Министерство науки и высшего образования
Российской Федерации
Братский педагогический колледж
федерального государственного бюджетного
образовательного учреждения высшего
образования
«Братский государственный университет»

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

методические рекомендации
по выполнению заданий лабораторных занятий

для студентов II курса
очной формы обучения
специальности
09.02.07 Информационные системы и программирование

Автор: Л.Д. Разумова

Братск, 2020

Основы алгоритмизации и программирования. Методические рекомендации по выполнению лабораторных работ. / Сост. Л.Д. Разумова.- Братск, 2020.- 139 с.

В методических рекомендациях излагаются в краткой форме основные принципы и правила построения и программирования базовых алгоритмических конструкций (линейных, разветвляющихся, циклических), характерных приемов парадигмы структурного программирования.

Методические рекомендации предназначены для студентов второго курса специальности 09.02.07 «Информационные системы и программирование», изучающих дисциплину «Основы алгоритмизации и программирования».

Печатается по решению научно-методического совета
Братского педагогического колледжа ФГБОУ ВО «БрГУ»
665709, г. Братск, ул. Макаренко, 40.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методические рекомендации предназначены для студентов второго курса специальности 09.02.07 «Информационные системы и программирование».

Цель методических рекомендаций – формирование у студентов начальных навыков алгоритмизации и программирования, составляющих основу профессиональной деятельности.

Написание данных методических рекомендаций преследовало цель изложить в краткой форме основные принципы и правила построения и программирования алгоритмов различных типов (линейных, разветвляющихся, циклических) и характерных приемов программирования, показать использование этих алгоритмов при решении практических задач.

При преподавании дисциплины «Основы алгоритмизации и программирования» в качестве базового принят язык Паскаль (Pascal).

Язык Паскаль (Pascal) обладает многими достоинствами в плане обучения, среди которых можно выделить следующие: простота и ясность конструкций; высокая типизация данных; контроль типов; возможность построения новых типов данных; возможность достаточно полного контроля правильности программы на этапе компиляции и во время ее выполнения; наличие набора структурных типов данных: массивов, записей, записей с вариантами, множеств, файлов; возможность удовлетворения требованиям структурного программирования, а в Object Pascal и требованиям объектно-ориентированного программирования; гибкость и надежность; возможность программирования задач различного профиля.

В методических рекомендациях рассматриваются стандартный язык Pascal, Turbo-Pascal, Object Pascal с применением интегрированной среды разработки программного обеспечения Delphi .

1. Рекомендации к лабораторной работе

Лабораторные работы выполняются индивидуально в соответствии с вариантом задания. Перед началом работы необходимо изучить теоретический минимум, который дается в начале описания каждой лабораторной работы; ответить на контрольные вопросы. Для лучшего усвоения теории к каждой лабораторной работе предлагается практический пример выполнения работы, тщательный разбор которого поможет студенту выполнить индивидуальное задание.

Написанная и отлаженная программа после запуска на выполнение должна выводить информацию об авторе, номере варианта, назначении программы (приводится лабораторное задание полностью). Вводу данных с клавиатуры обязательно должно предшествовать текстовое сообщение о типе и количестве вводимых данных.

По своей структуре разделы лабораторных работ содержат теоретическую часть, в которой кратко изложены основные аспекты языка по рассматриваемой теме; контрольные вопросы , предназначенные не только для повторения теории, но и для того, чтобы обратить внимание студента на различные приемы практического программирования предлагаемых задач; варианты индивидуальных заданий, которые выбираются из сборника заданий для лабораторных работ.

Для выполнения лабораторной работы необходимо:

1) изучить словесную постановку задачи;
2) сформулировать математическую постановку задачи;
3) выбрать метод решения задачи, если это необходимо;
4) разработать схему алгоритма;
5) записать разработанный алгоритм на языке Паскаль;
6) разработать контрольный тест программы;
7) отладить программу;
8) написать отчет.

Содержание отчета

1. Титульный лист.
2. Словесная постановка задачи.
3. Математическая формулировка задачи.
4. Схема алгоритма решения задачи.
5. Листинг программы.
6. Контрольный тест. Ручной расчет результатов(контрольных точек).
7. Результат тестирования программ (скриншоты результатов выполнения программного кода).
8. Ответы на контрольные вопросы к лабораторной работе по согласованию с преподавателем.

2. Краткая справка по разработке консольных приложений в среде Delphi

Система программирования Delphi предоставляет возможность разработки и отладки различных программных продуктов, в том числе приложений, работающих как с использованием графического интерфейса пользователя, так и в консольном режиме. Последние имеют интерфейс пользователя в виде текстового окна, называемого *окном программы*, в котором последовательно, строка за строкой отображаются данные, вводимые пользователем с клавиатуры, и данные, выводимые программой. Позицию начала ввода или вывода в окне программы указывает *курсор* - мигающий символ, имеющий вид подчеркивания в режиме вставки или прямоугольника – в режиме замена. По умолчанию длина строки равна 80, а количество строк – 50. Изменить эти и другие параметры окна программы позволяет диалог, открывающийся при вводе команды Свойства в Системном меню окна программы.

При вводе пользователь имеет возможность редактировать последние вводимые данные, используя клавиши с печатными символами, а также клавиши BackSpace (удаление последнего введённого символа), Delete (удаление символа справа от курсора), Insert (переключение режимов вставки и замены), Стрелка вверх

(удаление всех введённых символов), Стрелка влево (перемещение курсора в предыдущую позицию), Стрелка вправо (перемещение курсора в следующую позицию). Если в окне диалога, открывающегося при вводе из Системного меню окна программы команды Свойства, установить на вкладке Общие флажок Выделение мышью, то становится возможным выделять части текста буксировкой мыши, копировать выделенное в буфер обмена щелчком правой клавиши и затем вставлять в позицию курсора щелчком правой клавиши. Завершается ввод нажатием клавиши Enter, при этом курсор перемещается в начало новой строки. Максимальная длина вводимой последовательности символов равна 254.

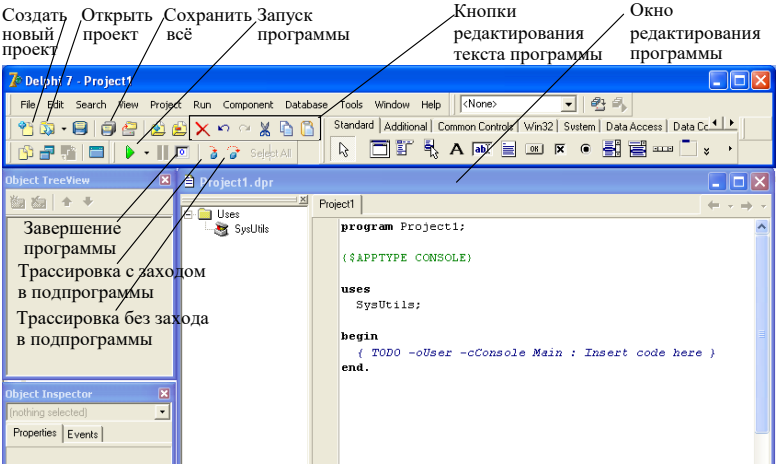


Рис. А. Окна Delphi при создании консольного приложения

Вывод данных из программы выполняется в виде текста, символ за символом при автоматическом перемещении курсора в очередную позицию строки, а при достижении её конца – в начало новой строки.

Консольный режим обычно используется в тех случаях, когда основными требованиями к программе являются минимизация времени счёта и расхода оперативной памяти. На подготовку таких программ требуется меньше времени, поэтому консольный режим удобно использовать для быстрой проверки и отладки отдельных алгоритмов. Именно поэтому, имея в виду, что данные методические

рекомендации ориентированы на развитие начальных навыков алгоритмизации и отладки небольших программ, предполагается использование консольного режима.

Для создания программы, работающей в консольном режиме, следует после запуска Delphi ввести команду File/New/Other... и выбрать вариант Console Application. При этом изменится набор окон, как показано на рис. А. В *окне редактирования программы* будет представлен стандартный *шаблон консольной программы*. Это работающая программа, которая при запуске только отображает окно программы и сразу же прекращает работу, закрывая окно. Запуск программы в среде Delphi выполняется командой Run/Run, или щелчком на кнопке  панели инструментов, или нажатием клавиши F9. Чтобы задержать окно программы до нажатия клавиши Enter, следует вставить в шаблон программы оператор ReadLn перед **end**. Впрочем, закрыть окно и завершить работу программы можно и любым другим способом закрытия окна Windows.

Элементами шаблона программы являются

заголовок программы,

директива {$APPTYPE CONSOLE},

предложение использования **uses** SysUtils;,

пустой *раздел операторов* – пустой *составной оператор*[1], состоящий только из *операторных скобок* **begin** и **end**,

символ . (точка), указывающий компилятору, что текст программы закончен.

Внутри раздела операторов (между *ключевыми словами*[2] **begin** и **end**) находится *комментарий* – поясняющий текст, предлагающий разместить в этом месте *операторы*, реализующие алгоритм программы. Комментарии, в отличие от операторов, не порождают команд процессора при обработке компилятором. Компилятор их просто пропускает. Назначение же комментариев состоит в том,

---

[1] Составным оператором называется конструкция, состоящая из операторных скобок **begin** и **end** , между которыми располагаются выполняемые последовательно операторы.

[2] Ключевыми называют слова, используемые как обязательные элементы синтаксических конструкций языка программирования (**begin**, **end**, **uses**, **type**, **var**, **if**, **for**, **do** и другие). Ключевыми слова запрещено использовать в программе по иному назначению. В тексте эти слова выделяются жирным шрифтом.

чтобы облегчить понимание алгоритма программы. Оформить текст в виде комментария можно одним из следующих способов:

поместив два символа // перед текстом строки,

заключив текст, который может занимать несколько строк, в фигурные скобки { и },

заключив текст, который может занимать несколько строк, в скобки вида (* и *).

Заголовок программы содержит её имя. Delphi даёт программе стандартное имя Project1, или Project2, или Project3 и так далее в порядке создания очередной программы командой File/New.... Это имя можно заменить на другое при сохранении текста программы командой File/Save All..., когда Delphi предложит сохранить программу в файле с расширением .dpr. Одновременно Delphi создаст и сохранит ещё файл с тем же именем и расширением .cfg и файл с расширением .drf. Вместе эти три файла образуют *проект* и для их сохранения желательно создать отдельную папку. В этой же папке будет размещена и созданная исполняемая программа, имеющая расширение .exe.

Директива {$APPTYPE CONSOLE} указывает компилятору, что он должен создать консольное приложение.

Предложение использования uses SysUtils; предписывает подключить к программе *стандартный* (входящий в систему программирования Delphi) *модуль*[3] с именем SysUtils. Этот модуль, в свою очередь, содержит предложение использования, подключающее другие стандартные модули, и всё, что объявлено в них, становится доступным в разрабатываемой программе. В частности, подключение модулей необходимо для создания консольного приложения и использования в нём некоторых стандартных *типов данных*[4] (например, типа Integer - для

[3] Модуль – это специальным образом оформленный набор именованных констант, типов, переменных, подпрограмм и директив, которые могут быть доступны в другом модуле или программе, если имя модуля включить в список предложения использования uses. Исходный текст модуля хранится в файле с расширением .pas, а откомпилированный модуль – в файле с расширением .dcu. Delphi позволяет создавать новые модули, автоматически или по запросу включая их в проект (см. главу Модули пользователей).

[4] Типом данных называют набор характеристик, которыми будут обладать *переменные*, объявленные с помощью этого типа. К числу характеристик

объявления переменных целого типа, типа Real - для объявления переменных вещественного типа), а также *стандартных подпрограмм*[5]: стандартных процедур (например, Write – для вывода данных в окно программы или в файл, Read – для чтения данных с клавиатуры или из файла) и стандартных функций (например, Sin – для вычисления синуса, Sqrt – для вычисления квадратного корня). Дополнительный набор стандартных подпрограмм (процедур и функций) содержит стандартный модуль с именем Math, который следует подключить к программе, добавив в предложение использования: **uses** SysUtils, Math;.

Объявления *именованных констант*[6], типов, переменных и подпрограмм, сделанные в подключенных к программе модулях, считаются сделанными до аналогичных объявлений в программе,

---

простых типов относятся правила записи констант, набор операций, набор стандартных подпрограмм. Можно объявлять в программе или в модуле новые типы данных на основе стандартных типов или ранее объявленных. Объявление новых именованных типов даётся в разделе объявления типов, начинающемся со слова **type**, например, **type** tMas=**array**[1..10] **of** Real; объявляет тип с именем tMas типом массивов из 10 вещественных чисел. Объявляются переменные в разделе, начинающемся со слова **var**, например, **var** N:Integer; X Y:tMas; объявляет переменную N целого типа и массивы X и Y типа tMas.

[5] Подпрограммой называют оформленный специальным образом алгоритм, для выполнения которого достаточно сделать вызов подпрограммы, указав после её имени список данных, подлежащих обработке. В языке Object Pascal есть два вида подпрограмм: процедуры и функции. Вызовы процедур являются отдельными операторами, а вызовы функций обычно используются внутри операторов в выражениях. Например, для вывода значения $\sin(\frac{\pi}{4})$ следует записать вызов стандартной процедуры Write с вызовом стандартной функции Sin, вычисляющей значение синуса, и стандартной функции Pi, вычисляющей значение числа π: Write(Sin(Pi/4)). Пользователь может создавать свои подпрограммы, размещая их тексты в любом месте перед разделом операторов .

[6] Именованные константы объявляются в разделе, начинающемся со слова **const**. Например, **const** Nmax=25;. Имена таких констант представляют в программе соответствующие значения. Использование именованных констант облегчает понимание алгоритма программы и позволяет минимизировать ошибки при модернизации программы, а именно, если имя константы используется в разных частях программы, то достаточно изменить значение только в объявлении именованной константы.

что соответствует правилу языка Object Pascal: в любом объявлении можно использовать только ранее объявленные имена.

Объявления и следующий за ними раздел операторов называют *блоком*. Используя этот термин, структуру консольной программы можно определить как последовательность, включающую заголовок программы, директиву {$APPTYPE CONSOLE}, предложение использования модулей, блок и точку. Кроме того, в тексте программы можно размещать директивы, управляющие работой компилятора.

Рассмотрим следующий пример программы.

```
program Project1;
{$APPTYPE CONSOLE}
uses  SysUtils;
{РАЗДЕЛ ОБЪЯВЛЕНИЯ КОНСТАНТ}
const
  g=9.8;            //Именованная константа, представляющая ускорение
{РАЗДЕЛ ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ}
var
  S: Integer;//Переменная целого типа, представляющая длину пути
  T: Real;//Переменная вещественного типа, представляющая время
{РАЗДЕЛ ОПЕРАТОРОВ}
begin
  //Вывод приглашения к вводу значения переменной S
  Write('Enter S: ');
  //Ввод значения переменной S
  ReadLn(S);
  //Вычисление времени движения тела на заданном пути
  T:=Sqrt(S*2/g);
  //Вывод результата с пояснениями
  WriteLn('T =', T);
  //Задержка закрытия окна программы до нажатия клавиши Enter
  ReadLn;
end.
```

Программа вычисляет время движения тела с ускорением 9,8 м/с^2 на пути заданной длины, вводимой с клавиатуры. В блоке программы используется раздел объявления именованных констант (константа g), раздел объявления переменных (переменные T и S) и раздел операторов. Протокол ввода-вывода программы представлен в окне программы на следующем рисунке.

Результат выводится с поясняющим кратким текстом, набранный в латинском алфавите.

Цвет фона (чёрный) и символов (белый) в окне программы можно изменить следующим образом: раскрыть Системное меню окна щелчком на кнопке  , выбрать пункт Свойства, в появившемся окне диалога на вкладке Цвета из представленной палитры выбрать для фона и символов желаемые цвета, нажать кнопку **OK**, выделить в появившемся окне Изменение свойств кнопку Сохранить свойства для других окон с тем же именем и щелкнуть на кнопке OK. Например, при выборе белого цвета для фона и чёрного для символов окно программы примет вид

.

Исполняемая программа может быть создана, только если устранены *синтаксические ошибки* (ошибки, связанные с нарушением в конструкциях программы правил языка программирования, которые обнаруживает компилятор) и подключены необходимые модули. При обнаружении таких ошибок Delphi выдаёт сообщения с кратким пояснением причины ошибки в специальном окне, появляющемся под окном редактирования программы. Ошибки следует исправить и вновь выполнить команду Run/Run.

Устранением указанных ошибок разработка программы не заканчивается. Это только начало важного этапа разработки программы, называемого *отладкой*. Суть отладки состоит в

приведении программы к полному соответствию заданию на её разработку. Программа должна на любых допустимых входных данных и их последовательностях давать требуемый результат.

На этапе отладки должны быть выявлены и устранены ошибки в её алгоритме и другие ошибки, которые могут проявиться только во время работы программы. Для их обнаружения обычно используют при отладке вывод промежуточных результатов вычислений, проверку конечных результатов (если они не очевидны) другими методами и другие приёмы.

Сократить время отладки позволяет использование *начальных значений переменных* (значений, заданных при объявлении переменных, которые они будут иметь в момент старта программы), что особенно заметно при необходимости ввода большого числа исходных данных, например, в программах обработки массивов. Использование начальных значений переменных позволяет также облегчить подготовку исходных данных для тестирования программы (если требуется изменять лишь часть данных), представить данные в удобном для восприятия виде (например, в виде матрицы) рядом с переменными, которые их представляют, быстро изменить данные (так как после перезагрузки программы сразу становится активным окно редактирования программы). При этом на время отладки программы её части, обеспечивающие ввод данных, превращают в комментарии, например, заключая между скобками (* и //*). В представленной ранее программе выполнить это можно так:

```
var
  S:Integer=15;//Переменная целого типа,
               //представляющая длину пути
  T:Real;//Переменная вещественного типа, представляющая время
{РАЗДЕЛ ОПЕРАТОРОВ}
begin
  //Вывод пояснения о назначении программы
  WriteLn(Rus('Программа вычисляет время, за которое тело')
       ,Rus(' в свободном падении пройдёт заданный путь.'));
(*
  //Вывод приглашения к вводу значения переменной S
  Write(Rus('Введите длину пути в метрах: '));
```

```
  //Ввод значения переменной S
  ReadLn(S);
//*)
```

После завершения отладки ввод данных восстанавливается превращением скобки (* в комментарий //(*. Для отключения операторов на время отладки или добавления отладочных операторов, которых не должно быть в готовой программе, удобно использовать *условную компиляцию*. Для этого в программу вводят конструкции вида

```
{$IFDEF <ИМЯ>}
. . . . .//операторы, используемые при отладке
{$ENDIF}
```

или

```
{$IFDEF <ИМЯ>}
. . . . .//операторы, используемые при отладке
{$ELSE}
. . . . .//операторы, используемые в готовой программе
{$ENDIF}
```

Обе конструкции обрабатываются компилятором следующим образом. Если ранее в исходном тексте программы встретилась директива {$DEFINE <ИМЯ>}, объявляющая некоторое уникальное имя, то компилятор включает в исполняемую программу операторы, расположенные между директивами {$IFDEF <ИМЯ>} и {$ENDIF} или {$IFDEF <ИМЯ>} и {$ELSE}. Если же директиву {$DEFINE <ИМЯ>} удалить или превратить в комментарий, например, так //{$DEFINE <ИМЯ>}, то компилятор включит в исполняемую программу операторы, расположенные между директивами {$ ELSE} и {$ENDIF}.[7] Например, если на время отладки в начале программы поместить директиву

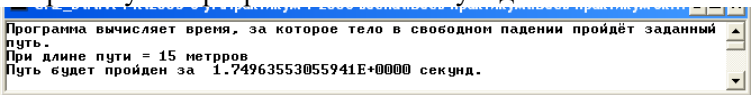
```
{$DEFINE Debug }
```

и в разделе операторов

[7] Есть и другие директивы условной компиляции, в частности {$IFNDEF <ИМЯ>} и {$UNDEF <ИМЯ>}, имеющие противоположный смысл по сравнению с директивами {$IFDEF <ИМЯ>} и {$DEFINE <ИМЯ>} соответственно.

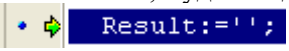
```
{$IFDEF Debug}
  Write(Rus('При длине пути = '), S, Rus(' метрров'));
{$ELSE}
  //Вывод приглашения к вводу значения переменной S
  Write(Rus('Введите длину пути в метрах: '));
  //Ввод значения переменной S
  ReadLn(S);
{$ENDIF}
```

то при запуске программы в её окне увидим



Рассмотренный приём ускорения отладки программы предполагается использовать по умолчанию при выполнении заданий.

