядок поиска библиотек для требуемой группы следующий: вначале просматривается *рабочий каталог учащегося* и в нем ищется библиотека с данным именем и явно указанной локалью; если она не найдена, то ищется библиотека с данным именем без маркера локали; если она не найдена, то поиск библиотек (в этом же порядке) выполняется в *каталоге Lib системного каталога задачника*.

Проект с описанием новой группы должен иметь определенную структуру. Приведем описание структуры проекта, который реализуется на языке Pascal в среде PascalABC.NET:

```
library PT4MakerDemo;  
  
uses PT4TaskMakerNET;  
  
// процедуры, реализующие конкретные задания  
...  
  
procedure InitTask(num: integer);  
begin  
    // в данной процедуре выполняются вызовы вспомогательных процедур,  
    // реализующих все задания группы; номер задания определяется
```

```

    // параметром num; для определения процедуры, соответствующей
    // требуемому номеру, обычно используется оператор case
    ...
end;

procedure inittaskgroup;
begin
    // вспомогательная процедура, в которой выполняется вызов стартовой
    // процедуры CreateGroup и могут вызываться процедуры, связанные
    // с добавлением комментариев (преамбул) для группы и ее подгрупп.
    // Все буквы в ее имени должны быть строчными

    CreateGroup('TMDemoAbc', 'Примеры различных задач (конструктор для
        'М. Э. Абрамян, 2013', 'qwqfsdf13dfttd', 8, InitTask);
    ...
end;

procedure activate(S: string);
begin
    // вспомогательная процедура, используемая при подключении библиотек
    // к задачнику. Все буквы в ее имени должны быть строчными

    ActivateNET(S);
end;

begin
end.

```

При определении новой группы заданий можно учитывать *текущий язык программирования*, установленный для задачника, и в зависимости от этого языка по-разному инициализировать некоторые задания группы. Эта возможность является особенно полезной при реализации заданий, связанных с обработкой динамических структур данных, поскольку в языках Visual Basic и 1С подобные задания должны быть недоступны, а в языках Pascal и C++ они должны оформляться по-другому, нежели в языках платформы .NET, Python и Java. См. также [пункт, посвященный динамическим структурам данных](#), в разделе «Примеры».

Разрабатываемые группы заданий желательно снабжать *дополнительными комментариями*. Эти комментарии не отображаются в окне задачника, однако включаются в html-описание группы в виде *преамбулы группы* (html-описание группы можно создать, либо вызвав в программе процедуру Task с параметром

вида '<имя группы>#', например, 'TMDemoAbc#', либо воспользовавшись кнопкой  в окне среды PascalABC.NET. Кроме того, большие группы заданий целесообразно разделять на несколько подгрупп, в каждую из которых также можно добавлять комментарии (*преамбулы подгрупп*). В преамбулах, как и в формулировках заданий, можно использовать специальные [управляющие последовательности](#), которые позволяют отформатировать текст требуемым образом (в частности, обеспечивают выделение переменных, позволяют использовать в тексте верхние и нижние индексы, специальные символы и т. д.). Форматирование используется и при отображении формулировки задания в окне задачника, и при генерации html-страницы с описанием группы заданий. См. также [пункт, посвященный добавлению комментариев](#), в разделе «Примеры».

Настройка проектов для языка PascalABC.NET и особенности их отладки

Для разработки новой группы заданий в среде PascalABC.NET достаточно подготовить pas-файл с именем, совпадающим с именем создаваемой библиотеки, и структурой, описанной в пункте [«Структура проекта с описанием группы заданий»](#). Для данного файла должен быть доступен модуль PT4TaskMakerNET; он может размещаться в этом же каталоге или в подкаталоге Lib системного каталога среды PascalABC.NET.

При нажатии клавиши [F9] будет создана динамическая библиотека с новой группой заданий.

Поскольку в среде PascalABC.NET не предусмотрены средства для определения главного приложения при разработке библиотек, для тестирования новой группы заданий необходимо использовать еще одно приложение, заготовку для которого можно создать с помощью программного модуля PT4Load уже после первой успешной компиляции динамической библиотеки. Впрочем, заготовка является настолько простой, что ее можно создать и непосредственно. Например, для тестирования группы с именем TMDemoAbc достаточно воспользоваться следующей программой:

```
uses PT4;  
begin  
  Task( 'TMDemoAbc1?' );  
end.
```

При запуске этой программы на экран будет выведено окно задачника для группы TMDemoAbc в демонстрационном режиме, причем в качестве текущего будет выбрано первое задание данной группы (демонстрационный режим устанавливается благодаря символу «?», указанному после имени задания). Для отображения в окне задачника сразу после запуска программы другого существующего задания этой группы достаточно указать его номер в параметре процедуры Task. Можно также удалить номер, указав символ «?» сразу после имени группы заданий: `Task('TMDemoAbc?')`. В этом случае при запуске программы будет отображаться последнее задание, входящее в группу.

Модуль PT4TaskMakerNET: основные компоненты

Если процедура имеет необязательные параметры, то в списке параметров они заключаются в квадратные скобки.

Определение общих характеристик группы заданий

При создании новой группы заданий требуется определить следующие характеристики этой группы:

- *имя группы* (GroupName) — текстовая строка, содержащая от 1 до 25 символов — цифр и латинских букв, причем последний символ не может быть цифрой (если имя группы содержит более 9 символов, то она считается особой *библиотечной группой*, работа с которой отличается от работы с обычной группой); запрещается использовать имена стандартных групп задачника (Begin, Integer и т. д.); имена, различающиеся только регистром букв, считаются совпадающими;
- *описание группы* (GroupDescription) — непустая текстовая строка с кратким описанием данной группы; при генерации полного описания группы в виде html-страницы данная строка указывается в качестве *заголовка* этого описания;
- *сведения об авторе* (GroupAuthor) — текстовая строка с информацией о разработчике данной группы (фамилия, инициалы, год разработки, e-mail и т. п.; строка может быть пустой);
- *ключ группы* (GroupKey) — непустая текстовая строка с произвольным набором символов, позволяющая в дальнейшем идентифицировать в файле результатов results.dat и results.abс те выполненные задания, которые относятся к данной группе;
- *количество заданий в группе* (TaskCount) — целое число в диапазоне от 1 до 999, определяющее количество заданий в группе;
- *основная процедура группы заданий* (InitTaskProc) — процедура с одним целочисленным параметром, обеспечивающая инициализацию всех заданий данной группы (параметр данной процедуры определяет номер задания в пределах группы).

Из перечисленных характеристик в дополнительном комментарии нуждается *ключ группы*. Если не использовать подобную характеристику, то становится невозможной идентификация группы, к которой относятся задания, выполненные учащимся.

Действительно, имя задания, сохраненное в файле результатов, не позволяет однозначно его идентифицировать, поскольку ничто не

мешает разработать другую группу с тем же именем и совершенно другими заданиями, после чего «подменить» ею исходную группу. Проблему решает использование ключа группы, который сложно подделать, так как он известен только разработчику группы. При успешном выполнении задания в файл результатов дополнительно записывается *идентификатор группы*, вычисляемый на основе ее ключа и позволяющий однозначно определить группу, к которой относится выполненное задание. Поскольку информация, связанная с идентификаторами групп, представляет интерес только для преподавателя, ознакомиться с ней можно только с помощью программы «Контрольный центр преподавателя», входящей в комплекс Teacher Pack.

Примечание. При выводе краткого описания группы в программных модулях PT4Demo и PT4Load первый символ этого описания преобразуется к нижнему регистру (поскольку текст описания располагается в этих модулях после двоеточия). Если понижать регистр первого символа не следует (в случае, если этот символ является началом фамилии или некоторой аббревиатуры, например, «ЕГЭ»), то в начале краткого описания группы надо указать дополнительный символ-метку «^» (шапочка). Пример использования символа «^» приводится в разделе «[Разработка групп заданий, связанных с ЕГЭ по информатике](#)».

Для определения характеристик новой группы необходимо вызвать процедуру CreateGroup, указав эти характеристики в качестве параметров:

```
procedure CreateGroup(GroupName, GroupDescription, GroupAuthor,  
    GroupKey: string; TaskCount: integer; InitTaskProc: TInitTaskProc)
```

Тип TInitTaskProc определяется следующим образом:

```
type TInitTaskProc = procedure(n: integer);
```

Процедуру CreateGroup необходимо вызывать в процедуре inittaskgroup, которая должна экспортироваться библиотекой, содержащей данную группу.

Процедура CreateGroup контролирует правильность переданных ей параметров и в случае ошибки выводит на экран информационное

окно с ее описанием. В подобной ситуации все последующие действия, связанные с определением данной группы, игнорируются, и группа не включается в список доступных для использования групп заданий. Перечислим некоторые из возможных ошибок:

- в процедуре `inittaskgroup` определяется более одной группы заданий (в этом случае определения всех групп, кроме первой, игнорируются);
- имя группы не соответствует имени `dll`-файла, в котором данная группа определяется (напомним, что имя `dll`-файла должно иметь вид `PT4<имя группы>` или `PT4<имя группы><маркер локали>`); при реализации группы в виде `rcs`-файла данное ограничение отсутствует;
- к задачнику `Programming Taskbook` уже подключена группа с указанным именем;
- имя группы не является допустимым (в частности, совпадает с именем одной из базовых групп задачника);
- не указано краткое описание группы;
- не указан ключ группы;
- количество заданий не принадлежит диапазону 1–999;
- процедурная переменная `InitTaskProc` содержит нулевую ссылку.

Базовые константы и процедуры для создания новых заданий

```
const  
  xCenter = 0;  
  xLeft = 100;  
  xRight = 200;
```

Эти константы, отвечают за выравнивание данных по горизонтали: константа `xCenter` центрирует текст, связанный с элементом данных, относительно всей экранной строки, константы `xLeft` и `xRight` центрируют текст в пределах левой и правой половины экранной строки соответственно. Используются в качестве параметра `X` в процедурах групп `Data` и `Result`, а также в процедуре `TaskText`.

```
procedure CreateTask([SubgroupName: string]);
```

Данная процедура должна быть вызвана первой при инициализации нового задания; в качестве необязательного параметра `SubgroupName` указывается заголовок *подгруппы*, в которую включается задание (задания целесообразно разбивать на подгруппы, если их количество в группе является достаточно большим; в случае деления группы на подгруппы *каждое задание рекомендуется связывать с какой-либо подгруппой*). Если параметр является пустой строкой или отсутствует, то задание не связывается с какой-либо подгруппой. В окне задачника заголовок подгруппы выводится над именем задания; если подгруппа для данного задания не указана, то выводится краткое описание всей группы (определенное в параметре `GroupDescription` процедуры `CreateGroup`). При выводе краткого описания группы или заголовка подгруппы в окне задачника его текст преобразуется к верхнему регистру.

В версии 4.9 конструктора учебных заданий к первоначальным двум вариантам процедуры `CreateTask` были добавлены еще два варианта, предназначенные для инициализации задания по [параллельному MPI-программированию](#).

```
procedure TaskText(S: string[; X, Y: integer]);
```

Данная процедура добавляет к *формулировке задания* строку *S*, которая располагается в строке *Y* (от 1 до 5) раздела формулировки задания, начиная с позиции *X*. Позиции нумеруются от 1; при указании параметра *X* следует учитывать, что ширина раздела формулировок (как и разделов исходных и результирующих данных) равна 78 символам. Кроме явного указания значения позиции *X* можно использовать специальные константы *xCenter*, *xLeft* и *xRight*; в частности, если параметр *X* равен 0, то строка центрируется. Рекомендуется всегда центрировать строки в формулировках заданий (как это делается в базовых группах, входящих в задачник); явное указание позиции *X* следует использовать лишь при выводе многострочных формул и в других случаях специального выравнивания текста. Все строки должны добавляться к формулировке *последовательно*; при этом если формулировка содержит 1 строку, то ее следует располагать на экранной строке с номером 3, если 2 строки — на экранных строках 2 и 4, если 3 строки — на экранных строках 2, 3 и 4, если 4 строки — на экранных строках с номерами от 2 до 5 (именно так оформляются задания в базовых группах задачника). Нарушение порядка добавления строк не проявится при отображении формулировки в окне задачника, однако приведет к неверному выводу формулировки в html-описании группы.

Кроме пяти строк с основным текстом формулировки, который отображается на экране при выводе задания, можно указывать *дополнительные строки*, отображаемые на экране при прокрутке текста задания (связанные с прокруткой кнопки отображаются в окне задачника справа от раздела формулировок, если в формулировке текущего задания имеются дополнительные строки). Все дополнительные строки, как и основные, должны добавляться к формулировке *последовательно*, причем параметр *Y* для таких строк надо положить равным 0. Максимальное количество дополнительных строк равно 200.

В строке *S* можно использовать [управляющие последовательности](#).

Если при выводе строки *S* часть ее не умещается на экранной строке, то выводится сообщение об ошибке «Ошибочное позиционирование по горизонтали». Если ошибка произошла при

выводе основной строки, то лишняя часть строки S отображается на следующей строке (или на первой строке, если ошибочной является пятая строка в разделе формулировок).

Определенный с помощью процедур TaskText текст формулировки задания используется также при формировании html-описания группы. В этом случае деление на строки, указанное для экранного вывода, игнорируется, однако учитываются дополнительные управляющие последовательности, позволяющие разбивать текст на абзацы с различным способом выравнивания (на отображение текста в окне задачника эти дополнительные последовательности не влияют).

Если при определении задания не указана его формулировка, то выводится сообщение об ошибке.

Вариант процедуры TaskText с единственным параметром S добавлен в версию 4.11 конструктора. В этом варианте строка S должна содержать весь текст формулировки, причем строки формулировки должны разделяться символами #13, #10 или их комбинациями #13#10. Начальные и конечные пробелы в каждой строке формулировки удаляются; если в результате какая-либо строка окажется пустой, то она не учитывается. Все строки формулировки автоматически центрируются по горизонтали; их вертикальное расположение определяется количеством строк и соответствует правилам, приведенным выше (если формулировка содержит одну строку, то она располагается на экранной строке 3, и т. д.). Если требуется специальное выравнивание какой-либо строки текста, то его можно добиться за счет добавления дополнительных пробелов в начало или конец строки; чтобы эти пробелы не были удалены, первый начальный (или последний конечный) пробел должен быть экранирован символом «\» (обратная косая черта).

```
procedure DataB ([Cmt: string;] B: boolean; X, Y: integer);  
procedure DataN([Cmt: string;] N: integer; X, Y, W: integer);  
procedure DataN2([Cmt: string;] N1, N2: integer; X, Y, W: integer);  
procedure DataN3([Cmt: string;] N1, N2, N3: integer; X, Y, W: integer);  
procedure DataR([Cmt: string;] R: real; X, Y, W: integer);  
procedure DataR2([Cmt: string;] R1, R2: real; X, Y, W: integer);  
procedure DataR3([Cmt: string;] R1, R2, R3: real; X, Y, W: integer);
```

```
procedure DataC([Cmt: string;] C: char; X, Y: integer);  
procedure DataS([Cmt: string;] S: string; X, Y: integer);
```

Процедуры группы Data добавляют к заданию *элементы исходных данных*. Добавленные элементы, вместе с необязательной строкой-комментарием Cmt, отображаются в разделе исходных данных, начиная с позиции X строки Y (позиции и строки нумеруются от 1; ширина экранной строки равна 78 позициям). Если используются значения параметра Y, большие 5, то разделе исходных данных будет доступна [прокрутка](#). Как и для процедуры TaskText, параметр X может принимать три особых значения: 0 (центрирование по горизонтали относительно всей экранной строки), 100 (центрирование по горизонтали относительно левой половины экранной строки), 200 (центрирование по горизонтали относительно правой половины экранной строки). Эти значения можно также задавать с помощью констант xCenter, xLeft и xRight.

Параметр W определяет *ширину поля вывода* для числовых данных (выравнивание всегда производится по правому краю поля вывода). Если ширины поля вывода недостаточно, то значение параметра W игнорируется, и для вывода элемента используется минимально необходимое число экранных позиций. При определении ширины поля вывода для вещественного числа следует учитывать размер отображаемой дробной части (который определяется процедурой SetPrecision, описываемой далее).

Для нечисловых данных ширина поля вывода полагается равной фактической ширине данных; в частности, для данных символьного типа отводятся 3 позиции, содержащие начальный апостроф, собственно символ и конечный апостроф, а для логического типа отводятся 5 позиций, достаточных для вывода названий обеих логических констант в любом используемом языке программирования. Для строки отводятся $L + 2$ позиции, где L — длина строки (начальная и конечная позиции используются для вывода апострофов). В зависимости от текущего языка программирования используются либо одинарные, либо двойные апострофы.

Используя процедуры группы Data, в задание можно включить до 200 различных скалярных исходных данных (при этом следует

учитывать, что некоторые процедуры, например, DataN2 и DataN3, добавляют в набор исходных данных несколько элементов). Наложение различных элементов в разделе исходных данных задачиком не контролируется, поэтому при размещении данных следует обращать особое внимание на то, чтобы последующие элементы не скрывали предыдущие. Кроме того, важен порядок определения исходных данных, так как именно в этом порядке данные будут передаваться программе учащегося, выполняющей это задание. Следует придерживаться стандартных правил, принятых в базовых группах задачника: данные должны перебираться по строкам (в направлении сверху вниз), а в пределах каждой строки — слева направо.

В параметре Cmt, содержащем текст комментария к определяемому элементу исходных данных, можно использовать управляющие последовательности (например, для отображения индексов).

В любом задании должен быть задан хотя бы один элемент исходных данных; в противном случае выводится сообщение об ошибке.

Варианты данных процедур, в которых параметр Cmt отсутствует, добавлены в версию 4.11 конструктора.

```
procedure DataComment(Cmt: string; X, Y: integer);
```

Процедура позволяет добавлять в раздел исходных данных отдельный комментарий Cmt, не связанный с каким-либо элементом исходных данных. Общее число отдельных комментариев, включаемых в разделы исходных и результирующих данных, не должно превосходить 200. Смысл параметров X и Y — тот же, что и для процедур группы [Data](#).

```
procedure ResultB ([Cmt: string;] B: boolean; X, Y: integer);  
procedure ResultN([Cmt: string;] N: integer; X, Y, W: integer);  
procedure ResultN2([Cmt: string;] N1, N2: integer; X, Y, W: integer);  
procedure ResultN3([Cmt: string;] N1, N2, N3: integer; X, Y, W: integer);  
procedure ResultR([Cmt: string;] R: real; X, Y, W: integer);  
procedure ResultR2([Cmt: string;] R1, R2: real; X, Y, W: integer);  
procedure ResultR3([Cmt: string;] R1, R2, R3: real; X, Y, W: integer);  
procedure ResultC([Cmt: string;] C: char; X, Y: integer);
```

```
procedure ResultS([Cmt: string;] S: string; X, Y: integer);
```

Процедуры данной группы добавляют к заданию *элементы результирующих данных* вместе с их контрольными значениями. Комментарии Cmt к результирующим данным сразу отображаются в разделе результатов. Контрольные значения отображаются в разделе «Пример верного решения». Смысл параметров X, Y и W — тот же, что и для процедур группы [Data](#). Задание может содержать до 200 скалярных элементов результирующих данных.

Как и в случае исходных данных, если элемент контрольных данных не помещается в поле, выделенном для его отображения (шириной W позиций), то параметр W игнорируется, и для вывода используется минимально необходимое число экранных позиций. Однако если элемент *результирующих* данных, переданный в задачник программой, решающей задание, не «уложится» в размер, выделенный для соответствующего элемента контрольных данных, то в правой позиции поля вывода для этого элемента отобразится символ «*» (звездочка) красного цвета. Подобная ситуация возможна как для чисел, так и для строк (если программа учащегося выведет число или строку, размер которых больше требуемого). Для того чтобы в этой ситуации увидеть полный текст всех подобных элементов результирующих данных, следует переместить курсор мыши в раздел результатов в окне задачника; через 1–2 секунды полный текст всех данных, размер которых превышает допустимый, появится во всплывающей подсказке.

Порядок вызова процедур группы Result важен, так как он соответствует порядку, в котором результирующие данные, полученные программой учащегося, должны передаваться задачнику для проверки их правильности. Поэтому, как и для исходных данных, для набора результирующих данных должен соблюдаться стандартный порядок их размещения: сверху вниз по строкам и слева направо в каждой строке.

В параметре Cmt, содержащем текст комментария к определяемому элементу результирующих данных, можно использовать управляющие последовательности.

Проверка правильности результатов, полученных программой

учащегося, выполняется путем сравнения *текста*, изображающего эти результаты в окне задачника, с текстом, изображающим соответствующие контрольные данные. Это означает, в частности, что вычислительная погрешность, возникающая при обработке вещественных чисел, не будет влиять на проверку правильности, если при отображении этих чисел не используется слишком большое количество дробных знаков (напомним, что число дробных знаков можно задать с помощью процедуры `SetPrecision`).

В любом задании должен быть задан хотя бы один элемент результирующих данных; в противном случае выводится сообщение об ошибке.

Варианты данных процедур, в которых параметр `Cmt` отсутствует, добавлены в версию 4.11 конструктора.

```
procedure ResultComment(Cmt: string; X, Y: integer);
```

Процедура позволяет добавлять в раздел результатов отдельный комментарий `Cmt`, не связанный с каким-либо элементом результирующих данных. Общее число отдельных комментариев, включаемых в разделы исходных и результирующих данных, не должно превосходить 200. Смысл параметров `X` и `Y` — тот же, что и для процедур группы [Data](#).

```
procedure SetPrecision(N: integer);
```

Процедура устанавливает количество `N` дробных знаков, используемое в дальнейшем при выводе всех элементов данных вещественного типа. По умолчанию количество дробных знаков равно 2. Если оно равно 0, то вещественные данные отображаются в экспоненциальном формате, а количество дробных знаков определяется шириной поля вывода, указанной для данного числа. Действие текущей настройки, определенной процедурой `SetPrecision`, продолжается до очередного вызова этой процедуры, однако не распространяется на другие учебные задания текущей группы. При отображении вещественных чисел в качестве десятичного разделителя всегда используется *точка*.

procedure SetRequiredDataCount(N: integer);

Процедура определяет минимально необходимое количество N элементов исходных данных, требуемое для правильного решения задания при текущем наборе исходных данных. По умолчанию это количество равно общему числу всех указанных в задании исходных данных. Если параметр N имеет нулевое или отрицательное значение, то выводится сообщение об ошибке; если значение параметра превышает общее число элементов исходных данных, то сообщение об ошибке не выводится, а требуемое количество исходных данных полагается равным их общему количеству.

Примером задания, в котором необходимо использовать данную процедуру, может служить задание Series10. В этом задании дается набор из N целых чисел и требуется вывести True, если данный набор содержит положительные числа, и False в противном случае. Ясно, что если при считывании элементов набора будет обнаружено положительное число, то можно сразу выводить значение True и завершать выполнение задания. Однако если при подготовке задания не указать минимально необходимое число исходных данных с помощью процедуры SetRequiredDataCount, то по умолчанию будет считаться, что для решения необходимо прочесть все исходные данные, и приведенный выше правильный вариант решения будет расценен как ошибочный (при этом будет выведено сообщение «Введены не все требуемые исходные данные»).

Если заданное с помощью процедуры SetRequiredDataCount количество требуемых исходных данных меньше их общего количества, то программа учащегося не обязана считывать все исходные данные: достаточно прочесть только требуемые. Однако если программа прочтет все данные и выведет правильный ответ, это также будет считаться верным вариантом решения.

При выполнении заданий по [параллельному программированию](#) во всех процессах параллельной программы должны быть введены все связанные с ними исходные данные. Поэтому попытка вызова процедуры SetRequiredDataCount(N) с параметром N, значение которого меньше общего числа исходных данных, приведет в задании по параллельному программированию к сообщению об ошибке.

```
procedure SetTestCount(N: integer);
```

Процедура определяет количество N успешных тестовых испытаний программы учащегося, необходимое для того, чтобы задание было зачтено как выполненное. По умолчанию количество тестовых испытаний полагается равным 5. Значение N должно находиться в пределах от 2 до 9; при указании других вариантов параметра N выводится сообщение об ошибке.

После каждого успешного тестового испытания в окне задачника выводится сообщение (на зеленом фоне), в котором указывается номер испытания и общее число тестов, необходимых для выполнения данного задания, например: «Верное решение. Тест номер 2 (из 5)». Если при очередном тестовом испытании программы ею будет получено ошибочное решение, то счетчик успешных тестов будет сброшен в 0, и тестирование (после исправления обнаруженной ошибки) придется начинать заново.

```
function CurrentTest: integer;
```

Данная функция добавлена в версию 4.11 конструктора учебных заданий. Она возвращает порядковый номер текущего тестового запуска, причем учитываются только успешные тестовые запуски. Если ранее успешных запусков не было, то функция возвращает 1. Если задание уже выполнено или было запущено в демонстрационном режиме, то функция возвращает 0.

При попытке подключения новой группы заданий, содержащей вызов функции CurrentTest, к задачнику более ранней версии (до 4.10 включительно) функция CurrentTest всегда возвращает 0.

Использование данной функции позволяет гарантировать включение в тестовые наборы специальных вариантов тестов (связанных с ситуациями, требующими особой обработки). В предыдущих версиях конструктора эти варианты выбирались только с применением датчика случайных чисел; это могло приводить к тому, что на протяжении требуемой серии тестов (которая не может превышать 9) особые варианты ни разу не генерировались. Используя функцию CurrentTest, особые варианты можно явно связать с тестовым

испытанием, имеющим определенный номер. Наряду с подобными «фиксированными» испытаниями при формировании задания следует предусматривать и испытания, при которых варианты тестов по-прежнему выбираются случайным образом; это позволит избежать ситуации (впрочем, маловероятной), при которой учащийся будет запускать разные программы для тестовых испытаний с различными порядковыми номерами. Необходимо также учитывать, что в ряде ситуаций функция CurrentTest возвращает особое значение 0, при котором также целесообразно выбирать тестовые варианты случайным образом.

В качестве примера приведем начальную часть процедуры, реализующей задание Array32, в котором требуется найти номер первого локального минимума. Очевидно, в данном задании следует предусмотреть особые варианты тестов, в которых первым локальным минимумом является первый или последний элемент исходного набора (а также «промежуточный вариант», в котором первый локальный минимум расположен во внутренней части набора). Ранее это обеспечивалось следующим образом (здесь n — размер исходного набора, k — порядковый номер первого локального минимума):

```
n := 5 + Random(6);
case Random(4) of
0: k := 1;
1: k := n;
2, 3: k := 2 + Random(n-2);
end;
...
```

При предусмотренных шести тестовых испытаниях вполне могло оказаться, что функция Random(4) ни разу не примет значения 0 или 1, и тем самым один или оба особых случая не будут протестированы. Возможна (хотя и менее вероятна) ситуация, при которой в течение всех испытаний ни разу не будут получены значения 2 и 3; тем самым не будет испытан «промежуточный вариант». В то же время было бы желательно, чтобы ошибка алгоритма, связанная с неверной обработкой одной из возможных ситуаций, была выявлена задачиком автоматически, до просмотра преподавателем текста программы.

В новом варианте процедуры после первого оператора case был добавлен второй, в котором для некоторых тестовых испытаний вариант для значения k задается явным образом:

```
n := RandomN(5, 10);
case Random(4) of
0: k := 1;
1: k := n;
2, 3: k := RandomN(2, n-1);
end;
case CurrentTest of
2: k := n;
3: k := RandomN(2, n-1);
5: k := 1;
end;
...
```

При этом для первого, четвертого и шестого (последнего) тестового испытания, а также при демонстрационных запусках, значение k по-прежнему выбирается случайным образом.

Заметим, что в новом варианте была также использована функция RandomN, добавленная в версию 4.11 конструктора.

После включения функции CurrentTest в конструктор были соответствующим образом модифицированы (без изменения формулировок) все группы заданий, включенные в базовый вариант задачника версии 4.11.

```
function RandomN(M, N: integer): integer;
function RandomR(A, B: real): real;
```

Вспомогательные функции, которые могут использоваться при генерации исходных данных с применением датчика случайных чисел. В явной инициализации датчика нет необходимости, поскольку подобная инициализация выполняется в процедуре CreateGroup. Указанные функции позволяют выполнять единообразную генерацию данных при использовании любого языка, поддерживаемого конструктором учебных заданий.

Функция `RandomN(M, N)` возвращает псевдослучайное целое число, лежащее в диапазоне от M до $N-1$ *включительно*. Если указанный диапазон пуст, то функция возвращает M .

Функция `RandomR(A, B)` возвращает псевдослучайное вещественное число, лежащее на полуинтервале $[A, B)$. Если указанный полуинтервал пуст, то функция возвращает A .