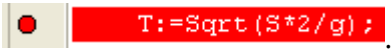
Среда Delphi также предоставляет средства отладки. Они позволяют остановить выполнение программы на любой строке её исходного текста и просмотреть текущие значения переменных и выражений, установить или отменить точки останова и продолжить работу программы. Рассмотрим некоторые из них.

Трассировка с заходом в подпрограммы (Команда  Run/Step Into. Быстрая клавиша F7). Ввод одной команды приводит к выполнению операторов одной строки программы и останов до ввода очередной команды продолжения работы программы. Если в строке есть вызов подпрограммы, то следующие такие команды будут выполнять операторы строк подпрограммы. В исходном тексте очередная строка, подлежащей исполнению, будет выделена и отмечена значком стрелки, например, 

Трассировка без захода в подпрограммы (Команда  Run/Step Over. Быстрая клавиша F8). То же, что и Run/Step Into, но строка с вызовами подпрограмм выполняется за один шаг трассировки.

Выполнить до курсора (Команда  Run/ Run to Cursor. Быстрая клавиша F4). Выполнение программы до строки её текста, в которой расположен курсор ввода.

Точка безусловного останова. Устанавливается щелчком в полосе перед строкой операторов на значке . В результате значок

и строка будут выделены. Например, 

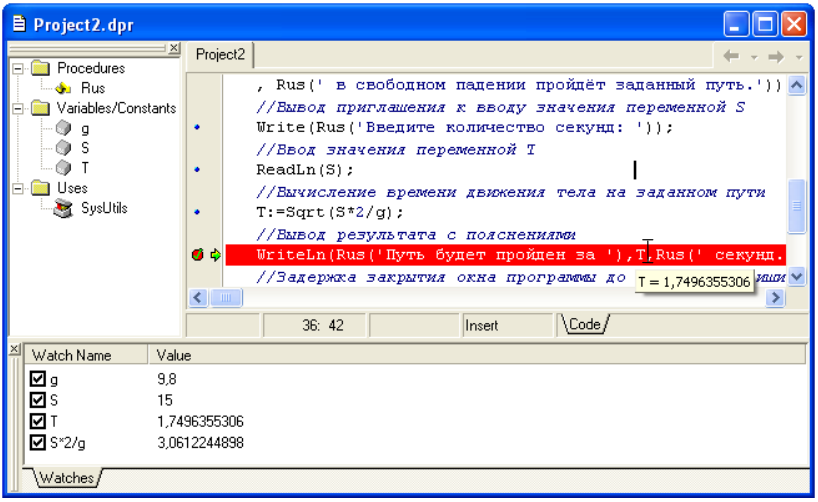
. Удаляется повторным щелчком на значке. При работе программы её останов будет перед выполнением операторов строк, отмеченных как точки безусловного останова.

Перезагрузка программы (Команда  Run/ program Reset. Быстрая клавиша Ctrl+F2). Выполняет выход из режима отладки и готовит программу к новому запуску.

Окно наблюдения (Команда  Run/Add Watch.... Быстрая клавиша Ctrl+F5). При вводе команды открывается окно диалога, в котором можно указать, значения какой переменной или выражения должно отображаться в окне наблюдения в момент ожидания системой ввода очередной команды продолжения выполнения программы. Выражения могут содержать переменные и именованные константы программы, обычные константы, обращения к некоторым стандартным функциям. Допускаются любые выражения, не обязательно те, что есть в программе. Окно наблюдения можно буксировкой разместить в окне редактирования программы или вне его. Значения переменных и выражений можно также просматривать, подводя к ним в окне редактирования программы курсор мыши.

Ниже представлен пример окна редактирования программы, вычисляющей выражение $\sqrt{\frac{2R}{g}}$. В нижней части этого окна расположено окно наблюдения.

Программа после запуска остановлена перед выполнением оператора WriteLn (эта строка объявлена точкой безусловного останова), и в окне наблюдения отображены значение именованной константы, текущие значения переменных и выражения. На вкладке с текстом программы виден курсор мыши, подведённый к переменной T, и под ним – значение этой переменной.



При выполнении программы могут возникать ошибки, связанные с ограничениями технических средств и особенностями выполнения некоторых операций, что может приводить к неверным результатам работы программы или снятию её с выполнения.

Устранение всех видов ошибок приводит, как правило, к многократной корректировке текста программы, поэтому желательно настроить панель инструментов Delphi, добавив в неё кнопки типовых команд редакторов текста, показанных на рис. А. Делается это так. Вводится команда View/Toolbars/Customize...; в открывшемся окне Customize на вкладке Commands выбирается строка Edit; значки из списка Commands: буксируются на панель инструментов. Для удаления значка с панели инструментов нужно просто, находясь в этом режиме, отбуксировать его за её пределы.

Аналогично можно добавить кнопки, используемые при отладке, раскрыв список Run на вкладке Commands окна Customize. На рис. А к расположенным по умолчанию кнопкам на панели инструментам добавлена кнопка перезагрузки программы  (команда Run/program Reset, быстрая клавиша Ctrl+F2).

3. Лабораторный практикум

3.1 Лабораторная работа № 1
Разработка программ линейной структуры. Стандартные типы данных и выражения .

Цель работы: ознакомление со стандартными типами данных и выражениями языка Паскаль, овладение навыками разработки алгоритмов линейных процессов, умением составлять математические модели алгоритмов и записывать программы на языке Паскаль.

Задание на лабораторную работу:

1. Составить код, позволяющий вычислить, упростив за счет использования скобочных форм и/или дополнительных переменных, значения по заданным формулам,
2. Для контроля правильности результатов выполнить вычисления по формулам без использования скобочных форм и дополнительных переменных.

Теоретический материал

Программой линейной структуры называется такая программа, каждый оператор которой выполняется один и только один раз. Она может строиться только из простых операторов, не меняющих естественный порядок вычислений, а именно, из операторов присваивания и операторов процедур. Из числа последних в этом разделе нас будут интересовать только операторы процедур ввода и вывода для стандартных устройств – клавиатуры и монитора.

Средства разработки программ линейной структуры

Рассмотрение вопросов алгоритмизации задач и приёмов программирования удобнее всего проводить на примерах обработки числовых данных. Рассмотрим в первую очередь стандартные типы (имеющиеся в Delphi и не требующие объявления в программе) числовых данных.

Целые типы данных

К числу стандартных целых типов относятся:

Int64 – тип, представляющий целые со знаком от –263 до +263-1, занимает 8 байт;

Integer – тип, представляющий целые со знаком от –2147483648 до +2147483647, занимает 4 байта;

LongInt – тип, эквивалентный типу Integer;

SmallInt – тип, представляющий целые со знаком от –32768 до +32767, занимает 2 байта;

ShortInt – тип, представляющий целые со знаком от –128 до +127, занимает 1 байт;

Byte – тип, представляющий целые без знака от 0 до 255, занимает 1 байт;

Word – тип, представляющий целые без знака от 0 до 65535 занимает 2 байта;

LongWord – тип, представляющий целые без знака от 0 до 4294967295, занимает 4 байта;

Cardinal – тип, эквивалентный типу LongWord .

Например, чтобы объявить переменные с именами I и K как переменные типа Integer и N – как Byte, в программе следует записать

```
var
  I, K:Integer; //Объявление целых переменных I и K типа Integer
  N:Byte  //Объявление целой переменной N типа Byte
```

Константы целого типа записываются в виде последовательности цифр, перед которой может стоять знак числа. Знак + перед положительным числом можно не писать. Например, константы +25 и 25 представляют одно и то же значение. Тип целой константы определяется как стандартный целый тип с наименьшим диапазоном значений, включающим зна-чение константы.

Именованные константы (имя такой константы представляет значение) объявляют в разделе const, связывая имя и значение знаком =, например,

```
const
  Nmax=10;   //Объявление именованной константы Nmax
```

Для данных целого типа определены следующие арифметические операции, результат выполнения которых также будет иметь целый тип такой, который имеет минимальный

диапазон, включающий вычисленное значение:

сложение (знак +),

изменение знака (унарный минус -),

вычитание (знак -),

умножение (знак *),

целочисленное деление (знак div),

взятие по модулю (знак mod).

Результатом выполнения операции div является целая часть частного, а операции mod – остаток от целочисленного деления (знак остатка всегда совпадает со знаком делимого). Например, выполнение –5 div -2 даст значение 2, а после выполнения -5 mod –2 получим –1.

Допустима над целыми также операция деления (знак /), приводящая к вещественному значению. Так, результатом выполнения –5/-2 будет вещественное число 2,5.

К числу целых относятся также интервальные (диапазонные) типы, объявляемые в программе. Например, в следующем фрагменте программы

```
type
  tBall = 2..5;      //Объявление типа tBall
  tIndex = 1..10;  //Объявление типа tIndex
```

объявляются тип tBall с диапазоном значений от 2 до 5 и тип tIndex с диапазоном значений от 1 до 10. Значение, представляющее начало диапазона должно быть меньше значения, представляющего конец диапазона, а разделителем между ними является составной символ из двух точек.

Вещественные типы данных

К числу стандартных вещественных (действительных) типов относятся:

Extended – вещественный тип с повышенной точностью, допускающий множество значений с 19 – 20 десятичными цифрами в диапазоне абсолютных значений от 3,610-4951 до 1,1○ 10+4932, занимает 10 байт;

Double – тип, допускающий множество значений с 15 – 16 десятичными цифрами в диапазоне абсолютных значений от 510-324 до 1,710+308, занимает 8 байт;

Real – тип, эквивалентный типу Double;

Real48 – тип, допускающий множество значений с 11 – 12 десятичными цифрами в диапазо-не абсолютных значений от 2,910-39 до 1,710+38, занимает 6 байт;

Single – тип, допускающий множество значений с 7 – 8 десятичными цифрами в диапазоне абсолютных значений от 1,510-45 до 3,410+38, занимает 4 байта;

Comp – тип, допускающий множество целых значений с 19 – 20 десятичными цифрами в диапазоне от -263+1до +263-1, занимает 8 байт;

Currency – тип, допускающий множество значений с 19 – 20 десятичными цифрами в диапа-зоне от -922337203685477,5808 до +922337203685477,5807, занимает 8 байт;

Константы вещественного типа записываются либо в естественной форме, например,

-12.345, либо в экспоненциальной форме, в которой то же самое число можно записать по–разному, например, -0.12345E+2, или -0.12345E2, или -0.12345e+2, или –1.2345E+1, или –1.2345E1, или –12.345E0, или –1234.5E-2, или –12345E-3, или и так далее. При представлении числа в такой форме безразлично, используется строчная или прописная латинская буква E. Чтобы получить значение числа, представленного в экспоненциальной форме, нужно умножить мантиссу, то есть то, что стоит перед символом E, на порядок, то есть на 10 в степени, значение которой представляет целое число, записанное после E. Так, константу -0.12345E+2 следует читать как –0,12345•10+2, а константу -1234.5E-2 – как –1234,5•10-2.

Следующий фрагмент программы представляет объявления вещественных переменных X и Y типа Real, Z – типа Extended и именованной константы H со значением 0,00000025:

```
var
  X, Y: Real;
  Z: Extended;
const
  H = 2.5E-7;
```

Для данных вещественного типа определены следующие арифметических операции, результат выполнения которых также будет иметь вещественный тип такой, который имеет минимальный диапазон, включающий вычисленное значение:

сложение (знак +),

изменение знака (унарный минус -),
вычитание (знак -),
умножение (знак *),
деление (знак /).

В отличие от языков программирования BASIC и Fortran, в языке Object Pascal нет операции возведения в степень.

Стандартные функции для обработки числовых данных

Стандартные функции, предназначенные для вычисления некоторых математических и тригонометрических функций, а также для преобразования данных вещественного типа к целому, содержатся в модуле System, подключаемом к программе по умолчанию, и в модуле Math.

Стандартные функции можно разбить на группы по признаку, – какого типа их аргументы (то есть фактические параметры, которыми могут быть константы, переменные, выражения) и какого типа возвращаемый ими результат.

Стандартные функции, не меняющие тип, то есть для целого аргумента возвращается целое значение, а для вещественного – вещественное:

Abs(X) – возвращает абсолютное значение аргумента,
Sqr(X) – возвращает квадрат аргумента
Max(X,Y) – возвращает максимальное из X и Y,
Min(X,Y) – возвращает минимальное из X и Y.

Стандартные функции, возвращающие в любом случае (аргумент целый, вещественный или вообще нет аргумента) вещественное значение:

Tan(X) – возвращает значение тангенса аргумента,
ArcTan(X) – возвращает значение арктангенса аргумента,

Sin(X) – возвращает значение синуса аргумента,
ArcSin(X) – возвращает значение арксинуса аргумента,
Cos(X) – возвращает значение косинуса аргумента,
ArcCos(X) – возвращает значение арккосинуса аргумента,
Exp(X) – возвращает значение ex,
Frac(X) – возвращает дробную часть аргумента,
Int(X) – возвращает целую часть аргумента,

Ln(X) – возвращает значение натурального логарифма аргумента,

Log10(X) – возвращает значение логарифма аргумента X по основанию 10,

LogN(N,X) – возвращает значение логарифма аргумента X по основанию N,

Pi – возвращает значение числа □,

Sqrt(X) – возвращает значение квадратного кореня аргумента,

IntPower(X,N) – возвращает значение X, возведенное в целую степень N,

Power(X,Y) – возвращает значение X, возведенное в степень Y (если Y не целое, то значение X должно быть не отрицательным).

Стандартные функции, возвращающие значение целого типа:

Round(X) – возвращает округленное значение вещественного аргумента,

Trunc(X) – возвращает значение вещественного аргумента после отбрасывания его дробной части,

Sign(X) – возвращает значение 1 для X>0, значение 0 – для X=0, значение -1 – для X<0.

Дополнительные сведения о подпрограммах модулей System и Math можно найти в справочных разделах Delphi Arithmetic routines и Trigonometry routines.

Арифметические выражения

Арифметическое выражение записывается в виде последовательности целых или вещественных констант, переменных и обращений к функциям, разделенных знаками арифметических операций и круглыми скобками. В арифметических выражениях могут использоваться одновременно переменные, константы и функции разных целых и вещественных типов (такую возможность обозначают термином «совместимость типов в выражениях»).

Результат вычисления выражения будет целого типа, если все используемые в нем константы, переменные и функции имеют целый тип и не используется знак операции /, иначе – вещественного типа.

В выражениях в первую очередь вычисляются обращения к функциям и содержимое круглых скобок, затем – операции типа

умножения (*, /, div, mod) в порядке слева направо, затем – операции типа сложения (+ и -) в порядке слева направо.

Например, для вычисления выражения можно записать в программе:

```
Sqr(Sin(X))*Cos(IntPower(Y,3))*1.2E-4/Sqrt(X)/Y/Power(Z,2/3).
```

Оператор присваивания

В процессе выполнения программы переменные могут получать новые значения либо с помощью процедур ввода, либо с помощью операторов присваивания (либо как выходные параметры подпрограмм, но об этом позже).

Оператор присваивания записывается в виде:

Выражение, записанное справа от знака присваивания := (состоит из знаков «двоеточие» и «равно»), вычисляется, преобразуется, при необходимости, к типу переменной, стоящей слева от знака присваивания, после чего эта переменная получает вычисленное значение. Например, для вещественной переменной X выполнение оператора

```
X:=5 mod 2;
```

будет складываться из следующих действий: вычисления выражения 5 mod 2 целого типа, результатом чего станет значение 1, преобразование этого значения к вещественному типу и сохранение преобразованного значения в ячейке памяти, соответствующей переменной X.

Возможность преобразования в операторе присваивания значения одного типа к другому называют совместимостью по присваиванию. Допустимо оно не во всех случаях. Например, целое можно преобразовать присваиванием к вещественному, но не наоборот. В последнем случае следует воспользоваться стандартными функциями Round или Trunc. Например, для переменной K целого типа допустимо использование операторов

```
K:=Round(5.6);          //Переменная K получит значение 6
```

или

```
K:=Trunc(5.6);  //Переменная K получит значение 5,
```

но запрещено

```
K:=5.6;
```

Ввод данных с клавиатуры

В консольных приложениях для ввода данных с клавиатуры есть два оператора вызова процедур с именами Read и ReadLn (далее их будем называть просто операторами ввода или операторами Read и ReadLn). Параметры этих процедур – переменные, значения которых вводятся, располагаются вслед за именами этих процедур в круглых скобках в виде списка через запятую, который называют списком ввода. Элементами списка ввода могут быть переменные только числовых типов, символьные и строковые. Вводимые данные для числовых типов, набираемые на клавиатуре, должны разделяться или пробелами, или символами табуляции (клавиша TAB), или символами конца строки (клавиша Enter). Набор данных заканчивается нажатием клавиши Enter. Набранные данные последовательно присваиваются переменным списка ввода. Возможно также, что число элементов списка ввода будет меньше или больше количества набранных на клавиатуре данных. Тогда, при использовании оператора Read, в первом случае не использованные этим оператором числа могут быть введены следующим оператором ввода, а во втором – следующий оператор ввода будет ждать набора на клавиатуре недостающих данных и нажатия клавиши Enter.

Например, ввести значения вещественных переменных X и Y оператором Read(X,Y) можно, подготовив вводимые числа на одной строке:

5.21 1e-3□

или в двух строках:

5.21□

1e-3□,

где символ □ обозначает нажатие клавиши Enter.

Использование для тех же целей двух операторов:

Read(X); Read(Y);

также допустимо при наборе данных в одной или двух строках.

Оператор ReadLn отличается от оператора Read тем, что после завершения ввода ос-тавшиеся неиспользованными набранные числа пропадают (не могут быть использованы следующими операторами ввода). Например, для ввода операторами

```
ReadLn(X); ReadLn(Y);
```

тех же данных их следует набирать в двух строка, как показано в предыдущем примере.

Вывод данных в окно программы

В консольных приложениях для вывода данных на экран есть два оператора вызова процедур с именами Write и WriteLn (далее их будем называть просто операторами вывода или операторами Write и WriteLn). Оператор WriteLn отличается от оператора Write тем, что после завершения вывода курсор в окне программы переводится в начало следующей строки, и если текущая строка была последней, то внизу окна программы появляется новая, пустая, а все другие смещаются вверх.

Подлежащие выводу данные перечисляются в списке вывода, который записывается в круглых скобках непосредственно за словами Write или WriteLn. Список вывода может содержать выражения, в частности константы, переменные, обращения к функциям, разделенные запятыми. Оператор WriteLn можно записывать и без списка вывода и скобок, если требуется просто перевести курсор в начало новой строки.

Элементы списка вывода могут иметь из простых типов любой числовой, булевский и символьный. Пока ограничимся рассмотрением только числовых типов и строковых констант.

Пример. Составить программу вычисления с повышенной точностью (Extended) тангенса угла, значение которого (целое число) в градусах вводится с клавиатуры в ответ на приглашение к вводу. Вывести на экран с пояснениями введенное значение угла в градусах, соответствующее ему значение в радианах и вычисленное значение тангенса этого угла.

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils, Math;
var
  R:Extended;
  Fi:Integer;
```

```
begin
  //Вывод приглашения к вводу угла в градусах
  Write('Введите значение угла в градусах: ');
  ReadLn(Fi);//Ввод значения угла в переменную Fi
  R:=Fi*Pi/180;//Перевод угла в радианы и присвоение переменной R
  //Вывод значения R
  WriteLn('Значение угла в радианах = ',R);
  WriteLn;                    //Пропуск строки
  //Вывод R и tg(R) с поясняющими текстами
  WriteLn('tg(', R, ') = ', Tan(R));
  ReadLn;
end.
```

где заключенные в апострофы тексты, например, 'Fi = ', 'tg(' и ') = ' – строковые константы, R и выражение Tan(R) представляют данные типа Extended.

Протокол ввода–вывода при выполнении этой программы при вводе для Fi значения 30 будет иметь вид:

```
Введите значение угла в градусах: 30
Значение угла в радианах =  5.23598775598299E-0001

tg( 5.23598775598299E-0001) =  5.77350269189626E-0001
```

а курсор перейдет в начало следующей, новой строки.

Первый оператор Write выводит приглашение к вводу, в ответ на которое пользователь вводит величину угла, в данном случае 30. Ввод данных заканчивается нажатием клавиши Enter, что приведет к переводу курсора в начало новой строки. Следующий далее оператор WriteLn('Значение угла в радианах = ',R) выводит значение угла в радианах и переводит курсор на начало следующей, пустой строки, оператор WriteLn без параметров оставит эту строку пустой и переведет курсор в начало еще одной новой пустой строки, в которую оператор WriteLn('tg(', R, ') = ', Tan(R)) выведет последовательность символов:

tg(5.23598775598299E-0001) = 5.77350269189626E-0001.

В этом примере для всех типов данных использовались принятые по умолчанию (что не всегда удобно) форматы вывода: вещественные выводятся в экспоненциальной форме в 23 позициях с

четырьмя цифрами порядка; целые и строковые занимают минимально необходимое число позиций, причем целые положительные изображаются без знака +.

Программисту, чтобы определить свою форму вывода, следует использовать один из форматов вывода, записываемый непосредственно за элементом списка вывода либо в виде :n – для любого типа данных, либо в виде :n:m – при выводе вещественного числа в естественной форме (<целая часть><точка><дробная часть>), где n – выражение целого типа, зна-чение которого задает длину поля (число знакомест) в строке на экране, отводимых для изо-бражения значения, а m – выражение целого типа, значение которого задает количество цифр в дробной части числа.

Значение выражения n может быть больше или меньше требуемого для представления значения количества знакомест. В первом случае выводимое значение будет расположено в правой части поля вывода. Во втором случае под вывод значения отводится минимально необходимое число позиций (для вещественных в экспоненциальной форме оно равно 10).

Например, при выполнении следующей программы:

```
program Project2;
{$APPTYPE CONSOLE}
uses
  SysUtils, Math;
var
  R:Extended;
  Fi:Integer;
begin
  //Вывод приглашения к вводу угла в градусах
  Write('Введите значение угла в градусах: ');
  ReadLn(Fi);//Ввод значения угла в переменную Fi
  R:=Fi*Pi/180;//Перевод угла в радианы и присвоение переменной R
  //Вывод значения R
  WriteLn('Значение угла в радианах = ',R:0);
  WriteLn;                    //Пропуск строки
  //Вывод R и Tan(R) с поясняющими текстами
  WriteLn('tg(', R:0:2, ') = ', Tan(R):14);
  ReadLn;
```

end.

протокол ввода–вывода при вводе для Fi значения 30 будет иметь вид:

```
Введите значение угла в градусах: 30
Значение угла в радианах =  5.2E-0001

tg(0.52) =  5.77350E-0001
```

Значение R первый раз выведено в экспоненциальной форме в минимально необходимое число позиций (формат вывода :0), а второй раз – в естественной форме также в минимально необходимое число позиций (формат вывода :0:2), так как в обоих случаях длина поля вывода указана равной нулю, то есть меньше минимально необходимой. Значение выражения Tan(R) выведено в экспоненциальной форме в поле из 14 позиций с 6 значащими цифрами мантиссы (формат вывода :14)

Приемы, используемые для минимизации вычислений

Одним из критериев, характеризующим качество составленной программы, является объем выполняемых ею вычислений для достижения требуемого результата. Чем он меньше, тем, как правило, быстрее будет работать программа. Существуют разные приемы, применение которых позволяет сократить объем вычислений за счет уменьшения в первую очередь количества вызовов функций, затем – количества операций типа умножения, и, наконец, – количества операций типа сложения. Вот некоторые из них, рассмотренные отдельно, хотя, как правило, они используются в сочетании.

Вынесение общих множителей за скобки.
Например, вместо оператора

```
Z:=Sin(X)*Y-Sin(X)*Sqr(Y)*Y+Sqr(X)*X+X;
```

лучше использовать оператор

```
Z:=Sin(X)*Y*(1-Sqr(Y))+X*(Sqr(X)+1);
```

Использование схемы Горнера для полиномов.
Например, полином

$2X^5 - 5X^4 + 2X^3 + 7X^2 - 4X + 6$

преобразованный по схеме Горнера, примет вид

((((2X-5)X+2)X+7)X-4)X+6

где явно меньше число операций умножения, если считать, что возведение в степень выполняется через умножение. В программе по второму варианту записи полинома будем иметь выражение

((((2*X-5)*X+2)*X+7)*X-4)*X+6,

а по первому –

2*IntPower(X,5)-5*IntPower(X,4)+2*IntPower(X,3)+7*IntPower(X,2)-4* X+6

где помимо такого же числа операций умножения требуется выполнить четыре обращения к функции возведения в степень.

Другой пример, где также применима схема Горнера:

1+2!+3!+4!+5! = 1+2(1+3(1+4(1+5)))

Использование дополнительных переменных.

Например, при вычислении значения функции

$$Z = (X - Y)\frac{1 + (X - Y)^3}{1 + (X - Y)^2}$$

целесообразно предварительно вычислить $A = X - Y$ и $B = A^2$,

а исходную формулу преобразовать к виду $Z = A\frac{1 + AB}{1 + B}$.

В этом случае в программе будут использованы три оператора присваивания:

```
A:=X-Y;
B:=Sqr(A);
Z:=A*(1+A*B)/(1+B);
```

Пример выполнения задания лабораторной работы

Найти значение функции

$$Y(X)=\frac{\left(\frac{a}{2}\right)^{x}-\lg\left(\frac{a}{2}+1\right)}{\left(\frac{a}{2}\right)^{3}-\left(\frac{a}{2}\right)^{2}} \quad (1.2),$$

упростив вычисления за счет использования скобочных форм и/или дополнительных переменных (в этом предложении и в дальнейшем конструкция «А и/или Б» обозначает «или А, или Б, или А и Б одновременно»). Для контроля правильности результата выполнить вычисление по формуле (1.2) без использования скобочных форм и дополнительных переменных.

Проверить работу программы на значениях A=(1; -1; 2; -2; 4; -4), X=(0,5; 2).

```
program Project1_2;
{$APPTYPE CONSOLE}
uses
SysUtils, Math;
var
A, B, C, X, Y1, Y2: Real;
begin
   {Ввод исходных данных}
  Write('Введите X и A : ');
  ReadLn(X,A);
  B:=A/2;
  C:=Sqr(B);
   {Вычисление выражения}
   { - с использованием дополнительных переменных}
  Y1:=(Power(B,X)-Log10(B+1))/C/(B-1);
   { - непосредственно по формуле (1.2)}
  Y2:=(Power(A/2,X)-Log10(A/2+1))/(IntPower(A/2,3)-Sqr(A/2));
   {Вывод вычисленных значений с надписями}
  WriteLn('     Y1             Y2 ');
  WriteLn(Y1:12:7,'  ',Y2:12:7);
  ReadLn;
end.
```

Контрольные вопросы

1. Перечислите стандартные типы данных в Паскале.
2. Укажите отличие данных действительного и целого типов.
3. Какие переменные называют логическими и какие значения они могут принимать?
4. Какие логические операции вы знаете?
5. Что представляет собой условие?
6. Что понимается под символьными данными?
7. Какие данные называют арифметическими?
8. Перечислите правила записи стандартных функций.
9. Перечислите стандартные функции, предназначенные для работы с арифметическими данными?
10. Перечислите стандартные функции, предназначенные для работы с символьными данными?
11. Что представляет собой арифметическое выражение?
12. Перечислите правила записи арифметических выражений.
13. Что представляет собой логическое выражение?
14. В чем отличие арифметического выражения от логического?
15. Каков порядок вычисления значения логического выражения?

3.2. Лабораторная работа № 2

Разработка программ разветвляющейся структуры (*условные операторы* if)

Цель работы: получение навыков разработки алгоритмов и программирования разветвляющихся вычислительных процессов

Задание на лабораторную работу:

Используя разветвляющуюся структуру, составить блок-схему вычисления значения составной функции, имеющей различный вид на разных участках аргумента, затем составить программу, реализующую данный алгоритм (значение аргумента функции вводится с клавиатуры).

Теоретический материал

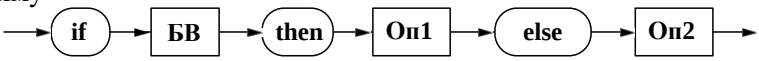
Программой разветвляющейся структуры называют такую программу, в которой, в зависимости от исходных данных, возможны различные последовательности выполнения операторов, причем на любой из них каждый оператор выполняется только один раз.

Для реализации программ или фрагментов программ с разветвляющейся структурой используются *сложные операторы*[8]: *условные операторы* **if** и *операторы выбора* **case**. В этом разделе ограничимся рассмотрением полной формы условного оператора - оператора **if then else** и его сокращенной формы – оператора **if then**.

При использовании условных операторов ветвление алгоритма обусловлено проверками логических выражений (в языке Object Pascal их называют булевскими выражениями), результатом вычисления которых могут быть лишь два значения: «истина» и «ложь». В условных операторах могут использоваться как простейшие булевские выражения, основанные на сравнении выражений других типов, так и сложные, использующие логические операции.

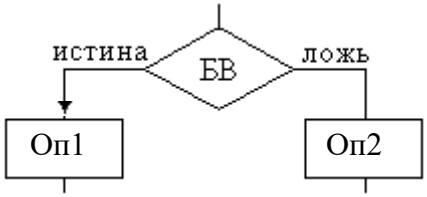
Условные операторы

Оператор **if then else** имеет следующую синтаксическую диаграмму

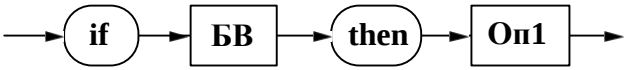


где БВ – *булевское выражение* , значением которого может быть либо «истина», либо «ложь». Оп1 и Оп2 – операторы, каждый из которых может быть пустым оператором. При выполнении оператора **if then else** вначале вычисляется выражение БВ и если результат – «истина», то выполняется оператор Оп1, иначе, то есть если результат имеет значение «ложь», – оператор Оп2. В схемах алгоритмов оператору **if then else** соответствует конструкция

[8] Сложные операторы включают в себя другие операторы



Оператор **if then** имеет синтаксическую диаграмму



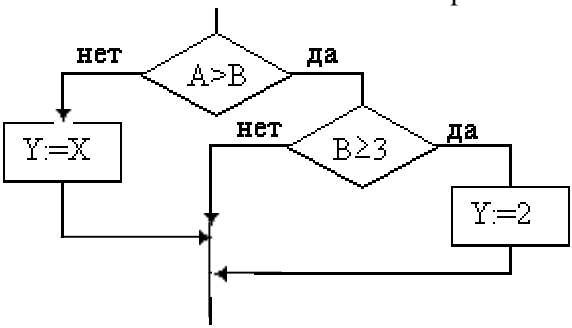
где БВ – булевское выражение, Оп1 – оператор. При выполнении оператора **if then** вначале вычисляется выражение БВ и если результат – «истина», то выполняется оператор Оп1, иначе, то есть если результат – «ложь», управление передается следующему по порядку оператору программы. В схемах алгоритмов оператору **if then** соответствует конструкция

Простейшими булевскими выражениями являются отношения. Знаки отношений записываются следующим образом: >, <, = – так же, как в математике, знаки $\leq, \geq, \neq$ записываются парами символов <=, >=, <> соответственно..

Пример. Требуется записать условный оператор, вычисляющий новое значение Y по заданным значениям A, B, X, Y по формуле

$$Y = \begin{cases} 2, \text{ если } (A > B) \,\&\, (B \geq 3), \\ Y, \text{ если } (A > B) \,\&\, (B < 3), \\ X, - \text{ во остальных случаях} \end{cases}$$

то есть в соответствии с алгоритмом



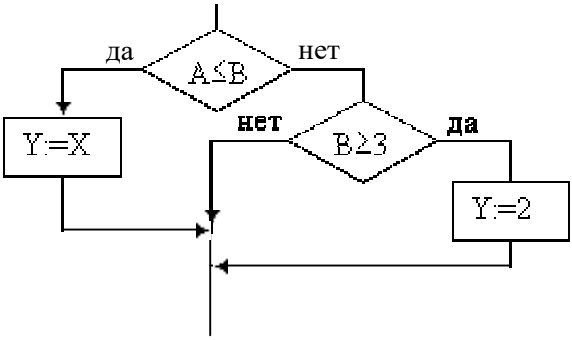
Вот этот оператор:

```
        if  A > B  then
 if  B >= 3  then
   Y:=2
 else
else
 Y:=X;
```

Этому оператору **if then else** подчинен оператор присваивания Y:=X и еще один оператор **if then else**, который, в свою очередь, содержит *пустой оператор* (после первого **else**) и оператор присваивания Y:=2. Необходимость использования else во вложенном условном операторе вытекает из следующего правила: **else** относится к ближайшему предшествующему **if**, у которого нет части **else**. Можно было бы не использовать **else** во вложенном условном операторе, но тогда пришлось бы заключить его в операторные скобки, то есть заменить его оператором

```
begin  if  B >= 3  then  Y:=2  end.
```

Пример. Для функции предыдущего примера можно составить другой алгоритм:



Тогда соответствующим ему оператором if then else будет

```
if  A<=B  then
  Y:=X
else
  if  B>=3  then
    Y:=2
```

и вложенный в него условный оператор естественно использовать в сокращенной форме.

Пример выполнения лабораторной работы

В качестве примера рассмотрим составную функцию вида

$$\mathbf{y} = \begin{cases} \mathbf{2} * \mathbf{x}, & \mathbf{x} > 2.5 \\ \mathbf{x^3} - \mathbf{x}, & \mathbf{0} \leq \mathbf{x} \leq \mathbf{2.5} \\ \mathbf{x} * (\mathbf{sin}(\mathbf{x})), & \mathbf{x} < 0 \end{cases}$$

Как видно из задания, функция вычисляется на трех диапазонах аргумента *x* по трем различным формулам. Составим блок-схему алгоритма решения данной задачи (рис.1.1).

Рис. 1.1. Блок-схема вычисления значения составной функции

Теперь **по данному алгоритму составим программу** на языке Pascal.

Любую программу рекомендуется (но не обязательно) начинать с заголовка.

Program Lab2;

Далее следует раздел описания переменных. В нашем примере понадобятся две переменные для хранения аргумента *x* и значения функции *y*. Так как и аргумент, и сама функция могут принимать дробные значения, то их необходимо описывать вещественным типом данных.

Var x,y: real;

Затем описываем основное тело программы. Как следует из блок-схемы алгоритма, вначале необходимо вывести информационный блок (автор, номер варианта, назначение

программы), после чего вывести текстовую строку, кото-рая подскажет пользователю, что нужно вводить значение аргумента . А затем считать с клавиатуры введенное пользователем число. Не забываем, что тело программы начинается с зарезервированного слова begin.

```
Begin

writeln;

writeln('        Автор – Иванов И.П., студент гр. 2 ПК ');

writeln('        Вариант № 10');

writeln('Программа вводит значение аргумента X и вычисляет зна-чение функции Y');

    writeln('        |2*x                x>2.5');
    writeln('Y       |x^3-x             0<=x<=2.5');
    writeln('        |x*(sin(x))        x<0');
    writeln;

writeln('введите x= ');

readln(x);
```

В данном фрагменте используется два вида оператора вывода: с параметром и без параметра. В первом случае на экран выводится текст, который указан в качестве параметра, а во втором случае – пустая строка (для того чтобы сделать отступ между строками).

После того, как значение X введено оператором ввода readln(x), нужно определить, по какой формуле должна вычисляться функция. Для этого в алгоритме предусмотрены проверки значения аргумента.

```
If x>2.5 then
```

```
Y:=2*x

Else

If x>=0 then y:=x*x*x-x

Else      Y:=x* sin(x);
```

Как видно, в данном фрагменте программы признак конца оператора (точка с запятой) ставится только один раз в самом конце, т. к. условный оператор заканчивается именно на последней строке, перед Else точку с запятой не ставят.

В конце программы нужно вывести результаты вычислений и завершить тело программы служебным словом end с точкой.

```
writeln('Y= ', y:7:3);

readln

end.
```

Последний оператор вывода отличается от всех предыдущих. В данном случае стандартная процедура вывода writeln содержит два параметра, перечисленных через запятую. Первый параметр – это текстовая строка ('Y= '), которую надо вывести на экран. Второй параметр – переменная Y, значение кото-рой требуется вывести в определенном формате, о чем говорят два числа, напи-санные через двоеточие. *Первое число – количество позиций, отводимых под вывод всего числа* (включая знак, целую часть числа, точку и дробную часть числа), а второе – число разрядов после запятой. В нашем случае под вывод всего числа запланировано 7 позиций, из них 3 позиции – под дробную часть, одна позиция – под точку, остается 3 позиции под целую часть и знак. Следует заметить , что если программист указал недостаточное количество позиций под вывод всего числа, то это число будет автоматически увеличено до требуемого для вывода значения.

Для того, чтобы результаты работы программы оставались на экране после выполнения программы, используем оператор ввода без параметров readln.

В этом случае программа выполнит все необходимые действия и будет ожидать от пользователя нажатия клавиши Enter.

Итак, программа написана, ниже приведен ее полный текст.

```
Program Lab2_variant10;

Var x,y: real;

Begin

writeln;

writeln('        Автор – Иванов И.П., студент гр. 2ПК');

writeln('        Вариант № 10');

writeln('Программа вводит значение аргумента X и вычисляет зна-чение функции Y');

        writeln('          |2*x                x>2.5');
        writeln('Y
=                          |x^3-x             0<=x<=2.5');
        writeln('          |x*(sin(x)))       x<0');
        writeln;
    writeln('введите x= ');

    readln(x);

    If x>2.5 then

    Y:=2*x

    Else

    If x>=0 then
```

```
y:=x*x*x-x

Else

Y:=x*sin(x);

writeln('Y= ', y:7:3);

readln

end.
```

Контрольные вопросы

1. Дайте классификацию разветвляющихся алгоритмов.
2. Какие блоки используются для описания разветвляющихся алгоритмов?
3. Какой оператор используется для записи обхода?
4. Какой оператор используется для записи альтернативы?
5. Каково назначение оператора перехода?
6. Сформулируйте ограничения использования оператора перехода.
7. Что такое метка и для чего она предназначена?

3.3. Лабораторная работа № 3

Разработка программ разветвляющейся структуры (*Оператор выбора case)*

Цель работы: получение навыков разработки алгоритмов и программирования разветвляющихся вычислительных процессов

Задание: Составить блок-схему и программу для выполнения действий по индивидуальному заданию, используя оператор выбора. Во всех вариантах предусмотреть проверку корректности исходных данных. При вводе некорректных данных должно выводиться сообщение об ошибке.

Теоретический материал

Оператор выбора case – это, по сути, усложненный оператор if. Но в отличие от условного оператора, когда программа может выполняться одним из двух способов в зависимости от выполнения условия, оператор выбора позволяет выполнять программу одним из нескольких способов в зависимости от значения некоторого выражения. В общем виде этот оператор выглядит так:

case Выражение-селектор **of**

Вариант1: Оператор1;

Вариант2: Оператор2;

...

ВариантN: ОператорN;

[**else** ОператорN1;]

end;

Пояснение: квадратные скобки означают то, что часть else может отсутствовать.

Селектор может быть целочисленным, символьным, булевским или пользовательским (перечисляемым или интервальным).

В качестве вариантов можно применять:

1) Константное выражение такого же типа, как и селектор. Константное вы-ражение отличается от обычного тем, что не содержит переменных и вы-зовов функций, тем самым оно может быть вычислено на этапе компиля-ции программы, а не во время выполнения.
2) Интервал, например: 1..5, 'a'..'z'.
3) Список значений или интервалов, например: 1, 3, 5..8, 10, 12.

Выполняется оператор case следующим образом: вычисляется выражение после слова case и по порядку проверяется, подходит полученное значение под какой-либо вариант, или нет. Если подходит, то выполняется соответствующий этому варианту оператор, иначе – есть два варианта. Если в операторе case записана часть else, то выполняется оператор после else, если же этой части нет, то не происходит вообще ничего.

Пример выполнения лабораторной работы

В качестве примера рассмотрим следующую задачу: пользователь вводит целое число от 1 до 10, программа должна приписать к нему слово «ученик» с необходимым окончанием (нулевое, «а» или «ов»).

Реализуем данный алгоритм в программе. Как и в лабораторной работе № 2, программу начинаем с заголовка и описания переменных. В нашей задаче понадобится всего одна переменная, которая будет принимать значения от 1 до 10. Поэтому целесообразно описать экономичным целочисленным типом byte (принимает значения от 0 до 255, занимает в памяти 1 байт).

```
program SchoolChildren;

var n: byte;
```

Далее следует основное тело программы, которое начинается с информационного блока.