При генерации заданий можно применять и стандартную функцию `Random` языка Pascal.

```
function Center(I, N, W, B: integer): integer;
```

Вспомогательная функция, которая позволяет размещать по центру экранной строки набор из N элементов данных одинаковой ширины. Эта функция возвращает горизонтальную координату, начиная с которой следует выводить I -й элемент набора (I меняется от 1 до N) при условии, что ширина каждого элемента равна W позициям, а между элементами надо указывать B пробелов. Функция `Center` обычно используется в качестве параметра X в процедурах групп `Data` и `Result` при выводе однотипных наборов данных (в частности, элементов массива).

В качестве примера приведем фрагмент, обеспечивающий формирование и вывод в разделе исходных данных массива вещественных чисел:

```
n := RandomN(2, 10);  
DataN('N = ', n, 0, 2, 1);  
for i := 1 to n do  
begin  
    a[i] := RandomR(-9.99, 9.99);  
    DataR(a[i], Center(i, n, 5, 1), 4, 5);  
end;
```

Вначале (во второй строке области исходных данных) выводится размер N массива, определяемый с помощью датчика случайных чисел и принимающий значения в диапазоне от 2 до 10 (он снабжается комментарием « $N =$ »). Затем (в четвертой строке) выводятся сами элементы массива, причем благодаря использованию функции `Center` весь список выравнивается относительно центра экранной строки независимо от количества элементов. Целые части всех элементов лежат в диапазоне от -9 до 9, т. е. представляются одной цифрой, одна позиция отводится под знак числа, еще одна — под отображение десятичного разделителя-

точки; наконец, по умолчанию указываются два дробных знака, поэтому для каждого элемента следует выделить 5 экранных позиций; это число указывается дважды: как второй параметр функции Center и как последний параметр процедуры DataR. Промежуток между элементами полагается равным 1 экранной позиции (это последний, четвертый параметр функции Center).

При использовании функции Center строку комментария следует оставлять пустой (начиная с версии 4.11 конструктора, в этом случае строку комментария можно просто не указывать).

Прокрутка разделов исходных данных и результатов

Начиная с версии 4.9 конструктора учебных заданий, для процедур групп Data и Result, в том числе DataComment и ResultComment, в качестве параметра Y разрешено указывать значение, *превышающее 5*. Если значение параметра Y для некоторого элемента раздела исходных данных превышает 5, то этот элемент размещается в строке с указанным номером, а в разделе исходных данных становится доступной *прокрутка*. Аналогичным образом прокрутка будет доступна в разделе результатов, если хотя бы один элемент этого раздела помещен в него процедурой с параметром Y, превышающим 5. Если оба раздела допускают прокрутку, то она выполняется независимо. Прокрутку в любом разделе можно выполнять с помощью клавиатуры или мыши; в последнем случае следует использовать полосы прокрутки, расположенные справа от прокручиваемого раздела. Прокрутка может также выполняться с помощью колесика мыши.

В задании запрещено использовать прокрутку раздела, если в нем уже имеется «внешний» объект ([файл](#) или [динамическая структура](#)). Если делается попытка вызвать какую-либо процедуру с параметром Y, большим 5, для раздела, уже содержащего внешний объект, то выводится сообщение об ошибке *«При наличии внешних объектов режим прокрутки для всего раздела недоступен»*. Если же в разделе, уже имеющем элементы данных или комментарии, размещенные в неотображаемых строках, делается попытка разместить внешний объект, то выводится сообщение об ошибке *«Раздел данных в режиме прокрутки не может содержать внешние объекты»*.

Возможность прокрутки разделов исходных и результирующих данных добавлена, прежде всего, для использования в заданиях по [параллельному программированию](#). Однако она может оказаться полезной и в других случаях, например, при использовании в качестве исходных данных нескольких двумерных массивов или массива строк. Заметим, что ни в одном из 1300 заданий, входящих в базовый набор задачника Programming Taskbook, прокрутка разделов исходных и результирующих данных не используется. Большое количество заданий с прокруткой разделов исходных и

результатирующих данных содержится в группе Align, входящей в задачник по строковым алгоритмам биоинформатике Programming Taskbook for Bio.

В режиме окна с динамической компоновкой, появившемся в версии 4.11 задачника, отдельная прокрутка разделов не поддерживается, поэтому описанные в данном пункте возможности приводят в данном режиме лишь к увеличению высоты соответствующего раздела задания.

Импортирование существующих заданий в новую группу

```
procedure UseTask(GroupName: string; TaskNumber: integer);
```

Данная процедура позволяет *импортировать* в создаваемую группу задание с номером TaskNumber из группы GroupName. Она обычно вызывается непосредственно в основной процедуре группы. Если импортируемое задание не найдено, то при попытке его запуска в окне задачника выводится сообщение «Задание не реализовано для текущего языка программирования», и этот же текст, выделенный курсивом, указывается в html-описании группы после имени, которое должно быть связано с импортированным заданием.

При использовании мини-варианта задачника импортированные задания будут доступны для выполнения только в том случае, если они доступны для выполнения в исходных группах.

В параметре GroupName после имени группы можно дополнительно указывать *поправку для вычисления ссылки на другое задание* (поправка является целым числом и отделяется от имени группы символом #). Например, если в группу Demo в качестве задания Demo10 импортируется задание Proc46, а в качестве Demo11 — задание Proc49, ссылающееся на Proc46, то при импортировании задания Proc49 необходимо указать поправку, равную 2. Если этого не сделать, то в формулировке задания Demo11 будет указана ссылка не на задание Demo10, а на задание Demo8 (поскольку оно находится «на том же расстоянии» от задания Demo11, что и задание Proc46 относительно задания Proc49). Добавление поправки 2 должно быть оформлено следующим образом:

```
UseTask( 'Proc#2' , 49).
```

Документирование группы заданий

Группы заданий можно снабжать комментариями, делая их «самодокументируемыми». Комментарии можно добавлять не только к группе, но и к ее *подгруппам*, т. е. наборам подряд идущих заданий в пределах группы (для включения задания в определенную подгруппу необходимо указать заголовок этой подгруппы в качестве параметра процедуры [CreateTask](#)).

Комментарии не отображаются в окне задачника, но включаются в html-описание группы. Они располагаются между заголовком группы (подгруппы) и формулировками заданий. Таким образом, эти комментарии представляют собой *преамбулы* к группе или ее подгруппам.

Определять преамбулу к подгруппе имеет смысл только в случае, если с этой подгруппой связаны некоторые задания, входящие в определяемую группу. Если группа не содержит заданий, связанных с некоторой подгруппой, то преамбула этой подгруппы в html-описании не выводится.

Для определения преамбул предназначены следующие процедуры.

```
procedure CommentText(S: string);
```

Данная процедура добавляет содержимое строки S к текущей преамбуле, отделяя это содержимое от предыдущего текста преамбулы пробелом. В строке S можно использовать [управляющие последовательности](#), обеспечивающие ее форматирование. Например, для перехода к новому абзацу преамбулы следует использовать последовательность \P (управляющие последовательности чувствительны к регистру букв).

```
procedure UseComment(GroupName: string[]; SubgroupName: string);
```

Процедура UseComment добавляет к текущей преамбуле текст преамбулы подгруппы SubgroupName группы GroupName или, если параметр SubgroupName является пустой строкой или отсутствует, текст преамбулы самой группы GroupName. Этот текст отделяется от предыдущего текста преамбулы пробелом. Регистр символов в

параметрах `GroupName` и `SubgroupName` может быть произвольным.

Если группа с именем `GroupName` не найдена или в ней отсутствует подгруппа `SubgroupName`, то процедура не выполняет никаких действий; сообщение об ошибке в этом случае не выводится.

Процедуры `CommentText` и `UseComment` должны вызываться после функции `CreateGroup`; при этом они определяют преамбулу данной группы. Для того чтобы они определяли преамбулу какой-либо подгруппы данной группы, перед их вызовом необходимо вызвать процедуру `Subgroup`, описываемую далее.

```
procedure Subgroup(SubgroupName: string);
```

Данная процедура устанавливает режим добавления текста к преамбуле *подгруппы* `SubgroupName` текущей группы. Этот режим сохраняется до следующего вызова данной процедуры или до завершения определения текущей группы заданий (определение группы, создаваемой в виде `dll`-файла, завершается при выходе из процедуры `inittaskgroup`).

Процедуру `Subgroup` можно вызывать несколько раз для одной и той же подгруппы, при этом ранее определенный текст преамбулы будет дополняться новыми данными. При вызове процедуры `Subgroup` с параметром — пустой строкой устанавливается режим дополнения преамбулы *группы* (напомним, что этот режим устанавливается также сразу после вызова процедуры `CreateGroup`).

Константы и функции для определения текущего состояния задачника

const

```
lgPascal = $0001;  
lgVB = $0002;  
lgCPP = $0004;  
lg1C = $0040;  
lgPython = $0080;  
lgCS = $0100;  
lgVBNET = $0200;  
lgPascalNET = $0400;  
lgJava = $10000;  
lgWithPointers = $003D;  
lgWithObjects = $FFF80;  
lgNET = $FF00;  
lgPascalABCNET = $0401;  
lgAll = $FFFFFF;
```

Данные константы, совместно с описываемой далее функцией `CurrentLanguage`, позволяют определить язык программирования, на который в данный момент (т. е. в момент инициализации текущей группы заданий) настроен задачник. Константы `lgPascal`, `lgVB`, `lgCPP`, `lgCS`, `lgVBNET`, `lgPascalABCNET`, `lg1C`, `lgPython`, `lgJava` соответствуют конкретному языку из числа тех, которые доступны в текущей версии задачника (Pascal, Visual Basic, C++, C#, Visual Basic .NET, PascalABC.NET, 1С:Предприятие, Python, Java). Эти константы являются *битовыми флагами*. Константа `lg1C` появилась в версии 4.9 конструктора учебных заданий (в связи с реализацией комплекса **PT for 1C** — варианта задачника для системы 1С:Предприятие), константа `lgPython` — в версии 4.10 (в связи с реализацией варианта задачника для языка Python), константа `lgJava` — в версии 4.11 (в связи с реализацией варианта задачника для языка Java).

Некоторые константы являются комбинациями битовых флагов (т. е. *битовыми масками*) и позволяют определить, к какой категории относится текущий язык:

- `lgAll` — любой язык,
- `lgNET` — язык платформы .NET (языки C# и Visual Basic .NET),
- `lgWithPointers` — язык, для которого можно разрабатывать

- группы заданий на обработку динамических структур с применением указателей (языки Pascal и C++),
- IgWithObjects — язык, для которого можно разрабатывать группы заданий на обработку динамических структур с применением объектов (все языки платформы .NET, а также Python и Java).

Особое место занимает язык, реализованный в системе PascalABC.NET, поскольку в нем объединяются свойства обычного языка Pascal и языка платформы .NET. Данному языку соответствует комбинация флагов IgPascal и IgPascalNET; это, в частности, означает, что он принадлежит одновременно к категориям IgWithPointers, IgWithObjects и IgNET. Для языка PascalABC.NET предусмотрена также именованная константа IgPascalABCNET.

```
function CurrentLanguage: integer;
```

Функция возвращает значение, соответствующее языку программирования, на который в данный момент настроен задачник. Помимо сравнения возвращаемого значения функции с константами, соответствующими конкретному языку, можно также использовать данную функцию для определения *категории*, к которой относится текущий язык программирования; в этом случае необходимо применять побитовые операции. Например, для проверки того, что текущий язык программирования относится к категории языков платформы .NET, достаточно проверить истинность следующего условия:

```
CurrentLanguage and IgNET <> 0
```

При использовании задачника совместно с системой PascalABC.NET функция CurrentLanguage возвращает значение [IgPascalABCNET](#).

```
function CurrentLocale: string;
```

Функция возвращает строку, соответствующую текущей *локали*, т. е. текущему языку интерфейса, используемому в задачнике. В версии 4.11 конструктора учебных заданий возможными возвращаемыми значениями функции CurrentLocale являются 'ru' (русский вариант задачника) и 'en' (английский вариант).

```
function CurrentVersion: string;
```

Данная функция добавлена в версию 4.10 конструктора учебных заданий. Она возвращает номер текущей версии задачника в виде строки числа формата 'd.dd'. Например, в случае версии 4.11 возвращается строка '4.11'. Для версий, предшествующих версии 4.10, функция возвращает строку '4.00'.

Образцы слов и предложений

Приведенные ниже элементы конструктора PT4TaskMaker позволяют получить доступ к встроенным в него образцам текстовых исходных данных: словам (Word), предложениям (Sentence) и многострочным текстам (Text).

const

```
SampleError = '#ERROR?';  
MaxLineCount = 50;
```

```
function WordCount: integer;  
function SentenceCount: integer;  
function TextCount: integer;  
function WordSample(N: integer): string;  
function SentenceSample(N: integer): string;  
function TextSample(N: integer): string;
```

```
function EnWordCount: integer;  
function EnSentenceCount: integer;  
function EnTextCount: integer;  
function EnWordSample(N: integer): string;  
function EnSentenceSample(N: integer): string;  
function EnTextSample(N: integer): string;
```

Функции WordSample, SentenceSample и TextSample возвращают текстовые данные, соответствующие текущей *локали*, т. е. текущему языку интерфейса, используемому в задачнике (см. функцию [CurrentLocale](#)): для русского варианта задачника возвращаются русские данные, для английского — английские. Варианты этих функций, снабженные префиксом En, возвращают английские текстовые данные в *любом* варианте задачника.

Функции, оканчивающиеся словом Count, возвращают *количество* соответствующих элементов данных. В версии 4.11, конструктора учебных заданий, как и в его предыдущих версиях, доступно 116 слов, 61 предложение и 85 текстов как на русском, так и на английском языке.

Функции WordSample/EnWordSample и SentenceSample/EnSentenceSample возвращают соответственно слово или предложение с индексом N (индексирование проводится

от 0).

Функция `TextSample/EnTextSample` возвращает строку, связанную с многострочным текстом, имеющим индекс `N` (индексирование также проводится от 0). При этом между соседними строками этого текста располагаются символы `#13#10` (маркеры конца строки). В конце текста маркер конца строки отсутствует, число строк в тексте не превышает значения константы `MaxLineCount`. Любой текст состоит из нескольких абзацев; между абзацами текста помещается одна пустая строка, отступы в начале абзацев («красная строка») не используются. В тексте не используются также переносы слов.

Если параметр `N` является недопустимым, то все функции возвращают особую строку, равную константе `SampleError`.

Буква «ё» в русских текстовых данных не используется.

Все слова-образцы состоят из заглавных (прописных) букв. Помимо слов «общего вида» в набор слов включены слова, обладающие следующими особенностями (наличие подобных особых слов может оказаться полезным при составлении заданий):

- слова, начинающиеся и оканчивающиеся одной и той же буквой;
- слова, содержащие три одинаковые буквы (в русском наборе — три буквы «А», в английском наборе — три буквы «Е»).

Длина предложений-образцов не превосходит 76 символов; таким образом, любое предложение умещается на одной экранной строке (напомним, что строки при выводе в окне задачника обрамляются апострофами).

Многострочные тексты предназначены для использования, прежде всего, в заданиях на обработку текстовых файлов (см. реализацию подобного задания в разделе «[Примеры](#)»).

Модуль PT4TaskMakerNET: дополнительные компоненты

Если процедура имеет необязательные параметры, то в списке параметров они заключаются в квадратные скобки.

Процедуры для включения в задание файлов

В конструкторе учебных заданий PT4TaskMaker предусмотрена возможность включения в каждое учебное задание (в качестве исходных или результирующих данных) до 10 файлов. Кроме текстовых файлов в заданиях можно использовать двоичные файлы, все элементы которых имеют один и тот же тип (целочисленный, вещественный, символьный, строковый). Каждый файл должен содержать не более 999 элементов (для текстовых файлов элементами считаются файловые строки); в случае, если файл содержит более 999 элементов, элементы с номерами, превышающими 999, в окне задачника не отображаются. Для корректного отображения на экране, а также для правильной проверки результирующих файлов необходимо, чтобы строки в двоичных строковых и текстовых файлах и text состояли из не более чем 70 символов.

Все процедуры, связанные с определением файловых данных, следует вызывать после вызова процедуры [CreateTask](#).

Файлы должны создаваться в текущем каталоге, поэтому при задании их имен не следует указывать имя диска и путь. Рекомендуется снабжать имена всех файлов, используемых в заданиях, расширением .tst.

Все данные из файла, как правило, нельзя одновременно отобразить в окне задачника, поэтому для файловых элементов предусмотрена возможность *прокрутки*. Для двоичных файлов прокрутка выполняется в горизонтальном направлении, а для текстовых файлов — в вертикальном.

```
procedure DataFileN(FileName: string; Y, W: integer);  
procedure DataFileR(FileName: string; Y, W: integer);  
procedure DataFileC(FileName: string; Y, W: integer);  
procedure DataFileS(FileName: string; Y, W: integer);  
procedure DataFileT(FileName: string; Y1, Y2: integer);
```

Процедуры группы DataFile с именами, завершающимися символами N, R, C, S, позволяют включить в задание в качестве исходного файла один двоичный файл с элементами целочисленного,

вещественного, символьного и строкового типа соответственно. Процедура DataFileT позволяет включить в задание в качестве исходного файла один текстовый файл. К моменту вызова процедуры файл, включаемый в задание, должен быть создан, заполнен исходными данными и закрыт. Имя этого файла передается параметром FileName.

Два последних параметра имеют разный смысл для процедур, обрабатывающих двоичные файлы, и для процедуры DataFileT, обрабатывающей текстовые файлы. Для процедур, связанных с двоичными файлами, параметр Y указывает номер экранной строки в области исходных данных, в которой будут отображаться элементы данного файла, а параметр W указывает количество позиций, отводимых под отображение *одного* элемента файла. Если фактическая длина элемента файла оказывается меньше параметра W, то изображение элемента дополняется пробелами (пробелы добавляются слева для числовых данных и справа для символьных); если длина элемента файла окажется больше значения W, то в конце поля, выделенного для его вывода, будет указан символ «*» (звездочка) красного цвета. При определении параметра W необходимо предусматривать дополнительные позиции для пробелов, служащих разделителями элементов, а в случае строковых и символьных файлов — для апострофов, автоматически добавляемых к каждому элементу при его отображении на экране. Способ отображения вещественных чисел устанавливается, как и для обычных исходных данных, процедурой [SetPrecision](#); по умолчанию вещественные числа отображаются в формате с фиксированной точкой и двумя знаками в дробной части. Количество элементов, отображаемых на экране, определяется автоматически так, чтобы заполнить по возможности всю экранную строку. Никакие другие исходные данные на этой строке размещать нельзя.

Для процедуры DataFileT параметры Y1 и Y2 определяют соответственно номер первой и последней экранной строки той части области исходных данных, которая отводится под отображение текстового файла. На каждой экранной строке размещается одна строка из текстового файла.

Параметры Y, Y1, Y2 должны принимать значения от 1 до 5 (Y и Y1

могут также принимать значение 0; этот случай описан в конце данного раздела); значение Y1 не должно превышать значение Y2. Параметр W должен лежать в диапазоне 1–72.

При наличии нескольких исходных файлов вызов соответствующих процедур группы DataFile может проводиться в любом порядке, независимо от порядка расположения этих файлов на экране. При попытке размещения двух файлов на одной экранной строке выводится сообщение об ошибке. Вызовы процедур группы DataFile могут проводиться как до, так и после вызовов процедур группы Data, определяющих «обычные», не файловые исходные данные.

Вызов процедур группы DataFile не влияет на содержимое включаемых в задание файлов. Он лишь приводит к копированию этого содержимого в специальный буфер в оперативной памяти. Созданная копия используется для отображения содержимого файла на экране; это позволяет просматривать начальное содержимое исходного файла и после его преобразования (или даже удаления) в ходе решения задания.

Поскольку при различных тестовых испытаниях учебного задания желательно не только изменять содержимое исходных файлов, но также и предлагать для обработки файлы с различными именами, возникает опасность «засорения» диска файлами, созданными при предыдущих при предыдущих тестовых испытаниях. Для того чтобы этого не произошло, в задачнике предусмотрено автоматическое удаление при завершении тестового испытания всех файлов, включенных в задание с помощью процедур группы DataFile. Заметим, что это удаление производится и в случае аварийного завершения программы, выполняющей учебное задание. Если же исходный файл был удален самой программой, выполняющей задание, то задачник не будет пытаться удалить этот файл еще раз.

Иногда (хотя и весьма редко — см., например, задание File4) при решении задания не требуется отображать содержимое исходного файла на экране. В этом случае в соответствующей процедуре группы DataFile параметр Y или Y1 надо положить равным 0. Если исходный файл не требуется ни отображать на экране, ни удалять после завершения тестового испытания, то в вызове процедуры группы DataFile нет необходимости.

Следует заметить, что формат *двоичных строковых файлов* является различным для разных языков программирования. В конструкторе предполагается, что подготовленный для включения в задание двоичный строковый файл имеет формат языка Pascal; в дальнейшем сам задачник выполняет автоматическое преобразование данного файла к формату того языка, на котором выполняется задание. При реализации задания в конструкторе для среды PascalABC.NET достаточно описать двоичный строковый файл как file of ShortString.

Пример реализации задания на обработку двоичных строковых файлов для различных языков программирования приводится в разделе «[Примеры](#)».

```
procedure ResultFileN(FileName: string; Y, W: integer);  
procedure ResultFileR(FileName: string; Y, W: integer);  
procedure ResultFileC(FileName: string; Y, W: integer);  
procedure ResultFileS(FileName: string; Y, W: integer);  
procedure ResultFileT(FileName: string; Y1, Y2: integer);
```

Процедуры группы ResultFile с именами, завершающимися символами N, R, C, S, позволяют включить в задание в качестве результирующего файла один двоичный файл с элементами целочисленного, вещественного, символьного и строкового типа соответственно. Процедура ResultFileT позволяет включить в задание в качестве результирующего файла один текстовый файл. К моменту вызова процедуры файл, включаемый в задание, должен быть создан, заполнен контрольными данными и закрыт. Под контрольными данными понимаются, как обычно, данные, которые должны содержаться в результирующем файле в случае правильного решения задания.

Смысл параметров процедур группы ResultFile совпадает со смыслом соответствующих параметров процедур группы DataFile, за исключением того, что теперь номера экранных строк Y, Y1, Y2 относятся к области результирующих данных. Ограничения на параметры для процедур группы ResultFile накладываются те же, что и для процедур группы DataFile.

В результате выполнения процедуры из группы ResultFile

содержимое указанного контрольного файла будет скопировано в специальный буфер в оперативной памяти, после чего контрольный файл будет *автоматически удален с диска*. В дальнейшем, при выполнении задания, файл с таким же именем должен быть создан и заполнен требуемыми данными программой самого учащегося. Контрольные данные, записанные в оперативную память процедурой из группы ResultFile, используются при проверке правильности содержимого результирующего файла (созданного в ходе выполнения задания). Кроме того, эти контрольные данные могут выводиться на экран в качестве примера правильного решения. Имя файла и другие параметры, указанные в процедуре ResultFile, будут также использоваться для поиска и отображения на экране результирующего файла, созданного при выполнении задания. Все результирующие файлы, созданные в ходе решения задания, автоматически удаляются с диска при завершении программы. Подобное удаление производится и при аварийном завершении программы; если же результирующий файл не создан, то попытка его удалить не производится.

Как и в случае процедур группы DataFile, отображение некоторых результирующих файлов можно отключить, положив в соответствующих процедурах ResultFile значения параметров Y или Y1 равными 0 (см., например, задание File1). Это, естественно, не отменит сравнения содержимого результирующих файлов с контрольными данными и удаления результирующих файлов при завершении программы.

При определении *двоичных строковых* файлов результатов необходимо следовать тем же правилам, что и при определении исходных двоичных строковых файлов (см. завершающую часть описания процедур [группы DataFile](#)).

Процедуры для включения в задание указателей и динамических структур данных

В учебные задания можно включать указатели только одного определенного в задачнике типа:

type

```
PNode = ^TNode;  
TNode = record  
  Data : integer;  
  Next : PNode;  
  Prev : PNode;  
  Left: PNode;  
  Right: PNode;  
  Parent: PNode;  
end;
```

Этот тип позволяет формировать одно- и двусвязные линейные динамические структуры (при этом используются поля связи Next и Prev), бинарные деревья и деревья общего вида (при этом используются поля связи Left и Right), а также бинарные деревья с обратной связью (при этом используются поля связи Left, Right и Parent). Поскольку значение адреса, хранящегося в указателе, не представляет интереса (и, кроме того, может изменяться при каждом тестовом запуске программы), на экране отображается не оно, а условное обозначение указателя «ptr», снабженное *обязательным* комментарием, например, $P_1 = ptr$, $P_x = ptr$ и т. д. В задании можно использовать до 36 различных указателей, которым присваиваются номера от 0 до 35. Указатели с номерами от 0 до 9 имеют комментарии с цифровым индексом (P_0 – P_9), а указатели с номерами от 10 до 35 — с буквенным (P_A – P_Z). Если некоторый указатель имеет нулевое значение, то вместо текста «ptr» отображается текст «nil» (или аналогичный текст с обозначением нулевого указателя, соответствующего текущему языку программирования), например, $P_1 = nil$ для языка Pascal, $P_1 = NULL$ для языка C++. Комментарии к указателям также используются при отображении на экране динамических структур, если они содержат элементы, с которыми связаны данные

указатели.

```
procedure SetPointer(NP: integer; P: PNode);
```

Эта процедура позволяет определить в учебном задании указатель с номером NP (значение этого указателя при инициализации задания будет равно P, однако при *выполнении* задания оно может измениться). Все прочие процедуры, связанные с этим указателем и описываемые далее, используют не его конкретное значение, а номер NP.

Номер NP должен лежать в диапазоне от 0 до 35; он указывается в обязательном комментарии к данному указателю (см. выше). Если процедура SetPointer для указателя с номером NP не вызвана, то указатель с этим номером будет иметь нулевое значение.

```
procedure DataP([Cmt: string;] NP: integer; X, Y: integer);  
procedure ResultP([Cmt: string;] NP: integer; X, Y: integer);
```

Процедуры DataP и ResultP помещают указатель с номером NP в список исходных или, соответственно, результирующих данных учебного задания и отображают его на экране. При отображении на экране указатель снабжается обязательным комментарием вида P_# =, где в качестве символа # указывается символ, связываемый с указателем (для NP от 0 до 9 — соответствующая цифра, для NP от 10 до 35 — заглавная латинская буква от A до Z). Само значение указателя на экран не выводится; вместо него указывается одно из двух условных обозначений: ptr для ненулевого указателя и nil для нулевого указателя. Как и прочие элементы данных, указатель может снабжаться дополнительным комментарием Cmt, который приписывается слева к обязательному (например, вызов процедуры с параметрами Cmt = 'Адрес начала стека: ' и NP = 6 приведет к выводу на экран следующей строки: **Адрес начала стека: P₆ = ptr**). В дополнительном комментарии Cmt можно использовать [управляющие последовательности](#). В процедурах DataP и ResultP задается также экранная позиция, начиная с которой элемент данных выводится в соответствующую экранную область. Параметр Y определяет номер строки (от 1 до 5), параметр X — позицию в

строке (от 1 до 78; как обычно, требуется, чтобы элемент данных вместе с комментарием полностью умещался на строке).

Варианты данных процедур, в которых параметр `Cmt` отсутствует, добавлены в версию 4.11 конструктора.

```
procedure DataList(NP: integer; X, Y: integer);  
procedure ResultList(NP: integer; X, Y: integer);
```

Процедуры `DataList` и `ResultList` предназначены для помещения структуры типа «одно- или двусвязный линейный динамический список» в набор исходных или, соответственно, результирующих данных, а также для вывода этой структуры на экран. Параметр `NP` задает номер указателя (предварительно определенный процедурой [SetPointer](#)), который указывает на *начало* данного списка, т. е. на его первый элемент. Если соответствующий указатель является нулевым, то список считается пустым. Пустой список на экране не отображается.

Непустой динамический список отображается на двух экранных строках; первая строка содержит имена указателей, входящих в задание и связанных с данным списком (они задаются процедурой [ShowPointer](#)), во второй строке — значения элементов списка (точнее, их полей `Data` целого типа) и виды связи между элементами.

Если память для элемента результирующего списка должна быть выделена программой учащегося, то значение его поля `Data` на экране обрамляется точками (например, `.23.`). Если в исходной динамической структуре требуется разрушить один или несколько элементов, то эти элементы выделяются более бледным цветом, а в случае, если программа учащегося не освободит память, занимаемую этими элементами, они будут выделены красным цветом. Наконец, если в списке, преобразованном программой учащегося, элементы располагаются не на требуемых местах, то они заключаются в скобки (например, `(23)`), а если элемент списка содержит ошибочную ссылку `Next`, то она помечается двумя красными звездочками (например `46 - **`). Красные звездочки указываются в конце списка также в случае, если его длина

превышает максимально допустимую. Специальные обозначения используются также для *циклических списков* (см. [пример 3](#)).