```
begin

writeln;

writeln(’        Автор – Иванов И.П., студент гр. 2 ПК’);

writeln(’        Вариант № 10’);
```

writeln(’Программа вводит целое число от 1 до 10 и приписывает окончание к слову ученик’);

```
writeln;
```

После вывода информации приступаем к вводу исходных данных, т. е.
нужно вывести подсказку пользователю и ввести число учеников.

```
write('Число учеников --> '); readln(n);
```

Далее программа должна вывести на экран введенное число, слово «ученик» и с помощью оператора выбора определить и вывести окончание этого слова.

```
write(n,' ученик');

case n of

2..4:     write('а');

5..10: write('ов');

end;
```

Заметим, что нулевое окончание не требует дополнительного вывода, по-этому если пользователь введет число 1, программа выведет слово «ученик», и не выберет никакой из вариантов оператора case.

Но представим ситуацию, если пользователь вводит число, выходящие за границы диапазона от 1 до 10, например, 12. В этом случае программа выдаст некорректный результат. Так как значение 12 не подходит ни под какой вариант оператора выбора, то слово «ученик» будет написано с нулевым окончанием (как в случае со значением 1). На экране появится текст «12 ученик», хотя должно быть «12 учеников».

Для устранения этой «слабости» программы добавим проверку введенного пользователем числа, и если это число входит

в допустимый диапазон, тогда нужно будет выполнить предыдущий фрагмент, в противном случае – выдать сообщение об ошибке.

```
if (n>=1) and (n<=10) then

begin

write(n,' ученик');

case n of

2..4:     write('а');

5..10: write('ов');

end;

end

else

writeln(' ошибка ввода!');
```

В данном случае условный оператор содержит, во-первых, сложное условие, составленное из двух условных выражений с помощью логической связки and (логическое И), и, во-вторых, при выполнении условия должно быть вы-полнено два оператора (write и case), поэтому они заключаются в оператор-ные скобки (begin end).

Полный текст программы будет выглядеть следующим образом:

```
program SchoolChildren;

var n: integer;

begin

writeln;
```

```
writeln(’ Автор – Иванов И.П., студент гр. 2ПК’);

writeln(’ Вариант № 10’);

writeln(’Программа вводит целое число от 1 до 10 и приписывает окончание к слову ученик’);

writeln;

write('Число учеников --> '); readln(n); if (n>=1) and (n<=10) then begin
write(n,' ученик');

case n of

2..4:  write('а');

5..10: write('ов');

end;

end

else

writeln(' ошибка ввода!');

readln;

end.
```

Контрольные вопросы

1. Дайте классификацию разветвляющихся алгоритмов.
2. Каково назначение оператора выбора и в каких случаях он используется?
3. Можно ли алгоритм выбора из многих возможностей записать с помощью оператора условного перехода?

3.4. Лабораторная работа № 4

Разработка программ циклической структуры

Цель работы: овладение теоретическими основами и практическими навыками программирования вычислительных процессов циклической структуры.

Задание: Составить блок-схему и программу для заполнения таблицы значений функции *y = f(x)* на отрезке с указанным шагом изменения аргумента. Вид функции задается в лабораторной работе № 2. Значение функции выводить с точностью до тысячных долей.

Результат вывести в следующем виде:

```
----------------------------------------

!        X        !        y = f(x) !

----------------------------------------
```

Теоретический материал

Программой циклической структуры называют такую программу, в которой операторы могут повторно, при изменяющихся значениях **переменных** выполняться несколько раз, образуя *цикл*. Различают следующие виды циклов (для их организации используются специальные сложные операторы - *операторы циклов)*:

цикл с заданным числом повторений или *цикл с параметром* (операторы цикла **for**: оператор **for to** и оператор **for downto**),

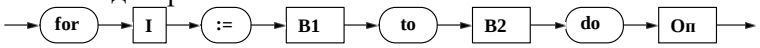
цикл с предусловием (оператор цикла **while**),

цикл с постусловием (оператор цикла **repeat until**).

В циклах можно выделить управляющие части, определяющие начало и условия выполнения цикла, и части из одного или нескольких операторов (*тело цикла*), выполняющие необходимые преобразования данных. Цикл называют *простым*, если в его теле нет других циклов.

Циклы с параметром

Структура оператора цикла **for to** описывается синтаксической диаграммой



где используются следующие обозначения:

I – *параметр цикла* - переменная *ординального (порядкового)*, в частности целого, типа,

B1 и B2 – выражения того же типа, что и параметр цикла, или совместимые с ним,

Оп – оператор, выполняемый внутри цикла.

Часть, предшествующая оператору Оп, - *заголовок цикла* является управляющей, а сам оператор Оп – *телом цикла*. Оператор Оп будет последовательно выполняться при автоматическом увеличении с минимальным шагом значения параметра цикла I от значения B1 до значения B2 включительно (для целых типов шаг равен 1). При B1>B2 оператор Оп не будет выполняться вообще.

Например, в цикле

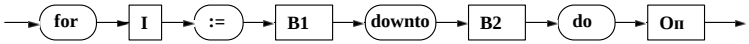
```
for I:=0 to 6 do
  WriteLn(I*10:2,Sin(I/18*Pi):8:2);
```

оператор WriteLn будет выполняться 7 раз при I, изменяющемся от 0 до 6 с шагом 1.

На экран будет выведена таблица, в первом столбце которой будут целые числа 0, 10, 20, …, 60, представляющие величины углов в градусах, а во втором – соответствующие им значения синуса:

```
 0  0.00
10  0.17
20  0.34
30  0.50
40  0.64
50  0.77
60  0.87
```

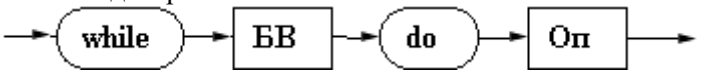
Структура оператора цикла **for downto** описывается синтаксической диаграммой



а его работа отличается от оператора **for to** тем, что параметр цикла I не увеличивается, а уменьшается от B1 до B2, а оператор Оп не будет выполняться вообще при B1<B2.

Цикл с предусловием

Структура оператора цикла **while** описывается синтаксической диаграммой



где используются следующие обозначения:

БВ – булевское выражение,

Оп – оператор, выполняемый внутри цикла (тело цикла).

Заголовок цикла – конструкция, предшествующая оператору ОП, управляет выполнением цикла следующим образом: оператор Оп будет последовательно выполняться, пока выражение БВ имеет значение True, или не будет выполняться вообще, если до выполнения оператора **while** БВ имеет значение False.

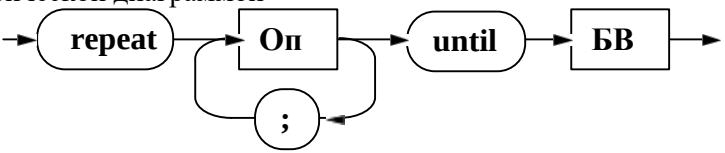
Например, фрагмент программы

```
        N:=0;
while N<=60 do
        begin
          WriteLn(N:2,Sin(N/180*Pi):8:2);
  N:=N+10
end
```

буде выполнять ту же работу, что и оператор **for to** в предыдущем примере.

Цикл с постусловием

Структура оператора цикла **repeat until** описывается синтаксической диаграммой



Внутри такого цикла может находиться произвольное число операторов Оп, которые будут выполняться один или более раз до получения булевским выражением БВ значения True. Например, такую же таблицу, что и в первом примере с применением оператора **for to**, будет выводить следующий фрагмент программы:

```
        N:=0;
repeat
  WriteLn(N:2,Sin(N/180*Pi):8:2);
  N:=N+10
until N>60;
```

В приведенных примерах переменные I и N изменялись по закону арифметической прогрессии. Нередко возникает необходимость иметь в цикле переменную – *дополнительный параметр цикла*, изменяющуюся по требуемому закону. Сделать это можно так: до входа в цикл этой переменной дается начальное значение, а внутри цикла значение переменной изменяется нужным образом с помощью оператора присваивания.

В следующем фрагменте программы

```
        R:=5;
for K:=1 to N do
        begin
  .          .          .          .          .          .
   R:=R*1.2;
  .          .          .          .          .          .
end;
```

R - дополнительный параметр, который в цикле при N равном 4 будет последовательно получать значения 5; 6; 7,2; 8,64, изменяясь по закону геометрической прогрессии умножением предыдущего значения на 1,2.

Выход из цикла по условию, объявленному в его управляющей части, будем называть естественным. При этом для циклов с параметром (организованным операторами **for**) рекомендуется считать, что значение параметра становится неопределённым.

Существует возможность и досрочного выхода из любого цикла, организованного рассмотренными операторами, либо с помощью оператора безусловного перехода **goto** (их мы не будем использовать), либо с помощью оператора **break**. В этом случае текущее значение параметра цикла **for** сохраняется (считается

определённым) и его можно использовать в дальнейших вычислениях.

В теле любого из рассмотренных циклов допускается использовать оператор **continue**. Его действие сводится к тому, что сразу происходит переход к очередному выполнению тела цикла (в циклах **for** с очередным значением параметра), или выход из цикла, если выполнено условие его завершения.

Простейшими примерами применения операторов цикла на практике являются программы вычисления значений функций при изменяющихся значениях аргумента и вывода данных в виде таблиц с заголовками. В качестве аргумента обычно выступает переменная – дополнительный параметр, изменяющаяся в цикле по требуемому закону. В циклах **while** и **repeat** эта переменная используется в условиях завершения цикла.

Пример выполнения лабораторной работы

Для примера рассмотрим функцию

$$\mathbf{y} = \begin{cases} \mathbf{2} * \mathbf{x}, & \mathbf{x} > 2.5 \\ \mathbf{x^3} - \mathbf{x}, & \mathbf{0} \leq \mathbf{x} \leq \mathbf{2.5} \\ \mathbf{x} * (\mathbf{sin}(\mathbf{x})), & \mathbf{x} < 0 \end{cases}$$

аргумент которой изменятся в интервале [-π; π] с шагом π/5.

Составим блок-схему алгоритма (рис. 3.1).

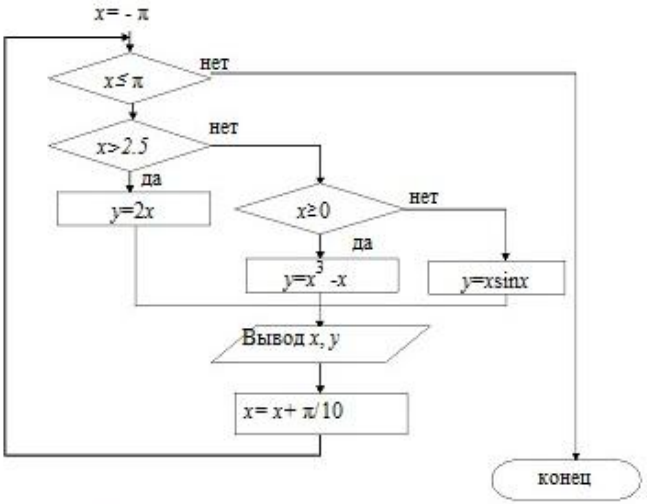


Рис. 3.1. Блок-схема задачи табулирования функции

Программную реализацию начинаем с описания переменных. Как видно из блок-схемы алгоритма, в программе понадобятся две переменные: *x* – для хранения значения аргумента, *y* – для хранения значения функции. Поскольку обе переменные могут иметь дробные значения, описываем их вещественным типом. Из соображений минимизации времени выполнения программы целесообразно шаг приращения аргумента *x* задать как величину постоянную, значение которой будет вычисляться единожды при описании константы. Заметим, что в языке Pascal имеется стандартная константа Pi, которую мы используем при вычислении шага.

```
Const h=Pi/5;

Var x,y:real;
```

Тело программы начинается с уже знакомого информационного блока и оператора, печатающего «шапку»

будущей таблицы. Затем задается начальное значение аргумента функции, которое определяется нижней границей изменения *x*.

```
x:=-Pi;
```

Далее организуется цикл, в ходе выполнения которого будут вычисляться и печататься значения аргумента и функции. Поскольку в данном случае заранее неизвестно число повторений цикла, будем использовать цикл с предусловием, в котором условием продолжения цикла будет $x \leq \pi$, а телом цикла – составной оператор, включающий группу операторов вычисления значения составной функции (см. лабораторную работу № 2) и оператор, увеличивающий текущее значение аргумента на величину шага.

```
While x<=Pi do
Begin
If x>2.5 then
Y:=2*x
Else
If x>=0 then
y:=sqr(x)*x-x {sqr(x) – возведение x в квадрат} Else
Y:=x* sin(x);
Writeln(x:7:3,y:10:3);
x:=x+h
end;
```

После выхода из цикла остается задержать результаты работы на экране с помощью пустого оператора ввода и завершить программу.

```
Readln

End.
```

Теперь соберем все фрагменты в единую программу.

```
Program Lab4;

Const h=Pi/5;

Var x,y:real;

Begin
writeln;
writeln('                    Автор – Иванов И.П., студент гр. 2ПК');
writeln('                    Вариант № 10');
writeln(' Программа для заполнения таблицы значений функции');
writeln('          |2*x                  x>2.5');
writeln('Y= |x^3-x                      0<=x<=2.5');
writeln('          |x*(sin(x))           x<0');
writeln('на отрезке [-Pi; Pi] с шагом Pi/5');
writeln;
x:=-Pi;
While x<=Pi do
Begin
    If x>2.5 then
    Y:=2*x
    Else
    If x>=0 then
        y:=sqr(x)*x-x
```

```
        Else
            Y:=x* sin(x);
        Writeln(x:7:3,y:10:3);
        x:=x+h
    end;
    Readln

    End.
```

Контрольные вопросы

1. Дайте определение арифметических итерационных циклов.
2. Перечислите особенности циклических процессов с предварительной проверкой условия.
3. Перечислите особенности циклических процессов с последующей проверкой условия.
4. В чем состоит отличие операторов цикла с "предусловием" и с "постусловием"?
5. Будет ли выполняться циклическая часть программы, если логическое выражение является ложным с самого начала в предложении WHILE?
6. Будет ли выполняться циклическая часть программы, если логическое выражение истинно с самого начала в предложении REPEAT?
7. Дайте определение арифметического цикла с известным числом повторений.
8. Какой тип данного может иметь параметр цикла?
9. Обязательно ли реализовывать арифметический цикл с известным числом повторений в программе с помощью предложения FOR?
10. В каких случаях необходимо использовать вложенные циклы?
11. Обязательно ли использование однотипных операторов цикла при организации вложенных циклов?
12. Какова максимальная глубина вложенности циклов?
13. Можно ли изменять программным путем параметр цикла циклической части предложения For?

14. В чем отличия итерационных циклов и циклов с фиксированным числом повторений.
15. В чем состоят преимущества использования операторов цикла?
16. Укажите основные правила организации вложенных циклов.
17. Возможен ли выход из внутреннего цикла до его полного завершения ?

3.5.Лабораторная работа № 5

Обработка одномерных массивов

Цель работы: овладение практическими навыками работы с векторами и матрицами при программировании

Задание: составить программу заданной обработки массива целых чисел. В процессе обработки использовать перестановки элементов внутри массива, не создавая новых массивов. Заполнение исходного массива организовать с помощью генератора случайных чисел, если иное не предусмотрено вариантом задания. Исходный и обработанный массив выводить на экран.

Теоретический материал

При выполнении лабораторных работ 5 и 6 будет использоваться структурированный тип данных *массив*, который представляет собой фиксированное количество упорядоченных однотипных компонент, снабженных индексами. Он может быть *одномерным* и *многомерным* . В лабораторной работе № 5 будут использоваться одномерные массивы, в лабораторной работе № 6 – двумерные.

Тип-массив описывается в разделе описания типов следующим образом:

type <имя типа> = **array**[<тип индекса(индексов)>] **of** <тип компонент>;

Размерность массива может быть любой, компоненты массива могут быть любого (в том числе и структурированного) типа, индекс (индексы) может (могут) быть только интервального или перечисляемого типа. При употреблении в качестве типа индекса типа *Integer* или *Word* можно использовать только его диапазон.

При описании типа индекса (индексов) можно использовать константы, которые должны быть определены до определения типа.

Определенный в разделе описания типов тип-массив можно использовать для описания переменных и типизированных констант. Тип-массив можно вводить непосредственно и при определении соответствующих переменных или типизированных констант.

При задании значений константе-массиву компоненты указываются в круглых скобках и разделяются запятыми , причем, если массив многомерный, внешние круглые скобки соответствуют левому индексу, вложенные в них круглые скобки – следующему индексу и т. д. При этом все компоненты массива должны быть заполнены.

Доступ к компонентам массива осуществляется указанием имени массива, за которым в квадратных скобках помещается значение индекса (индексов) компоненты. В многомерных массивах значения индексов перечисляются через запятую.

Для работы с массивом как с единым целым используется идентификатор массива без указания индексов в квадратных скобках. Массив может участвовать только в операциях отношения («=», «<>») и в операторе присваивания.

Массивы, участвующие в этих действиях, должны быть идентичны по структуре, то есть иметь одинаковые базовые типы и типы индексов.

Задание значений переменной типа массив может осуществляться с клавиатуры, путем генерирования случайным образом либо иным способом. Когда значения элементов задаются с клавиатуры или генерируются случайным образом, необходима организация цикла, в котором последовательно происходит обращение к каждому элементу массива. Вывод значений элементов массива на экран или в файл также происходит в цикле.

При написании программ необходимо следить за тем, чтобы значения индексов не превышали границ, указанных при объявлении массива, так как выход индекса за границы массива

приводит к сбою в работе программы. Контроль значений индексов массивов можно организовать при помощи директивы компилятора {$R+}, которая приводит к проверке всех индексных выражений на соответствие их значений диапазону индекса.

Вычисление суммы и произведения

Прием *накопления суммы* часто используется в различных приложениях. Суммируемыми элементами могут быть элементы массива. В любом случае накопление суммы производится в цикле по рекуррентной формуле S=S+ Y, где S- промежуточная сумма, Y- слагаемое. Перед циклом в большинстве случаев начальное значение S должно быть равным нулю.

Фрагмент программы для вычисления суммы первых n элементов массива Y будет иметь вид:

```
S:=0;
for i:=1 to n do
  S:= S+Y[i];
```

Фрагмент программы для вычисления среднего первых n значений функции Y(x), вычисляемых при значениях x, изменяемых от x0 с шагом dx будет иметь вид:

```
S:=0;
x:=x0;
for i:=1 to n do
          begin
   S:=S+Y(x)
   x:=x+dx
 end;
 S:=S/n;
```

Вычисление произведения выполняется аналогичным образом, но перед циклом в большинстве случаев начальное значение P должно быть равным единице P:=1, а в цикле произведение накапливается по рекуррентной формуле P:=P·Y.

Пример. Вычислить значение суммы элементов массива X(n), n≤30.

```
const
  nMax=30;
```

```
var S:Real;
  X: array [1..nMax] of  Real;
  i:Integer;
begin
  Write(' Введите количество элементов: ');
  ReadLn (n);
  WriteLn(' Введите элементы массива: ');
  for i:=1 to n do
    Read (X[i]);
  ReadLn;
  S:=0;  //Начальное значение суммы
  for i:=1 to n do
    S:= S+X[i]; //Накопление суммы
  WriteLn('Сумма элементов массива равна ', S:6:2);
. . . . . .
end.
```

Нахождение наибольшего или наименьшего значения

Поиск наибольшего или *наименьшего* значения может выполняться в имеющемся массиве или при вычислении значения функции.

При поиске в массиве начальному значению наибольшего Xmax (начальному значению наименьшего Xmin) присваивается значение первого элемента массива, а затем в цикле для очередного элемента массива X_i, i=2, 3,…,n, производится проверка: если X_i>Xmax
(X_i < Xmin), то переменной Xmax (переменной Xmin) присваивается значение X_i, иначе значение Xmax (Xmin) не изменяется.

Поиск максимального и минимального значений в массиве представляют следующие фрагменты программ

```
Xmax= X[1];
for i:=2 to n do
  if X[i] > Xmax then
    Xmax:=X[i];
WriteLn('Xmax=', Xmax:6:2);
```

```
Xmin = X[1];
for i:=2 to n do
  if X[i] < Xmin then
    Xmin:=X[i];
WriteLn('Xmin=',Xmin:6:2);
```

Наряду с нахождением максимального/минимального значения в массиве может потребоваться нахождение значения индекса элемента массива, при котором оно достигается. Нахождение указанных значений при поиске максимума поясняют следующие фрагменты программ.

Поиск максимального элемента
и соответствующего значения аргумента массива
и его индекса

```
Xmin = X[1];
Imin:=1;
for i:=2 to n do
if X[i] > Xmax then
begin
Xmax:=X[i];
Imax:=i;
end;
WriteLn('Xmax = ', Xmax:6:2);
WriteLn('Imax = ', Imax);
```

Пример выполнения лабораторной работы

Для примера рассмотрим следующую задачу. Дан одномерный массив целых чисел. Удалить все отрицательные элементы, расположенные справа от максимального.

Прежде всего, определим алгоритм решения данной задачи. Вначале надо найти максимальный элемент среди элементов массива и запомнить его индекс. Затем, начиная со следующего за максимальным элементом, проверять элементы массива на знак, и если число оказывается отрицательным, то удаляем этот элемент из массива путем «сдвига» элементов массива влево и уменьшаем длину массива. При этом следует заметить, что в памяти переменная типа массив будет занимать столько же места, как и до удаления элементов.

В графическом виде этот алгоритм будет блок-схему, представленную на рис. 5.1.

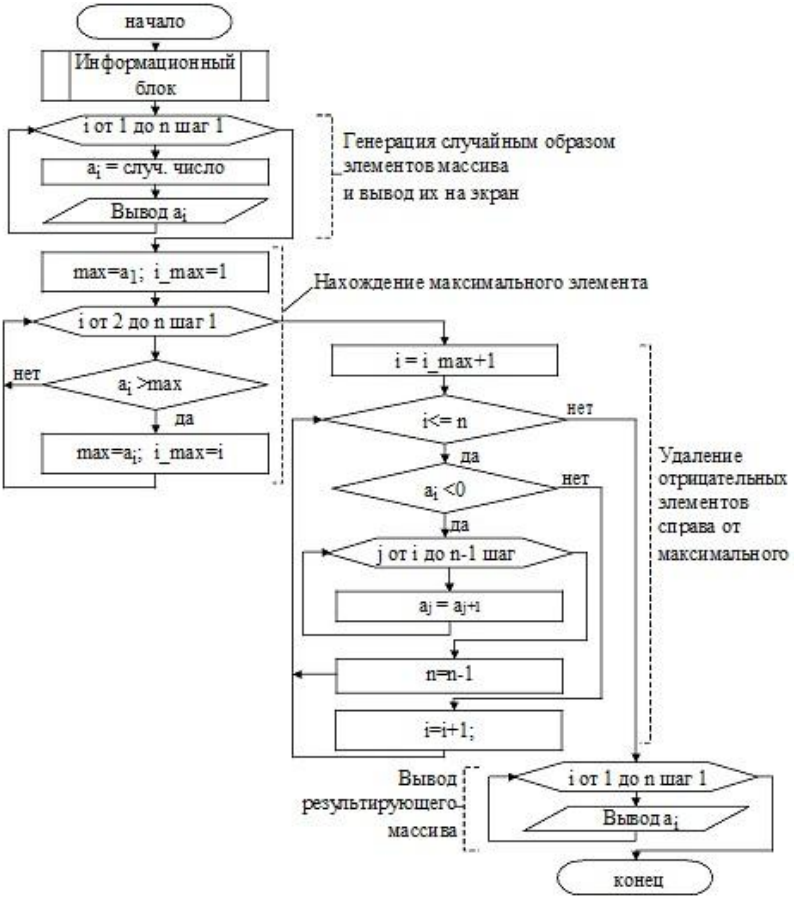


Рис. 5.1. Блок-схема алгоритма удаления из массива отрицательных элементов, расположенных справа от максимального элемента массива

Теперь пишем код программы.

В разделе описания типов определим тип-массив:

type arr=array[1..15] of integer;

В разделе описания переменных описываем необходимые переменные:

```
var a:arr;          {массив}

i,j,n: byte;        {индексы, длина массива}

i_max: byte;        {индекс максимального элемента массива}

max:integer;        {максимальный элемент}
```

Далее описываем основное тело программы. Как отмечено на рисунке, алгоритм состоит из нескольких частей. Первая часть – генерация исходного массива:

```
begin

writeln;

writeln('          Автор – Иванов И.П., студент гр. 2ПК');

writeln('          Вариант № 10');

writeln(' Дан одномерный массив целых чисел.');
writeln(' Удалить все отрицательные элементы, расположенные справа от максимального ');

writeln;

{$R+}  {Включаем контроль значений индексов}

clrscr;

n:=15;

randomize;

writeln('Исходный массив:');

for i:=1 to n do  {генерация элементов массива}

begin
```

```
a[i]:=-20+random(41); {в диапазоне [-20; 20]} write(a[i]:4);

end;

writeln;
```

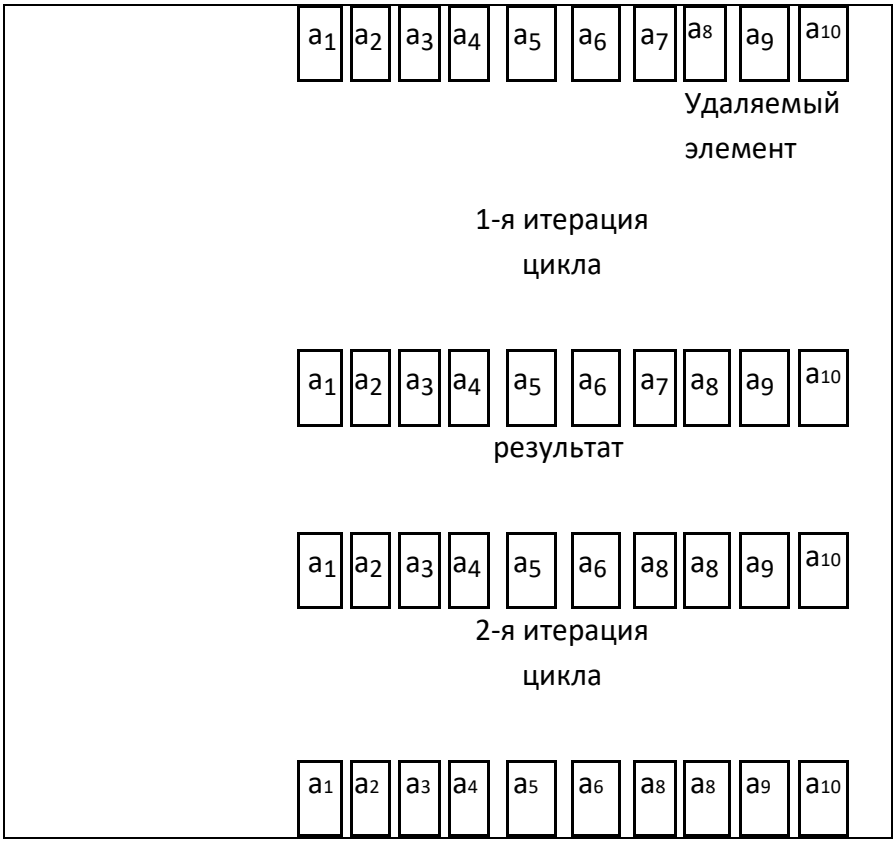
Генератор случайных чисел активизируется командой randomize. Далее в цикле случайным образом задаются элементы массива. Функция random(41) возвращает случайное число в диапазоне от 0 до 40. Получается, что минимально возможное значение элементов массива будет равно -20 (к -20 прибавить 0, сгенерированный функцией random), максимально возможное – 20 (-20 плюс число 40, сгенерированное функцией random). Сразу же выводим значение элемента массива на экран. В цикле все элементы будут выводиться в строчку, т. к. используется команда вывода write. После генерации и вывода всех элементов массива осуществляется переход на новую строку.

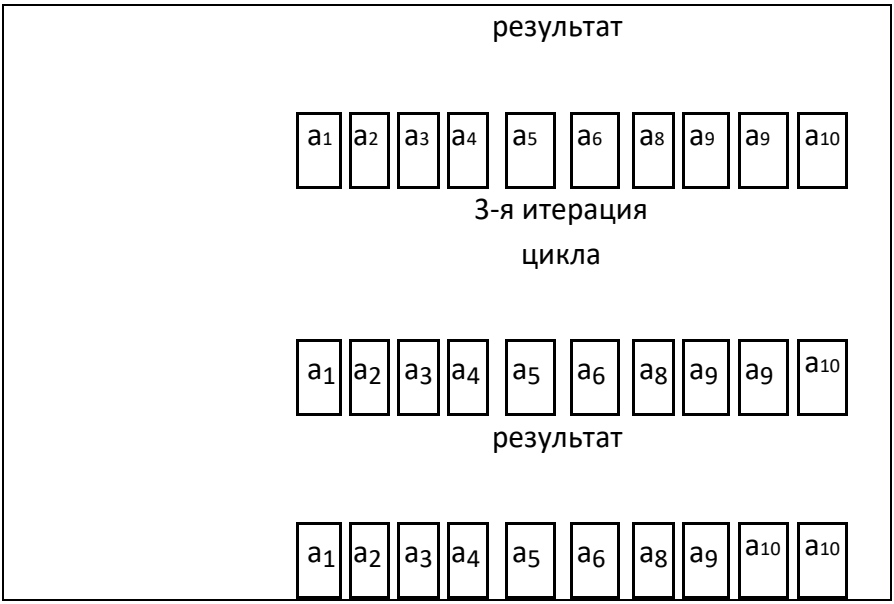
Вторая часть алгоритма – поиск максимального элемента. Предположим, что первый элемент – максимальный, тогда запомним значение первого элемента в переменной max, а в переменную i_max, где будет запоминаться положение максимального элемента, запишем 1. Далее организуется цикл, в котором ищется элемент, больший чем max.

```
max:=a[1];

i_max:=1;

for i:=2 to n do

if a[i]>max       then

begin

max:=a[i];

i_max:=i;

end;
```

Далее, начиная с элемента, следующим за максимальным , организуется цикл проверки элементов массива на знак (важно заметить, что в данном случае нужно использовать цикл while, т . к. длина массива будет изменяться , и, соответственно, число итераций цикла также будет меняться) . Если число оказывается отрицательным, то удаляем этот элемент из массива, для чего организуется еще один цикл, в котором на место удаляемого записывается стоящий следом элемент.

Внутренний цикл сдвига можно проиллюстрировать на примере удаления седьмого элемента из массива, содержащего 10 элементов (рис. 5.2).





В программе удаление элементов, находящихся справа от максимального, будет выглядеть следующим образом:

```
i:=i_max;
while i<=n do
begin
if a[i]<0 then
begin
for j:=i to n-1 do
a[j]:=a[j+1];
n:=n-1;
end
```

```
else

i:=i+1;

end;
```

Еще одна тонкость данного фрагмента состоит в том, что переход к следующему элементу (i:=i+1) для проверки осуществляется только тогда, когда эле-мент не удовлетворяет условию. Если же i-й элемент окажется отрицательным, то

после удаления на его месте окажется i+1-й элемент, который и нужно проверять на следующей итерации цикла.

Остается вывести результирующий массив на экран.

```
writeln('Результат обработки:');

for i:=1 to n do

write(a[i]:4);

readln;

{$R-} {Выключаем контроль значений индексов} end.
```

Ниже представлен полный текст программы.

```
program lab5_var10;


type arr=array[1..15] of integer;

var a:arr;          {массив}

i,j,n: byte;        {индексы, длина массива}

i_max: byte;        {индекс максимального элемента массива}
```

```
max:integer;      {максимальный элемент}

begin

writeln;

writeln('          Автор – Иванов И.П., студент гр. 2ПК');

writeln('          Вариант № 10');

writeln(' Дан одномерный массив целых чисел.');

writeln(' Удалить все отрицательные элементы, расположенные справа от максимального ');

writeln;

{$R+} {Включение контроля значений индексов}
clrscr; {Очистка экрана}
n:=15;

randomize;

writeln('Исходный массив:');

for i:=1 to n do  {генерация элементов массива}

begin

a[i]:=-20+random(41); {в диапазоне [-20; 20]}
 write(a[i]:4);

end;

writeln;

max:=a[1];
```

```
        i_max:=1;

        for i:=2 to n do {поиск максимального элемента массива}
        if a[i]>max then

        begin

        max:=a[i];

        i_max:=i;
        end;
        i:=i_max+1;
        While i<=n do                    {цикл для проверки элементов на
знак }
        Begin

        If a[i]<0 then {если элемент отрицательный, то}
        Begin
        for j:=i to n-1 do  {цикл удаления i-го элемента}
        a[j]:=a[j+1];
        n:=n-1; {уменьшение длины массива}
        end
        else
        i:=i+1; {иначе переход к следующему элементу}
        end;

        writeln('Результат обработки:');

        for i:=1 to n do

        write(a[i]:4);

        readln;

        {$R-}   {Выключение контроля значений индексов}

        end.
```

3.6.Лабораторная работа № 6

Обработка двумерных массивов

Цель работы : овладение практическими навыками работы с векторами и матрицами при программировании

Задание: составить программу заданной обработки матрицы целых чисел. В процессе обработки использовать перестановки элементов внутри массива, не создавая новых массивов. Заполнение исходного массива организовать с помощью генератора случайных чисел. Исходный и обработанный массив выводить на экран.

Во всех вариантах работать с ЗАДАНОЙ матрицей, не создавая дополнительных массивов и матриц (кроме случаев, где это предусмотрено вариантом задания).

Теоретический материал

Матрица, по сути, является двумерным массивом элементов, поэтому для работы с матрицей будем использовать двумерный массив. Правила создания, обработки двумерных массивов, в общем, остаются такими же, как и для одномерных массивов. Отличие состоит в количестве индексов.

Объявление двумерного массива можно осуществлять двумя способами. Первый предполагает объявление одномерного массива, а затем использование этого типа для объявления типа-массива массивов:

```
Type mas = array[1..10] of integer; matr = array[1..5] of mas;

Var a: matr;
```

Однако проще сразу объявить тип-массив, описывающий матрицу, состоящую из 5 строк и 10 столбцов:

```
Type matr = array[1..5,1..10] of integer;
Var a: matr;
```

Доступ к элементам двумерного массива также может осуществляться двумя способами:

a[[1],[1]] или гораздо проще a[1,1]

В обоих случаях первый индекс указывает номер строки, второй – номер столбца матрицы.

В памяти элементы матрицы располагаются последовательно:

a[1,1], a[1,2],…, a[1,10], a[2,1], a[2,2],…, a[5,10]

однако последовательный доступ к элементам матрицы может быть организован как в порядке расположения их в памяти, так и иным способом.

Для организации доступа последовательно ко всем элементам матрицы организуются два цикла, причем порядок «вложения» циклов определяется тем, как осуществляется перебор элементов: по столбцам или по строкам. Чаще используется проход по матрице по строкам, т. е. сначала в первой строке перебираются элементы всех столбцов, затем – во второй строке и т. д. В этом случае «внешним» будет цикл по строкам (т. е. по индексу i), а вложенным – цикл по столбцам (т. е. по j). Ниже представлен пример заполнения матрицы значениями, вводимыми с клавиатуры, причем элементы строк матрицы могут вводиться в строчку через пробел, а для ввода значений новой строки надо нажать клавишу <Enter>.

```
for i:=1 to 5 do

begin

for j:=1 to 10 do

read(a[i,j]);

readln

end;
```

Обработка и вывод матрицы осуществляется аналогично.

Пример выполнения лабораторной работы

Для примера рассмотрим следующую задачу. Дана матрица A(n*m) целых положительных двузначных чисел. Минимальные элементы строк заменить нулями. Исходную матрицу сгенерировать случайным образом.

Программу, как обычно, начнем с описания констант, типов и переменных. Размеры матрицы (количество строк и столбцов) в ходе выполнения программы не будут изменяться, поэтому они объявляются как константы и задаются в разделе описания констант.

```
const n=5;

m=7;
```

При описании типа эти константы будут использованы для определения диапазонов изменения индексов матрицы.

```
type matr=array[1..n,1..m] of byte;
```

Тип элементов массива выбран byte (возможные значения этого типа данных [0..255]), т. к. по условию задачи элементы матрицы положительны и двузначны, т. е. находятся в интервале [10..99]. Это уточнение пригодится и при заполнении матрицы.
В разделе описания переменных объявляются переменные для хранения матрицы, индексов, а также минимального значения. Индексы и переменная, которая будет хранить значение минимального элемента, также описываются целочисленным типом byte.

```
var a:matr;

i,j,j_min,min:byte;
```

Далее следует тело программы. Информационный блок подробно рассматриваться не будет. Блок формирования случайным

образом элементов матрицы и вывода исходной матрицы выглядит следующим образом:

```
writeln('Исходная матрица');

for i:=1 to n do

begin

for j:=1 to m do

begin

a[i,j]:=random(90)+10;

write(a[i,j]:3);

end;

writeln;

end;
```

До начала цикла по формированию матрицы выводится надпись «Исходная матрица». Далее в цикле формируются элементы сначала первой строки и выводятся на экран в строчку (оператор write), затем осуществляется переход на новую строку (writeln) и начинается новая итерация цикла по i.

Следующий шаг – поиск минимального элемента в каждой строке и замена его на ноль.

```
for i:=1 to n do

begin

j_min:=1;

min:=a[i,1];

for j:=1 to m do
```

```
if a[i,j]<min then

begin

min:=a[i,j];

j_min:=j

end;

a[i,j_min]:=0;;

end;
```

Здесь также используется два цикла. Тело «внешнего» цикла содержит операторы присваивания начальных значений переменным j_min,min и цикл перебора элементов i-й строки по столбцам.

Тело «внутреннего» цикла состоит лишь из одного условного оператора, поэтому в нем не нужны операторные скобки. А вот внутри условного оператора – несколько операторов, поэтому операторные скобки в нем обязательны!

После завершения цикла по столбцам будет найден минимальный элемент i-й строки и его положение (j_min), и произведена замена его на ноль. Далее программа возвращается во «внешний» цикл и происходит аналогичная обработка следующей строки матрицы. Выход из внешнего цикла означает, что обработка матрицы завершена. Остается вывести результат на экран, снабдив выведенную матрицу соответствующим текстом.