Элемент данных типа «линейный список» должен содержать не более 14 элементов типа TNode, причем значения их полей Data должны лежать в диапазоне от -9 до 99, поскольку для каждого поля Data отводится по две экранные позиции. Для большей наглядности рекомендуется использовать числа из диапазона 10–99, резервируя однозначные и отрицательные числа для особых элементов (например, барьерного элемента циклического списка — см. задание Dynamic70). Если значение элемента списка не уместится в поле вывода, то в его последней экранной позиции выводится красная звездочка — признак ошибки.

Для отображения списка как двусвязного необходимо, чтобы в его элементах были определены поля связи Next и Prev; в этом случае связи между соседними элементами списка обозначаются двойными линиями: «=». Если в задании требуется использовать односвязный список, то для его элементов надо определить поле связи Next, а для полей Prev следует указать значение, *не связанное* с элементами этого списка (например, адрес какой-либо глобальной переменной типа TNode). Связи между элементами односвязных списков обозначаются одинарными линиями: «-».

Вызов процедуры DataList или ResultList приводит к тому, что соответствующий список становится *текущей динамической структурой* для данного задания. Все последующие вызовы процедур [ShowPointer](#), [SetNewNode](#) и [SetDisposedNode](#) будут влиять на эту текущую структуру.

```
procedure DataBinTree(NP, X, Y1, Y2: integer);  
procedure ResultBinTree(NP, X, Y1, Y2: integer);  
procedure DataTree(NP, X, Y1, Y2: integer);  
procedure ResultTree(NP, X, Y1, Y2: integer);
```

Эти процедуры предназначены для включения в задание *бинарных деревьев* и [деревьев общего вида](#) (называемых также *деревьями с произвольным ветвлением*) в качестве исходных (DataBinTree и DataTree) или результирующих (ResultBinTree и ResultTree) данных. Для деревьев общего вида используется представление «левая

дочерняя вершина — правая сестра» («left child — right sibling»). Как и для процедур `DataList` и `ResultList`, описанных выше, первым параметром этих процедур является номер указателя типа `PNode`, ранее определенного с помощью процедуры `SetPointer`. В данном случае этот указатель должен указывать на корень добавляемого в задание дерева; если он является нулевым указателем, то дерево считается пустым и не отображается на экране. В отличие от процедур, связанных с линейными списками, для отображения дерева можно (и рекомендуется) выделять на экране более двух строк; номера начальной и конечной экранной строки задаются параметрами `Y1` и `Y2` соответственно. В отличие от других «прокручиваемых» данных (а именно типизированных и текстовых файлов), дерево может не занимать выделенные для него строки по всей ширине: параметр `X` показывает, начиная с какой позиции экранных строк будет отображаться дерево и связанная с ним информация. Следует отметить, что использовать для других целей можно только левую часть строки, связанной с деревом; все позиции строки, начиная с позиции `X`, будут использоваться для отображения дерева.

Вызов любой из описываемых процедур приводит к тому, что соответствующее дерево становится *текущей динамической структурой* для данного задания. Все последующие вызовы процедур [ShowPointer](#), [SetNewNode](#) и [SetDisposedNode](#) будут влиять на эту текущую структуру.

При отображении дерева используются обозначения, аналогичные тем, которые применяются при отображении линейных динамических структур. В частности, в качестве вершины изображается значение ее поля `Data`, причем для вывода этого значения выделяются две экранные позиции (если двух позиций недостаточно, например, в случае значения 234, то на второй из выделенных позиций изображается красная звездочка: 2*). В качестве обозначения связей между вершинами используются одинарные и двойные линии («-» и «=»); двойные линии, как и для линейных списков, означают, что связь между вершинами является двусторонней (так называемые *деревья с обратной связью* — см. [пример 6](#)). Обратная связь обеспечивается полем `Parent`; ее можно

использовать только для бинарных деревьев.

Для деревьев предусмотрены два варианта отображения. Первый вариант предназначен для отображения бинарного дерева; он применяется для деревьев, включенных в задание процедурами `DataBinTree` и `ResultBinTree`. В этом варианте обе дочерние вершины располагаются ниже родительской вершины (на следующем уровне — см. [примеры 5 и 6](#)). Второй вариант предназначен для отображения *дерева общего вида* (вершины которого могут содержать более двух дочерних вершин); он применяется для деревьев, включенных в задание процедурами `DataTree` и `ResultTree`. В этом варианте вершина, определяемая полем `Left` вершины `P`, как обычно, располагается ниже и левее вершины `P` и задает ее первую (левую) дочернюю вершину, а вершина, определяемая полем `Right`, моделирует следующую вершину-«сестру» вершины `P` и поэтому располагается *на том же уровне*, что и вершина `P`. Такой способ отображения деревьев позволяет, в частности, легко определить глубину дерева общего вида и номер уровня для любой его вершины (см. [пример 7](#)).

Перечислим другие обозначения, имеющие тот же смысл, что и для линейных структур:

- если вершина дерева должна быть создана в программе учащегося, то данная вершина выделяется слева и справа точками, например, `.23.` (для этого используется процедура [SetNewNode](#));
- если в дереве, преобразованном программой учащегося, существующие вершины располагаются не на своих местах, то они заключаются в скобки: `(23)`;
- если в исходном дереве требуется разрушить одну или несколько вершин, то эти вершины выделяются более бледным цветом (а в случае, если программа учащегося не освободит память, занимаемую этими вершинами, они будут выделены красным цветом);
- если переход по ссылке `Left` или `Right` для данной вершины дерева невозможен, то, как и в случае линейных структур, это отмечается красными звездочками, которые, однако, изображаются не рядом с данной вершиной, а ниже вершины

(что подчеркивает тот факт, что ошибка возникла при попытке перехода на следующий уровень дерева).

Для деревьев, в отличие от линейных структур, нулевые поля связи не отображаются. Если поле Left или Right равно нулевому указателю, то на изображении дерева у соответствующей вершины просто отсутствует левая или правая связь.

Максимальное число вершин в дереве равно 18; это объясняется тем, что каждая вершина занимает 4 позиции экранной строки и, кроме того, 4 начальных позиции отводятся под дополнительную информацию (номера уровней дерева). Поэтому, с учетом того, что ширина экранной строки равна 78, вписать в нее можно только дерево с не более чем 18 вершинами. Впрочем, в заданиях рекомендуется использовать не более 16 вершин, начиная вывод дерева с 11 экранной позиции; это дает возможность использовать левую часть строк для отображения других данных, например, указателей, связанных с данным деревом.

Количество уровней дерева ограничивается только количеством его вершин и, таким образом, может достигать 18. В соответствии с общепринятой практикой, уровни дерева нумеруются от 0. Номер уровня отображается в левой части области, отведенной под изображение дерева; он выделяется цветом и отделяется от изображения дерева двоеточием.

Если количество уровней превышает число экранных строк, выделенных для отображения дерева, то для дерева становится возможной *прокрутка*, подобная прокрутке файловых данных (точнее, данных из текстовых файлов, поскольку для деревьев, как и для текстовых файлов, прокрутка выполняется в вертикальном направлении). На возможность прокрутки указывают дополнительные символы, которые изображаются слева от номера уровня. Символ «стрелка вверх», расположенный на первой экранной строке, отведенной для отображения дерева, означает, что изображение дерева можно пролистать вверх, а символ «стрелка вниз», расположенный на последней экранной строке, отведенной для отображения дерева, означает, что изображение дерева можно пролистать вниз (см. [пример 8](#)). В режиме окна с динамической компоновкой все деревья отображаются полностью, поэтому

отдельная прокрутка для них не требуется.

Обычно первая строка, отводимая под изображение дерева, содержит его корень и помечается слева числом 0 (нулевой уровень дерева). Единственная ситуация, когда это правило нарушается, связана с ошибочным формированием бинарного дерева с обратной связью в случае, если поле Parent корня не содержит значение nil. В этой ситуации перед строкой с изображением корня дерева помещается еще одна строка, в которой над корнем изображается красная звездочка — признак ошибки.

Изображение дерева может также содержать строки, расположенные ниже последнего уровня; эти строки могут потребоваться для вывода имен указателей, связанных с вершинами-листьями, расположенными на последнем уровне, а также для вывода звездочек, отмечающих ошибочные ссылки Left или Right для вершин, расположенных на последнем уровне дерева. Заметим, что ссылка считается ошибочной в двух случаях:

- если она содержит неверный адрес,
- если она ссылается на вершину, которую нельзя отобразить, поскольку для этой вершины не предусмотрено экранного места.

В частности, звездочки обязательно будут выведены при попытке отобразить на экране дерево с количеством вершин, превышающим 18.

Приведем примеры изображений линейных списков и деревьев. В этих примерах предполагается, что в качестве текущего языка задачника выбран язык Pascal; для других языков вместо nil используются обозначения нулевых указателей или *нулевых объектов* — (см. описание процедуры [SetObjectStyle](#)), соответствующие этим языкам.

Пример 1

```
P1  
24 - 23 >nil
```

Первый элемент данного списка связан с указателем P₁; список содержит два элемента и является *односвязным*: на это указывает

символ «—» между элементами, означающий, что поле Next первого элемента (со значением 24) указывает на второй элемент (со значением 23). Поле Next второго элемента равно nil.

Пример 2

$$\begin{array}{c} P_1 \qquad P_2 \\ \text{nil} < 14 = 23 = 34 > \text{nil} \end{array}$$

Данный список является двусвязным (двойная связь, использующая оба поля связи — Next и Prev, — обозначается знаком «=»), причем в задании с этим списком связаны два указателя: P_1 указывает на его первый, а P_2 — на его последний элемент.

Пример 3

$$\begin{array}{c} P_0 \\ << = 15 - 23 = 34 = >> \end{array}$$

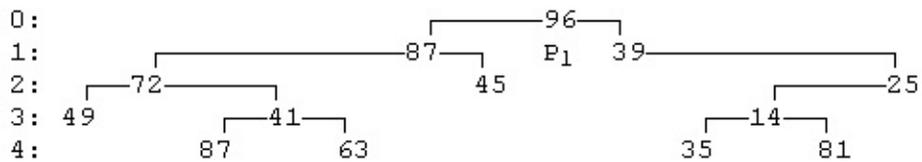
Данный список является двусвязным *циклическим* списком (на его цикличность указывают символы «<<>>» и «>><<»), однако одна из его связей отсутствует. А именно, элемент 23 (на который указывает указатель P_0) не связан с предыдущим элементом 15, т. е. поле Prev элемента 23 содержит ошибочное значение (например, равно nil). При правильно разработанном задании подобная ситуация может возникнуть только для ошибочных списков, созданных в программе учащегося. Заметим, что связь в другом направлении (от 15 к 23) имеется, т. е. поле Next элемента 15 указывает на элемент 23.

Пример 4

$$\begin{array}{c} P_X P_Y \\ 95 - 63 - .34. - >> \end{array}$$

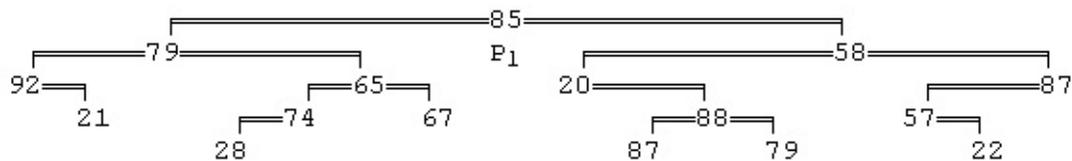
Данный список является односвязным циклическим списком. Он имеет две особенности. Во-первых, на элемент 95 указывают сразу два указателя (P_X и P_Y), и, во-вторых, элемент 34 должен быть размещен в памяти процедурой New при *выполнении* задания (на это указывают обрамляющие его точки). Подобные элементы, естественно, могут содержаться только в *результатирующих* списках. Они определяются с помощью процедуры [SetNewNode](#).

Пример 5



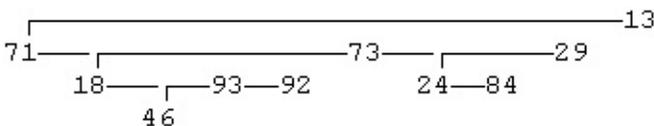
Так выглядит на экране бинарное дерево глубины 4. С корнем этого дерева (поле Data которого равно 96) связан указатель P₁.

Пример 6



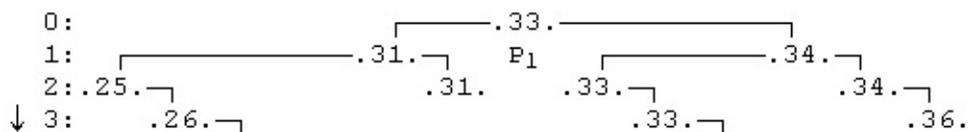
Так выглядит на экране бинарное дерево с обратной связью (номера уровней на данной иллюстрации не указаны).

Пример 7



Так выглядит на экране дерево общего вида (номера уровней и имена связанных с деревом указателей на данной иллюстрации не указаны). В данном случае корень дерева 13 имеет три непосредственных потомка: вершины 71, 73 и 29. Напомним, что в дереве общего вида поле Left определяет первую (левую) дочернюю вершину, а поле Right — очередную (правую) вершину-сестру.

Пример 8



Так выглядит на экране бинарное дерево с включенным режимом прокрутки. Дополнительной особенностью этого дерева является наличие точек около каждой его вершины. Это означает, что данное дерево является результирующим, причем память для всех его вершин должна быть выделена в программе учащегося.

```
procedure ShowPointer(NP: integer);
```

Процедура обеспечивает отображение указателя с номером NP при выводе текущего линейного списка или дерева. Например, ее вызов вида ShowPointer(1) обеспечил отображение указателя P₁ в [примерах 1, 2 и 5](#). Если указатель номер NP является нулевым, то вызов процедуры ShowPointer игнорируется без вывода сообщения об ошибке. Если указатель с номером NP не является нулевым и не связан ни с одним из элементов списка, то выводится сообщение об ошибке.

С одним элементом списка или дерева можно связать не более двух указателей (исключение составляет последний элемент списка, с которым можно связать не более трех указателей). Порядок вызова процедур ShowPointer для одного и того же элемента списка является произвольным; при отображении указателей, связанных с одним и тем же элементом, они выводятся в отсортированном порядке (например, P₃P₆). В случае списков имена указателей отображаются *над* элементом, и при наличии нескольких указателей на один элемент их имена располагаются слева направо. В случае деревьев имена указателей располагаются *под* элементом, и при наличии нескольких указателей на один элемент их имена располагаются одно под другим. Если количество указателей, связываемых процедурами ShowPointer с данным элементом списка, превосходит максимально допустимое (например, с последним элементом связывается четыре различных указателя), то список связанных указателей дополняется символом ошибки — звездочкой (например, P₁P₂P₃*). Если с элементом бинарного дерева связывается более двух указателей, то под вторым указателем изображается еще один указатель вида P*. Символ ошибки * выделяется красным цветом.

Если указатель надо связать с элементом списка или дерева, помеченным точками (см. [пример 8](#)), то вызов процедуры ShowPointer для данного указателя надо выполнить *до того*, как для соответствующего элемента списка или дерева будет вызвана процедура SetNewNode (в противном случае при вызове процедуры ShowPointer будет выведено сообщение об ошибке вида «Не найден элемент с адресом P1»).

[Pascal]

```
procedure SetNewNode(NNode: integer);
```

[C++]

```
void SetNewNode(int NNode);
```

Процедура определяет для текущего списка элемент с номером NNode (нумерация ведется от 1) как элемент, который требуется разместить в памяти с помощью процедуры New в ходе выполнения задания (подобные элементы выделяются в списке с помощью обрамляющих точек — см. [пример 4](#)). Она также позволяет аналогичным образом выделить элемент текущего дерева (см. [пример 8](#)); при этом предполагается, что элементы дерева нумеруются в префиксном порядке (в частности, корень дерева всегда имеет номер 1; по поводу префиксного порядка см. задание Tree13).

Данная процедура может применяться только к *результатирующим* спискам и деревьям (для определения которых используются процедуры группы Result: ResultList, ResultBinTree, ResultTree). Если результирующий список или дерево не содержит элемента с номером NNode, то выводится сообщение об ошибке.

В случае, если указатель на элемент номер NNode требуется отобразить на экране, вызов соответствующей процедуры ShowPointer необходимо выполнить до вызова процедуры SetNewNode.

Если при выполнении задания учащийся будет выделять память (процедурой New для языка Pascal или аналогичными средствами для других языков) для тех элементов результирующего списка или дерева, для которых это не предусмотрено заданием, то соответствующие элементы в результирующем списке (дереве) будут обрамлены точками, что приведет к сообщению «Ошибочное решение».

```
procedure SetDisposedNode(NNode: integer);
```

Процедура определяет для текущего списка или дерева элемент с

номером NNode (нумерация ведется от 1, элементы дерева нумеруются в префиксном порядке), который требуется удалить из динамической памяти в ходе выполнения задания. Данная процедура может применяться только к исходным спискам и деревьям (для определения которых используются процедуры группы Data: DataList, DataBinTree, DataTree). Если исходный список или дерево не содержит элемента с номером NNode, то выводится сообщение об ошибке.

Элементы, помечаемые с помощью процедуры SetDisposedNode, выделяются на экране цветом меньшей яркости. Если они не удаляются из памяти в ходе выполнения задания, то их цвет изменяется на красный и выводится соответствующее сообщение об ошибке.

```
procedure SetObjectStyle;
```

Данная процедура устанавливает «объектный стиль» для динамических структур и связанных с ними ссылок при выполнении задания в среде PascalABC.NET. Она должна вызываться при формировании заданий, ориентированных на использование не записей TNode и связанных с ними указателей PNode, а *объектов* класса Node (данный класс определен в вариантах задачника для языков платформы .NET и, в частности, для языка PascalABC.NET).

При разработке заданий класс Node *не используется*. Даже если разрабатываемая группа заданий ориентирована на его применение, сами задания надо создавать с помощью записей TNode, указателей PNode и описанных выше процедур. Однако для *решения* подобных задач на языках платформы .NET, а также на языках Python и Java, вместо записей и связанных с ними указателей надо применять объекты класса Node. Среди языков, поддерживаемых версией 4.11 задачника, имеется единственный язык, позволяющий использовать как указатели, так и объекты: это язык PascalABC.NET. Именно поэтому для данного языка предусмотрена процедура SetObjectStyle (в прочих языках, использующих объекты Node, *настройка объектного стиля в заданиях на обработку динамических структур выполняется автоматически*).

Создавая задания на обработку динамических структур для языка PascalABC.NET, разработчик должен указать учащемуся требуемый способ решения, используя соответствующие термины в формулировке задания («запись» или «объект», «указатель» или «ссылка» и т. п.), а также настроив, при необходимости, вывод динамических структур на «объектный стиль», вызвав процедуру SetObjectStyle (по умолчанию применяется «стиль указателей», подробно описанный выше). Процедура SetObjectStyle должна быть вызвана после процедуры CreateTask (необходимо также, чтобы ее вызов располагался *перед* вызовами любых процедур, обеспечивающих добавление к заданию динамических структур и связанных с ними указателей). В результате ее вызова изменяется отображение этих элементов данных, а именно:

- вместо текста ptr для непустого указателя указывается текст Node (т. е. имя непустого *объекта* типа Node);
- в стандартном комментарии к указателю вместо буквы P указывается буква A, например, $A_1 = \text{Node}$ (эту особенность следует учитывать в формулировке задания, используя в ней вместо имен указателей P_1, P_2 и т. д. имена *объектов* A_1, A_2 и т. д.).

Аналогичные изменения (символа P на символ A) выполняются и при отображении указателей, связанных с динамическими структурами. Например, односвязный список, приведенный в [примере 1](#), при установке объектного стиля будет иметь следующий вид:

```
A1  
24 - 23 >nil
```

Слово nil осталось неизменным, так как в PascalABC.NET оно применяется для обозначения как нулевых указателей, так и «пустых» объектов. При использовании других языков платформы .NET обозначения «пустых» объектов соответствующим образом корректируются; так, для языка C# применяется обозначение null, а для языка VB.NET — обозначение Noth (от слова Nothing).

Заметим, что объектный стиль используется в базовых группах ObjDyn и ObjTree, имеющих в варианте задачника для системы

PascalABC.NET. Эти группы с содержательной точки зрения полностью аналогичны группам Dynamic и Tree, ориентированным на применение указателей.

Процедуры для разработки заданий по параллельному MPI-программированию

Процедуры, описанные в настоящем разделе, связаны с разработкой заданий по параллельному программированию и доступны в конструкторе учебных заданий, начиная с версии 4.9 задачника *Programming Taskbook*. Подробное описание этих процедур приводится в соответствующем разделе описания задачника **Programming Taskbook for MPI**.

```
procedure CreateTask([SubgroupName: string;] var ProcessCount: integ
```

Данные перегруженные варианты процедуры `CreateTask` предназначены для инициализации задания по параллельному программированию. От исходных вариантов процедуры `CreateTask` их отличает наличие параметра `ProcessCount`. Параметр `SubgroupName` имеет тот же смысл, что и для исходных вариантов процедуры: он определяет заголовок *подгруппы*, в которую включается задание, если разрабатываемую группу заданий целесообразно разбить на подгруппы. Если параметр `SubgroupName` является пустой строкой или отсутствует, то задание не связывается с какой-либо подгруппой.

Параметр `ProcessCount` определяет количество процессов при выполнении задания в параллельном режиме. Допускается использовать от 2 до 36 процессов. При определении параметра `ProcessCount` желательно применять датчик случайных чисел; это позволит протестировать предложенный алгоритм решения при различном количестве процессов параллельного приложения.

Если параметр `ProcessCount` меньше или равен 1, то для инициализации задания используется соответствующий вариант процедуры `CreateTask` без данного параметра (при этом выходное значение параметра `ProcessCount` полагается равным 1, а задание выполняется в обычном, «непараллельном» режиме).

Если параметр `ProcessCount` превосходит 36, то в окне задачника выводится сообщение об ошибке.

Способ использования параметра `ProcessCount` при инициализации

задания по параллельному программированию зависит от того, какую «роль» играет программа, вызвавшая процедуру CreateTask с параметром ProcessCount (см. таблицу).

«Роль» программы	Входное значение параметра ProcessCount	Выходное значение параметра ProcessCount
Непараллельная программа-загрузчик, обеспечивающая запуск параллельного варианта программы	Используется (определяет число процессов при запуске параллельного варианта программы)	Всегда равно 0
Главный процесс параллельной программы (процесс ранга 0)	Не используется	Равно числу процессов в параллельной программе; используется при формировании входных и выходных данных
Подчиненный процесс параллельной программы	Не используется	Всегда равно 0
Непараллельная программа, обеспечивающая демонстрационный запуск учебного задания	Используется	Всегда равно входному значению; используется при формировании входных и выходных данных

```
procedure SetProcess(ProcessRank: integer);
```

Данная процедура устанавливает в качестве *текущего процесса* параллельного приложения процесс ранга ProcessRank. Все *числовые* исходные и контрольные данные связываются с текущим процессом. До первого вызова данной процедуры текущим процессом считается процесс ранга 0. Процедуру можно вызывать несколько раз с одним и тем же параметром (например, первый раз процесс делается текущим при определении связанных с ним исходных данных, а второй раз — при определении его контрольных

данных).

Параметр `ProcessRank` должен принимать значения в диапазоне от 0 до $N - 1$, где N — количество процессов, возвращаемое параметром `ProcessCount` процедуры `CreateTask`. При нарушении этого условия выводится сообщение об ошибке «*Параметр процедуры `SetProcess` находится вне диапазона $0..N-1$, где N — количество используемых процессов*».

Модуль PT4TaskMakerNET: форматирование текста заданий

Общие сведения

В конструкторе учебных заданий PT4TaskMaker предусмотрена возможность *форматирования* текста заданий, а также преамбул для группы и ее подгрупп. Форматирование выполняется с помощью набора *управляющих последовательностей* (команд), большинство из которых имеет вид `\символ`.

Используя управляющие последовательности, можно выполнять следующие действия по форматированию текста в окне задачника:

- добавлять в текст специальные символы, в том числе символы шрифта Symbol и буквы западноевропейских языков;
- выделять фрагмент текста полужирным шрифтом;
- использовать в тексте нижние и верхние индексы;
- добавлять в текст задания ссылки на другие задания этой же группы, не указывая при этом название группы (что позволяет корректно изменять эти ссылки при включении задания в другие группы);
- добавлять в текст задания элементы, зависящие от текущего языка программирования (в частности, обозначения логических констант).

Все описанные выше действия обеспечивают требуемое форматирование текста задания как в окне задачника, так и в html-описании данного задания. Аналогичное форматирование (в частности, использование нижних и верхних индексов) можно применять и в текстах комментариев, которые выводятся в окне задачника в разделах исходных и результирующих данных.

Кроме того, имеются управляющие последовательности, не влияющие на текст задания в окне задачника, однако обеспечивающие дополнительное форматирование этого текста (и текста комментариев для группы и ее подгрупп) в html-описании задания или группы заданий. Данные управляющие последовательности позволяют:

- выделять имена переменных курсивом;
- использовать более разнообразное выделение фрагментов текста (помимо полужирного начертания можно установить курсивное начертание, выделение моноширинным шрифтом и

- специальное выделение);
- разбивать текст задания и преамбулы на отдельные абзацы;
 - устанавливать для требуемых фрагментов текста режим вывода с центрированием или с отступом;
 - обеспечивать вывод фрагментов текста в несколько столбцов, с возможностью установки способа выравнивания для каждого столбца.

Напомним, что для вывода на экран html-страницы с описанием задания или группы заданий достаточно вызвать процедуру Task, указав в конце ее параметра (имени задания или группы заданий) суффикс #. Кроме того, html-страницы с описанием групп заданий можно генерировать с помощью модуля PT4Demo, используя кнопку  в окне этого модуля.

Таблица управляющих последовательностей

Управляющие последовательности, приведенные в таблице, можно использовать в формулировках заданий (параметр S процедуры [TaskText](#)), комментариях к исходным и результирующим данным (параметр Cmt в процедурах групп [Data](#) и [Result](#)), а также в дополнительных описаниях (преамбулах) групп и подгрупп учебных заданий (параметр S в процедуре [CommentText](#)).

Управляющие последовательности, использованные в параметрах процедур групп Data и Result, влияют только на представление соответствующих комментариев в окне задачника (см. столбец «Окно задачника»). Управляющие последовательности, использованные в параметре S процедуры CommentText, обеспечивают соответствующее форматирование преамбулы к группе заданий и ее подгруппам в тексте html-страницы с описанием группы заданий (см. столбец «Html-страница»). Управляющие последовательности, использованные в параметре S процедуры TaskText, влияют на вид формулировок заданий как в окне задачника, так и в html-описаниях.

Все последовательности вида `\символ`, не указанные в приведенной ниже таблице, игнорируются как при выводе текста в окне задачника, так при его отображения в виде html-страницы.

В заголовках подгрупп, указываемых в процедуре [CreateTask](#) (параметр SubgroupName), а также в *тексте краткого описания группы*, указываемого в процедуре [CreateGroup](#) (параметр GroupDescription), управляющие последовательности не обрабатываются. При указании в этих строках управляющих последовательностей они дословно воспроизводятся и в окне задачника, и в html-описании. Для указания короткого (–) или длинного (—) тире в кратких описаниях и заголовках подгрупп можно использовать двойные и тройные дефисы соответственно: -- и ---. Начиная с версии задачника 4.10, двойные и тройные дефисы в заголовках подгрупп при их отображении в html-описаниях заменяются на короткие и длинные тире.