```
writeln('Результирующая матрица');
for i:=1 to n do
begin

for j:=1 to m do

write(a[i,j]:3);
```

```
writeln;

end;
```

Как можно заметить, этот фрагмент очень похож на блок формирования и вывода исходной матрицы . Отличие состоит в том, что тело «внутреннего» цикла содержит всего один оператор вывода, поэтому он не заключается в операторные скобки begin..end.

Итак, соединив фрагменты в единое целое, получим текст программы:

```
program lab_6;

const n=5;

m=7;

type matr=array[1..n,1..m] of byte;

var a:matr;

i,j,j_min,min:byte;

begin

writeln;

writeln('          Автор – Иванов И.П., студент гр. 2 ПК');

writeln('          Вариант № 10');

writeln(' Дана матрица A(n*m) целых положительных двузначных чисел.');

writeln(' Минимальные элементы строк заменить нулями.');
writeln(' Исходную матрицу сгенерировать случайным.');
writeln;
{$R+} {Включение контроля значений индексов}
 clrscr; {Очистка экрана} writeln('Исходная матрица');
```

```
for i:=1 to n do
begin
for j:=1 to m do
begin
a[i,j]:=random(90)+10;
write(a[i,j]:3);
end;
writeln;
end;
for i:=1 to n do
begin
j_min:=1;
min:=a[i,1];
for j:=1 to m do
if a[i,j]<min then
begin
min:=a[i,j];
j_min:=j
end;
a[i,j_min]:=0;;
```

```
end;

writeln;

writeln('Результирующая матрица');

for i:=1 to n do

begin

for j:=1 to m do

write(a[i,j]:3);

writeln;

end;

readln;

{$R-}

end.
```

Контрольные вопросы

1. Дайте определение массива.
2. Напишите формулу для вычисления объема памяти, занимаемого массивом.
3. В оперативной памяти любой многомерный массив располагается линейно. Выведите формулу, по которой процессор определяет порядковый номер элемента многомерного массива в памяти при известных значениях индексов.
4. Как организовать ввод матрицы размером N*M элементов ?
5. Как организовать вывод матрицы?

3.7.Лабораторная работа № 7

Разработка программ с использованием алгоритмов сортировки

Цель: приобрести опыт разработки программ с использованием алгоритмов сортировки

Задание на лабораторную работу:

1. Для каждого из примеров, демонстрирующих методы сортировки, скомпилируйте программу.
2. Подготовьте таблицу исходных данных.
3. Результат работы программы представьте преподавателю в электронном виде.
4. Заполните массив случайными числами. Для этого сделайте изменения в программе.
5. Результат работы программы представьте преподавателю в электронном виде.

Теоретический материал

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа.

Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – это не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться «на том же месте» в исходном массиве. Сортировку массивов принято называть **внутренней** в отличие от сортировки файлов (списков), которую называют **внешней**.

Наиболее популярны:

1. Метод прямого обмена (пузырьковая сортировка).
2. Метод прямого выбора.
3. Сортировка с помощью прямого (двоичного) включения.

4. Шейкерная сортировка (модификация пузырьковой).
5. Метод Д. Шелла , усовершенствование метода прямого включения.
6. Сортировка с помощью дерева, метод HeapSort, Д. Уильямсон.
7. Сортировка с помощью разделения, метод QuickSort, Ч. Хоар, улучшенная версия пузырьковой сортировки.

.

Сортировка включением

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста. Пусть имеется массив ключей a[1], a[2], ..., a[n]. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...) и до тех пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i], или если достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива).

Таблица 2.1 Пример сортировки методом простого включения

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1	8 23 5 65 44 33 1 6
Шаг 2	8 5 23 65 44 33 1 6 5 8 23 65 44 33 1 6
Шаг 3	5 8 23 65 44 33 1 6
Шаг 4	5 8 23 44 65 33 1 6
Шаг 5	5 8 23 44 33 65 1 6 5 8 23 33 44 65 1 6
Шаг 6	5 8 23 33 44 1 65 6 5 8 23 33 1 44 65 6 5 8 23 1 33 44 65 6 5 8 1 23 33 44 65 6

	5 1 8 23 33 44 65 6 1 5 8 23 33 44 65 6
Шаг 7	1 5 8 23 33 44 6 65 1 5 8 23 33 6 44 65 1 5 8 23 6 33 44 65 1 5 8 6 23 33 44 65 1 5 6 8 23 33 44 65

Дальнейшим развитием метода сортировки с включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Мы не будем описывать алгоритм в общем виде, а ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2n элементами алгоритм работает следующим образом. На первой фазе производится сортировка включением всех пар элементов массива, расстояние между которыми есть 2(n-1). На второй фазе производится сортировка включением элементов полученного массива, расстояние между которыми есть 2(n-2). И так далее, пока мы не дойдем до фазы с расстоянием между элементами, равным единице, и не выполним завершающую сортировку с включениями. Применение метода Шелла к массиву, используемому в наших примерах, показано в таблице 2.2.

Таблица 2.2. Пример сортировки методом Шелл

Начальное состояние массива	8 23 5 65 44 33 1 6
Фаза 1 (сортируются элементы, расстояние между которыми четыре)	8 23 5 65 44 33 1 6 8 23 5 65 44 33 1 6 8 23 1 65 44 33 5 6 8 23 1 6 44 33 5 65
Фаза 2 (сортируются элементы, расстояние между которыми два)	1 23 8 6 44 33 5 65 1 23 8 6 44 33 5 65 1 23 8 6 5 33 44 65 1 23 5 6 8 33 44 65 1 6 5 23 8 33 44 65 1 6 5 23 8 33 44 65 1 6 5 23 8 33 44 65
Фаза 3 (сортируются элементы,	1 6 5 23 8 33 44 65 1 5 6 23 8 33 44 65

расстояние между которыми один)	1 5 6 23 8 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65

Обменная сортировка

Простая обменная сортировка (в просторечии называемая "методом пузырька") для массива a[1], a[2], ..., a[n] работает следующим образом. Начиная с конца массива сравниваются два соседних элемента (a[n] и a[n-1]). Если выполняется условие a[n-1] > a[n], то значения элементов меняются местами. Процесс продолжается для a[n-1] и a[n-2] и т.д., пока не будет произведено сравнение a[2] и a[1]. Понятно, что после этого на месте a[1] окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются a[3] и a[2]. И так далее. На последнем шаге будут сравниваться только текущие значения a[n] и a[n-1]. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые "легкие") постепенно "всплывают" к верхней границе массива. Пример сортировки методом пузырька показан в таблице 2.3.

Таблица 2.3. Пример сортировки методом пузырька

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1	8 23 5 65 44 33 1 6 8 23 5 65 44 1 33 6 8 23 5 65 1 44 33 6 8 23 5 1 65 44 33 6 8 23 1 5 65 44 33 6 8 1 23 5 65 44 33 6 1 8 23 5 65 44 33 6
Шаг 2	1 8 23 5 65 44 6 33 1 8 23 5 65 6 44 33 1 8 23 5 6 65 44 33 1 8 23 5 6 65 44 33 1 8 5 23 6 65 44 33 1 5 8 23 6 65 44 33
Шаг 3	1 5 8 23 6 65 33 44 1 5 8 23 6 33 65 44

	1 5 8 23 6 33 65 44 1 5 8 6 23 33 65 44 1 5 6 8 23 33 65 44
Шаг 4	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 5	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 6	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 7	1 5 6 8 23 33 44 65

Метод пузырька допускает три простых усовершенствования. Во-первых, как показывает таблица 2.3, на четырех последних шагах расположение значений элементов не менялось (массив оказался уже упорядоченным). Поэтому, если на некотором шаге не было произведено ни одного обмена, то выполнение алгоритма можно прекращать. Во-вторых, можно запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса. В-третьих, метод пузырька работает неравноправно для "легких" и "тяжелых" значений. Легкое значение попадает на нужное место за один шаг, а тяжелое на каждом шаге опускается по направлению к нужному месту на одну позицию.

На этих наблюдениях основан метод шейкерной сортировки (ShakerSort). При его применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге "всплывает" очередной наиболее легкий элемент, а на другом "тонет" очередной самый тяжелый. Пример шейкерной сортировки приведен в таблице 2.4.

Таблица 2.4. Пример шейкерной сортировки

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1	8 23 5 65 44 33 1 6 8 23 5 65 44 1 33 6 8 23 5 65 1 44 33 6 8 23 5 1 65 44 33 6 8 23 1 5 65 44 33 6 8 1 23 5 65 44 33 6 1 8 23 5 65 44 33 6
Шаг 2	1 8 23 5 65 44 33 6 1 8 5 23 65 44 33 6 1 8 5 23 65 44 33 6 1 8 5 23 44 65 33 6 1 8 5 23 44 33 65 6 1 8 5 23 44 33 6 65
Шаг 3	1 8 5 23 44 6 33 65 1 8 5 23 6 44 33 65 1 8 5 6 23 44 33 65 1 8 5 6 23 44 33 65 1 5 8 6 23 44 33 65
Шаг 4	1 5 6 8 23 44 33 65 1 5 6 8 23 44 33 65 1 5 6 8 23 44 33 65 1 5 6 8 23 33 44 65
Шаг 5	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65

Сортировка выбором

При сортировке массива a[1], a[2], ..., a[n] методом простого выбора среди всех элементов находится элемент с наименьшим значением a[i], и a[1] и a[i] обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] до тех пор, пока мы не дойдем до подмассива a[n], содержащего к этому моменту наибольшее значение. Работа алгоритма иллюстрируется примером в таблице 2.5.

Таблица 2.5. Пример сортировки простым выбором

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1	1 23 5 65 44 33 8 6
Шаг 2	1 5 23 65 44 33 8 6
Шаг 3	1 5 6 65 44 33 8 23
Шаг 4	1 5 6 8 44 33 65 23
Шаг 5	1 5 6 8 33 44 65 23
Шаг 6	1 5 6 8 23 44 65 33
Шаг 7	1 5 6 8 23 33 65 44
Шаг 8	1 5 6 8 23 33 44 65

Сортировка разделением (Quicksort)

Метод сортировки разделением был предложен Чарльзом Хоаром (он любит называть себя Тони) в 1962 г. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть "методом быстрой сортировки - Quicksort".

Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент массива x, после чего массив просматривается слева, пока не встретится элемент a[i] такой, что a[i] > x, а затем массив просматривается справа, пока не встретится элемент a[j] такой, что a[j] < x. Эти два элемента меняются местами, и процесс просмотра, сравнения и обмена продолжается, пока мы не дойдем до элемента x. В результате массив окажется разбитым на две части - левую, в которой значения ключей будут меньше x, и правую со значениями ключей, большими x. Далее процесс рекурсивно продолжается для левой и правой частей массива до тех пор, пока каждая часть не будет содержать в точности один элемент. Понятно, что как обычно, рекурсию можно заменить итерациями, если запоминать соответствующие индексы

массива. Проследим этот процесс на примере нашего стандартного массива (таблица 2.6).

Таблица 2.6. Пример быстрой сортировки

Начальное состояние массива	8 23 5 65 |44| 33 1 6

Шаг 1 (в качестве x выбирается a[5])	`       |--------|`
	8 23 5 6 44 33 1 65
	`          |---|`
	8 23 5 6 1 33 44 65

Шаг 2 (в подмассиве a[1], a[5] в качестве x выбирается a[3])	8 23 |5| 6 1 33 44 65
	`|--------|`
	1 23 5 6 8 33 44 65
	`  |--|`
	1 5 23 6 8 33 44 65

Шаг 3 (в подмассиве a[3], a[5] в качестве x выбирается a[4])	1 5 23 |6| 8 33 44 65 `    |----|` 1 5 8 6 23 33 44 65
Шаг 4 (в подмассиве a[3], a[4] выбирается a[4])	1 5 8 |6| 23 33 44 65 `    |--|` 1 5 6 8 23 33 44 65

Примеры для выполнения лабораторной работы (рассмотрим реализацию в коде некоторых методов сортировки):

1.Метод пузырька (обменная сортировка методом «пузырька)

В алгоритме сортировки методом пузырька сравниваются два соседних элемента. Если они расположены в неправильной последовательности, то выполняется перестановка этих элементов. Сортировка осуществляется путем многократного прохождения по списку элементов. При сортировке по возрастанию элементы с малыми значениями поднимаются вверх в начало списка, подобно пузырькам воздуха в воде.

Более подробно алгоритм.

Алгоритм начинается со сравнения 1-го и 2-го элементов массива. Если элементы расположены не по порядку, то они меняются местами. Этот процесс повторяется со 2-м и 3-им, 3-м и 4-м, и т.д. элементами, пока пара (n-1) –й и n-й элемент не будет отработана.

За один просмотр массива самый большой элемент встанет на старшее (n-е) место.

Далее алгоритм повторяется, причем на p- ом просмотре уже только первые (n-p) элементов сравниваются со своими правыми соседями. Если при очередном просмотре перестановок не было или p=n, то алгоритм окончен.

Программа, реализующая алгоритм пузырьковой сортировки:

```
const n=6;
type vec = array[1..n] of real;
ind = array[1..n] of integer;
var k,kol:integer;
a:vec;
w:real;
p:boolean;
i:integer;
begin

{ввод исходных в данных в массив}
writeln('ВВЕДИТЕ ДАННЫЕ В МАССИВ');
for i:=1 to n do
begin
write (i:2,':');
readln(a[i]);
end;

{МЕТОД ПУЗЫРЬКА}
begin
kol:=1;
repeat
p:=true;
for k:=1 to n-kol do
if a[k]>a[k+1] then
```

```
begin
w:=a[k];
a[k]:=a[k+1];
a[k+1]:=w;
p:=false;
end;
inc(kol);
until p;
end;

{Вывод на экран результата}
writeln('Результат сортировки');
for k:=1 to n do
writeln(a[k]:4);
end.
```

В общем случае для сортировки требуется n – 1 проход по массиву.

Чтобы исключить ненужные проходы, если массив уже полностью или частично отсортирован, используется не цикл с заданным количеством повторений (for kol := 1 to n – 1 do), а оператор repeat с флажком p.

```
{ Заполнение массива случайными числами }
ClrScr; Randomize;
WriteLn(' Значения элементов массива ');
FOR I:=1 TO N DO Begin
A[I]:=Random(50);
Write(A[I]:4);
End;
 WriteLn;
```

2.Метод прямого выбора

Сортировка осуществляется путем многократного прохождения по списку элементов. На каждом (k-ом) проходе находится минимальный элемент с к-го по n- ый элементы, который затем переставляется с к-ым элементом

Программа, реализующая алгоритм этой сортировки:

```
const n=6;
type vec = array[1..n] of real;
ind = array[1..n] of integer;
var k:integer;
a:vec;
w:real;
i,m:integer;
begin

{ввод исходных в данных в массив}
writeln('ВВЕДИТЕ ДАННЫЕ В МАССИВ');
for i:=1 to n do
begin
write (i:2,':');
readln(a[i]);
end;

{МЕТОД ПРЯМОГО ВЫБОРА}
begin
for k:=1 to n-1 do
begin
m:=k;
for i:=k+1 to n do
if a[i]<a[m] then m:=i;
w:=a[m];
a[m]:=a[k];
 a[k]:=w;
end;
end;

{Вывод на экран результата}
writeln('Результат сортировки');
for k:=1 to n do
writeln(a[k]:4);
end.
```

3. Метод Шелла

Алгоритм Шелла намного эффективнее, чем метод пузырька. Сначала сравниваются отдаленные, а затем близкорасположенные элементы. Переменная kol содержит интервал, разделяющий сравниваемые элементы. Начальное значение kol равно половине количества элементов. В процессе сортировки значение kol уменьшается в два раза на каждом проходе, пока не начнет выполнятся сравнение соседних элементов, как в методе пузырька.

Программа, реализующая алгоритм сортировки Шелла :

```
const n=6;
type vec = array[1..n] of real;
ind = array[1..n] of integer;
var k,kol:integer;
a:vec;
w:real;
p:boolean;
i:integer;
begin

{ввод исходных в данных в массив}
writeln('ВВЕДИТЕ ДАННЫЕ В МАССИВ');
for i:=1 to n do
begin
write (i:2,':');
readln(a[i]);
end;

{МЕТОД ШЕЛЛА}
begin
kol:=n div 2;
repeat
repeat
p:=true;
for k:=1 to n-kol do
if a[k]>a[k+kol] then
```

```
begin
w:=a[k];
a[k]:=a[k+kol];
a[k+kol]:=w;
p:=false;
end;
until p;
kol:=kol div 2;
until kol=0;
end;

{Вывод на экран результата}
writeln('Результат сортировки');
for k:=1 to n do
writeln(a[k]:4);
end.
```

Полезный совет: перестановка элементов со сложными типами данных

– довольно длительный процесс. Поэтому рекомендуется вместо перестановки самих данных переставлять индексы. Такой прием используется практически во всех коммерческих приложениях.

В этом случае программа сортировки Шелла имеет вид:

```
const n=6;
type vec = array[1..n] of real;
ind = array[1..n] of integer;
var k,kol:integer;
var nom:ind;
a:vec;
w:integer;
p:boolean;
i:integer;
begin

{ввод исходных в данных в массив}
```

```
writeln('ВВЕДИТЕ ДАННЫЕ В МАССИВ');
for i:=1 to n do
begin
write (i:2,':');
readln(a[i]);
end;

{МЕТОД ШЕЛЛА для индексов}
begin
for k:=1 to n do nom[k]:=k;
kol:=n div 2; // Зазор
repeat
repeat
p:=true;
for k:=1 to n-kol do
if a[nom[k]]>a[nom[k+kol]] then
begin
w:=nom[k];
nom[k]:=nom[k+kol];
nom[k+kol]:=w;
p:=false;
end;
until p;
kol:=kol div 2;
until kol=0;
end;

{Вывод на экран результата}
writeln('Результат сортировки');
for k:=1 to n do
writeln('ИНДЕКС   ',nom[k]:4);
end.
```

3.8.Лабораторная работа № 8

Обработка строк с использованием множественного типа данных

Цель работы: ознакомиться со строковыми данными; получить навыки в организации работы со строковыми переменными: удалением, вставкой, копированием, заменой одной строки на другую и т.д.; получить навыки в задании переменных типа множество и организации ввода и вывода данных типа множество; получить практические навыки в выполнении операций над множествами.

Задание: составить программу заданной обработки массива слов. В процессе обработки использовать множественных тип данных. Заполнение исходных данных – с клавиатуры. Исходный и обработанный массив выводить на экран.

Теоретический материал

Строковый тип данных занимает промежуточное положение между простыми и структурированными типами данных. С одной стороны, данные строкового типа имеют структуру (строка, по сути, – это последовательность символов). С другой стороны, строковый тип , как все простые типы, является стандартным, а ни один структурированный тип не является стандартным. Кроме того, над строками можно выполнять некоторые действия, которые допустимы для данных простых типов и недопустимы для структурированных типов данных. Например, строку s можно ввести с клавиатуры или вывести на экран с помощью стандартных процедур ввода-вывода read(s) и write(s). В то же время , если описать переменную s как массив символов, то для ввода (или вывода) необходимо организовывать цикл, в котором стандартные процедуры ввода-вывода будут применяться к элементу массива s, т. е. read(s[i]) или
write(s[i]).

Значениями строковых переменных могут быть последовательности раз - личной длины (от нуля до 255 символов, длине 0 соответствует пустая строка

Для описания данных строкового типа используется зарезервированное слово string. По аналогии с любым стандартным типом, это слово можно ис-пользовать для описания переменных в разделе переменных следующим обра-зом:

```
var s:string;
```

Такое описание будет означать, что для переменной s в памяти зарезерви-ровано место под 255 символов (т. е. максимальная длина строки). Если же максимально возможное значение строковой переменной меньше (например, имя человека обычно содержит не более 20 символов), то можно ограничить длину строки, сэкономив тем самым оперативную память:

```
var name:string[20];
```

Вместе с тем, по аналогии со структурированным типом, можно опреде-лить строковый тип в разделе описания типов:

```
type str_20=string[20];
```

Затем использовать данный тип при описании переменных:

```
var name:str_20;
```

Стандартные процедуры и функции для работы со строками приведены ниже.

Функции

Concat	объединяет несколько строк
copy	возвращает подстроку, содержащуюся в строке
lenght	возвращает длину строки
pos	осуществляет поиск подстроки в строке
ord	возвращает код символа
chr	возвращает символ по его коду

Процедуры

delete удаляет подстроку из строки
insert вставляет подстроку в строку
str преобразует числовое значение в строку цифр
val преобразует строку цифр в числовое значение

Еще один тип данных, который потребуется при выполнении лабораторной работы, – множество. Множество – это структурированный тип данных, представляющий набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое.

Для задания типа-множества следует использовать зарезервированные слова set и of, а затем указать элементы этого множества, как правило в виде перечисления или диапазона:

Type <имя типа> = set of <el1, el2, ..., eln>;

Введя тип-множество, можно задать переменные или типизированные кон-станты этого типа-множества. Так же, как и для других структурированных ти-пов, тип-множество можно ввести непосредственно при задании переменных или типизированных констант.

var alf:set of char;

Множеству можно в программе присвоить значение, которое обычно зада-ется с помощью конструктора множества:

alf:=['A'..'Z'];

В каждое множество включается и так называемое пустое множество – [ ], не содержащее никаких элементов.

Операции над множествами:

+ объединение множеств;
- разность множеств;
* пересечение множеств;
= проверка эквивалентности двух множеств;
<> проверка неэквивалентности двух множеств;
<= проверка того, является ли левое множество подмножеством правого множества;

>= проверка того, является ли правое множество подмножеством левого множества;

in оператор вхождения, т. е. проверка того, входит ли элемент, указанный слева, в множество, у

Результатом операций объединения, разности или пересечения является соответствующее множество, остальные операции дают результат логического типа.

Пример выполнения лабораторной работы

Для примера рассмотрим следующую задачу. Дан массив из n слов произвольной длины (длина слова не превышает 80 символов). Символами могут быть буквы латинского алфавита и цифры. Определить сумму цифр во введенном с клавиатуры тексте.

Программу, как обычно, начнем с заголовка, разделов описания типов и переменных. Тип T_str определяем для работы с массивом слов, тип T_set – тип-множество символов.

```
program lab_8;


type T_str=array [1..10] of string[80];
T_set=set of char;
```

В разделе описания переменных объявляются переменные для хранения массива слов, множества символов, которому в дальнейшем присвоим значение допустимых символов, индексов, длины слова и количества слов (описываются целочисленным типом byte). Кроме того, в программе понадобятся целочис-ленные переменные, которые будут использоваться для преобразования симво-лов цифр в число и вычисления необходимых сумм.

```
var s:T_str;        {массив слов}

alf:T_set;          {множество символов}

n,          {количество слов}

i,j:byte; {индексы}
```

```
cod,      {код ошибки преобразования строки в число}

k,        {число, полученное преобразованием строки}
tmp, {сумма цифр слова} sum:integer; {сумма цифр всех слов}
```

Далее следует тело программы. Информационный блок подробно рассмат-риваться не будет. Перед началом ввода массива слов необходимо присвоить значение переменной множественного типа alf. По условию в массиве слов допускаются только латинские буквы и цифры, поэтому с помощью конструк-тора множеств задаем соответствующее значение. Латинские буквы ограничим лишь диапазоном заглавных букв, поскольку в дальнейшем при проверке сим-волов, входящих в массив слов будет применять функцию преобразования сим-вола в верхний регистр UpCase.

```
alf:=['A'..'Z','0'..'9'];
```

Если программист не использует эту функцию, тогда необходимо указы-вать диапазон как заглавных, так и строчных латинских букв:

```
alf:=['A'..'Z','a'..'z','0'..'9'];
```

Далее необходимо задать количество слов в массиве. Напомним, что при описании типа T_ str было определено максимально возможное количество слов в массиве, но не указано количество слов, которое будет обрабатываться. Для хранения количества слов в массиве определена переменная n, значение которой можно задать непосредственно в тексте программы, а можно запросить у пользователя. Во втором случае для предотвращения ошибок ввода необходимо предусмотреть проверку корректности вводимых значений:

```
repeat

write ('Введите количество слов (не более 10) ');

readln(n); {ввод числа слов} if not (n in [1..10]) then begin
```

```
writeln;

writeln (’Ошибка ввода!’);

writeln;

end;

until (n in [1..10]);
```

Цикл Repeat позволяет выполнить операторы тела цикла, по крайней мере, один раз. После ввода клавиатуры количества слов проверяется условие «значение переменной n не входит в множество целых чисел от 1 до 10». Если это условие выполняется, то выдается сообщение об ошибке и тело цикла повторяется еще раз, т.к. условие выхода из цикла («значение переменной n входит в множество целых чисел от 1 до 10») ложно. Если же пользователь вводит допустимое количество слов, то цикл завершается.

Теперь можно организовывать цикл по вводу и обработке массива слов. Используем для этого цикл for. В теле цикла, прежде всего, осуществляется ввод слова с клавиатуры и обнуление переменной, в которой накапливается сумма цифр i-го слова.

```
for i:=1 to n do

begin

write ('введите ',i,'-е слово ');

readln(s[i]); {ввод i-го слова} tmp:=0; {обнуление суммы цифр i-го слова}
```

Далее в следует проверка символа на принадлежность его к цифрам, для чего организуется еще один цикл, в котором будет осуществляться перебор символов i- го слова. В программе мы несколько усложним алгоритм за счет добавления проверки символов на допустимость значений, т. е. условия , которое будет проверять принадлежность символа к множеству допустимых

символов alf. В этом используется функция преобразования символа в верхний регистр UpCase (строчные буквы становятся заглавны-ми, заглавные – остаются без изменений).

for j:=length(s[i]) downto 1 do

if UpCase(s[i,j]) in alf then {проверка допустимости j-го символа i-го слова}

begin

Если проверяемый символ допустимый, то проверяется его принадлежность к множеству цифр.

if s[i,j] in ['0'..'9'] {если j-й символ i-го слова -

цифра}

then

Когда символ оказывается цифрой, то происходит его преобразование в число и увеличение суммы цифр текущего слова.

begin

val(s[i,j],k,cod); {преобразование символа в число} tmp:=tmp+k; {накапливается сумма цифр i-го слова}

end

end

Если же проверяемый символ оказался недопустимый, то выдается сообщение об ошибке, обнуляется сумма цифр слова и уменьшается номер слова. Цикл For автоматически увеличивает параметр цикла, и такое «топтание» на месте позволит еще раз повторить ввод и обработку ошибочного слова. Команда break позволит досрочно выйти из цикла проверки символов i-го слова.

```
else

begin

writeln('Встретился недопустимый символ!');

writeln;

tmp:=0;

dec(i); {Уменьшение номера слова} break; {Досрочный выход из цикла по j}

end;
```

После выхода из цикла проверки символов i-го слова происходит увеличение суммы цифр текущего слова.

```
sum:=sum+tmp; {накапливается сумма цифр слов} end;
```

Когда все слова массива будут обработаны, остается вывести результаты на экран и завершить программу.

```
writeln('Сумма цифр = ',sum);

readln;

end.
```

Итак, соединив фрагменты в единое целое, получим текст программы.

```
{Дан массив слов произвольной длины, но не превышающих 80 симво-лов.

Символами могут быть буквы латинского алфавита и цифры.

Определить сумму цифр во введенном с клавиатуры тексте.}

program lab_7;
```

```
uses crt;

type T_str=array [1..10] of string[80];

T_set=set of char;

var s:T_str;        {массив слов}

alf:T_set;          {множество символов}

n,          {количество слов}

i,j:byte; {индексы}

cod,        {код ошибки преобразования строки в число}

k,          {число, полученное преобразованием строки}

tmp,        {сумма цифр слова}

sum:integer;        {сумма цифр всех слов}

begin

clrscr;

writeln;

writeln('           Автор - Иванов И.П., студент гр. 2 ПК');

writeln('           Вариант № 10');

writeln('Дан массив слов произвольной длины, но не превышающих
80 символов. ');

writeln('Символами могут быть буквы латинского алфавита и
цифры.');
```

```
writeln('Определить сумму цифр во введенном с клавиатуры
тексте.');

writeln;

alf:=['A'..'Z','0'..'9'];       {задается множество допустимых

значений символов}

repeat

write ('Введите количество слов (не более 10) ');

readln(n);          {ввод числа слов}
if not (n in [1..10]) then
begin

writeln;

writeln (’Ошибка ввода!’);

writeln;

end;
until (n in [1..10]);

for i:=1 to n do

begin

write ('введите ',i,'-е слово ');

readln(s[i]);       {ввод i-го слова}

tmp:=0;{обнуление суммы цифр i-го слова}

for j:=length(s[i]) downto 1 do

if UpCase(s[i,j]) in alf then {проверка допустимости j-
```

```
го символа i-го слова}
begin

if s[i,j] in ['0'..'9'] then {если j-й символ i-го

слова - цифра}

begin

val(s[i,j],k,cod); {преобразование символа в число}

tmp:=tmp+k;     {накапливается сумма цифр i-го слова}
end
end
else
begin
writeln('Встретился недопустимый символ!');
writeln;
tmp:=0;
dec(i);   {Уменьшение номера слова}
break;   {Досрочный выход из цикла по j}
end;
sum:=sum+tmp; {накапливается сумма цифр всех слов}
end;
writeln('Сумма цифр = ',sum);
readln;
end.
```

Контрольные вопросы

1. Дайте определение строковой переменной.
2. Какие типы данных используются в качестве базовых в строковых данных?
3. Каким образом распределяется память под строковые переменные?
4. Какие операции выполняются над строковыми переменными?
5. В чем состоит сходство и различие строковых переменных и символьных массивов?

6. Возможно ли преобразование строковых переменных?
7. Назовите основные функции над строковыми переменными и их назначение.
8. Каково назначение процедур DELETE,INSERT.
9. Как реализуется ввод и вывод строковых переменных.
10. Предложите схему преобразования действительных чисел в стринги.
11. Что такое литерный ряд?
12. В чем состоит отличие стринга от литерного ряда?
13. Что понимается под множеством?
14. Какие вы знаете операции над множествами в математике?
15. Как записываются операции над множествами в языке Турбо-Паскаль?
16. Как задаются множества на языке Турбо-Паскаль?
17. Что такое пустое множество и как оно задается?
18. Как организовать вывод элементов множества?

3.9. Лабораторная работа № 9

Работа с файлами

Цель работы: изучение файловых типов в языке Турбо-Паскаль; получение навыков в организации файлов и использовании их для обработки информации.

Задание на лабораторную работу.

1. Скомпилировать программыl.
2. Программы сохранить, представить преподавателю результат работы программ и продемонстрировать понимание работы программ с файлами.

Теоретический материал

В Паскале понятие файла употребляется в двух смыслах:

– как поименованная информация на внешнем устройстве (внешний файл);

– как переменная файлового типа в Паскаль-программе (внутренний файл).

В программе между этими объектами устанавливается связь.

Вследствие этого все, что происходит в процессе выполнения программы с внутренним файлом, дублируется во внешнем файле.

С элементами файла можно выполнять только две операции:

читать из файла и записывать в файл.

Файловый тип переменной — это структурированный тип, представляющий собой совокупность однотипных элементов, количество которых заранее (до исполнения программы) не определено.

Структура описания файловой переменной:

Var <имя переменной>: **F i l e Of** <тип элемента>;

где <тип элемента> может быть любым, кроме файлового.

Для создания и заполнения файла требуется следующая последовательность действий:

1. Описать файловую переменную.
2. Описать переменную того же типа, что и файл.
3. Произвести назначение (**Assign)**.
4. Открыть файл для записи (**Rewrite**).
5. Записать в файл данные (**Write)**.
6. Закрыть файл (**Close**).

Для последовательного чтения данных из файла требуется выполнить следующие действия:

1. Описать файловую переменную.
2. Описать переменную того же типа.
3. Выполнить назначение (**Assign**).
4. Открыть файл для чтения (**Reset**).
5. В цикле читать из файла (**Read**).
6. Закрыть файл **(close**).

Процедуры и функции для работы с файлами

Все рассматриваемые функции и процедуры принадлежат стандартному модулю DOS, поэтому его необходимо подключить к программе с помощью предложения USES.

1. ReName(< файловая переменная >,< новое имя файла >) - переименование файла.
2. Erase(< файловая переменная >) - удаление файла.
3. ChDir(< путь >) - изменение директория, где <путь> - путь к новому директорию.
4. GetDir(< устройство >, < директорий >) - определение текущего каталога, где <устройство> задается следующим образом:

0 - текущее устройство;
1 - устройство A;
2 - устройство B и т.д.

5. MkDir(< директорий >) - создание нового каталога. В аргументе <директорий> указывается полный путь до того каталога, который создается.
6. PmDir(< директорий >) - удаление каталога. В качестве аргумента указывается полный путь до удаляемого каталога. При этом удаляемый каталог должен быть обязательно пустым.
7. IOResult - проверка правильности завершения работы той или иной операции ввода-вывода. Эта функция имеет тип WORD и возвращает значение 0, если операция ввода-вывода выполнилась успешно, и в противном случае следующие значения:

1 - файл не найден,
2 - путь не найден,
3 - слишком много открытых файлов,
5 - запрет доступа к файлу,
12 - некорректный код доступа к файлу
и так далее.

При применении этой функции в программе необходимо с помощью директивы компилятора отключить стандартную проверку - {$I-}, а после выполнения операций ввода-вывода включить - {$I+}.
Данная функция записана в стандартном модуле SYSTEM.

8. DiskFree(< устройство >) - определение числа свободных байтов на заданном диске. Эта функция типа LONGINT.

В качестве аргумента указывается номер устройства. Если указано несуществующее устройство, то вместо объема свободной памяти на

диске эта функция возвращает значение-1. Функцию рекомендуется применять перед созданием файла, чтобы выяснить,
достаточно ли места для создаваемого файла на указанном накопителе.
9. DiskSize(< устройство >) - определение числа свободных байтов на диске. Тип функции
LONGINT. Аргумент задается так же, как и в предыдущей функции.
10. FindFirst(< уточненное имя файла>, < атрибуты >, < доп.инф_____-я >) - поиск указанного файла.
В процедуре входным параметром является только первый. Два последних параметра являются выходными. Параметр < атрибуты > имеет тип BYTE, параметр < дополнительная информация > должен быть объявлен как SearchRec. Этот тип описан в стандартном модуле Dos.
11. FindNext(< следующий файл >) - поиск указанного файла. Процедуры FindFirst и FindNext зачастую используются для просмотра всех файлов, находящихся в каталоге.
12. FSearch(< имя файла>,<список каталогов >) - поиск файла в списке каталогов. Функция имеет тип PathStr (описана в стандартном модуле Dos).
13. FSplit(< уточненное имя файла >, < путь >,< имя >,< расширение >) - выделение из уточненного имени файла трех переменных: < путь >, < имя файла >, < расширение >.
14. FExpand(< имя файла >) - добавление к имени файла, находящегося в текущем каталоге, полного пути доступа к нему.
Примечание: перед использованием первых четырех процедур файл должен быть обязательно закрыт.

Типизированные файлы

Пример № 1.

В файловую переменную Fx занести 20 вещественных чисел, последовательно вводимых с клавиатуры.

```
program file1;
uses crt;
var FX:file of Real;
   x:real;
```

```
   i:byte;
begin
clrscr;
Assign(Fx,'file1.dat');
  rewrite(fx);
  for i:=1 to 20 do
  begin
  write('ВВОДИТЕ ДАННЫЕ :  ');
  readln(x);
  write(Fx,x);
  end;
  close(fx);
  writeln('На диске создан файл FILE.DAT ');
  writeln('Найдите файл FILE.DAT  в каталоге');
  end.
```

Пример № 2.

В переменной **X** получить 10-й элемент вещественного файла **Fx**.

```
program file2;
uses crt;
var FX:file of Real;
   x:real;
   i:byte;
begin
clrscr;
Assign(Fx,'file1.dat');
  reset(fx);
  for i:=1 to 10 do
 read(fx,x);
  close(fx);
  writeln('Результат ',x:4:2);
  end.
```

Пример № 3

Просуммировать все числа из файла Fx, описанного в предыдущем примере.

Функция **Eof (FV**) проверяет маркер конца файла *(end of file).*

Это логическая функция, которая получает значение t r u e , если указатель установлен на маркер конца, в противном случае — f a l se .

```
program file3;
uses crt;
var FX:file of Real;
    x:real;
    sx:real;
    i:byte;
begin
clrscr;
Assign(Fx,'file1.dat');
   reset(fx);
  while not eof(fx) do
  begin
  read(fx,x);
  sx:=sx+x;
    end;
   close(fx);
   writeln('Результат ',sx:4:2);
end.
```

Пример № 3.1

То же самое с помощью цикла Repeat можно делать следующим образом:

```
program file31;
uses crt;
var FX:file of Real;
    x:real;
    sx:real;
```

```
   i:byte;
begin
clrscr;
Assign(Fx,'file1.dat');
   reset(fx);
   sx:=0;
   repeat
  read(fx,x);
  sx:=sx+x;
until eof(fx);
   close(fx);
   writeln('Результат ',sx:4:2);
end.
```

Ответьте на вопрос:

В каком варианте (пример 3 или пример 3.1.) возможна ошибка чтения, если файл Fx пустой ?.

Пример № 4.

Создать файл, содержащий среднесуточные температуры за некоторое количество дней. При этом необязательно предварительно указывать количество чисел во вводимой информации. Можно договориться о каком-то условном значении, которое будет признаком конца ввода. Пусть, например, признаком конца ввода будет число 9999.

```
program file4;
uses crt;
var ft:file of real;
t:real;
begin
clrscr;
   assign(ft,'temp.dat');
   rewrite(ft);
    writeln('ВВОДИТЕ ДАННЫЕ.ПРИЗНАК КОНЦА-9999');
   readln(t);
   while t<>9999 do
   begin
```

```
    write(ft,t); write(' ');readln(t);
    end;
    writeln('Ввод данных закончен');
     writeln('На диске создан файл temp.DAT ');
  writeln('Найдите файл temp.DAT  в каталоге');
    close(ft);
end.
```

В результате работы этой программы на диске будет создан файл с именем Temp. dat, в котором сохранится введенная информация.

Пример № 5.

Определить среднюю температуру для значений, хранящихся в файле Temp.dat.

В этой программе использована функция определения размера файла:

FileSize(<имя файловой переменной>);

Ее результат — целое число, равное текущей длине файла.

```
program file5;
uses crt;
var ft:file of real;
t,st:real;
n:integer;
begin
clrscr;
  assign(ft,'temp.dat');
  reset(ft);
  st:=0;
  while not eof(ft) do
  begin
  read(ft,t);
  st:=st+t;
  end;
  n:=filesize(ft);
```

```
  st:=st/n;
     writeln('СРЕДНЯЯ ТЕМПЕРАТУРА ЗА ',N:3,'  СУТОК РАВНА
',st:4:2,'  градусов');
    close(ft);
end.
```

Текстовые файлы.

Текстовый файл — наиболее часто употребляемая разновидность файлов.

В программе файловая переменная текстового типа описывается следующим образом:

Var <идентификатор>:text;

Наряду с процедурами Read и Write употребляются процедуры ReadLn и WriteLn.

ReadLn(FV,<список ввода>)

Эта процедура читает строку из файла с именем FV, помещая прочитанное в переменные из списка ввода.

WriteLn(FV,<список вывода>)

Процедура записывает в файл FV значения из списка вывода, после чего выставляет маркер конца строки.

Для обнаружения конца строки в текстовом файле используется функция

Eoln(FV)

(End of line — конец строки).

Это логическая функция, которая принимает значение t r u e, если указатель файла достиг маркера конца строки и f a l s e — в противном случае.

Процедура **ReadLn** может использоваться без списка ввода. В этом случае происходит пропуск текущей строки в читаемом файле.

Употребление процедуры **WriteLn** без списка вывода обозначает вывод пустой строки (в файле выставляется маркер конца строки).

Пример № 6

Пусть файл с именем Note.txt содержит некоторый текст. Требуется подсчитать количество строк в этом тексте.

Для отладки программы, создайте в текстовом редакторе файл note.txt , заполните любой информацией в несколько строк . Файл поместите в каталог /PRG.

```
program file6;
var note:text;
    k:integer;
begin
  assign(note,'note.txt');
    reset(note);
    k:=0;
    while not eof(note) do
    begin
    readln(note);
    k:=k+1;
    end;
    writeln('КОЛИЧЕСТВО СТРОК РАВНО  ',k);
    close(note);
end.
```

Используемый здесь оператор ReadLn (Note) «пролистывает» строки из текстового файла Note, не занося их в какую-либо переменную.

Пример № 7

В текстовом файле Note. t x t определить длину самой большой строки.