Команда	Описание	Окно задачника	Html-страница

Ссылки на другие задания данной группы

<code>\число</code>	Имя задания из текущей группы с номером, меньшим текущего величину указанного десятичного числа (или <code>*****</code> , если результирующий номер оказывается меньшим 1)
<code>\0число</code>	Имя задания из текущей группы с номером, большим текущего величину указанного десятичного числа (или <code>*****</code> , если результирующий номер оказывается большим 999)

Пробелы

<code>\,</code>	Малый неразрывный пробел	<i>Игнорируется</i>	<code>&nbsp;</code>
<code>\;</code>	Средний неразрывный пробел	<i>Пробел</i>	<code>&nbsp;</code>
<code>~</code>	Обычный неразрывный пробел	<i>Пробел</i>	<code>&nbsp;</code>
<code>\q</code>	Большой пробел	<i>Игнорируется</i>	<i>Пробел&nbsp;</i> ; <i>Пробел</i>
<code>\Q</code>	Двойной большой пробел	<i>Игнорируется</i>	<i>2 копии большого про</i>

Символы

<code>\\</code>	Обратная косая черта (\)	<code>\</code>	<code>\</code>
<code>\&</code>	Амперсанд (&)	<code>&</code>	<code>&amp;</code>
<code>\{</code>	Открывающая фигурная скобка ({)	<code>{</code>	<code>{</code>
<code>\}</code>	Закрывающая фигурная скобка (})	<code>}</code>	<code>}</code>
<code>\~</code>	Символ «волна» (~)	<code>~</code>	<code>~</code>
<code>\^</code>	Символ «шапочка» (^)	<code>^</code>	<code>^</code>
<code>_</code>	Символ подчеркивания (_)	<code>_</code>	<code>_</code>
<code>\.</code>	Многоточие (...)	<code>...</code> (<i>три точки</i>)	<code>&#8230;</code>

<code>\=</code>	Длинное тире (—)	—	<code>&#8212;</code>
<code>\:</code>	Символ диапазона, или короткое тире (–)	–	<code>&#8211;</code>
<code>\-</code>	Символ «минус» (–)	–	<code>&#8722;</code>
<code>\<</code>	Открывающие угловые кавычки («	«	<code>&#171;</code>
<code>\></code>	Закрывающие угловые кавычки (»)	»	<code>&#187;</code>
<code>*</code>	Символ умножения (·)	·	<code>&#183;</code>
<code>\+</code>	Символ «плюс-минус» (±)	±	<code>&#177;</code>
<code>\o</code>	Символ градуса (°)	°	<code>&#176;</code>
<code>\x</code>	Символ «косой крест» (×)	×	<code>&#215;</code>
<code>\a</code>	Греческая буква «альфа» (α)	α	<code>&#945;</code>
<code>\p</code>	Греческая буква «пи» (π)	π	<code>&#960;</code>
<code>\e</code>	Греческая буква «эпсилон» (ε)	ε	<code>&#949;</code>
<code>\l</code>	Символ «меньше или равно» (≤)	≤	<code>&#8804;</code>
<code>\g</code>	Символ «больше или равно» (≥)	≥	<code>&#8805;</code>
<code>\n</code>	Символ «не равно» (≠)	≠	<code>&#8800;</code>
<code>\X</code>	Символ «bullet» (•)	•	<code>&#8226;</code>
<code>\hXX</code> , где XX — двузначное 16-ричное	Символ с кодом XX из второй половины кодовой таблицы для западноевропейских	Соответствующий символ	<code>&#DDD;</code> (DDD — десятичный соответствующего с

число	языков ANSI Latin-1 (кодировка страница 1252)		в кодировке Unicode)
$\backslash NXX$, где XX — двузначное 16-ричное число	Символ с кодом XX Windows-шрифта Symbol	Соответствующий символ	$\&\#DDD;$ (DDD — десятичный соответствующего с в кодировке Unicode) или <code>&\#D </code> (DDD — десятичное равное 16-ричному чи

Элементы текста, зависящие от текущего языка программирования

$\backslash R$	Метка начала квадратного корня	<code>Sqrt(</code> (Pascal, PascalABC.NET, VB.NET, C#) <code>sqrt(</code> (C++, Python, Java) <code>Sqr(</code> (Visual Basic версий 5 и 6)	(
$\backslash r$	Метка конца квадратного корня)) ^{1/2}
$\backslash t$	Логическая константа «True»	<code>True</code> (Pascal, PascalABC.NET, Visual Basic VB.NET, Python) <code>true</code> (C++, C#, Java)	
$\backslash f$	Логическая константа «False»	<code>False</code> (Pascal, PascalABC.NET, Visual Basic VB.NET, Python) <code>false</code> (C++, C#, Java)	
$\backslash N$	Нулевой указатель	<code>nil</code> (Pascal) <code>NULL</code> (C++)	
$\backslash 0$	Нулевая ссылка для объекта	<code>null</code> (C#, Java) <code>Nothing</code> (VB.NET) <code>nil</code> (PascalABC.NET)	

		None (Python)
\d	Имя функции для освобождения ресурсов, выделенных объекту	Dispose (PascalABC.NET, C#, VB.NET) dispose (Python, Java)

Индексы

_символ	Односимвольный нижний индекс	Символ выводится как нижний индекс	_{символ}
^символ	Односимвольный верхний индекс	Символ выводится как верхний индекс	^{символ}
_ {	Метка начала многосимвольного нижнего индекса	Переход в режим нижнего индекса	<sub>
^{	Метка начала многосимвольного верхнего индекса	Переход в режим верхнего индекса	<sup>
}	Метка конца текущего (верхнего или нижнего) многосимвольного индекса	Выход из режима индекса	</sub> или </sup>

Выделение (в окне задачника любой режим выделения приводит к выделению полужирным шрифтом)

\B	Метка начала полужирного выделения	Переход в режим выделения	
\b	Метка конца полужирного выделения	Выход из режима выделения	
\I	Метка начала курсивного выделения	Переход в режим выделения	<i>
\i	Метка конца курсивного выделения	Выход из режима выделения	</i>

	выделения		
\S	Метка начала специального выделения	<i>Переход в режим выделения</i>	<span class="ptSpecial":
\s	Метка конца специального выделения	<i>Выход из режима выделения</i>	

Дополнительное форматирование html-страниц (в окне задачника данн управляющие последовательности игнорируются)

{	Метка начала выделения переменной		<i>
}	Метка конца выделения переменной		</i>
\M	Метка начала моноширинного текста		<tt>
\m	Метка конца моноширинного текста		</tt>
\	Разрыв строки		
\P	Начало нового абзаца		<i>Завершается предыду абзац и создается абз стилиа ptTaskContin использования в формулировке заданиа ptComment (при использовании в текст преамбулы)</i>
\[Метка начала абзаца с центрированием		<i>Завершается предыду абзац и создается абз стилиа ptTaskCenter использования в формулировке заданиа ptCommentCenter (п</i>

			использовании в тексте преамбулы)
\]	Метка конца абзаца с центрированием		Завершается предыдущий абзац и создается абзац стиля <code>ptTaskContinue</code> (использовании в формулировке задания <code>ptCommentContinue</code> (использовании в тексте преамбулы))
\(Метка начала абзаца с отступом		Завершается предыдущий абзац и создается абзац стиля <code>ptTaskQuote</code> (использовании в формулировке задания <code>ptCommentQuote</code> (использовании в тексте преамбулы))
\)	Метка конца абзаца с отступом		Завершается предыдущий абзац и создается абзац стиля <code>ptTaskContinue</code> (использовании в формулировке задания <code>ptCommentContinue</code> (использовании в тексте преамбулы))
\J	Метка начала режима выравнивания по столбцам (после нее указывается последовательность символов <code>r</code> , <code>l</code> , <code>c</code> , которая должна оканчиваться символом <code>&</code>)		<code><table></code>
	Метка конца		

\j	режима выравнивания по столбцам		</table>
&	Переход к новому столбцу в режиме выравнивания по столбцам		Добавляется тег <td соответствующим выравниванием; для n столбца предварител указывается тег <tr:

Дополнительные сведения об использовании управляющих последовательностей

Необходимость в специальных командах для генерации ссылок на другие задания группы объясняется тем, что любое имеющееся задание может быть импортировано в группу с другим именем (с помощью процедуры [UseTask](#)), и поэтому все ссылки на другие задания этой группы также потребуются откорректировать, указав в них новое имя группы. Разумеется, в подобной ситуации необходимо переносить в новую группу *все* задания, содержащие ссылки друг на друга. Следует заметить, что *разность* между номерами ссылающихся друг на друга заданий не обязана быть такой же, как в исходной группе заданий. Если в новой группе задания находятся на другом «расстоянии» друг от друга, то для указания правильной ссылки достаточно внести соответствующую *поправку* в параметр процедуры UseTask.

Наличие нескольких видов *неразрывных пробелов*, не различающихся в тексте заданий и html-страниц, связано с планируемой в дальнейшем возможностью генерации текста заданий в других форматах (в частности, в формате системы TeX, в котором данные виды пробелов различаются). Приведем рекомендации по использованию неразрывных пробелов:

- вокруг символов $=$, $<$, $>$ указывается обычный неразрывный пробел \sim ; исключением являются фрагменты текста в скобках вида $(> \theta)$, в которых рекомендуется использовать малый пробел: $(> \backslash, \theta)$;
- неразрывный пробел \sim указывается также между текстом и переменной: $\text{стороны} \sim \{a\}$ и $\sim \{b\}$;
- вокруг символов $+$ и $-$ ставится средний пробел \backslash ;
- символы умножения \backslash^* и деления $/$ пробелами не обрамляются; исключением служит ситуация, когда слева и справа от символа деления указываются прописные буквы; в этом случае желательно использовать обрамление малыми пробелами.

Приведем пример оформления формул (данный пример взят из задания Be9in39; обратите внимание на выделение переменных с помощью фигурных скобок, а также на команды, обеспечивающие

вывод индексов, выделение квадратного корня и центрирование формулы):

```
TaskText('Найти корни \Иквaдрaтного уравнения\и ' +
  '{A}\*{x}^2\;\+;\{B}\*{x}\;\+;\{C}\~=-\0, задaнного', 0, 1);
TaskText('своими коэффициентами~{A}, {B}, {C} ' +
  '(коэффициент~{A} не равен~0), если известно,', 0, 2);
TaskText('что дискриминaнт уравнения положителен. ' +
  'Вывести вначале меньший, а затем',0,3);
TaskText('больший из найденных корней. Корни квадратного ' +
  'уравнения находятся по формуле', 0, 4);
TaskText('\[\{x}\_{1,\,2}\~=-(\-\{B}\;\;\+;\;\R{D}\r)/(2\*{A}),\] ' +
  'где {D}\~=\ \Идискриминaнт\и, ' +
  'равный {B}^2\;\-\;\;4\*{A}\*{C}.' , 0, 5);
```

В результате обработки данной формулировки задания в окне задачника будет выведен текст:

Найти корни **квадратного уравнения** $A \cdot x^2 + B \cdot x + C = 0$, заданного своими коэффициентами A , B , C (коэффициент A не равен 0), если известно, что дискриминант уравнения положителен. Вывести вначале меньший, а затем больший из найденных корней. Корни квадратного уравнения находятся по формуле $x_{1,2} = (-B \pm \text{sqrt}(D)) / (2 \cdot A)$, где D — **дискриминант**, равный $B^2 - 4 \cdot A \cdot C$.

В html-описании этот же текст будет отформатирован следующим образом:

Begin39. Найти корни *квадратного уравнения* $A \cdot x^2 + B \cdot x + C = 0$, заданного своими коэффициентами A , B , C (коэффициент A не равен 0), если известно, что дискриминант уравнения положителен. Вывести вначале меньший, а затем больший из найденных корней. Корни квадратного уравнения находятся по формуле $x_{1,2} = (-B \pm (D)^{1/2}) / (2 \cdot A)$, где D — *дискриминант*, равный $B^2 - 4 \cdot A \cdot C$.

Для указания кавычек в тексте задания следует использовать управляющие последовательности `<` и `>`.

Управляющие последовательности `t`, `f`, `N`, `0` для логических констант, нулевых указателей и объектов генерируют текст, зависящий от выбранного в данный момент языка программирования.

Обычные пробелы, указанные после управляющих последовательностей `q`, `Q`, `P`, `[`, `(`, `]`, `)`, `|` и `&`, учитываются только в тексте задачника (и пропускаются в тексте html-страниц).

Режим специального выделения, устанавливаемый парными командами `S` и `s`, в окне задачника приводит к выделению

полужирным шрифтом, а в html-описании обеспечивает выделение фрагмента текста, аналогичное выделению, используемому для *имени задания* в начале его формулировки (в приведенном выше фрагменте html-описания так выделено имя задания «Begin39»). Данный режим рекомендуется использовать для выделения *заголовков*, размещаемых в начале абзаца (например, если формулировка задания завершается абзацем, содержащим указание, то с помощью специального выделения целесообразно выделить текст «Указание» в начале этого абзаца).

Команды выделения переменной { и } не влияют на ее вид в окне задачника, но обеспечивают ее выделение курсивом в тексте html-страницы. В индексах команды выделения переменной не учитываются, а любые *латинские* буквы в них автоматически выделяются курсивом.

В односимвольных индексах нельзя указывать управляющие последовательности для вывода специальных символов, поэтому при необходимости применения в индексах специальных символов следует использовать режим многосимвольных индексов. Метки индексов не могут быть вложенными. В индексах не допускается использование меток выделения `\I`, `\B`, `\S` и наоборот, внутри выделенного текста не допускается указывать индексы.

Выделенные фрагменты не могут содержать меток выделения другого вида. Режим индексов и выделения, заданный в одной процедуре `TaskText` или `CommentText`, не переносится на текст, определяемый при последующих вызовах этих процедур. Если в тексте отсутствуют команды завершения текущего режима (индексов или выделения), то режим автоматически завершается при достижении конца текста, определяемого в текущей процедуре `TaskText` или `CommentText`.

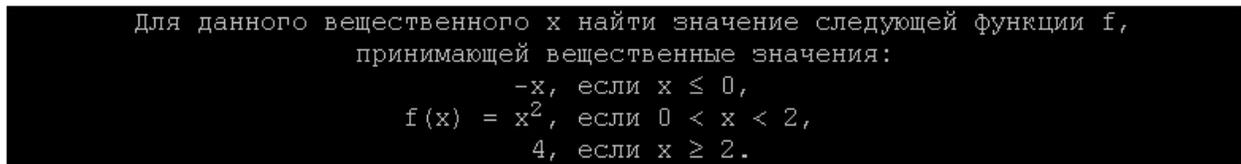
В команде начала режима выравнивания по столбцам символы `r`, `l`, `c` определяют *способ выравнивания* в каждом столбце текста (`r` — выравнивание по правому краю, `l` — выравнивание по левому краю, `c` — выравнивание по центру). Их количество должно быть равно числу столбцов. В каждом столбце должен быть хотя бы один непробельный символ (для пустого столбца достаточно указать

малый неразрывный пробел \,).

Ввиду сложности управляющих последовательностей, связанных с выравниванием по столбцам, приведем пример их использования (пример взят из задания If26):

```
TaskText('Для данного вещественного~{x} найти значение ' +  
  'следующей функции~{f},', 0, 1);  
TaskText('принимаящей вещественные значения:', 0, 2);  
TaskText('\[\Jrcr1&\,&\,&          \-~{x},& если {x}~\l~0,',  
  26, 3);  
TaskText('&{f}({x})&~=-&{x}^2,& если 0~<~{x}~<~2,', 26, 4);  
TaskText('&\,&\,&          4,& если {x}~\g~2.\j\]', 26, 5);
```

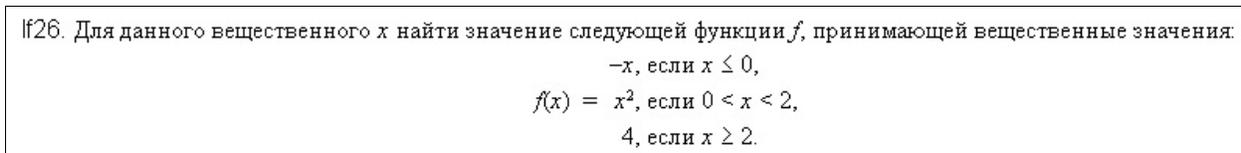
В результате обработки данной формулировки задания в окне задачника будет выведен текст:



Для данного вещественного x найти значение следующей функции f ,
принимаящей вещественные значения:

$$f(x) = \begin{cases} -x, & \text{если } x \leq 0, \\ x^2, & \text{если } 0 < x < 2, \\ 4, & \text{если } x \geq 2. \end{cases}$$

В html-описании этот же текст будет отформатирован следующим образом:



If26. Для данного вещественного x найти значение следующей функции f , принимающей вещественные значения:

$$f(x) = \begin{cases} -x, & \text{если } x \leq 0, \\ x^2, & \text{если } 0 < x < 2, \\ 4, & \text{если } x \geq 2. \end{cases}$$

Команду разрыва строки `\|` можно использовать как в обычном тексте, так и в режиме выделения с отступом или центрированием (в режиме выравнивания по столбцам переход на новую строку выполняется автоматически). Для более наглядного отображения в html-документе текста, выровненного по столбцам, данный текст рекомендуется дополнительно центрировать (как в приведенном выше примере) или использовать для него режим выравнивания с отступом.

Следует обратить внимание еще на одну особенность формулировок, содержащих выравнивание по столбцам: строки с подобным выравниванием, как правило, не нужно центрировать в окне задачника. В приведенном примере в трех последних вызовах процедуры `TaskText` в качестве параметра `X` указывается не нулевое

значение (означающее центрирование по горизонтали), а число 26 — значение позиции по горизонтали, начиная с которой требуется вывести указанную строку.

Указанное обстоятельство может затруднить использование для подобных формулировок нового варианта процедуры TaskText, появившегося в конструкторе версии 4.11. В этом варианте указывается единственный строковый параметр, содержащий все строки формулировки, разделенные символами разрыва строки (#13 или #10). При разборе этого параметра из каждой строки формулировки удаляются начальные и конечные пробелы, причем строки формулировки, оказавшиеся в результате пустыми, игнорируются, а все непустые строки выводятся в окне задачника в режиме центрирования. Тем не менее, и этот вариант процедуры TaskText позволяет обеспечить особое выравнивание требуемых строк в окне задачника. Для этого надо добавить некоторое количество вспомогательных начальных или конечных пробелов к строке, требующей специального выравнивания, причем первый начальный или последний конечный пробел надо *экранировать* символом \, чтобы не допустить его автоматического удаления. Прочие пробелы (следующие за начальным пробелом или предшествующие конечному пробелу) экранировать необязательно. Приведем пример определения формулировки, в котором используется новый вариант процедуры TaskText (следует обратить внимание на конечные пробелы, указанные в последних трех строках; в остальном текст формулировки не отличается от приведенного выше):

```
TaskText('Для данного вещественного~{x} найти значение следующей фун  
'принимаящей вещественные значения: '#13 +  
'\[\Jrcr1&\,&\,& \- {x},& если {x}~\1~0, \ '#13 +  
'&{f}({x})&~=&{x}^2,& если 0~<~{x}~<~2, \ '#13 +  
'&\,&\,& 4,& если {x}~\g~2.\j\ \ ');
```

Управляющая последовательность \P предназначена для разделения абзацев. В тексте, отображаемом в окне задачника, данная команда игнорируется (подобно прочим командам, связанным с разделением на абзацы). Для нее не предусмотрено парной завершающей команды, поскольку необходимые теги при

переходе к новому абзацу добавляются в текст html-страницы автоматически. Пробелы после команды `\P` при генерации html-страницы игнорируются, однако они учитываются при отображении текста в окне задачника.

Генерация специальных символов

Используя две «универсальные» управляющие последовательности `\h` и `\H`, можно включать в текст задания или преамбулы специальные символы, входящие во вторую половину кодовой таблицы для западноевропейских языков ANSI Latin-1 (команда `\h`) или содержащиеся в Windows-шрифте Symbol (команда `\H`). После имени каждой из этих команд следует указать двузначное шестнадцатеричное число, определяющее код требуемого символа; при этом шестнадцатеричные цифры A, B, C, D, E, F можно указывать в любом регистре. Если двухсимвольный текст после команд нельзя преобразовать в шестнадцатеричное число или число не является допустимым, то команды возвращают символ «?» (знак вопроса).

В случае команды `\h` (символы таблицы Latin-1) допустимыми считаются числа из диапазона 128–255, за исключением кодов неотображаемых символов, например, кода неразрывного пробела 160 (A0) или «мягкого» переноса 173 (AD). Символы таблицы Ansi Latin-1 с кодами 128–159 имеют в кодировке Unicode другие значения кодов; при генерации html-описаний для этих символов используются их коды в таблице Unicode.

С помощью команды `\H` можно получить только *часть* символов, определенных в Windows-шрифте Symbol. Исключены символы, уже присутствующие в таблицах ASCII и ANSI Latin-1 (например, цифры и знаки препинания) или имеющие идентичное начертание с символами из этих таблиц (например, заглавные греческие буквы, совпадающие по начертанию с латинскими: A, B, E, H, X и т. д.). Кроме того, исключены символы с кодами 230–239 и 243–254, представляющие собой фрагменты больших скобок.

Следует заметить, что для части математических символов нельзя обеспечить их правильное отображение в каждом из трех наиболее популярных веб-браузеров (Microsoft Internet Explorer, Mozilla Firefox и Opera) без использования средств веб-программирования. В браузерах Internet Explorer и Firefox можно подключать шрифты Windows, в том числе шрифт Symbol, однако в Opera это сделать нельзя. С другой стороны, в Opera и Firefox для отображения всех

стандартных математических символов достаточно указать их код в Unicode-кодировке, однако в стандартных Windows-шрифтах, используемых браузером Internet Explorer, часть символов с требуемыми кодами отсутствует. При реализации команды \N для вывода подобных символов в html-документе был выбран вариант, обеспечивающий их правильное отображение в браузере Internet Explorer (и Mozilla Firefox): для этого используется Windows-шрифт Symbol. Однако в браузере Opera (и других браузерах, не поддерживающих шрифты Windows) *данные символы будут отображаться неправильно.*

Примечание. Для возможности использования Windows-шрифтов в браузере Mozilla Firefox следует установить режим «Разрешить веб-сайтам использовать свои шрифты вместо установленных».

Соответствующий флажок находится в окне «Шрифты», которое можно отобразить с помощью следующей последовательности действий: выполнить команду меню «Инструменты | Настройки...», в появившемся окне «Настройки» перейти на вкладку «Содержимое» и в разделе «Шрифты и цвета» нажать кнопку «Дополнительно...».

С некоторыми часто используемыми специальными символами связаны особые управляющие последовательности (см. таблицу управляющих последовательностей, раздел «[СИМВОЛЫ](#)»). Все подобные символы правильно отображаются во всех перечисленных выше браузерах.

Хотя символ пересечения (\cap , код 8745) имеется в стандартных Windows-шрифтах, прочие символы, связанные с множествами (объединение, вложение, принадлежность и т. д.), в этих шрифтах отсутствуют. Для того чтобы все обозначения, связанные с множествами, выглядели в html-документе единообразно, для отображения символа пересечения (команда \Nc7) используется соответствующий символ из шрифта Symbol.

Ниже приводятся таблицы всех символов, которые можно получить с помощью универсальных команд \h и \H. Первая таблица содержит символы, генерируемые командой \h, а вторая — символы, генерируемые командой \H. Команды из второй таблицы, связанные с теми символами, которые будут неверно отображаться в браузере

Opera, выделены полужирным шрифтом.

Таблица 1. Символы, генерируемые командой \h

\h80		\h82	\h83	\h84	\h85	\h86	\h87	\h88	\h89
€		,	f	„	...	†	‡	^	‰
\h8a	\h8b	\h8c		\h8e			\h91	\h92	\h93
Š	‹	Œ		Ž			‘	’	“
\h94	\h95	\h96	\h97	\h98	\h99	\h9a	\h9b	\h9c	
”	•	–	—	~	™	š	›	œ	
\h9e	\h9f		\ha1	\ha2	\ha3	\ha4	\ha5	\ha6	\ha7
ž	ÿ		i	¢	£	¤	¥	¦	§
\ha8	\ha9	\haa	\hab	\hac		\hae	\haf	\hb0	\hb1
¨	©	ª	«	¬		®	¯	°	±
\hb2	\hb3	\hb4	\hb5	\hb6	\hb7	\hb8	\hb9	\hba	\hbb
²	³	´	µ	¶	·	¸	¹	º	»
\hbc	\hbd	\hbe	\hbf	\hc0	\hc1	\hc2	\hc3	\hc4	\hc5
¼	½	¾	¿	À	Á	Â	Ã	Ä	Å
\hc6	\hc7	\hc8	\hc9	\hca	\hcb	\hcc	\hcd	\hce	\hcf
Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
\hd0	\hd1	\hd2	\hd3	\hd4	\hd5	\hd6	\hd7	\hd8	\hd9
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù
\hda	\hdb	\hdc	\hdd	\hde	\hdf	\he0	\he1	\he2	\he3
Ú	Û	Ü	Ý	Þ	ß	à	á	â	ã
\he4	\he5	\he6	\he7	\he8	\he9	\hea	\heb	\hec	\hed
ä	å	æ	ç	è	é	ê	ë	ì	í
\hee	\hef	\hf0	\hf1	\hf2	\hf3	\hf4	\hf5	\hf6	\hf7
î	ï	ð	ñ	ò	ó	ô	õ	ö	÷
\hf8	\hf9	\hfa	\hfb	\hfc	\hfd	\hfe	\hff		
ø	ù	ú	û	ü	ý	þ	ÿ		

Таблица 2. Символы, генерируемые командой \H

\H22	\H24	\H27	\H2d	\H40	\H44	\H46	\H47	\H4c	\H50
∇	∃	ə	–	≅	Δ	Φ	Γ	Λ	Θ
\H51	\H53	\H56	\H57	\H58	\H59	\H5c	\H5e	\H61	\H62
Θ	Σ	ς	Ω	Ξ	Ψ	∴	⊥	α	β
\H63	\H64	\H65	\H66	\H67	\H68	\H69	\H6a	\H6b	\H6c

χ	δ	ε	ϕ	γ	η	ι	φ	κ	λ
<code>\H6d</code> μ	<code>\H6e</code> ν	<code>\H70</code> π	<code>\H71</code> θ	<code>\H72</code> ρ	<code>\H73</code> σ	<code>\H74</code> τ	<code>\H75</code> υ	<code>\H76</code> ω	<code>\H77</code> ω
<code>\H78</code> ξ	<code>\H79</code> ψ	<code>\H7a</code> ζ	<code>\Ha1</code> Υ	<code>\Ha2</code> '	<code>\Ha3</code> \leq	<code>\Ha5</code> ∞	<code>\Ha7</code> \clubsuit	<code>\Ha8</code> \diamond	<code>\Ha9</code> \heartsuit
<code>\Haa</code> \spadesuit	<code>\Hab</code> \leftrightarrow	<code>\Hac</code> \leftarrow	<code>\Had</code> \uparrow	<code>\Hae</code> \rightarrow	<code>\Haf</code> \downarrow	<code>\Hb2</code> "	<code>\Hb3</code> \geq	<code>\Hb5</code> μ	<code>\Hb6</code> ∂
<code>\Hb9</code> \neq	<code>\Hba</code> \equiv	<code>\Hbb</code> \approx	<code>\Hbd</code> $ $	<code>\Hbe</code> $-$	<code>\Hbf</code> \lrcorner	<code>\Hc0</code> κ	<code>\Hc1</code> \Im	<code>\Hc2</code> \Re	<code>\Hc3</code> \wp
<code>\Hc4</code> \otimes	<code>\Hc5</code> \oplus	<code>\Hc6</code> \emptyset	<code>\Hc7</code> \cap	<code>\Hc8</code> \cup	<code>\Hc9</code> \supset	<code>\Hca</code> \supseteq	<code>\Hcb</code> $\not\subset$	<code>\Hcc</code> \subset	<code>\Hcd</code> \subseteq
<code>\Hce</code> \in	<code>\Hcf</code> \notin	<code>\Hd0</code> \angle	<code>\Hd1</code> ∇	<code>\Hd5</code> Π	<code>\Hd6</code> \surd	<code>\Hd9</code> \wedge	<code>\Hda</code> \vee	<code>\Hdb</code> \leftrightarrow	<code>\Hdc</code> \leftarrow
<code>\Hdd</code> \Uparrow	<code>\Hde</code> \Rightarrow	<code>\Hdf</code> \Downarrow	<code>\He0</code> \diamond	<code>\He1</code> '	<code>\He5</code> Σ	<code>\Hf1</code> '	<code>\Hf2</code> \int		

Модуль PT4TaskMakerNET: примеры разработки учебных заданий

Создание простейшей сводной группы

Вначале опишем действия по созданию наиболее простого варианта группы заданий — [сводной группы](#), в которой не разрабатываются новые задания, а лишь производится переконфигурация заданий из имеющихся групп.

Создадим группу заданий MakerDemo, в которую импортируем два первых задания из базовой группы Begin. Следуя правилам об именовании dll-файлов с группами заданий, дадим нашей библиотеке имя PT4MakerDemo.

Файл PT4MakerDemo.pas, содержащий сводную группу заданий, является кратким и имеет стандартную структуру:

```
library PT4MakerDemo;

uses PT4TaskMakerNET;

procedure InitTask(num: integer);
begin
  case num of
    1..2: UseTask('Begin', num);
  end;
end;

procedure inittaskgroup;
begin
  CreateGroup('MakerDemo', 'Примеры различных задач',
    'М. Э. Абрамян, 2013', 'qwqfsdf13dfttd', 2, InitTask);
end;

procedure activate(S: string);
begin
  ActivateNET(S);
end;

begin
end.
```

К библиотеке подключается модуль PT4TaskMakerNET, после чего в ней описывается основная процедура группы заданий [InitTask](#), определяющая задание по его номеру. Поскольку мы не создавали своих заданий, в данной процедуре используется только

стандартная процедура [UseTask](#), позволяющая импортировать задания из имеющихся групп. В нашем случае импортируются задания с номерами 1 и 2 из группы Begin.

Затем описывается процедура инициализации данной группы заданий. Она должна иметь стандартное имя `inittaskgroup` (**набранное строчными, т. е. маленькими буквами**). В этой процедуре вызывается процедура [CreateGroup](#), в которой задаются настройки создаваемой группы: имя ('`MakerDemo`'), описание ('`Примеры различных задач`'), сведения об авторе, строковый ключ, число заданий (2) и основная процедура группы ([InitTask](#)).

После процедуры `inittaskgroup` описывается вспомогательная процедура `activate` (ее имя также должно быть набрано строчными буквами), в которой необходимо вызвать процедуру `ActivateNET`, описанную в модуле `PT4TaskMakerNET`.

Тестирование созданной группы

Для успешной компиляции программы с созданной группой необходимо, чтобы ей был доступен модуль PT4TaskMakerNET. Этот модуль входит в число стандартных модулей библиотеки системы PascalABC.NET и размещается в подкаталоге LIB системного каталога PascalABC.NET, поэтому копировать его в рабочий каталог не требуется. Однако даже при успешной компиляции программы просмотреть задания группы не удастся, так как созданную библиотеку (dll-файл) нельзя запускать на выполнение (при успешной компиляции будет выведено сообщение «Невозможно запустить динамическую библиотеку»).

Для тестирования полученной библиотеки необходимо создать вспомогательную программу, являющуюся заготовкой для выполнения заданий из созданной группы. Так как после успешной компиляции библиотеки в рабочем каталоге уже содержится файл PT4MakerDemo.dll, для создания программы-заготовки можно использовать программный модуль PT4Load. Вызвав его окно на экран (для этого достаточно использовать клавиатурную комбинацию [Shift]+[Ctrl]+[L]) и удалив, при необходимости, имя ранее введенного задания, мы должны увидеть в списке доступных групп заданий созданную нами группу MakerDemo. Если имя группы MakerDemo не отображается, значит, задачник не смог успешно загрузить эту группу из библиотеки PT4MakerDemo.dll. В этом случае необходимо проверить имя созданной библиотеки (в частности, наличие в нем префикса PT4) и наличие в файле библиотеки процедур inittaskgroup и activate, определенных по описанным выше правилам.

Если имя группы появилось в списке, то надо ввести в поле «Задание» имя «MakerDemo1» и нажать клавишу [Enter] (или кнопку «Загрузка»); в результате будет создан файл MakerDemo1.pas, который сразу загрузится в редактор среды PascalABC.NET. Приведем содержимое этого файла:

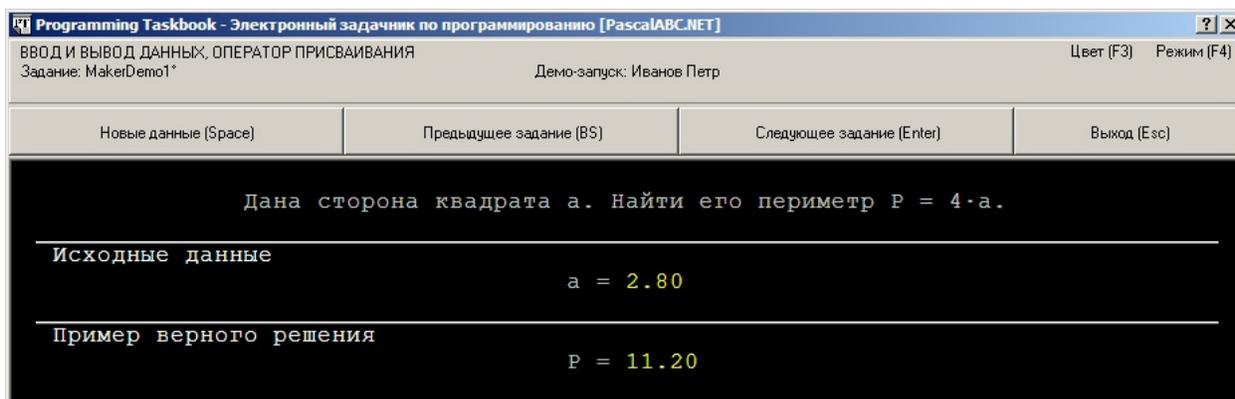
```
uses PT4;  
  
begin  
    Task( 'MakerDemo1' );
```

end.

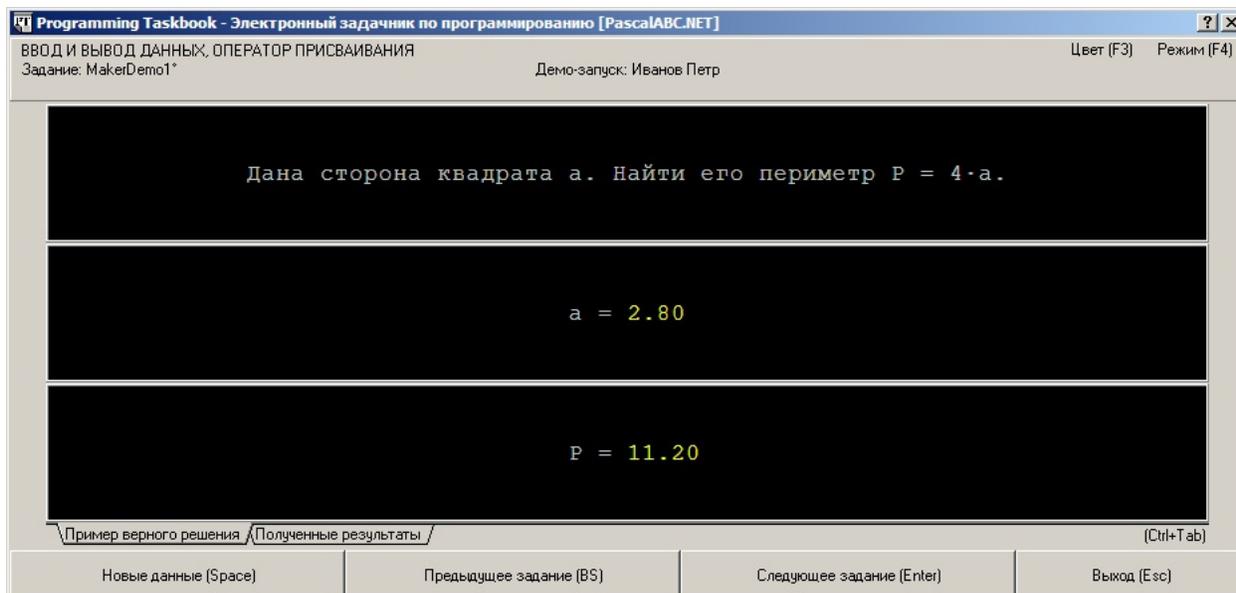
Поскольку мы собираемся просматривать задания группы в демо-режиме, добавим в конец строки с именем задания символ «?»:

```
Task( 'MakerDemo1?' );
```

После компиляции и запуска полученной программы на экране отобразится окно задачника с указанным заданием данной группы:



По умолчанию окно задачника отображается в режиме с динамической компоновкой, который появился в версии 4.11 и является более наглядным, чем режим с фиксированной компоновкой. Однако при разработке заданий желательно применять режим с фиксированной компоновкой, поскольку он позволит выявить недостатки форматирования (в частности, вертикального выравнивания данных), присущие только этому режиму. Для переключения между режимами отображения данных достаточно нажать клавишу [F4]. После выполнения этого действия окно задачника изменится следующим образом:



В окне задачника можно просматривать все имеющиеся задания данной группы (нажимая клавиши [Enter] и [Backspace]), а также генерировать различные варианты исходных данных и связанных с ними контрольных (т. е. «правильных») результатов. При закрытии окна программа немедленно завершит работу, и мы вернемся в редактор среды PascalABC.NET. Заметим, что при последующих запусках программы будет автоматически выбираться тот режим окна задачника, в котором оно находилось в момент его предшествующего закрытия.

Примечание. После добавления в группу нового задания было бы желательно, чтобы при запуске тестирующей программы на экране сразу отображались данные, связанные с последним добавленным заданием. Чтобы не приходилось каждый раз изменять номер задания в процедуре Task, можно удалить этот номер, указав символ «?» сразу после имени группы: `Task('MakerDemo?')`. В этом случае при запуске программы на экране будет отображаться последнее задание данной группы.

Добавление описания группы и ее подгрупп

По тексту, расположенному выше названия задания MakerDemo1 (см. приведенные выше рисунки), мы видим, что импортированные из группы Begin задания входят в подгруппу с заголовком «Ввод и вывод данных, оператор присваивания». В сводной группе MakerDemo мы можем добавить комментарий (*преамбулу*) как к самой группе, так и к любой имеющейся в ней подгруппе. Кроме того, мы можем *импортировать* преамбулу любой имеющейся группы или подгруппы. Для иллюстрации этих возможностей добавим в процедуру inittaskgroup новые операторы (их надо указать после вызова процедуры CreateGroup):

```
CommentText('Данная группа демонстрирует различные возможности');
CommentText('\Иконструктора учебных заданий\i \MPT4TaskMaker\m. ');

Subgroup('Ввод и вывод данных, оператор присваивания');
CommentText('В этой подгруппе содержатся задания, импортированные');
CommentText('из группы Begin.\PПриводимый ниже абзац преамбулы');
CommentText('также импортирован из данной группы.\P');
UseComment('Begin');
```

Два первых вызова процедуры [CommentText](#) определяют текст преамбулы для группы MakerDemo. Обратите внимание на [управляющие последовательности](#): пара последовательностей `\I` и `\i` выделяет *курсивный* фрагмент, а пара `\M` и `\m` выделяет фрагмент, в котором используется *моноширинный* шрифт. Последующий вызов процедуры Subgroup устанавливает режим определения преамбулы для подгруппы с указанным именем. В тексте этой преамбулы, который, как и текст преамбулы группы, определяется с помощью процедуры CommentText, используется управляющая последовательность `\P`, обеспечивающая переход к новому абзацу.

Наконец, последняя процедура ([UseComment](#)) импортирует преамбулу группы Begin в преамбулу нашей подгруппы «Ввод и вывод данных, оператор присваивания». Имеется также вариант процедуры UseComment, позволяющий импортировать преамбулу подгруппы; в этом варианте следует указать два параметра: имя группы и заголовок требуемой подгруппы, входящей в эту группу.

Импортировать преамбулы подгрупп можно только для тех групп заданий, в которых имеется разделение на подгруппы (обычно это группы, содержащие большое количество заданий). В группе Begin деления на подгруппы нет, поэтому из нее можно импортировать только преамбулу самой группы.

Для того чтобы ознакомиться с результатом сделанных изменений, следует сгенерировать html-страницу с текстом группы MakerDemo. Для этого достаточно внести небольшое изменение в тестирующую программу, а именно, следует заменить символ «?» в параметре процедуры Task на «#»: `Task('MakerDemo?')`. Теперь при запуске данной программы на экране вместо окна задачника появится html-браузер с описанием созданной группы:

Примеры различных задач

М. Э. Абрамян, 2013

Данная группа демонстрирует различные возможности конструктора учебных заданий PT4TaskMaker.

Ввод и вывод данных, оператор присваивания

В этой подгруппе содержатся задания, импортированные из группы Begin.

Приводимый ниже абзац преамбулы также импортирован из данной группы.

Все входные и выходные данные в заданиях этой группы являются вещественными числами.

MakerDemo1°. Дана сторона квадрата a . Найти его периметр $P = 4 \cdot a$.

MakerDemo2°. Дана сторона квадрата a . Найти его площадь $S = a^2$.

Дата генерации страницы: 04.02.2013.

Обратите внимание на последний абзац в описании подгруппы («Все входные и выходные данные в заданиях этой группы являются вещественными числами»), который был импортирован из группы Begin.

Примечание. Если указать в параметре процедуры Task символ «#», не удаляя номер задания (например, `Task('MakerDemo2#')`), то в html-описание будет включено только задание с указанным номером. При этом будут также выведены комментарии ко всей группе и к той подгруппе, к которой относится выбранное задание. Для включения в html-страницу нескольких заданий (или групп заданий) достаточно для каждого из них вызвать процедуру Task с параметром, оканчивающимся символом «#».

Добавление нового задания

Добавим к нашей группе новое задание. Фактически это задание будет дублировать задание Begin3, однако вместо импортирования этого задания мы разработаем его самостоятельно. Все действия по созданию нового задания удобно реализовать во вспомогательной процедуре, которую можно назвать MakerDemo3 (таким образом, название процедуры будет соответствовать имени создаваемого задания, хотя это и не является обязательным):

```
procedure MakerDemo3;
var
  a, b: real;
begin
  CreateTask('Ввод и вывод данных, оператор присваивания');
  TaskText('Даны стороны прямоугольника~{a} и~{b}.', 0, 2);
  TaskText('Найти его площадь {S}~==~{a}\*{b} и периметр {P}~==~2\*({a}
    0, 4);
  a := RandomN(1, 99) / 10;
  b := RandomN(1, 99) / 10;
  DataR('a = ', a, xLeft, 3, 4);
  DataR('b = ', b, xRight, 3, 4);
  ResultR('S = ', a * b, 0, 2, 4);
  ResultR('P = ', 2 * (a + b), 0, 4, 4);
  SetTestCount(3);
end;
```

Описание процедуры MakerDemo3 (как и описания всех других процедур, обеспечивающих формирование новых заданий) следует разместить перед описанием процедуры InitTask.

Процедура MakerDemo3 включает все основные действия, используемые при формировании нового задания:

- *инициализацию* нового задания (процедура [CreateTask](#); мы указали в этой процедуре, что данное задание должно входить в подгруппу «Ввод и вывод данных, оператор присваивания», т. е. в ту же подгруппу, что и два предыдущих задания);
- определение его *формулировки* (процедуры [TaskText](#); обратите внимание на используемые в этих процедурах [управляющие последовательности](#));
- определение исходных (процедуры [DataR](#)) и результирующих данных (процедуры [ResultR](#)); при этом исходные данные

генерируются с помощью датчика случайных чисел (процедура [RandomN](#));

- указание количества успешных тестовых запусков программы учащегося, достаточных для регистрации задания как выполненного (процедура [SetTestCount](#); для нашего простого задания достаточно трех *проведенных подряд* успешных тестовых запусков).

Необходимо также включить вызов созданной процедуры в основную процедуру группы MakerDemo, связав его с номером 3:

```
procedure InitTask(num: integer);
begin
  case num of
    1..2: UseTask('Begin', num);
    3: MakerDemo3;
  end;
end;
```

Наконец, следует откорректировать число заданий в вызове процедуры CreateGroup, изменив его на 3.

Запустив тестирующую программу, мы увидим в html-описании группы MakerDemo формулировки трех заданий, а выполнив обратную замену в этой программе символа «#» на «?» (в результате вызов процедуры Task опять примет вид `Task('MakerDemo?')`) и повторно запустив программу на выполнение, мы увидим окно задачника с загруженным заданием MakerDemo3. Заметим, что при последующих запусках проекта мы будем получать в окне задачника различные исходные данные; это связано с тем, что при генерации исходных данных используется датчик случайных чисел.

Добавление заданий на обработку двумерных массивов и символьных строк

Добавим к группе MakerDemo еще два задания: первое из них дублирует задание Matrix7 (подгруппа «Двумерные массивы (матрицы): вывод элементов»), а второе не имеет полного аналога в группе String, однако может быть отнесено к ее первой подгруппе: «Символы и строки: основные операции». Реализуем эти задания в процедурах MakerDemo4 и MakerDemo5:

```
procedure MakerDemo4;
var
  m, n, i, j, k: integer;
  a: array [1..5, 1..8] of real;
begin
  CreateTask('Двумерные массивы (матрицы): вывод элементов');
  TaskText('Дана матрица размера~{M}\;\x\;{N} и целое число~{K} (1~\
    0, 2);
  TaskText('Вывести элементы {K}-й строки данной матрицы.', 0, 4);
  m := RandomN(2, 5);
  n := RandomN(4, 8);
  k := 1;
  if m = 5 then k := 0;
  DataN('M = ', m, 3, 1, 1);
  DataN('N = ', n, 10, 1, 1);
  for i := 1 to m do
    for j := 1 to n do
      begin
        a[i, j] := RandomR(-9.99, 9.99);
        DataR(a[i, j], Center(j, n, 5, 1), i + k, 5);
      end;
    k := RandomN(1, m);
    DataN('K = ', k, 68, 5, 1);
  for j := 1 to n do
    ResultR(a[k, j], Center(j, n, 5, 1), 3, 5);
  SetTestCount(5);
end;

procedure MakerDemo5;
var
  s: string;
begin
  CreateTask('Символы и строки: основные операции');
  TaskText('Дана непустая строка~{S}.', 0, 2);
  TaskText('Вывести ее первый и последний символ.', 0, 4);
```

```

s := WordSample(RandomN(0, WordCount-1));
if CurrentTest = 3 then
  while s[1] = s[Length(s)] do
    s := WordSample(RandomN(0, WordCount-1));
  DataS('S = ', s, 0, 3);
  ResultC('Первый символ: ', s[1], xLeft, 3);
  ResultC('Последний символ: ', s[Length(s)], xRight, 3);
  SetTestCount(4);
end;

```

Обратите внимание на использование вспомогательной функции [Center](#) для центрирования строк матрицы в области исходных и результирующих данных: каждый элемент матрицы занимает 5 экранных позиций, а между элементами размещается по одному пробелу. В процедуре `MakerDemo4` не вызывается процедура `SetTestCount`; в этом случае число успешных тестов, необходимых для регистрации задания как выполненного, по умолчанию полагается равным 5.

При выводе элементов исходной матрицы и результирующей матричной строки дополнительные комментарии указывать не требуется, поэтому используется вариант процедур `DataR` и `ResultR`, в котором комментарий отсутствует (этот вариант процедур групп `Data` и `Result` добавлен в версию 4.11 конструктора учебных заданий).

В процедуре `MakerDemo5` для получения исходных символьных строк используются функции [WordCount](#) и [WordSample](#). С помощью этих функций можно получать различные варианты русских слов. Заметим, что в конструкторе `PT4TaskMaker` имеются также функции `EnWordCount` и `EnWordSample`, с помощью которых можно получать варианты английских слов.

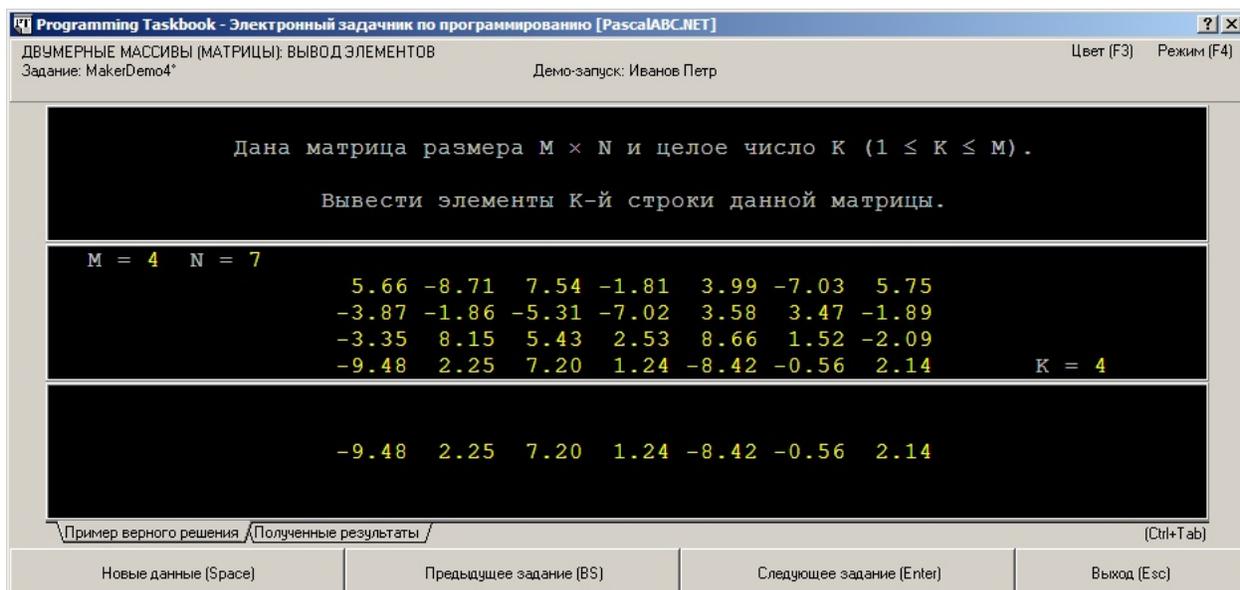
В процедуре `MakerDemo5` использована еще одна возможность, появившаяся в версии 4.11 конструктора: функция [CurrentTest](#), возвращающая порядковый номер текущего тестового запуска. Использование этой функции позволяет связать какой-либо особый вариант теста с некоторым номером тестового испытания, и тем самым гарантировать, что программа с решением задачи обязательно будет проверена на этом особом варианте теста. В нашем случае строка `S` выбирается из набора слов-образцов, среди

которых имеется сравнительно большое число слов, начинающихся и оканчивающихся одной и той же буквой. Для более надежного тестирования решения желательно гарантировать, что в наборе тестов будет *хотя бы один тест*, в котором начальный и конечный символ исходной строки различаются. Разумеется, можно было бы *всегда* выбирать подобные строки, используя соответствующий цикл `while`. Однако при наличии функции `CurrentTest` в этом нет необходимости: достаточно выполнять подобный цикл для единственного теста, например, с номером 3, как это сделано в приведенной реализации задания. В дальнейшем мы рассмотрим более содержательный пример использования функции `CurrentTest`.

Осталось изменить количество заданий в вызове процедуры `CreateGroup` на 5 и включить вызовы новых процедур в основную процедуру группы `InitTask`:

```
procedure InitTask(num: integer);
begin
  case num of
    1..2: UseTask('Begin', num);
    3: MakerDemo3;
    4: MakerDemo4;
    5: MakerDemo5;
  end;
end;
```

Приведем вид окна задачника для новых заданий:



Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

СИМВОЛЫ И СТРОКИ: ОСНОВНЫЕ ОПЕРАЦИИ Цвет (F3) Режим (F4)
Задание: MakerDemo5* Демо-запуск: Иванов Петр

Дана непустая строка S.
Вывести ее первый и последний символ.

S = 'ДИАФРАГМА'

Первый символ: 'Д' Последний символ: 'А'

Пример верного решения / Полученные результаты / (Ctrl+Tab)

Новые данные (Space)	Предыдущее задание (BS)	Следующее задание (Enter)	Выход (Esc)
----------------------	-------------------------	---------------------------	-------------

Добавление заданий на обработку файлов

Добавим к группе MakerDemo еще два задания: первое из них дублирует задание File63 (подгруппа «Символьные и строковые файлы»), а второе — задание Text16 (подгруппа «Текстовые файлы: основные операции»). Реализуем эти задания в процедурах MakerDemo6 и MakerDemo7:

```
function FileName(Len: integer): string;
const
  c = '0123456789abcdefghijklmnopqrstuvwxyz';
var
  i: integer;
begin
  result := '';
  for i := 1 to Len do
    result := result + c[RandomN(1, Length(c))];
end;

procedure MakerDemo6;
var
  k, i, j, jmax: integer;
  s1, s2, s3: string;
  fs1: file of ShortString;
  fs2: file of ShortString;
  fc3: file of char;
  s: ShortString;
  c: char;
begin
  CreateTask('Символьные и строковые файлы');
  TaskText(
    'Дано целое число~{K} (>\,0) и строковый файл.#13 +
    'Создать два новых файла: строковый, содержащий первые {K}~символо
    'каждой строки исходного файла, и символьный, содержащий {K}-й сим
    'каждой строки (если длина строки меньше~{K}, то в строковый файл'
    'записывается вся строка, а в символьный файл записывается пробел)
  );
  s1 := '1' + FileName(5) + '.tst';
  s2 := '2' + FileName(5) + '.tst';
  s3 := '3' + FileName(5) + '.tst';
  Assign(fs1, s1);
  Rewrite(fs1);
  Assign(fs2, s2);
  Rewrite(fs2);
  Assign(fc3, s3);
```

```

Rewrite(fc3);
k := RandomN(2, 11);
jmax := 0;
for i := 1 to RandomN(10, 20) do
begin
  j := RandomN(2, 16);
  if jmax < j then
    jmax := j;
  s := FileName(j);
  write(fs1, s);
  if j >= k then
    c := s[k]
  else
    c := ' ';
  write(fc3, c);
  s := copy(s, 1, k);
  write(fs2, s);
end;
Close(fs1);
Close(fs2);
Close(fc3);
DataN('K = ', k, 0, 1, 1);
DataS('Имя исходного файла: ', s1, 3, 2);
DataS('Имя результирующего строкового файла: ', s2, 3, 4);
DataS('Имя результирующего символьного файла: ', s3, 3, 5);
DataComment('Содержимое исходного файла:', xRight, 2);
DataFileS(s1, 3, jmax + 3);
ResultComment('Содержимое результирующего строкового файла:',
  0, 2);
ResultComment('Содержимое результирующего символьного файла:',
  0, 4);
ResultFileS(s2, 3, k + 3);
ResultFileC(s3, 5, 4);
end;

procedure MakerDemo7;
var
  p: integer;
  s, s1, s2, s0: string;
  t1, t2: text;
begin
  CreateTask('Текстовые файлы: основные операции');
  TaskText('Дан текстовый файл.', 0, 2);
  TaskText('Удалить из него все пустые строки.', 0, 4);
  s1 := FileName(6) + '.tst';
  s2 := '#' + FileName(6) + '.tst';
  s := TextSample(RandomN(0, TextCount-1));

```

```

Assign(t2, s2);
Rewrite(t2);
Assign(t1, s1);
Rewrite(t1);
writeln(t2, s);
Close(t2);
s0 := #13#10#13#10;
p := Pos(s0, s);
while p <> 0 do
begin
  Delete(s, p, 2);
  p := Pos(s0, s);
end;
writeln(t1, s);
Close(t1);
ResultFileT(s1, 1, 5);
Rename(t2, s1);
DataFileT(s1, 2, 5);
DataS('Имя файла: ', s1, 0, 1);
SetTestCount(3);
end;

```

При реализации этих заданий используется вспомогательная функция `FileName(Len)`, позволяющая создать случайное имя файла длины `Len` (без расширения). Имя файла при этом будет содержать только цифры и строчные (маленькие) латинские буквы.

Имена файлов, полученные с помощью функции `FileName`, дополняются расширением `.tst` (заметим, что в базовых группах `File`, `Text` и `Param` это расширение используется в именах всех исходных и результирующих файлов).

Функция `FileName` используется также для генерации элементов строкового файла в процедуре `MakerDemo6`.

Для того чтобы предотвратить возможность случайного совпадения имен файлов, в процедуре `MakerDemo6` к созданным именам добавляются префиксы: **1** для первого файла, **2** для второго, **3** для третьего. В процедуре `MakerDemo7` имя временного файла дополняется префиксом `#`, что также гарантирует его отличие от имени основного файла задания.

В процедуре `MakerDemo6` использован новый вариант [процедуры TaskText](#), появившийся в версии 4.11 задачника. В этом варианте

процедура `TaskText` принимает один строковый параметр, который определяет *всю формулировку задания*, причем в качестве разделителей строк, входящих в формулировку, можно использовать символы `#13`, `#10` или их комбинацию `#13#10` (в указанном порядке). Новый вариант процедуры `TaskText` позволяет более наглядно отобразить формулировку задания и не требует указания дополнительных параметров.

При реализации задания на обработку текстовых файлов для генерации содержимого файла используются функции [TextCount](#) и [TextSample](#). Строка, возвращаемая функцией `TextSample`, представляет собой текст, содержащий *маркеры конца строки* — символы `#13#10`. Указанные символы *разделяют соседние строки текста* (в конце текста маркер конца строки не указывается). Благодаря наличию маркеров конца строки полученный текст можно записать в текстовый файл с помощью единственной процедуры `writeln`, которая, кроме записи текста, обеспечивает добавление маркера конца строки в конец файла.

После разработки новых заданий необходимо изменить количество заданий в вызове процедуры `CreateGroup` на 7 и включить вызовы новых процедур в основную процедуру группы `InitTask`:

```
procedure InitTask(num: integer);
begin
  case num of
    1..2: UseTask('Begin', num);
    3: MakerDemo3;
    4: MakerDemo4;
    5: MakerDemo5;
    6: MakerDemo6;
    7: MakerDemo7;
  end;
end;
```

Приведем вид окна задачника для новых заданий:

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] [?] X

СИМВОЛЬНЫЕ И СТРОКОВЫЕ ФАЙЛЫ Цвет (F3) Режим (F4)
 Задание: MakerDemo6* Демо-запуск: Иванов Петр

Дано целое число $K (>0)$ и строковый файл.
 Создать два новых файла: строковый, содержащий первые K символов каждой строки исходного файла, и символьный, содержащий K -й символ каждой строки (если длина строки меньше K , то в строковый файл записывается вся строка, а в символьный файл записывается пробел).

(-) $K = 3$

- Имя исходного файла: '1uq9el.tst' Содержимое исходного файла:
 1: 'va9x0bsu2' 'z3' 'lwx114fuyq4'

Имя результирующего строкового файла: '2ba571.tst'
 Имя результирующего символьного файла: '3eoydt.tst'

Содержимое результирующего строкового файла:
 1: 'va9' 'z3' 'lwx' 'w9g' 'nwj' 'dnb' 'w4a' 'hx8' 'hi5' 'ah' 'js8'

Содержимое результирующего символьного файла:
 1: '9' ' ' 'x' 'g' 'j' 'b' 'a' '8' '5' ' ' '8'

(+) \Пример верного решения / Полученные результаты / (Ctrl+Tab)

Новые данные (Space)	Предыдущее задание (BS)	Следующее задание (Enter)	Выход (Esc)
----------------------	-------------------------	---------------------------	-------------

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] [?] X

ТЕКСТОВЫЕ ФАЙЛЫ: ОСНОВНЫЕ ОПЕРАЦИИ Цвет (F3) Режим (F4)
 Задание: MakerDemo7* Демо-запуск: Иванов Петр

Дан текстовый файл.
 Удалить из него все пустые строки.

(-) Имя файла: '802zf7.tst'

- 1: 'А потом Винни-Пух закричал:'
 ' '
 '- Ай, спасите! Я лучше полезу назад!'
 ' '

1: 'А потом Винни-Пух закричал:'
 '- Ай, спасите! Я лучше полезу назад!'
 'Еще потом он закричал:'
 '- Ай, помогите! Нет, уж лучше вперед!'
 'И, наконец, он завопил отчаянным голосом:'

(+) \Пример верного решения / Полученные результаты / (Ctrl+Tab)

Новые данные (Space)	Предыдущее задание (BS)	Следующее задание (Enter)	Выход (Esc)
----------------------	-------------------------	---------------------------	-------------

Добавление заданий на обработку динамических структур данных

Наконец, добавим в нашу группу задание, посвященное обработке динамических структур данных, причем представим его в двух вариантах: традиционном, основанном на использовании записей типа TNode и связанных с ними указателей типа PNode, и «объектном», характерном для .NET-языков (C#, Visual Basic .NET, PascalABC.NET), а также языков Python и Java. Следует подчеркнуть, что при *разработке* как традиционного, так и объектного варианта заданий на динамические структуры надо использовать типы TNode и PNode и связанные с ними процедуры конструктора учебных заданий. В то же время, при *выполнении* объектного варианта задания на соответствующем языке требуется использовать объекты типа Node (которые при разработке задания не применяются).

Задание, которое мы реализуем, дублирует задание Dynamic30, посвященное преобразованию односвязного списка в двусвязный (подгруппа «Динамические структуры данных: двусвязный список»). Оформим два варианта этого задания в виде процедур MakerDemo8 и MakerDemo8Obj:

```
var WrongNode: TNode;

procedure MakerDemo8Data;
var
  i, n: integer;
  p, p1, p2: PNode;
begin
  if RandomN(1, 4) = 1 then
    n := 1
  else
    n := RandomN(2, 9);
  case CurrentTest of
  2: n := 1;
  4: n := RandomN(2, 9);
  end;
  new(p1);
  p1^.Data := RandomN(10, 99);
  p1^.Prev := nil;
  p2 := p1;
```

```

for i := 2 to n do
begin
    new(p);
    p^.Data := RandomN(10, 99);
    p^.Prev := p2;
    p2^.Next := p;
    p2 := p;
end;
p2^.Next := nil;
SetPointer(1, p1);
SetPointer(2, p2);
ResultP('Последний элемент: ', 2, 0, 2);
ResultList(1, 0, 3);
ShowPointer(2);
DataP(1, 0, 2);
p := p1;
for i := 1 to n do
begin
    p^.Prev := @WrongNode;
    p := p^.Next;
end;
DataList(1, 0, 3);
ShowPointer(1);
end;

procedure MakerDemo8;
begin
    CreateTask('Динамические структуры данных: двусвязный список');
    TaskText('Дан указатель~{P}_1 на начало непустой цепочки ' +
        'элементов-записей типа TNode,', 0, 1);
    TaskText('связанных между собой с помощью поля Next. Используя ' +
        'поле Prev записи TNode,', 0, 2);
    TaskText('преобразовать исходную (\Иодносвязную\i) цепочку ' +
        'в \Идвусвязную\i, в которой каждый', 0, 3);
    TaskText('элемент связан не только с последующим элементом ' +
        '(с помощью поля Next),', 0, 4);
    TaskText('но и с предыдущим (с помощью поля Prev). Поле Prev ' +
        'первого элемента положить', 0, 5);
    TaskText('равным \N. Вывести указатель на последний элемент ' +
        'преобразованной цепочки.', 0, 0);
    MakerDemo8Data;
end;

procedure MakerDemo8Obj;
begin
    CreateTask('Динамические структуры данных: двусвязный список');
    TaskText(

```

```

'Dана ссылка~{A}_1 на начало непустой цепочки элементов-объектов т
'связанных между собой с помощью своих свойств Next. Используя сво
'данных объектов, преобразовать исходную (\Iодносвязную\i) цепочку
'в которой каждый элемент связан не только с последующим элементом
'свойства Next), но и с предыдущим (с помощью свойства Prev). Свой
'первого элемента положить равным \0. Вывести ссылку~{A}_2 на посл
'элемент преобразованной цепочки.'
);
SetObjectStyle;
MakerDemo8Data;
end;

```

Анализируя приведенные варианты процедур, легко заметить, что они отличаются лишь деталями формулировки задания. Алгоритмы генерации исходных и контрольных данных для традиционного и объектного вариантов совпадают, поэтому они выделены в отдельную вспомогательную процедуру MakerDemo8Data. В то же время *представления* динамических структур и связанных с ними указателей или объектов будут отличаться (см. рисунки, приведенные ниже). Необходимые корректировки в представлении динамических структур выполняются задачиком автоматически, с учетом используемого языка программирования.

Однако для языка PascalABC.NET требуемую настройку необходимо выполнить явно, так как в нем можно использовать оба варианта представления динамических структур: традиционный (как для обычного Паскаля в системах Delphi и Free Pascal Lazarus) и объектный (как в языках C#, Visual Basic .NET, Python и Java). Для того чтобы представление динамических данных при выполнении задания в среде PascalABC.NET соответствовало объектному варианту, следует в начале процедуры, реализующей задание (перед вызовом любых процедур, связанных с указателями и динамическими структурами), вызвать специальную процедуру без параметров [SetObjectStyle](#). Для остальных языков данная процедура не выполняет никаких действий.

Обратите внимание на возможность использования в формулировке задания более 5 экранных строк. Строки, которые не уместятся в области формулировки задания, следует добавлять к заданию процедурой TaskText, указывая в качестве последнего параметра процедуры число 0 (см. процедуру MakerDemo8). Еще проще

задавать «длинные» формулировки заданий с помощью нового варианта процедуры TaskText с единственным строковым параметром, содержащим все строки формулировки (см. процедуру MakerDemo9). При наличии подобных строк в окне задачника (если окно находится в режиме с фиксированной компоновкой) слева от области формулировки появятся кнопки, обеспечивающие прокрутку формулировки задания; кроме этих кнопок для прокрутки можно также использовать стандартные клавиши, в частности, клавиши со стрелками.

Для того чтобы имя нулевого указателя (или объекта) соответствовало используемому языку программирования, в формулировке задания применяются управляющие последовательности `\N` (имя нулевого указателя) и `\0` (имя нулевого объекта). Для языка PascalABC.NET обе эти последовательности генерируют текст `nil`.

Достаточно часто алгоритмы, разработанные учащимися для обработки динамических структур данных, дают неверные результаты в случае особых (хотя и допустимых) структур, например, состоящих только из одного элемента. Поэтому желательно предусмотреть появление подобных структур в тестовых наборах исходных данных. В наших заданиях исходный список, состоящий из *одного* элемента, будет предлагаться программе учащегося в среднем один раз при каждом четырех тестовых испытаниях. Кроме того, благодаря использованию функции [CurrentTest](#), появившейся в версии 4.11 конструктора, вариант списка с единственным элементом будет предложен программе учащегося для обработки в тесте номер 2, а вариант списка с более чем одним элементом — в тесте номер 4. Таким образом, можно гарантировать, что при прохождении набора из 5 тестовых испытаний программе будут предложены как «стандартные», так и «особые» наборы исходных данных.

При формировании односвязной структуры неиспользуемые поля Prev для каждого элемента структуры следует положить равными адресу «фиктивного» элемента (в нашем случае — переменной WrongNode), не связанного с данной структурой. Заметим, что для всех элементов, кроме первого, значения поля Prev можно было бы

положить равными nil, однако это не подходит для первого элемента: если поле Prev первого элемента будет равно nil, то слева от него будет выведен «лишний» (в данной ситуации) текст nil<.

Характерной особенностью разработки заданий на динамические структуры является *обратный порядок* создания этих структур: вначале создаются *контрольные* структуры (которые сразу передаются в задачник), а затем они преобразуются в соответствующие *исходные* структуры, которые должны не только передаваться в задачник, но и *оставаться в памяти*, чтобы в дальнейшем их можно было использовать в программе учащегося, выполняющей это задание.

Если в группу включаются задания на динамические структуры, то необходимо *анализировать текущий язык программирования*, используемый задачиком. Это обусловлено двумя причинами:

- имеются языки, для которых отсутствует возможность выполнять задания на обработку динамических структур (например, Visual Basic и 1С);
- в языках платформы .NET, а также Python и Java, необходимо использовать «объектный» стиль формулировок вместо стиля, основанного на указателях и применяемого для языков Pascal и C++.

Кроме того, следует определиться с выбором стиля для языка PascalABC.NET, поскольку в нем можно использовать как стиль указателей, так и стиль объектов. Можно, например, включить в группу заданий для языка PascalABC.NET оба варианта каждого задания.

Отмеченные обстоятельства приводят к тому, что для разных языков программирования создаваемая группа может содержать разное число заданий и, кроме того, для этих заданий будут использоваться разные инициализирующие процедуры.

С учетом этих замечаний изменим основную процедуру группы InitTask следующим образом:

```
procedure InitTask(num: integer);
begin
  case num of
```

```

1..2: UseTask('Begin', num);
3: MakerDemo3;
4: MakerDemo4;
5: MakerDemo5;
6: MakerDemo6;
7: MakerDemo7;
8: if CurrentLanguage and lgWithPointers <> 0 then
    MakerDemo8
    else
    MakerDemo8Obj;
9: MakerDemo8Obj;
end;
end;

```

В этой процедуре используется функция [CurrentLanguage](#), позволяющая определить текущий язык программирования, используемый задачиком. Если текущий язык относится к категории языков, поддерживающих указатели (в том числе PascalABC.NET), то в качестве задания номер 8 вызывается процедура MakerDemo8, в которой задание формулируется в терминах указателей. В противном случае вызывается вариант задания, использующий объектную терминологию. При использовании языка PascalABC.NET число заданий в группе будет равно 9; при этом дополнительное задание номер 9 будет представлять собой «объектный» вариант задания номер 8.

Функцию CurrentLanguage потребуется использовать и в начале процедуры inittaskgroup для того, чтобы правильно определить количество заданий в группе для разных языков программирования. Приведем фрагмент, на который надо заменить вызов процедуры CreateGroup и предшествующее ему ключевое слово begin (обратите внимание на то, что теперь в качестве предпоследнего параметра процедуры CreateGroup используется переменная n):

```

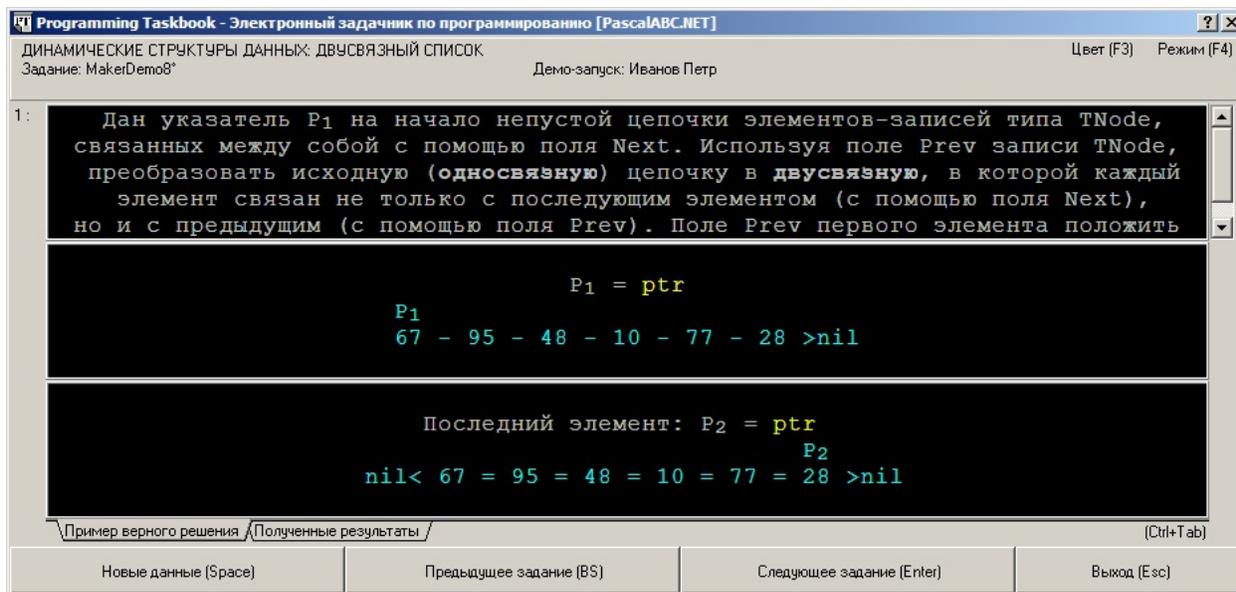
var
  n: integer;
begin
  n := 7;
  if CurrentLanguage = lgPascalABCNET then
    n := 9
  else
    if CurrentLanguage and (lgWithPointers or lgWithObjects) <> 0 then
      n := 8;

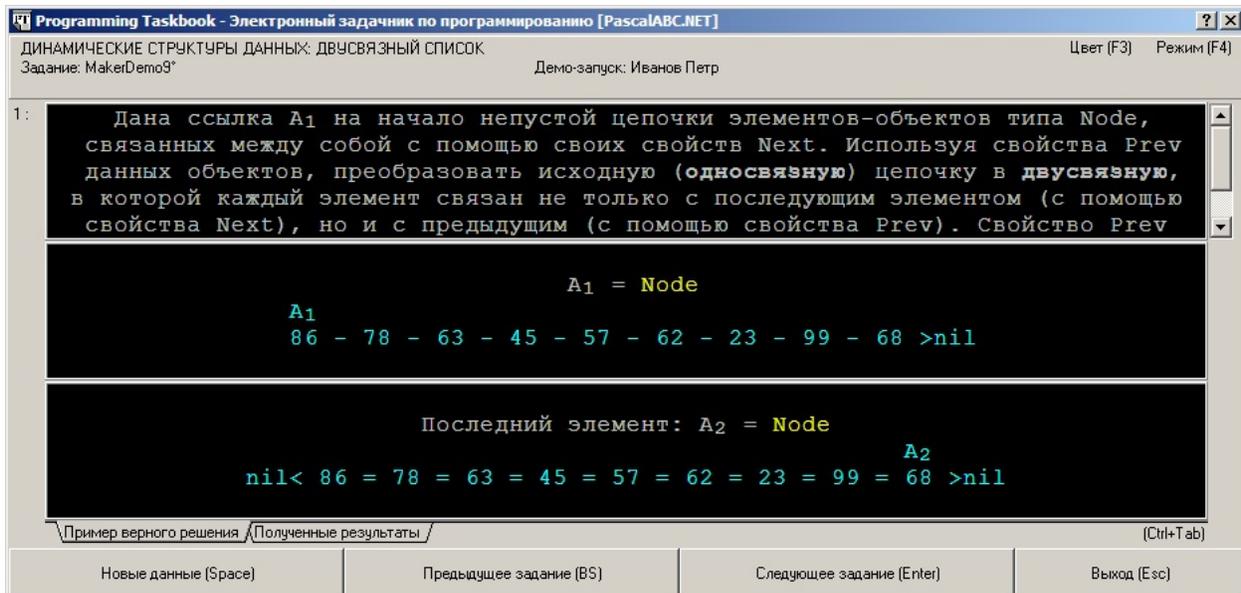
```

```
CreateGroup('MakerDemo', 'Примеры различных задач',  
    'М. Э. Абрамян, 2013', 'qwqfsdf13dfttd', n, InitTask);
```

Приведенный набор условий будет правильно определять количество заданий и в случае, если состав языков, поддерживаемых задачиком, будет расширен. Это обеспечивается тем, что в условиях используются не константы для конкретных языков (за исключением константы `IgPascalABCNET`), а битовые маски `IgWithPointers` и `IgWithObjects`. Первая из этих масок включает все языки, для которых в задачнике можно использовать варианты заданий на динамические структуры, основанные на указателях, а вторая — все языки, позволяющие использовать варианты аналогичных заданий в объектной терминологии.

В среде `PascalABC.NET` можно протестировать оба варианта реализованного задания. Приведем вид окна задачника для этого задания (первый рисунок соответствует варианту задания, использующему указатели, второй — варианту, использующему объекты). Обратите внимание на кнопки, расположенные справа от формулировки задания и обеспечивающие ее прокрутку.





Завершая оформление модуля `PT4MakerDemo`, добавим комментарии к новым подгруппам заданий (указанные операторы следует поместить в конец процедуры `inittaskgroup`):

```

Subgroup('Двумерные массивы (матрицы): вывод элементов');
CommentText('Данное задание дублирует задание Matrix7.');
```

```

Subgroup('Символьные и строковые файлы');
CommentText('Данное задание дублирует задание File63.');
```

```

CommentText('Оно демонстрирует особенности, связанные с двоичными');
CommentText('\Истроковыми\i файлами.');
```

```

Subgroup('Текстовые файлы: основные операции');
CommentText('Данное задание дублирует задание Text16.');
```

```

Subgroup('Динамические структуры данных: двусвязный список');
CommentText('Данное задание дублирует задание Dynamic30.');
```

```

CommentText('\РЗадание реализовано в двух вариантах: основанном на');
CommentText('(для языков Pascal и C++) и основанном на использован');
CommentText('а также Python и Java). Для языка Visual Basic это за');
CommentText('В системе PascalABC.NET доступны оба варианта задания');

```

Приведем заключительную часть `html`-страницы с описанием данной группы:

Двумерные массивы (матрицы): вывод элементов

Данное задание дублирует задание Matrix7.

MakerDemo4°. Дана матрица размера $M \times N$ и целое число K ($1 \leq K \leq M$). Вывести элементы K -й строки данной матрицы.

Символы и строки: основные операции

MakerDemo5°. Дана непустая строка S . Вывести ее первый и последний символ.

Символьные и строковые файлы

Данное задание дублирует задание File63. Оно демонстрирует особенности, связанные с двоичными строковыми файлами.

MakerDemo6°. Дано целое число K (> 0) и строковый файл. Создать два новых файла: строковый, содержащий первые K символов каждой строки исходного файла, и символьный, содержащий K -й символ каждой строки (если длина строки меньше K , то в строковый файл записывается вся строка, а в символьный файл записывается пробел).

Текстовые файлы: основные операции

Данное задание дублирует задание Text16.

MakerDemo7°. Дан текстовый файл. Удалить из него все пустые строки.

Динамические структуры данных: двусвязный список

Данное задание дублирует задание Dynamic30.

Задание реализовано в двух вариантах: основанном на использовании указателей (для языков Pascal и C++) и основанном на использовании объектов (для языков платформы .NET, а также Python и Java). Для языка Visual Basic это задание недоступно. В системе PascalABC.NET доступны оба варианта задания.

MakerDemo8°. Дан указатель P_1 на начало непустой цепочки элементов-записей типа TNode, связанных между собой с помощью поля Next. Используя поле Prev записи TNode, преобразовать исходную (*односвязную*) цепочку в *двусвязную*, в которой каждый элемент связан не только с последующим элементом (с помощью поля Next), но и с предыдущим (с помощью поля Prev). Поле Prev первого элемента положить равным nil. Вывести указатель на последний элемент преобразованной цепочки.

MakerDemo9°. Дана ссылка A_1 на начало непустой цепочки элементов-объектов типа Node, связанных между собой с помощью своих свойств Next. Используя свойства Prev данных объектов, преобразовать исходную (*односвязную*) цепочку в *двусвязную*, в которой каждый элемент связан не только с последующим элементом (с помощью свойства Next), но и с предыдущим (с помощью свойства Prev). Свойство Prev первого элемента положить равным nil. Вывести ссылку A_2 на последний элемент преобразованной цепочки.

Модуль PT4TaskMakerNET: разработка заданий, связанных с ЕГЭ по информатике

Группы заданий Exam и их особенности

Начиная с версии 4.10, в базовый набор задачника Programming Taskbook для языков Pascal и C++ входят специальные группы заданий, связанные с ЕГЭ по информатике: ExamBegin и ExamTaskC. Порядок выполнения заданий из этих групп имеет ряд особенностей, основной из которых является отказ от применения специальных средств ввода-вывода, входящих в состав задачника. В заданиях групп Exam для ввода-вывода надо применять стандартные средства используемого языка программирования. Это позволяет максимально приблизить вид программы, выполняющей задание, к виду, требуемому на экзамене, а также учесть при выполнении задания его дополнительные особенности, связанные с организацией ввода исходных данных и форматированием результатов.

С использованием конструктора учебных заданий PT4TaskMaker преподаватель может разрабатывать новые группы заданий, связанные с ЕГЭ по информатике. При этом необходимо следовать дополнительным правилам, основные из которых приводятся ниже.

1. Любые группы заданий, связанные с ЕГЭ по информатике, должны иметь имена, начинающиеся с префикса Exam (для групп с этим префиксом задачник генерирует специальные программы-заготовки, позволяющие использовать при выполнении задания стандартные средства ввода-вывода).
2. Необходимо проверять номер текущей версии задачника и текущий язык программирования, создавая новую группу только в случае, если версия имеет номер не ниже 4.10, а языком программирования является Pascal или C++.
3. В преамбуле к группе заданий желательно отметить тот факт, что для ввода-вывода необходимо использовать стандартные средства языка.
4. В новые группы Exam следует импортировать только те задания, которые также относятся к группам Exam.
5. Набор исходных и контрольных данных надо сохранять в текстовых файлах, передавая задачнику информацию об именах этих файлов (процедурами DataS) и связывая содержимое этих файлов с разделами исходных и

результатирующих данных (процедурами DataFileT и ResultFileT соответственно).

6. При любых вариантах наборов исходных данных соответствующие контрольные файлы не должны быть пустыми (при наличии пустого файла результатов задачник считает запуск программы ознакомительным).

Проиллюстрируем эти правила, разработав в среде PascalABC.NET демонстрационную группу заданий ExamDemo. Задания, связанные с ЕГЭ, вполне допустимо разрабатывать и на других языках, поддерживаемых конструктором учебных заданий, в частности, на языке C++ или на языке Pascal в средах Delphi или Lazarus, причем полученные реализации не будут иметь никаких существенных отличий от реализации, приведенной ниже. Задания можно разрабатывать даже на языке C#, несмотря на то что *выполнять* их на этом языке будет нельзя.

Реализация сводной группы заданий

Напомним, что *сводной группой* называется группа, все задания которой импортированы из уже имеющихся групп. Сводные группы оказываются очень полезными при составлении вариантов проверочных работ, поскольку позволяют дать заданиям новые имена и тем самым затрудняют применение разного рода шпаргалок.

Так как в сводных группах отсутствуют новые задания, при разработке сводных групп для заданий, связанных с ЕГЭ по информатике, достаточно учесть правила 1–4, приведенные в предыдущем пункте.

Будем предполагать, что общие правила разработки новых групп заданий в среде PascalABC.NET читателю известны (см. раздел «[Примеры](#)»).

Следуя правилам именования групп (имя должно состоять из латинских букв и цифр, иметь длину не более 9 символов и не оканчиваться цифрой), а также правилу 1 из предыдущего пункта (наличие префикса Exam), назовем нашу группу ExamDemo. Динамическая библиотека в этом случае должна иметь имя PT4ExamDemo.

Импортируем в группу ExamDemo несколько заданий из обеих групп Exam, входящих в базовый набор. Из группы ExamBegin возьмем задания ExamBegin71 и ExamBegin72, входящие в подгруппу «Преобразование массивов» и связанные перестановкой элементов массива. Из группы ExamTaskC возьмем серию из 12 заданий ExamTaskC25–ExamTaskC36, объединенных общей предметной областью: сведениями об абитуриентах из различных школ.

Учитывая правила подготовки динамических библиотек с группами учебных заданий, а также правила 2 и 3 из предыдущего пункта, получим следующий вариант нашей библиотеки (файл PT4ExamDemo.pas):

```
library PT4ExamDemo;  
  
uses PT4TaskMakerNET;
```

```

procedure InitTask(num: integer);
begin
  case num of
    1..2: UseTask('ExamBegin', 70 + num);
    3..14: UseTask('ExamTaskC', 22 + num);
  end;
end;

procedure inittaskgroup;
begin
  if (CurrentVersion < '4.10') or
    (CurrentLanguage and (lgPascal or lgCPP) = 0) then
    exit;
  CreateGroup('ExamDemo', '^ЕГЭ по информатике: примеры различных за
    'М. Э. Абрамян, 2013', 'qdfedsag33gbg45j', 14, InitTask);
  CommentText('\РПри выполнении заданий данной группы вместо');
  CommentText('специальных операций ввода-вывода, предоставляемых');
  CommentText('задачником, необходимо применять стандартные операции
  CommentText('используемого языка программирования: процедуры');
  CommentText('\MRead\m\MReadln\m:\Mwrite\m\Mwriteln\m для языка'
  CommentText('Pascal, потоки \Mcin\m\:\Mcout\m для языка C++.'));
end;

procedure activate(S: string);
begin
  ActivateNET(S);
end;

begin
end.

```

Кратко опишем полученную программу. Вначале к ней подключается модуль PT4TaskMakerNET, в котором реализован конструктор учебных заданий для среды PascalABC.NET. Затем следует описание основной процедуры группы заданий [InitTask](#), определяющей задание по его номеру. Поскольку мы не создавали своих заданий, в данной процедуре используется только стандартная процедура [UseTask](#), позволяющая импортировать задания из имеющихся групп. В нашем случае импортируются задания с номерами 71 и 72 из группы ExamBegin и задания с номерами 25–36 из группы ExamTaskC (всего 14 заданий).

Затем описывается процедура инициализации данной группы заданий. Она имеет стандартное имя inittaskgroup (набранное

строчными, т. е. маленькими буквами). В этой процедуре вызывается процедура [CreateGroup](#), в которой задаются характеристики создаваемой группы: имя ('ExamDemo'), описание ('^ЕГЭ по информатике: примеры различных задач'), сведения об авторе, строковый ключ, число заданий (14) и основная процедура группы ([InitTask](#)).

Поскольку надо гарантировать, что группа будет создана только в случае использования задачника версии не ниже 4.10 и только для языков Pascal и C++, перед вызовом процедуры инициализации группы [CreateGroup](#) выполняется проверка перечисленных выше условий. Если хотя бы одно из условий нарушено, то выполняется немедленный выход из процедуры [inittaskgroup](#), и группа ExamDemo не создается.

Следует обратить внимание на наличие символа-метки «^» в начале строки-описания группы. Этот символ отменяет автоматическое преобразование к нижнему регистру первой буквы описания при его выводе в программных модулях PT4Demo и PT4Load (если бы символ «^» отсутствовал, то строка с описанием данной группы имела бы вид «Тема: еГЭ по информатике: примеры различных задач»).

Наконец, в соответствии с правилом 3, мы включили в создаваемую группу *преамбулу*, в которой отмечается основная особенность данной группы: необходимость использования стандартных средств ввода-вывода (для определения текста преамбулы вызываются процедуры [CommentText](#)). Обратите внимание на управляющие последовательности `\M-\m`, позволяющие отобразить фрагмент текста *моноширинным шрифтом*.

После процедуры [inittaskgroup](#) описывается вспомогательная процедура [activate](#) (ее имя также должно быть набрано строчными буквами), в которой необходимо вызвать процедуру [ActivateNET](#), описанную в модуле [PT4TaskMakerNET](#).

Для успешной компиляции программы с созданной группой необходимо, чтобы ей был доступен модуль [PT4TaskMakerNET](#). Этот модуль входит в число стандартных модулей библиотеки системы [PascalABC.NET](#) и размещается в подкаталоге LIB системного

каталога PascalABC.NET, поэтому копировать его в рабочий каталог не требуется.

Для того чтобы при успешной компиляции можно было сразу просмотреть содержимое созданной группы, достаточно использовать вспомогательную тестирующую программу, являющуюся заготовкой для выполнения заданий из созданной группы. Эту программу проще всего создать с помощью программного модуля PT4Load, нажав комбинацию клавиш [Shift]+[Ctrl]+[L] и введя в поле «Задание»; имя первого задания группы: ExamDemo1. В результате будет создан файл с именем ExamDemo1.pas со следующим содержимым:

```
uses PT4Exam;  
  
begin  
    Task( 'ExamDemo1' );  
  
end.
```

Следует обратить внимание на то, что к созданной программе подключается не традиционный модуль задачника PT4, а специальный его вариант PT4Exam, используемый для заданий, связанных с ЕГЭ.

Для того чтобы тестирующая программа отображала задания в демонстрационном режиме, причем при ее запуске на экране отображалось последнее из заданий, включенных в группу, достаточно заменить в процедуре Task номер задания на символ «?»: `Task('ExamDemo?')`.

Теперь при запуске тестирующей программы на экране отобразится окно задачника с заданием ExamDemo14:

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ОБРАБОТКА СЛОЖНЫХ НАБОРОВ ДАННЫХ
Задание: ExamDemo14
Демо-запуск: Иванов Петр
Цвет (F3) Режим (F4)

Новые данные (Space) Предыдущее задание (BS) Следующее задание (Enter) Выход (Esc)

На вход подаются сведения об абитуриентах. В первой строке указывается количество абитуриентов N, каждая из последующих N строк имеет формат <Фамилия> <Номер школы> <Год поступления>

Номер школы содержит не более двух цифр, годы лежат в диапазоне от 1990 до 2010. Найти школы, для которых общее число абитуриентов за все годы было не меньше среднего значения по всем школам (вначале указывать число абитуриентов для данной школы, затем номер школы). При вычислении среднего значения учитывать только школы, присутствующие в исходных данных; среднее значение может иметь ненулевую дробную часть. Сведения о каждой школе выводить на новой строке и упорядочивать по возрастанию числа абитуриентов, а для одинаковых чисел – по убыванию номера школы.

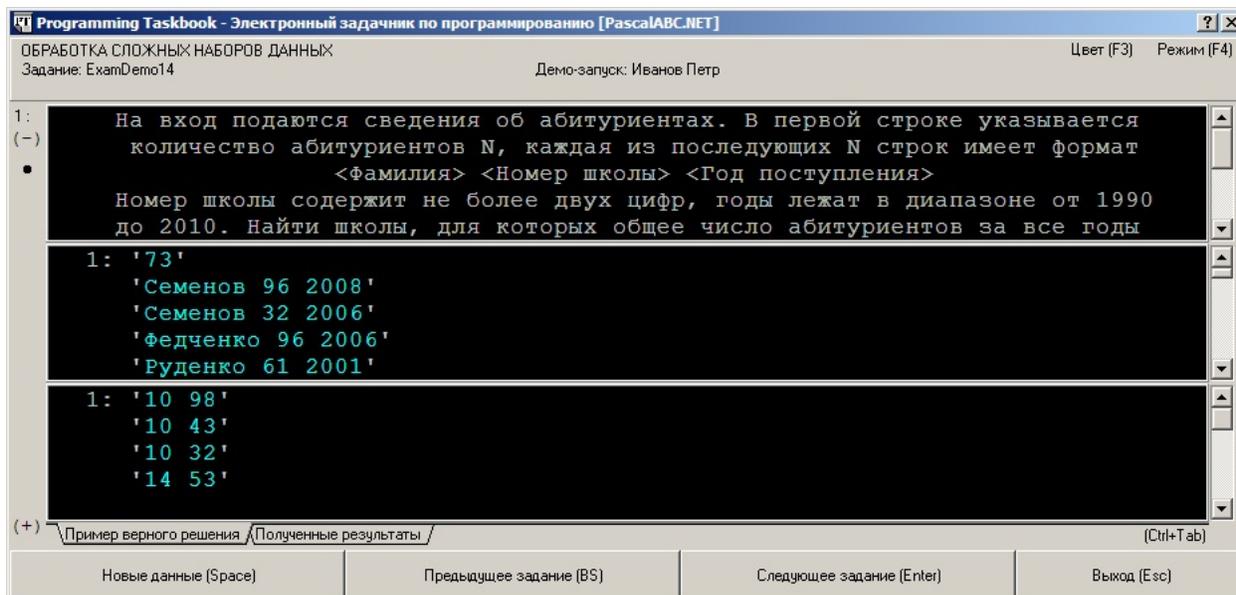
Исходные данные

```
1: '73'  
   'Семенов 96 2008'  
   'Семенов 32 2006'  
   'Федченко 96 2006'  
   'Руденко 61 2001'
```

Пример верного решения

```
1: '10 98'  
   '10 43'  
   '10 32'  
   '14 53'
```

По умолчанию окно задачника отображается в режиме с динамической компоновкой, который появился в версии 4.11 и является более наглядным, чем режим с фиксированной компоновкой. Однако при разработке заданий желательно применять режим с фиксированной компоновкой, поскольку он позволит выявить недостатки форматирования (в частности, вертикального выравнивания данных), присущие только этому режиму. Для переключения между режимами отображения данных достаточно нажать клавишу [F4]. После выполнения этого действия окно задачника изменится следующим образом:



В окне задачника можно просматривать все имеющиеся задания данной группы (нажимая клавиши [Enter] и [Backspace]), а также генерировать различные варианты исходных данных и связанных с ними контрольных (т. е. «правильных») результатов. При закрытии окна программа немедленно завершит работу, и мы вернемся в редактор среды PascalABC.NET. Заметим, что при последующих запусках программы будет автоматически выбираться тот режим окна задачника, в котором оно находилось в момент его предшествующего закрытия.

Для того чтобы сгенерировать html-страницу с описанием созданной группы (это позволяет, в частности, увидеть текст преамбулы группы), достаточно в процедуре Task тестирующей программы заменить символ «?» на символ «#»: `Task('ExamDemo#')`. Теперь при запуске этой программы на экране вместо окна задачника с заданием ExamDemo14 появится html-браузер с описанием созданной группы:

ЕГЭ по информатике: примеры различных задач

М. Э. Абрамян, 2013

При выполнении заданий данной группы вместо специальных операций ввода-вывода, предоставляемых задачником, необходимо применять стандартные операции используемого языка программирования: процедуры Read/Readln-Write/Writeln для языка Pascal, потоки cin-cout для языка C++.

Преобразование массивов

ExamDemo1°. На вход в первой строке подается целое положительное число N , а во второй строке — массив из N целых чисел. Поменять порядок следования элементов массива на обратный. Вывести преобразованный массив в одной строке, для каждого элемента отводить 5 экранных позиций.

ExamDemo2. На вход в первой строке подаются целые положительные числа K_1 , K_2 и N ($K_1 \leq K_2 \leq N$), а во второй строке — массив из N вещественных чисел. Поменять в массиве порядок следования элементов с номерами от K_1 до K_2 включительно на обратный (элементы нумеруются от 1). Вывести преобразованный массив в одной строке, для каждого элемента отводить 7 экранных позиций.

Обработка сложных наборов данных

ExamDemo3°. На вход подаются сведения об абитуриентах. В первой строке указывается количество абитуриентов N , каждая из последующих N строк имеет формат

<Номер школы> <Год поступления> <фамилия>

Номер школы содержит не более двух цифр, годы лежат в диапазоне от 1990 до 2010. Для каждого года, присутствующего в исходных данных, вывести общее число абитуриентов, поступивших в этом году (вначале выводить год, затем число абитуриентов). Сведения о каждом году выводить на новой строке и упорядочивать по возрастанию номера года.

ExamDemo4. На вход подаются сведения об абитуриентах. В первой строке указывается количество абитуриентов N , каждая из последующих N строк имеет формат

<Год поступления> <Номер школы> <фамилия>

Номер школы содержит не более двух цифр, годы лежат в диапазоне от 1990 до 2010. Определить, в какие годы общее число абитуриентов для всех школ было наибольшим, и вывести это число, а также количество таких лет. Каждое число выводить на новой строке.

ExamDemo5. На вход подаются сведения об абитуриентах. В первой строке указывается количество

Добавление новых заданий

Добавим к нашей группе новые задания. Подобно заданиям, импортированным из группы ExamBegin, они будут посвящены преобразованию массивов путем перестановки их элементов. Если импортированные задания были посвящены инвертированию массива (или его части), то в новых заданиях надо будет выполнить перестановку всех пар элементов или перестановку первой и второй половины массива. Чтобы не уточнять действия в случае массивов нечетного размера, добавим в задание условие о том, что исходный массив всегда имеет четный размер. Тип элементов массива для подобных заданий является несущественным, поэтому будем обрабатывать массивы вещественных чисел. Таким образом, набор исходных данных будет иметь вид, подобный набору из задания ExamDemo2 (см. формулировку этого задания, приведенную на предыдущем рисунке). Оформление вывода результатов также не будет отличаться от оформления, требуемого в задании ExamDemo2.

Прежде чем приступить к реализации заданий, добавим в начало нашей библиотеки описание ряда вспомогательных переменных, процедур и функций.

Первая функция упрощает генерацию случайных исходных данных вещественного типа с не более чем двумя дробными знаками:

```
function RandR(a, b: integer): real;
begin
  result := RandomN(a*100, b*100)/100;
end;
```

При реализации функции RandR мы использовали функцию [RandomN](#), входящую в состав конструктора учебных заданий, начиная с версии 4.11 (функция RandomN(M, N) возвращает случайное целое число, лежащее в диапазоне M..N, включая границы диапазона). Функция RandR(a, b) (a и b — целые) возвращает вещественное число, лежащее в диапазоне a..b и *имеющее не более двух дробных знаков*. Такие числа можно без потери точности записывать в текстовый файл в формате с двумя дробными знаками; таким образом, программа учащегося прочтет из

файла именно то число, которое было сгенерировано при инициализации задания. Напомним, что в версии 4.11 конструктора имеется функция RandomR, также предназначенная для генерации случайных вещественных чисел, однако она не позволяет фиксировать число дробных знаков, и поэтому менее пригодна для генерации данных, предназначенных для записи в текстовые файлы.

Поскольку в обоих заданиях нам потребуется выполнять обмен значений, содержащихся в вещественных переменных, опишем процедуру, которая выполняет подобный обмен:

```
procedure SwapR(var a, b: real);
var
  c: real;
begin
  c := a;
  a := b;
  b := c;
end;
```

Следующая группа вспомогательных переменных и процедур связана с особенностью заданий типа Exam, описанной в правиле 5 (см. первый пункт данного раздела). Ввиду важности этого правила приведем его еще раз:

- Набор исходных и контрольных данных надо сохранять в текстовых файлах, передавая задачнику информацию об именах этих файлов (процедурами DataS) и связывая содержимое этих файлов с разделами исходных и результирующих данных (процедурами DataFileT и ResultFileT соответственно).

Таким образом, при инициализации каждого задания группы Exam надо выполнить следующие действия:

1. Сгенерировать имена файлов, содержащих исходные и контрольные данные (эти имена должны быть различными и меняться при каждом тестовом испытании программы; кроме того, подобно всем файлам, используемым в заданиях, они должны иметь расширение .tst).
2. Связать созданные имена с файловыми переменными и открыть эти файлы на запись.

3. Заполнить файлы необходимыми данными.
4. Закрывать файлы с исходными и контрольными данными.
5. Передать задачнику информацию об именах созданных файлов, чтобы при выполнении задания эта информация была использована при связывании файлов со стандартными потоками ввода-вывода.
6. Передать задачнику информацию о том, что первый из созданных файлов должен быть включен в раздел исходных данных, а второй — в раздел результатов; это, во-первых, позволит отобразить содержимое файлов в окне задачника и, во-вторых, обеспечит проверку правильности результирующего файла, созданного программой учащегося (путем его сравнения с данными контрольного файла).

От условий конкретного задания будет зависеть только действие 3, связанное с заполнением файлов нужными данными. Все остальные действия являются стандартными и должны выполняться при инициализации любого задания групп Exam. Поэтому удобно оформить эти действия в виде двух вспомогательных процедур, одна из которых (StartExam) выполняет начальные действия 1–2, а другая (EndExam) — завершающие действия 4–6. Поскольку в каждой из этих процедур необходимо использовать имена созданных файлов и связанные с ними файловые переменные, эти переменные удобно описать как глобальные:

```
var
  f1, f2: text;
  f1name, f2name: string;

procedure StartExam;
var
  s: string;
begin
  Str(RandomN(10000, 99999), s);
  f1name := 'pt1' + s + '.tst';
  f2name := 'pt2' + s + '.tst';
  Assign(f1, f1name);
  Rewrite(f1);
  Assign(f2, f2name);
  Rewrite(f2);
end;
```

```
procedure EndExam;  
begin  
  Close(f1);  
  Close(f2);  
  DataS(f1name, 3, 1);  
  DataS(f2name, 45, 1);  
  DataFileT(f1name, 1, 5);  
  ResultFileT(f2name, 1, 5);  
end;
```

Обсудим особенности этих процедур. Имена файлов, создаваемых в процедуре StartExam, имеют вид pt1#####.tst (для файла с исходными данными) и pt2#####.tst (для файла с контрольными данными), причем в позициях, помеченных символом «#», располагаются цифры, выбираемые случайным образом. Тем самым обеспечиваются все требования к именам файлов: они генерируются случайным образом, имеют расширение .tst, и имя файла с исходными данными всегда отличается от имени контрольного файла. Напомним, что все файлы с расширением .tst автоматически удаляются из рабочего каталога после проверки учебного задания.

При анализе процедуры EndExam следует обратить внимание на то, что информация о содержимом исходного файла занимает всю область исходных данных (строки с первой по пятую — см. вызов процедуры [DataFileT](#)) и, таким образом, она *скрывает* информацию об именах файлов, ранее выведенную в первой строке области исходных данных (см. вызовы процедур [DataS](#)). В обычном задании такая реализация была бы ошибочной, поскольку учащийся не увидел бы на экране имена файлов и не понял бы, что эти имена необходимо ввести и обработать в его программе. Однако в задании групп Exam именно такая реализация является правильной, поскольку ввод имен файлов и связывание этих файлов со стандартными потоками ввода-вывода выполняется автоматически («незаметно» для программы учащегося), и поэтому информацию об именах файлов на экране отображать не следует.

Итак, наличие процедур StartExam и EndExam позволяет нам упростить реализацию заданий: после определения формулировки любого задания нам достаточно вызвать процедуру StartExam, заполнить файлы f1 и f2 исходными и, соответственно,

контрольными данными и вызвать процедуру EndExam.

Приступим к непосредственной реализации заданий. Поскольку эти задания являются однотипными, реализуем их в одной процедуре Exam1, снабдив ее параметром m: при $m = 1$ будет инициализироваться первое задание, а при $m = 2$ — второе:

```
procedure Exam1(m: integer);
var
  n, i: integer;
  a: array[1..10] of real;
begin
  CreateTask('Преобразование массивов');
  case m of
    1:
      begin
        TaskText('На вход в первой строке подается целое положительное ч
        TaskText('а во второй строке \= массив из {N} вещественных чисел
        TaskText('его первый элемент со вторым, третий с четвертым, и т.
        TaskText('преобразованный массив в одной строке, для каждого эле
        TaskText('отводить 7 экранных позиций.', 0, 5);
      end;
    2:
      begin
        TaskText('На вход в первой строке подается целое положительное ч
        TaskText('а во второй строке \= массив из {N} вещественных чисел
        TaskText('первую и вторую половину элементов массива. Вывести пр
        TaskText('в одной строке, для каждого элемента отводить 7 экранн
      end;
  end;
  StartExam;
  n := 2 * RandomN(1, 5);
  for i := 1 to n do
    a[i] := RandR(-99, 99);
  writeln(f1, n);
  for i := 1 to n - 1 do
    write(f1, a[i]:0:2, ' ');
  writeln(f1, a[n]:0:2);
  for i := 1 to n div 2 do
    case m of
      1: SwapR(a[2*i - 1], a[2*i]);
      2: SwapR(a[i], a[i + n div 2]);
    end;
  for i := 1 to n do
    write(f2, a[i]:7:2);
  writeln(f2);
```

```
EndExam;  
SetTestCount(3);  
end;
```

Обратите внимание на то, что при вызове процедуры CreateTask ей передается строковый параметр, содержащий имя подгруппы «Преобразование массивов». Это обеспечивает включение новых заданий в подгруппу, с которой связаны ранее импортированные в нашу группу задания ExamBegin71 и ExamBegin72.

Размер исходного массива всегда будет четным и не превосходящим 10; последнее условие необходимо для того, чтобы все исходные данные можно было разместить на одной экранной строке.

В обоих заданиях значения элементов исходного массива можно выбирать произвольным образом из некоторого диапазона. Мы выбрали диапазон от -99 до 99, поскольку в этом случае при отображении чисел с двумя дробными знаками они будут занимать не более 6 экранных позиций.

При записи в файл элементов исходного массива между ними всегда располагается по одному пробелу, поскольку такой порядок организации исходных данных принят во всех заданиях групп ExamBegin и ExamTaskC. Чтобы обеспечить при этом отображение вещественных чисел с двумя дробными знаками, используется специальный набор форматирующих атрибутов: «:0:2». При выводе результатов, согласно формулировке задания, необходимо отводить для каждого элемента массива по 7 экранных позиций и выводить его с двумя дробными знаками (последнее условие принято по умолчанию во всех заданиях групп ExamBegin и ExamTaskC, использующих вещественные данные). Поэтому при выводе применяются другие форматирующие атрибуты: «:7:2».

Так как алгоритм решения обеих задач не содержит ветвлений, для проверки его правильности достаточно небольшого числа тестовых запусков. Мы установили это число равным трем, указав его в качестве параметра процедуры [SetTestCount](#).

Нам осталось включить вызовы процедуры Exam1 (с параметрами, равными 1 и 2) в основную процедуру группы InitTask, связав эти вызовы с номерами заданий. Следует разместить новые задания

сразу после импортированных заданий ExamBegin71 и ExamBegin72, так как все эти задания относятся к одной и той же подгруппе «Преобразование массивов». При этом номера последних 12 заданий увеличатся на 2:

```
procedure InitTask(num: integer);
begin
  case num of
    1..2:  UseTask('ExamBegin', 70 + num);
    3..4:  Exam1(num - 2);
    5..16: UseTask('ExamTaskC', 20 + num);
  end;
end;
```

Необходимо также увеличить на 2 пятый параметр процедуры CreateGroup, определяющий общее количество заданий в группе (теперь это количество должно быть равно 16).

Для просмотра новых заданий в окне задачника надо заменить в параметре процедуры Task тестирующей программы символ «#» на «?»: `Task('ExamDemo?')`.

При нажатии клавиши [F9] мы увидим на экране окно задачника в демо-режиме, в котором можно выбрать и просмотреть все задания, включенные к настоящему моменту в нашу группу. Приведем вид окна для задания ExamDemo4 (напомним, что это задание инициализируется посредством вызова процедуры Exam1 с параметром, равным 2):

Programming Taskbook - Электронный задачник по программированию [PascalABC.NET] ? X

ПРЕОБРАЗОВАНИЕ МАССИВОВ Цвет (F3) Режим (F4)
Задание: ExamDemo4* Демо-запуск: Иванов Петр

На вход в первой строке подается целое положительное четное число N, а во второй строке – массив из N вещественных чисел. Поменять местами первую и вторую половину элементов массива. Вывести преобразованный массив в одной строке, для каждого элемента отводить 7 экранных позиций.

(-)

- 1: '10'
'-43.23 16.74 -33.74 90.48 17.75 -16.60 -93.33 12.30 -60.96 -11.22'

1: ' -16.60 -93.33 12.30 -60.96 -11.22 -43.23 16.74 -33.74 90.48 17.75'

(+) (Ctrl+Tab)

Новые данные (Space)	Предыдущее задание (BS)	Следующее задание (Enter)	Выход (Esc)
----------------------	-------------------------	---------------------------	-------------

Добавление заданий повышенной сложности

Наша группа ExamDemo к настоящему моменту содержит 12 заданий повышенной сложности, импортированных из группы ExamTaskC. Все эти задания связаны с общей предметной областью; они содержат сведения об абитуриентах и включают их фамилии, номера школ и годы поступления в вузы. Для того чтобы проиллюстрировать некоторые особенности, связанные с разработкой подобных заданий, дополним набор уже имеющихся заданий двумя новыми заданиями из той же предметной области.

Новые задания будут связаны с группировкой абитуриентов по школам: для каждой школы надо найти связанный с ней минимальный (или максимальный) год поступления абитуриента. Второе из двух заданий мы усложним, дополнительно потребовав, чтобы полученные результаты были отсортированы по убыванию максимального года (а для одинаковых годов — по возрастанию номера школы). Первое задание сделаем более простым: в нем результирующие данные надо располагать по возрастанию номеров школ.

При генерации наборов исходных данных нам потребуются не только числа (номера школ и годы поступления), но и строковые данные — фамилии абитуриентов (хотя для выполнения этих заданий они не требуются). Проще всего определить массив возможных фамилий достаточно большого размера, из которого выбирать элементы случайным образом. Заметим, что в условии заданий не говорится о том, что все фамилии в исходном наборе должны быть различными, поэтому совпадения фамилий вполне допустимы (если в некоторой группе заданий все фамилии должны быть уникальными, то целесообразно дополнять их *инициалами*, чтобы обеспечить большее разнообразие; кроме того, для таких заданий при добавлении к набору исходных данных новой фамилии необходимо проверять, что среди уже имеющихся элементов набора отсутствует данная фамилия с теми же инициалами).

Добавим к нашей библиотеке вспомогательный массив фамилий из 40 элементов (обратите внимание на то, что по правилам языка PascalABC.NET между описанием массива и списком

инициализирующих значений указывается знак присваивания):

```
const
  famcount = 40;
var
  fam: array[1..famcount] of string :=
  ('Иванов', 'Петров', 'Сидоров', 'Кузнецов', 'Филиппов',
  'Сергеев', 'Александров', 'Петухов', 'Пономарев', 'Яшин',
  'Греков', 'Иванова', 'Кузнецова', 'Алексеева', 'Зайцев',
  'Волкова', 'Фролов', 'Юрьев', 'Бондарев', 'Семенов',
  'Семенова', 'Федченко', 'Марченко', 'Борисова', 'Петровский',
  'Беляева', 'Белкин', 'Лысенко', 'Сорокина', 'Пастухов',
  'Юрьева', 'Кондратьев', 'Тимофеев', 'Степанова', 'Якимов',
  'Юсов', 'Степанов', 'Руденко', 'Демидов', 'Леонидов');
```

Оба новых задания, как и два предыдущих, мы реализуем в виде одной процедуры с параметром *m*, принимающим значения 1 или 2:

```
procedure Exam2(m: integer);
var
  n, i, y, num, max, k: integer;
  a: array[1..100] of integer;
  nums: array[1..10] of integer;
begin
  CreateTask('Обработка сложных наборов данных');
  case m of
    1:
      begin
        TaskText('На вход подаются сведения об абитуриентах. В первой ст
        TaskText('количество абитуриентов {N}, каждая из последующих {N}
        TaskText('\(\M<Год поступления> <Фамилия> <Номер школы>\m\)', 0, 3
        TaskText('Номер школы содержит не более двух цифр, годы лежат в
        TaskText('до 2010. Для каждого номера школы, присутствующего в и
        TaskText('определить связанный с ним минимальный год поступления
        TaskText('номер школы, затем минимальный год). Сведения о каждой
        TaskText('на новой строке и упорядочивать по возрастанию номера
        k := 1;
        for i := 1 to 100 do
          a[i] := 2100;
        end;
      2:
      begin
        TaskText('На вход подаются сведения об абитуриентах. В первой ст
        TaskText('количество абитуриентов {N}, каждая из последующих {N}
        TaskText('\(\M<Номер школы> <Фамилия> <Год поступления>\m\)', 0, 3
        TaskText('Номер школы содержит не более двух цифр, годы лежат в
        TaskText('до 2010. Для каждого номера школы, присутствующего в и
```

```

TaskText('определить связанный с ним максимальный год поступления
TaskText('максимальный год, затем номер школы). Сведения о каждо
TaskText('на новой строке и упорядочивать по убыванию максимальн
TaskText('а для совпадающих годов \= по возрастанию номера школь
k := -1;
for i := 1 to 100 do
  a[i] := 0;
end;
end;
StartExam;
if Random(2)=0 then
  n := RandomN(50, 100)
else
  n := RandomN(10, 20);
case CurrentTest of
1: n := RandomN(10, 20);
2: n := RandomN(50, 100);
end;
if n <= 20 then
  for i := 1 to 10 do
    nums[i] := RandomN(1, 100);
writeln(f1,n);
for i := 1 to n do
begin
  y := RandomN(1990, 2010);
  if n <= 20 then
    num := nums[RandomN(1, 10)]
  else
    num := RandomN(1, 100);
  case m of
1: writeln(f1, y, ' ', fam[RandomN(1, famcount)], ' ', num);
2: writeln(f1, num, ' ', fam[RandomN(1, famcount)], ' ', y);
end;
  if k*a[num] > k*y then
    a[num] := y;
end;
case m of
1: for i := 1 to 100 do
  if a[i] < 2100 then
    writeln(f2, i, ' ', a[i]);
2: while true do
  begin
    max := 0;
    for i := 1 to 100 do
      if a[i] > max then
        begin
          max := a[i];

```

```

        num := i;
    end;
    if max = 0 then
        break
    else
        begin
            writeln(f2, max, ' ', num);
            a[num] := 0;
        end;
    end;
end;
EndExam;
SetTestCount(5);
end;

```

Обсудим детали реализации этих заданий. Начальная часть их формулировки посвящена описанию предметной области и является стандартной для данной серии заданий. Обратите внимание на то, что поля исходных записей в заданиях указываются в различном порядке. Этот прием используется во всех сериях группы ExamTaskC, чтобы обеспечить большее разнообразие входящих в них задач.

При определении завершающей части формулировки заданий последний параметр процедур TaskText полагается равным 0. Это означает, что в режиме окна с фиксированной компоновкой данные строки при первоначальном отображении задания не видны на экране, однако их можно просмотреть, используя *прокрутку* раздела с формулировкой задания. В режиме с динамической компоновкой полный текст формулировки задания сразу отображается на экране.

В процедуре используются два массива: массив a предназначен для хранения контрольных (правильных) результатов, а массив numS является вспомогательным (его назначение описывается далее). Для хранения исходных данных массив не предусматривается, поскольку после генерации полей очередной записи они будут немедленно записываться в исходный файл и обрабатываться.

Количество записей в исходном наборе данных записывается в переменную n. Наборы, содержащие небольшое число записей, удобны при отладке программы (благодаря своей «обозримости»), в то время как большие наборы позволяют проверить программу в

«реальной» ситуации и тем самым окончательно убедиться в правильности алгоритма. Используя функцию [CurrentTest](#), добавленную в версию 4.11 конструктора PT4TaskMaker, мы обеспечили дополнительную «настройку» процесса генерации исходных данных: при первом тестовом запуске программы с решением задачи ей всегда предлагается набор из небольшого количества записей (что упрощает поиск и исправление ошибок), а при втором тестовом запуске — большой набор записей (что позволяет проверить предложенный алгоритм «на прочность»). При последующих тестовых запусках (а также при демонстрационном и ознакомительном запуске программы) значение n с равной вероятностью выбирается либо из диапазона 10..20, либо из диапазона 50..100.

В случае генерации исходных данных для указанных заданий при небольших значениях n (10–20) возникает дополнительная проблема: если выбирать случайным образом номера школ из всего допустимого диапазона 1–100, то с большой вероятностью каждый номер школы появится в наборе исходных данных всего по одному разу, что не позволит проверить правильность реализованного в программе алгоритма нахождения минимального/максимального значения. Чтобы решить эту проблему, используется вспомогательный массив `numts` из 10 элементов, в который заносятся 10 случайно выбранных номеров школ, после чего номера школ для исходного набора записей выбираются уже из этого набора номеров.

В любом задании, связанном с нахождением набора записей, обычно требуется *отсортировать* полученный набор. Задание должно быть сформулировано таким образом, чтобы обеспечить *однозначный* порядок вывода полученных данных. В частности, если поле, по которому выполняется сортировка (*главный ключ сортировки*), может содержать одинаковые значения, то обязательно следует указать дополнительное поле (*подчиненный ключ сортировки*), по которому надо сортировать записи с одинаковым главным ключом. При выводе отсортированных данных вначале надо располагать главный ключ, после него — подчиненные ключи (если они имеются), затем — остальные поля (такой порядок вывода принят во всех заданиях группы ExamTaskC).

Для упорядочивания результатов во втором задании вместо сортировки массива *a* по убыванию используется другой алгоритм, связанный с последовательным нахождением максимального элемента, выводом этого элемента и его «порчей» (заменой его значения на 0). Обычная сортировка массива в данном случае не позволит получить требуемый набор данных, так как при перемене местами значений элементов в массиве *a* будет потеряна связь с номером школы (который определяется по индексу элемента). Заметим, что возможен и вариант получения упорядоченного набора данных с помощью сортировки, однако для этого надо использовать массив *записей*, полями которых являются максимальный год и номер школы.

Во втором задании результаты должны упорядочиваться по *набору ключей*: первый (главный) ключ — максимальный год (сортируется по убыванию), второй (подчиненный) ключ — номер школы (сортируется по возрастанию). Используемый нами способ упорядочивания обеспечивает *автоматическую* сортировку по подчиненному ключу, так как при поиске очередного максимума массив *a* перебирается по *возрастанию* индексов, и поэтому в результате находится номер *первого* максимального элемента.

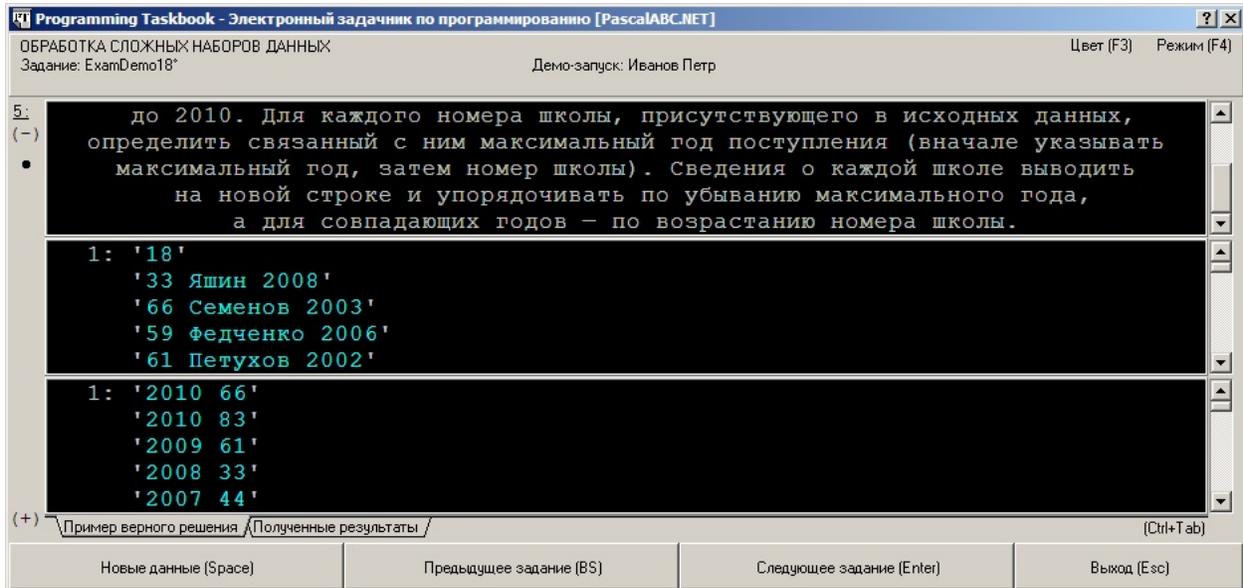
Прочие фрагменты процедуры Exam2 дополнительных комментариев не требуют.

Вызов процедуры Exam2 надо добавить в конец оператора case процедуры InitTask, связав его с номерами 17 и 18:

```
procedure InitTask(num: integer);
begin
  case num of
    1..2:  UseTask('ExamBegin', 70 + num);
    3..4:  Exam1(num - 2);
    5..16: UseTask('ExamTaskC', 20 + num);
    17..18: Exam2(num - 16);
  end;
end;
```

Кроме того, необходимо опять откорректировать параметр процедуры CreateGroup, определяющий количество заданий, положив его равным 18.

При нажатии клавиши [F9] на экране появится окно задачника с последним заданием данной группы (на рисунке приведен вид окна после прокрутки раздела с формулировкой задания):



Если теперь опять заменить символ «?» на символ «#» в параметре процедуры Task тестирующей программы, то при нажатии [F9] мы увидим html-страницу с описанием группы, которое теперь содержит не только импортированные, но и реализованные нами задания:

ЕГЭ по информатике: примеры различных задач

М. Э. Абрамян, 2013

При выполнении заданий данной группы вместо специальных операций ввода-вывода, предоставляемых задачиком, необходимо применять стандартные операции используемого языка программирования: процедуры Read/Readln-Write/Writeln для языка Pascal, потоки cin-cout для языка C++.

Преобразование массивов

ExamDemo1°. На вход в первой строке подается целое положительное число N , а во второй строке — массив из N целых чисел. Поменять порядок следования элементов массива на обратный. Вывести преобразованный массив в одной строке, для каждого элемента отводить 5 экранных позиций.

ExamDemo2°. На вход в первой строке подаются целые положительные числа K_1 , K_2 и N ($K_1 \leq K_2 \leq N$), а во второй строке — массив из N вещественных чисел. Поменять в массиве порядок следования элементов с номерами от K_1 до K_2 включительно на обратный (элементы нумеруются от 1). Вывести преобразованный массив в одной строке, для каждого элемента отводить 7 экранных позиций.

ExamDemo3°. На вход в первой строке подается целое положительное четное число N , а во второй строке — массив из N вещественных чисел. Поменять местами его первый элемент со вторым, третий с четвертым, и т. д. Вывести преобразованный массив в одной строке, для каждого элемента отводить 7 экранных позиций.

ExamDemo4°. На вход в первой строке подается целое положительное четное число N , а во второй строке — массив из N вещественных чисел. Поменять местами первую и вторую половину элементов массива. Вывести преобразованный массив в одной строке, для каждого элемента отводить 7 экранных позиций.

Обработка сложных наборов данных

ExamDemo5°. На вход подаются сведения об абитуриентах. В первой строке указывается количество абитуриентов N , каждая из последующих N строк имеет формат

<Номер школы> <Год поступления> <фамилия>

Номер школы содержит не более двух цифр, годы лежат в диапазоне от 1990 до 2010. Для каждого года, присутствующего в исходных данных, вывести общее число абитуриентов, поступивших в этом году (вначале выводить год, затем число абитуриентов). Сведения о каждом году выводить на новой строке и упорядочивать по возрастанию номера года.

Поскольку при создании новых заданий мы указали в качестве параметра процедур CreateTask названия подгрупп (и разместили новые задания после заданий из данных подгрупп), новые задания отображаются в составе этих подгрупп: ExamDemo3 и ExamDemo4 — в подгруппе «Преобразование массивов», а ExamDemo17 и ExamDemo18 — в подгруппе «Обработка сложных наборов данных».

ABCObjects: быстрое введение

Основными типами графических объектов, определенными в модуле `ABCObjects`, являются `RectangleABC`, `SquareABC`, `EllipseABC`, `CircleABC`, `TextABC`, `RegularPolygonABC`, `StarABC`, `PictureABC`, `MultiPictureABC`, `BoardABC` и `ContainerABC`.

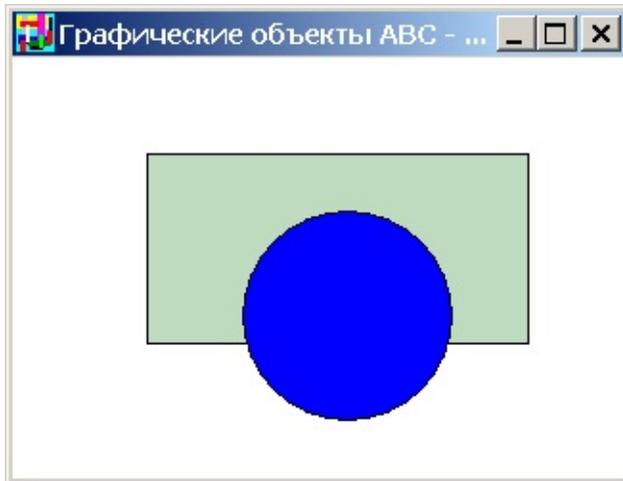
Типы графических объектов представляют собой классы, состоящие из методов и свойств, а также нуждающиеся в конструировании перед первым использованием. Изменение свойств влияет на внешний вид и поведение графических объектов. Например, при изменении свойств `Width` и `Height` меняются размеры графического объекта, при изменении свойства `Color` - цвет графического объекта и т.д. Вызов методов графического объекта возвращает или меняет его характеристики. Например, при вызове метода `ToFront` графический объект перемещается на передний план, а вызов метода `Intersect(g)` возвращает, пересекается ли текущий объект с объектом `g`.

Все графические объекты являются разновидностями класса `ObjectABC`, который содержит общие для всех свойства и методы.

Создадим два перекрывающихся графических объекта:

```
uses ABCObjects, GraphABC;  
var   r: RectangleABC;  
      c: CircleABC;  
begin  
  r := new RectangleABC(70, 50, 200, 100, clMoneyGreen);  
  c := new CircleABC(120, 80, 110, clBlue);  
end.
```

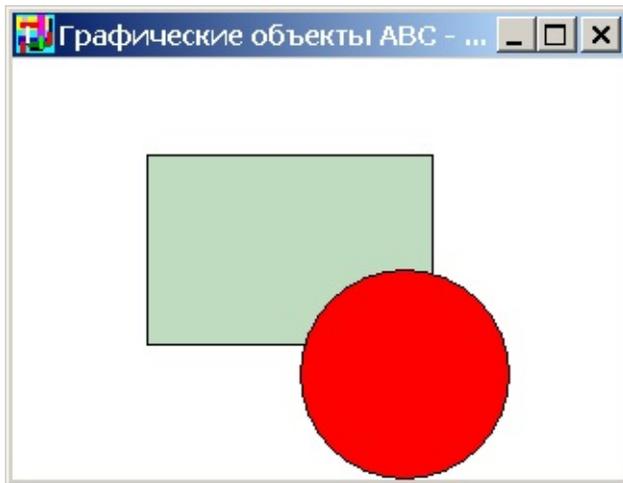
После запуска программы увидим на экране следующее:



Поменяем некоторые свойства графических объектов и вызовем метод `MoveOn` для окружности, дописав в конец программы следующие строки:

```
r.Width := 150;  
c.Color := clRed;  
c.MoveOn(30, 30);
```

После запуска программы:



Добавим в конец программы следующие строки:

```
c.Number := 8;  
r.Text := 'Hello';  
r.ToFront;
```

После запуска программы:



ABCObjects: контейнеры графических объектов

Класс `ContainerABC` представляет собой контейнер графических объектов. Он также является потомком `ObjectABC`, но при создании не содержит ни одного объекта. Он добавляет следующий интерфейс:

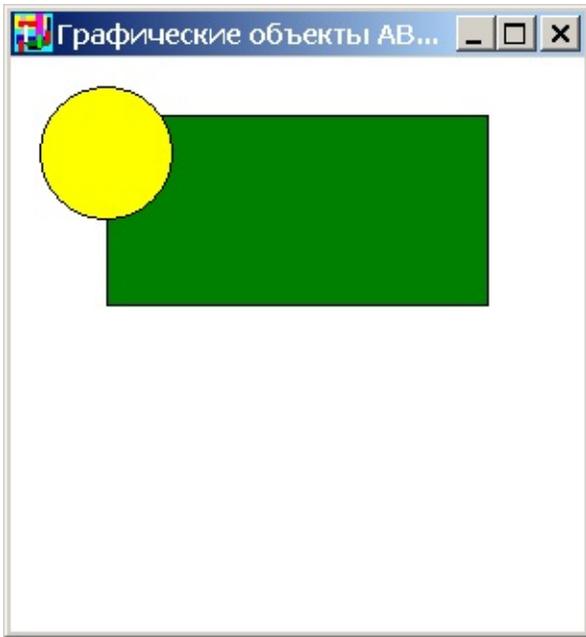
```
procedure Add(g: ObjectABC);  
property Count: integer; // количество объектов  
property Objects[i: integer]: ObjectABC; // i-тый  
объект
```

При масштабировании `ContainerABC` производится масштабирование всех входящих в него объектов. При добавлении объекта в `ContainerABC` его свойство `Owner` становится равным этому `ContainerABC`. При присваивании свойству `Owner` объекта его владелец меняется, при этом объект перерисовывается как принадлежащий новому владельцу. При присваивании свойству `Owner` объекта значения `nil` он перестает иметь владельца и отображается непосредственно в графическом окне.

Рассмотрим следующую программу:

```
uses ABCObjects, GraphABC;  
var  
  c1, c2: ContainerABC;  
  r: CircleABC;  
begin  
  SetWindowSize(300, 300);  
  c1 := new ContainerABC(50, 30);  
  c1.Add(new RectangleABC(0, 0, 200, 100, clGreen));  
  r := new CircleABC(15, 15, 70, clYellow);  
end.
```

После ее запуска графический экран имеет вид:



Контейнер `c1` содержит зеленый прямоугольник, а объект `r` не имеет владельца (`r.Owner=nil`). Нетрудно убедиться, что `ObjectsCount=2` (контейнер и круг), а `c1.Count=1`.

Добавим круг в контейнер, дописав в конец программы строчку
`c1.Add(r);`

После запуска программы графический экран примет вид:



Круг `r` теперь принадлежит контейнеру (`r.Owner=c2`),

`ObjectsCount=1` (только контейнер), а `c1.Count=2`. Кроме этого, координаты круга пересчитываются относительно координат контейнера-владельца (они по-прежнему равны (15,15), но относительно левого верхнего угла контейнера `c1`).

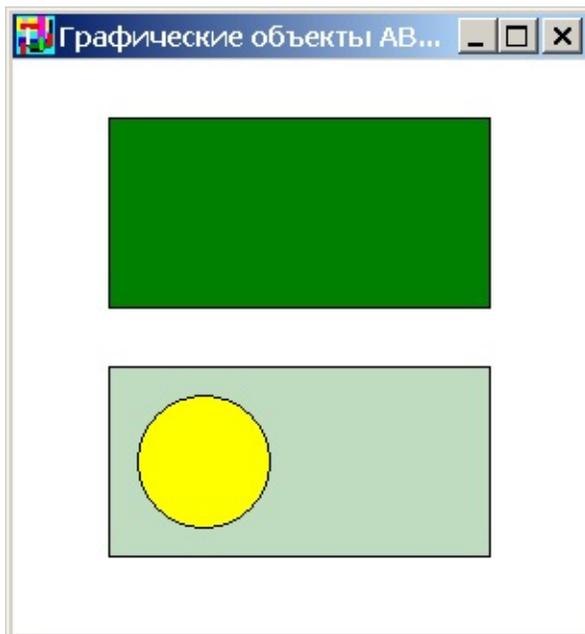
Такой же эффект можно было получить от оператора

```
r.Owner := c1;
```

Создадим второй контейнер `c2` и поменяем владельца у `r` на `c2`. Для этого допишем в конец строки:

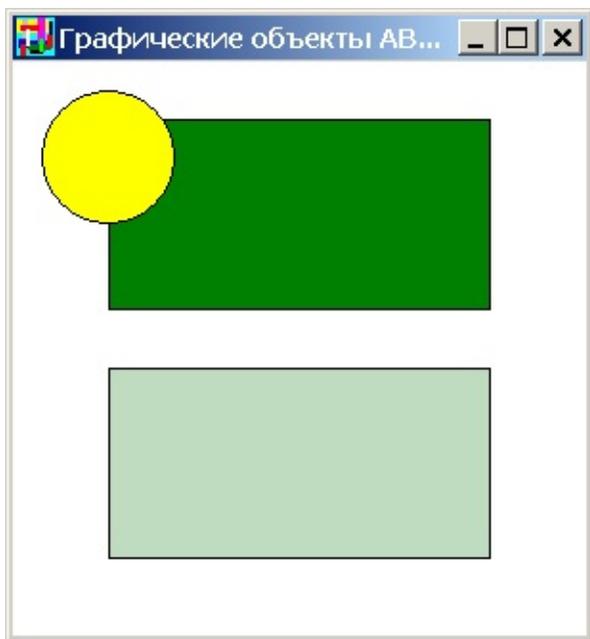
```
c2 := new ContainerABC(50,160);  
c2.Add(new RectangleABC(0,0,200,100,c1MoneyGreen));  
r.Owner := c2;
```

После запуска программы графический экран примет вид:



Как мы видим, круг `r` поменял владельца, и теперь имеет координаты (15,15), но относительно левого верхнего угла нового владельца `c2`.

Если вместо строчки `r.Owner:=c2`; написать `r.Owner:=nil`; , то круг `r` потеряет владельца и снова будет позиционироваться относительно левого верхнего угла экрана:



Анимация без мерцания

Данная программа иллюстрирует применение процедур `LockDrawing` и `Redraw` для реализации анимации без мерцания:

```
uses GraphABC;  
begin  
  LockDrawing;  
  for var i:=1 to 500 do  
    begin  
      Window.Clear;  
      Brush.Color := clGreen;  
      Ellipse(i,100,i+100,200);  
      Redraw;  
      Sleep(1);  
    end;  
  end.  
end.
```

Основная идея состоит в следующем: отключим рисование на экране, вызвав `LockDrawing` (рисование будет осуществляться только во внеэкранный буфере), после чего будем всякий раз формировать новый кадр изображения и выводить его целиком на экран, вызывая `Redraw`. При вызове `Redraw` перерисовывается все графическое окно, поэтому скорость анимации ограничена скоростью вывода внеэкранный буфера на экран.

Рисование мышью в графическом окне

Данная программа осуществляет рисование мышью в графическом окне:

```
uses GraphABC;  
  
procedure MouseDown(x,y,mb: integer);  
begin  
    MoveTo(x,y);  
end;  
  
procedure MouseMove(x,y,mb: integer);  
begin  
    if mb=1 then LineTo(x,y);  
end;  
  
begin  
    // Привязка обработчиков к событиям  
    OnMouseDown := MouseDown;  
    OnMouseMove := MouseMove  
end.
```

Перемещение окна с помощью клавиатуры

Данная программа осуществляет перемещение графического окна с помощью клавиатуры:

```
uses GraphABC;  
  
procedure KeyDown(Key: integer);  
begin  
  case Key of  
    VK_Left: Window.Left := Window.Left - 2;  
    VK_Right: Window.Left := Window.Left + 2;  
    VK_Up: Window.Top := Window.Top - 2;  
    VK_Down: Window.Top := Window.Top + 2;  
  end;  
end;  
  
begin  
  // Привязка обработчиков к событиям  
  OnKeyDown := KeyDown;  
end.
```

Пример использования таймера

Данная программа выводит 1 каждые 100 миллисекунд в течение 3 секунд:

```
uses Timers;  
  
procedure TimerProc;  
begin  
    write(1);  
end;  
  
begin  
    var t := new Timer(100,TimerProc);  
    t.Start;  
    Sleep(3000);  
end.
```

Вызов `Sleep` здесь обязателен, иначе программа после создания таймера сразу закончится, и обработчик таймера ни разу не сработает.

Оператор вызова процедуры

Оператор вызова процедуры имеет вид:

имя процедуры

или

имя процедуры (список фактических параметров)

Количество фактических параметров должно совпадать с количеством формальных, а типы фактических параметров должны соответствовать типам соответствующих формальных (исключение составляют подпрограммы с параметрами по умолчанию и подпрограммы с переменным числом параметров). Фактические параметры, передаваемые по значению, должны быть совместимы по присваиванию с соответствующими формальными параметрами, типы фактических параметров, передаваемых по ссылке, должны быть эквивалентны типам соответствующих формальных параметров.

Функцию можно вызывать как процедуру, возвращаемое значение при этом игнорируется.

Методы типа char

Тип [char](#) в PascalABC.NET содержит ряд методов.

Статические методы класса char

Метод	Описание
<code>char.IsDigit(c: char): boolean</code>	Возвращает, является ли символ цифрой
<code>char.IsLetter(c: char): boolean</code>	Возвращает, является ли символ буквой
<code>char.IsWhiteSpace(c: char): boolean</code>	Возвращает, является ли символ пробельным
<code>char.IsUpper(c: char): boolean</code>	Возвращает, является ли символ буквой в верхнем регистре
<code>char.IsLower(c: char): boolean</code>	Возвращает, является ли символ буквой в нижнем регистре
<code>char.IsPunctuation(c: char): boolean</code>	Возвращает, является ли символ знаком препинания
<code>char.IsLetterOrDigit(c: char): boolean</code>	Возвращает, является ли символ буквой или цифрой
<code>char.ToLower(c: char): char</code>	Возвращает символ, преобразованный к нижнему регистру
<code>char.ToUpper(c: char): char</code>	Возвращает символ, преобразованный к верхнему регистру

Выражение вызова функции

В выражениях можно использовать вызов функции, имеющий такой же вид как и [оператор вызова процедуры](#):

имя функции

или

имя функции (список фактических параметров)

Количество фактических параметров должно совпадать с количеством формальных, а типы фактических параметров должны соответствовать типам соответствующих формальных (исключение составляют подпрограммы с параметрами по умолчанию и подпрограммы с переменным числом параметров). Фактические параметры, передаваемые по значению, должны быть совместимы по присваиванию с соответствующими формальными параметрами, типы фактических параметров, передаваемых по ссылке, должны быть эквивалентны типам соответствующих формальных параметров.

Приведение типов объектов

Объект производного класса неявно преобразуется к типу базового класса. Обратное преобразование от типа базового класса к типу производного класса может быть выполнено только явно с помощью [операции приведения типа](#).

Например:

```
type    Person = class
    ...
end;
Student = class(Person)
    ...
    procedure IncCourse;
end;
...
var
    p: Person;
    s: Student;
begin
    p := new Student('Иванов',20,3,1); // неявное
    преобразование к типу базового класса
    s := Student(p); // явное приведение к типу
    производного класса
end.
```

При выполнении приведения к производному классу может возникнуть исключение (если приведение невозможно; например, если в переменной `p` на момент выполнения операции будет храниться объект типа `Person`).

После приведения к типу производного класса можно обращаться к любым полям, свойствам и методам производного класса:

```
s.IncCourse;
```

или без присваивания промежуточной переменной:

```
Student(p).IncCourse;
```

Проверка на возможность приведения к типу производного класса осуществляется с помощью [операций `is` и `as`](#).

Синтаксис лямбда-выражений

Лямбда-выражение имеет следующий синтаксис:

секция_параметров секция_типа -> выражение

или

секция_параметров секция_типа -> составной оператор

или

function *секция_параметров секция_типа -> выражение*

или

procedure *секция_параметров -> составной оператор*

Секция типа - либо пуста, либо имеет вид:

: тип

Секция параметров устроена следующим образом:

идентификатор

или

(список идентификаторов)

Список идентификаторов состоит из секций. Каждая секция содержит идентификаторы, перечисляемые через запятую, после которых может следовать : тип. Секции отделяются одна от другой символом "точка с запятой". Список идентификаторов не может состоять из одного идентификатора без указания типа.

В инициализаторах процедурных переменных вида

var *имя: тип := лямбда-выражение*

если лямбда-выражение не начинается с ключевого слова **function** или **procedure**, то в секции параметров допустимы только переменные без указания типа, перечисляемые через запятую; секция типа также должна быть пустой

Например:

```
x -> x+1
() -> 1
(x, y) -> x*y
```

```
(x,y: integer) -> x*y
(x,y: integer): integer -> x*y
(x: integer; y: integer) -> x*y
(x,y: integer) -> begin Result := x*y end
(x,y: integer) -> begin Result := x*y end
function -> 1
function (x,y) -> x*y
function (x,y: integer) -> x*y
function (x,y: integer): integer -> x*y
function (x,y: integer): integer -> begin Result := x*y
end
procedure -> begin write(1); write(2) end
procedure (x: integer; s: string)-> begin write(x,s) end
```

Метод ForEach

Описание методов

Методы приведены для последовательности **sequence of T**.
procedure ForEach(action: T->()); Применяет действие к
каждому элементу последовательности.

Пример

```
begin  
  var a := Lst(2,3,5);  
  var p := 1;  
  a.ForEach(procedure(x) -> p *= x);  
  Println(p);  
end.
```

Перенаправление ввода-вывода

procedure SetConsoleIO; Устанавливает консольный ввод-вывод

procedure SetGraphABCI0;

Устанавливает ввод-вывод через графическое окно (по умолчанию)

Класс ContainerABC

Класс `ContainerABC` является потомком класса `ObjectABC` и представляет графический объект "Контейнер графических объектов". Он предназначен для группировки других графических объектов.

Конструкторы класса ContainerABC

constructor `Create(x, y: integer);` Создает пустой контейнер графических объектов в позиции (x, y) . Для его наполнения следует использовать метод `Add`. Координаты всех помещаемых в него графических объектов пересчитываются относительно точки (x, y)

constructor `Create(g: ContainerABC);`

Создает контейнер графических объектов - копию контейнера графических объектов `g`

Свойства класса ContainerABC

property `Count: integer;`

Количество графических объектов в контейнере

property `Objects[i: integer]: ObjectABC;`

Массив графических объектов в контейнере

Методы класса ContainerABC

procedure `Add(g: ObjectABC);`

Добавляет в контейнер графический объект `g`

procedure `Remove(g: ObjectABC);`

Удаляет из контейнера графический объект `g`

procedure `UnLink(g: ObjectABC);`

Отсоединяет от контейнера графический объект `g`. Объект `g` перестает иметь владельца и продолжает отображаться на экране в той же позиции

function `Clone: ContainerABC;`

Возвращает клон контейнера графических объектов

Свойства, унаследованные от класса ObjectABC

property `Left: integer;`

Отступ графического объекта от левого края

property `Top: integer;`

Отступ графического объекта от верхнего края

property Width: integer;

Ширина графического объекта

property Height: integer;

Высота графического объекта

property dx: integer;

x-координата вектора перемещения объекта при вызове метода **Move**.

По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property dy: integer;

y-координата вектора перемещения объекта при вызове метода **Move**.

По умолчанию установлено в 0. Для неподвижных объектов может быть использовано для хранения любой дополнительной информации, связанной с объектом

property Center: Point;

Центр графического объекта

property Position: Point;

Левый верхний угол графического объекта

property Visible: boolean;

Видим ли графический объект

property Color: GColor;

Цвет графического объекта

property FontColor: GColor;

Цвет шрифта графического объекта

property Text: string;

Текст внутри графического объекта

property TextVisible: boolean;

Видимость текста внутри графического объекта

property TextScale: real;

Масштаб текста относительно размеров графического объекта, $0 \leq \text{TextScale} \leq 1$. При $\text{TextScale} = 1$ текст занимает всю ширину или высоту объекта. По умолчанию $\text{TextScale} = 0.8$

property FontName: string;

Имя шрифта для вывода свойства **Text**

property FontStyle: FontStyleType;

Стиль шрифта для вывода свойства **Text**

property Number: integer;

Целое число, выводимое в центре графического объекта. Для вывода используется свойство `Text`

property RealNumber: real;

Вещественное число, выводимое в центре графического объекта. Для вывода используется свойство `Text`. Вещественное число выводится с одним знаком после десятичной точки

property Owner: ContainerABC;

Владелец графического объекта, ответственный также за перерисовку графического объекта внутри себя (по умолчанию `nil`)

Методы, унаследованные от класса ObjectABC

procedure MoveTo(x, y: integer);

Перемещает левый верхний угол графического объекта к точке (x, y)

procedure MoveOn(a, b: integer);

Перемещает графический объект на вектор (a, b)

procedure Move; **override**;

Перемещает графический объект на вектор, задаваемый свойствами dx, dy

procedure Scale(f: real); **override**;

Масштабирует графический объект в f раз (f>1 - увеличение, 0<f<1 - уменьшение)

procedure ToFront;

Переносит графический объект на передний план

procedure ToBack;

Переносит графический объект на задний план

function Bounds: System.Drawing.Rectangle;

Возвращает прямоугольник, определяющий границы графического объекта

function PtInside(x, y: integer): boolean; **override**;

Возвращает `True`, если точка (x, y) находится внутри графического объекта, и `False` в противном случае

function Intersect(g: ObjectABC): boolean;

Возвращает `True`, если изображение данного графического объекта пересекается с изображением графического объекта g, и `False` в противном случае. Белый цвет считается прозрачным и не принадлежащим объекту

function IntersectRect(r: System.Drawing.Rectangle):
boolean;

Возвращает **True**, если прямоугольник графического объекта пересекается прямоугольником *r*, и **False** в противном случае

function Clone0: ObjectABC; **override**;

Возвращает клон графического объекта

procedure Draw(x,y: integer; g: Graphics); **override**;

Защищенная. Не вызывается явно. Переопределяется для каждого графического класса. Рисует объект на объекте *g: Graphics*

destructor Destroy;

Уничтожает графический объект