```
program file7;
var note:text;
   max,k:integer;
    c:char;
begin
  assign(note,'note.txt');
```

```
   reset(note);
   max:=0;
   while not eof(note) do
   begin
   k:=0;
  while not eoln(note) do
  begin
  read(note,c);
   k:=k+1;
   end;
   if k>max then
   max:=k;
   readln(note);
  end;
   writeln('НАИБОЛЬШАЯ СТРОКА ИМЕЕТ  ',max,'   знаков');
   close(note);
end.
```

Здесь каждая строчка прочитывается посимвольно, при этом в переменной к работает счетчик числа символов в строке. В переменной Max отбирается наибольшее значение счетчика.

Пример № 8

```
Var  f: Text;
  s: string;
begin
  assign(f,'Text1.pas');
  reset(f);
  while not eof(f) do
  begin
    readln(f,s);
    writeln(s);
  end;
  close(f);
end.
```

Сохраните программу обязательно с именем text1.pas

Пример № 9

```
Const n = 30;
  filename = 'pifagor.txt';

var f: Text;
    i,j:integer;
begin
  assign(f,filename);
  rewrite(f);
  for i:=1 to n do
  begin
    for j:=1 to n do
      write(f,i*j:4);
    writeln(f);
  end;
  close(f);
  writeln('Результат умножения в файле  ',filename);
end.
```

Контрольные вопросы

1. Укажите режимы ввода информации.
2. В каких случаях удобно использовать файлы?
3. Дайте определение файла и укажите его характеристики.
4. Что такое путь доступа к файлу?
5. Где хранятся файлы ?
6. Выведите формулу подсчета объема файла в байтах.
7. Каким образом описываются переменные файловых типов ?
8. Как подразделяются файлы по видам доступа к его компонентам ? Как осуществляется
доступ к компонентам файлов ?
9. Какие операции определены над файлами

3.10. Лабораторная работа № 10

Работа с файлами записей

Цель работы: получить навыки в организации ввода и вывода значений комбинированных типов данных; получить навыки программирования задач с использованием записей; приобрести опыт разработки программ с файлами записей.

Задание на лабораторную работу.

1. Скомпилировать программы .
2. Программы сохранить, представить преподавателю результат работы программ и продемонстрировать понимание работы программ с файлами.

Теоретический материал

Запись (комбинированный тип данных) — это структурированный тип, состоящий из фиксированного числа компонент (полей) разного типа.

Обычно запись содержит совокупность разнотипных атрибутов, относящихся к одному объекту.

Например, анкетные сведения о студенте вуза могут быть представлены в виде информационной структуры :

Анкета студента
Ф.И.О. Пол Дата рождения Адрес Курс Группа Стипендия

Задать тип и описать соответствующую переменную можно следующим образом:

```
Type Anketal=Record
FIO: String[50];
Pol: Char;
Dat: String[16];
Adres: String[50];
Curs: 1..5;
Grup: 1..10;
```

Stip: Real
End;
Var Student: Anketal;

Такая запись, так же как и соответствующее ей дерево, называется двухуровневой.

К каждому элементу записи можно обратиться, используя *составное имя,* которое имеет следующую структуру:

<имя переменной>.<имя поля>

Например, student. f io; student. dat и т.п. Если, например, требуется полю *курс* присвоить значение 3, то это делается так:

Student.Curs:=3;

Поля записи могут иметь любой тип, в частности сами могут быть записями. Такая возможность используется в том случае, когда требуется представить многоуровневое дерево (более 2 уровней).

Например, те же сведения о студентах можно отобразить трехуровневым деревом

Анкета студента

Ф.И.О.	*Пол*	*Дата рождения*		*Адрес*		*Курс*	*Группа*	*Стипендия*
		Год	*Месяц*	*День*	*Город*	*Улица, дом, квартира*		

Такая организация данных позволит, например, делать выборки информации по году рождения или по городу, где живут студенты.

В этом случае описание соответствующей записи будет выглядеть так:

Type Anketa2=Record
FIO: S t r i n g [5 0];
Pol: Char;
Dat: Record
God: I n t e g e r;
Mes: S t r i n g [1 0];
Den: 1..31;
End;
Adres: Record
Gorod: S t r i n g [2 0];
UlDomKv: S t r i n g [3 0];

```
End;
Curs: 1..5;
Grup: 1. . 1 0;
S t i p : Real
End;

Var Student: Anketa2;
```

Поля такой записи, находящиеся на третьем уровне, идентифицируются тройным составным именем.

Например, s t u d e n t . D a t . G o d ;
student.Adres.Gorod.

Любая обработка записей, в том числе ввод и вывод, производится поэлементно. Например, ввод сведений о 500 студентах можно организовать следующим образом:

```
For I:=l To 500 Do
With Student[I] Do
Begin
Write('Ф.И.О.:'); ReadLn(FIO) ;
Write('Пол (м/ж):'); ReadLn(Pol);
Write("Дата рождения:'); ReadLn(Dat);
Write('Адрес:'); ReadLn(Adres);
Write('Курс:'); ReadLn(Curs);
Write('Группа:'); ReadLn(Grup);
Write('Стипендия (руб.):'); ReadLn(Stip)
End;
```

В этом примере использован *оператор присоединения,* который имеет следующий вид:

With <переменная типа запись> Do <оператор>;

Он позволяет, один раз указав имя переменной типа *запись* после слова With, работать в пределах оператора с именами полей как с обычными переменными, т.е. не писать громоздких составных имен.

Работа с файлами записей.

Чаще всего записи используются как элементы файлов, составляющих компьютерные информационные системы. Рассмотрим примеры программ, работающих с файлами записей.

Пример 1.

Сформировать файл FM. DAT, содержащий экзаменационную ведомость одной студенческой группы. Записи файла состоят из следующих элементов:

фамилия, имя, отчество; номер зачетной книжки; оценка.

```
Program Examen;
Type Stud=Record
FIO: String[30];
Nz: String[6];
Mark: 2..5;
End;
Var Fstud: File Of Stud;
S: Stud;
N,I: Byte;
Begin
n:=0;
Write('Количество студентов в группе?');
ReadLn(N) ;
Assign(Fstud,'FM.DAT'); Rewrite(Fstud);
For I:=1 To N Do
Begin
Write (i:1, '-Й,Фамилия И.О.'); ReadLn(S.FIO);
Write('Номер зачетки:'); ReadLn(S.Nz);
Write('Оценка:'); ReadLn(S.Mark);
Write(Fstud,S)
End;
WriteLn('Формирование файла закончено!');
Close(Fstud);
End.
```

Пример 2.

Имеется файл, сформированный программой из предыдущего примера. Пусть некоторые студенты пересдали экзамен и получили новые оценки. Составить программу внесения результатов переэкзаменовки в файл. Программа будет запрашивать номер студента в ведомости и его новую оценку. Работа заканчивается, если вводится несуществующий номер (9999).

```
Program New_Marks;
Type Stud=Record
FIO: String[30];
Nz: String[6];
Mark: 2..5
End;
Var Fstud: File Of Stud;
S: Stud;
N: Integer;
Begin
Assign(Fstud,'FM.DAT');
Reset(Fstud);
Write('Номер в ведомости?');
ReadLn(N);
While N<>9999 Do
Begin
Seek(Fstud,N-1);
Read(Fstud,S);
Write(S.FIO,'оценка?');
ReadLn(S.Mark);
Seek(Fstud,N-1);
Write(Fstud,S);
Write('Номер в ведомости?');
ReadLn (N);
End;
WriteLn('Работа закончена!');
Close(Fstud);
End.
```

Пример требует некоторых пояснений. Список студентов в ведомости пронумерован, начиная от 1, а записи в файле нумеруются от 0. Поэтому, если п — это номер в ведомости, то номер соответствующей записи в файле равен п - 1 .

После прочтения записи «номер п-1» указатель смещается к следующей п-й записи.

Для повторного занесения на то же место исправленной записи повторяется установка указателя.

Контрольные вопросы

1. Что понимается под записью в языке Паскаль?
2. Как объявляются записи?
3. Какие операции допустимы над полями записи?
4. Как организовать ввод и вывод данных типа записи?
5. Как осуществляется доступ к полям записи?
6. Можно ли использовать в записи поля одного типа?
7. Чем отличается запись от массива?
8. Каково назначение оператора присоединения?

3.11.Лабораторная работа № 11

Процедуры и функции. Рекурсивные функции.

Цель работы: приобретение практических навыков в программировании процедур и функций; изучение механизма передачи параметров; знакомство с локальными и глобальными переменными.

Задание к лабораторной работе

1. Модифицируйте подпрограмму, вычисляющую степенную функцию так, чтобы она вычисляла и отрицательные степени.
2. Напишите подпрограмму, способную вычислять любые степени: положительные и отрицательные, целочисленные и действительные.

Теоретический материал

Процедуры и функции являются мощным средством языка программирования. Это средство удобно применять в тех случаях, когда при решении задачи возникает необходимость в программе некоторую совокупность операторов повторять несколько раз. Так например, может возникнуть необходимость один и тот же цикл использовать в нескольких местах программы.

Функцию или процедуру можно сравнить с мини-программой, именно поэтому их называют иногда одним общим именем - "подпрограмма (ПП)". ПП оформляется подобно программе: в начале записывается заголовок ПП, затем следует декларативная часть ПП и после процедурная. В декларативной части описываются все данные, область действия которых ограничена телом данной ПП. Эти данные называются локальными. Данные, объявленные в основной (главной) программе, называются глобальными и они могут использоваться в любой ПП, входящей в основную программу. В процедурной части описывается тело ПП, реализующее алгоритм решения, и которое заключается в операторные скобки BEGIN, END.

ПП помещается сразу же после объявления всех переменных. Заголовок ПП для подпрограмм-функций начинается с ключевого слова FUNCTION, для подпрограмм- процедур с ключевого слова PROCEDURE. Эти ключевые слова играют роль признаков, которые распознает компилятор. Так как ПП исполняется не сразу и возможно не один раз, то компилятор, встретив тело ПП, должен его пропустить.

Функции

Общая схема функции следующая:

FUNCTION < идентификатор функции >(<параметр: тип параметра> [,<параметр: тип параметра>, ...]) : <тип результата функции>;

< декларативная часть (разделы LABEL, CONST, TYPE, VAR) – объявление локальных данных >
BEGIN
< процедурная часть функции - тело функции >
END;

В теле основной программы функция вызывается по имени, это значит, что после FUNCTION необходимо записать

идентификатор функции, а затем в круглых скобках перечислить все параметры функции. Так как язык Паскаль сильно типизирован, то он требует, чтобы после каждого параметра был указан его тип. Результатом вычисления функции всегда получается одно значение. Поэтому после круглых скобок в заголовке функции необходимо указать тип результата, вычисляемого функцией.
Ко всем функциям обращаются одинаково: в любом предложении программы они играют роль переменной.

Самым простым и наглядным примером использования функций являются стандартные функции, например, функция LENGTH(St). Эта функция может применяться в программе всякий раз, когда необходимо вычислить длину стринга, в данном случае стринга St. Все стандартные функции входят в состав компилятора, то есть они описаны в теле самого компилятора. LENGTH - это идентификатор функции, St - аргумент функции.

Заголовок этой функции может быть следующий:

FUNCTION Lenght(St : string) : byte;

Тип результата, возвращаемого этой функцией, BYTE.

***Пример*. № 1**

Напишем подпрограмму, позволяющую вычислять степенную функцию **a** в степени **n,** причем **n** может иметь только положительные целые значения.

В этом примере параметрами функции должны быть: основание и степень.

```
{ вычисление степенной функции }
FUNCTION Degree(Base: real; Power: integer) : real;
{ обьявление локальных переменных i, Y
i - параметр цикла, считающий степени;
Y - промежуточная переменная, содержащая текущие значения
степенной функции }
VAR
i : integer;
Y : real;
{ тело функции или процедурная часть }
BEGIN
Y:= 1;
For i:= 1 to Power do
```

```
Y:= Y * Base;
Degree:= Y;
END;
```

В основной программе обращаться к этой подпрограмме (функции) можно многократно,
например:

```
. . .
VAR
B, X, Z : real;
P : integer;
. . .
BEGIN { основная или главная программа }
. . .
B:= 1.5; P:= 3;
Z:= Degree(B, P); { основание = 1.5, степень = 3 }
. . .
WriteLn('Значение степенной функции = ', Degree(X*2, 10));
. . .
END.
```

Проследим работу функции по первому вызову. Значения переменных B и P: 1.5 и 3 передаются подпрограмме DEGREE. Однако после входа в подпрограмму мы эти переменные называем уже не B и P, а Base и Power. Имена Base и Power, фигурирующие в подпрограмме, являются просто "пустышками", замещаемыми при работе функции (вычислении) конкретными значениями. По терминологии ПП параметры, используемые в самой подпрограмме, называются формальными, а параметры, используемые в основной программе при ссылке к функции, называются фактическими. В нашем примере B и P -
фактические параметры, Base и Power - формальные. Этот прием введения фактических и формальных параметров удобен тем, что при многократном вызове подпрограммы мы можем передавать различные параметры, как это сделано в представленном выше фрагменте программы:
Degree(B, P) и Degree(X*2, 10).

Имена формальных и фактических параметров не обязаны совпадать, хотя если случайно они и оказываются одинаковыми, никаких проблем не возникнет. Однако необходимо,

чтобы типы формального и фактического параметров были согласованы (например, оба были типа INTEGER или REAL).
При выполнении любой ПП в первую очередь происходит сопоставление формальных и фактических параметров. С этого момента ссылка на Base и Power в подпрограмме фактически означает обращение к B и P. Следовательно, команды, вычисляющие степенную функцию, используют значения B и P (1.5 и 3). Результат вычислений помещается в имя DEGREE, в этом и состоит механизм, посредством которого функция "посылает" свой ответ в основную программу. После возврата в основную программу результат, переданный функцией, назначается переменной Z. При этом значение, вычисленное функцией и
помещаемое в DEGREE, имеет тот же тип, что и переменная Z.

Процедуры

Подпрограмму следует оформить в виде процедуры, если она предназначена для решения задачи одного из двух типов. Задача первого типа: требуется выполнить некую последовательность действий, не возвращая результирующего значения. Задача второго типа: требуется изменить значения одного или нескольких фактических параметров.

Процедура, как и функция, помещается в конце декларативной части программы.

Компилятор распознает процедуру по ключевому слову PROCEDURE. С этого слова начинается заголовок процедуры, после которого следует идентификатор процедуры, а затем в круглых скобках перечисляются формальные параметры процедуры.

В Паскале передать параметры подпрограмме можно двумя способами. До сих пор мы имели дело только с одним из них. Этот способ называют передачей параметров по значению. Он состоит в том, что значение фактического параметра назначается соответствующему формальному параметру. Другими словами, перед началом выполнения процедуры вычисляется конкретное значение фактического параметра (например, 1.5 или 3).

Затем полученное значение копируется в соответствующий формальный параметр, принадлежащий процедуре. Как только начинается выполнение процедуры, никакие изменения значения формального параметра уже не оказывают влияния на значение

соответствующего фактического параметра. Это значит, что по окончании работы процедуры фактический параметр будет иметь точно такое же значение, каким он обладал до начала работы процедуры, вне зависимости от того, что происходило с формальным параметром. Такой способ передачи параметров действует при работе с функциями.

Второй способ передачи параметров называется передачей параметров по ссылке или по адресу. При передаче параметра по ссылке в процедуру пересылается уже не значение аргумента, а его местоположение (адрес) в памяти компьютера. Чтобы сообщить компилятору о намерении передать такой параметр, в заголовке процедуры в списке формальных параметров следует указать слово VAR. Если формальный параметр снабжен атрибутом VAR, а соответствующий ему фактический параметр является переменной, то любые изменения формального параметра будут отражаться в значениях фактического параметра, поскольку теперь формальный и фактический параметры занимают одну и ту же область памяти.

Общая схема процедуры аналогична схеме функции со следующими изменениями: ключевое слово FUNCTION заменяется на PROCEDURE; отсутствует тип результата.

Вызов процедуры в основной программе оформляется как отдельное предложение, состоящее из имени процедуры и пары круглых скобок, в которых через запятую перечислены фактические параметры. Предложение заканчивается как обычно символом ";".

Пример № 2

Рассмотрим пример на задачу первого типа: необходимо выполнить перемножение двух
матриц A(3x4) и B(3x4).

Оформим в виде процедуры ввод матрицы. Программа в этом случае может иметь вид:

```
{*************************************************************
***}
{* программа перемножения двух матриц A, B *}
{*************************************************************
***}
Program Main;
```

```
Const
N = 3; { количество строк в матрице }
M = 4; { количество столбцов в матрице }
Type
MatrTyp = array[1..N, 1..M] of real;
Var
A, B, C : MatrTyp; { массивы можно передавать только как
простой тип }
i1, i2, i3 : byte; { индексы массивов }
{************************************************************
***}
{* процедура ввода значений матрицы *}
{************************************************************
***}
Procedure GetMatr(NameMatr : string; var Matr : MatrTyp);
Var
i, j : byte;
Begin { GetMatr }
WriteLn('Введите значения матрицы ', NameMatr);
For i:= 1 to N do
begin
WriteLn('Строка ', i);
For j:= 1 to M do
Read(Matr[i,j]);
WriteLn; { перевод на новую строку }
end;
End; { GetMatr }
{************************************************************
***}
80
{* главная программа *}
{************************************************************
***}
Begin
GetMatr('A', A); {Вызов процедуры для ввода значений матрицы A}
GetMatr('B', B); {Вызов процедуры для ввода значений матрицы B}
{ алгоритм перемножения матриц }
For i1:= 1 to N do
For i2:= 1 to M do
```

```
begin
C[i1,i2]:= 0;
For i3:= 1 to N do
C[i1,i2]:= C[i1,i2] + A[i1,i2] * B[i3,i1];
end;
{ вывод значений результирующей матрицы }
For i1:= 1 to N do
begin
For i2:= 1 to M do
Write(' C[',i1:1, ',', i2:1, '] = ', C[i1,i2]:8:3);
WriteLn; { перевод на новую строку }
end;
End. { конец главной программы }
```

Различия между процедурами и функциями

Главное различие (из которого следуют все остальные) состоит в том, что функция всегда возвращает, причем в явной форме, одно-единственное значение, которое может быть использовано в качестве составной части выражения; процедура такого значения не возвращает. Однако применительно к процедуре все же можно говорить о возвращаемой информации - процедура способна изменять значения своих параметров (тех, что описаны с атрибутом VAR).

Помимо главного различия можно отметить ряд второстепенных различий синтаксического характера. Так, например, заголовок функции всегда завершается указанием типа возвращаемого значения. В заголовке процедуры такая информация не нужна. Для функции типично, чтобы в качестве последнего шага имени функции было назначено некоторое вычисленное значение. В процедурах этого нет. И наконец, еще одно различие.

Поскольку функция возвращает какое-то значение, вызов функции может появляться прямо в выражении. Вызов процедуры не может быть частью выражения - это всегда отдельное предложение.

Рекурсивные функции

Под ***рекурсией*** в программировании понимают подпрограмму, которая вызывает сама себя. Рекурсивные функции чаще всего используют для компактной реализации рекурсивных алгоритмов. Классическими рекурсивными алгоритмами могут быть возведение числа в целую положительную степень, вычисление факториала. С другой стороны, любой рекурсивный алгоритм можно реализовать без применения рекурсий. Достоинством рекурсии является компактная запись, а недостатком расход памяти на повторные вызовы функций и передач параметров, кроме того, существует опасность переполнения памяти.

В рекурсивной функции необходимо обязательно предусмотреть завершение рекурсивного вызова. Иначе функция никогда не завершит свою работу.

Рассмотрим применение рекурсии на примерах.

Пример 3.

Вычислить факториал числа *n*.

Для решения этой задачи с применением рекурсии создадим функцию factorial, алгоритм которой представлен на рис.1.

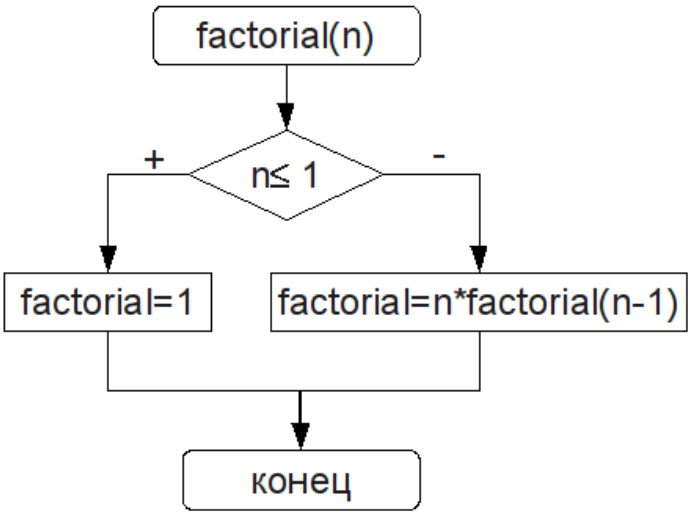


Рисунок 1.: Алгоритм вычисления факториала

Текст подпрограммы с применением рекурсии:

```
function factorial(n:word):longint;
begin
if n<=1 then factorial:=1
else factorial:=n*factorial(n-1)
end;
var i:integer;
begin

write('i=');
read(i);
write(i,'!=',factorial(i));
end.
```

Пример 4.

Вычислить *n*-ю степень числа *a* (*n* – целое число)

Результатом возведения числа a в целую степень n является умно жение этого числа на себя n раз. Но это утверждение верно только для положительных значений n. Если n принимает отрицательные значения, то $a^{-n} = 1/a^n$

В случае если n=0, то a^0=1.

Для решения задачи создадим рекурсивную функцию stepen, алгоритм которой представлен на рис. 2.

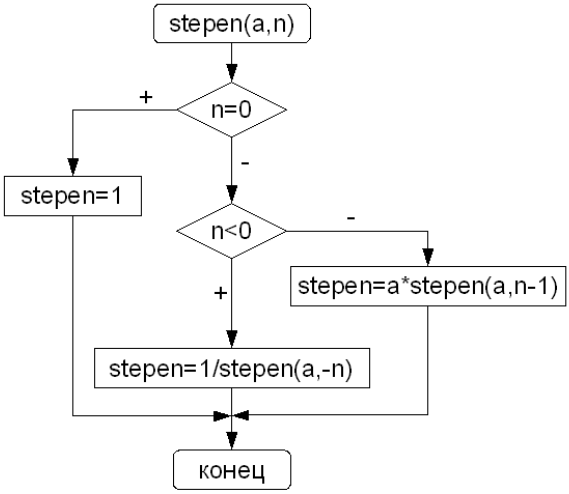


Рисунок 2. Рекурсивный алгоритм вычисления степени числа

Фрагмент программы с применением рекурсии:

```
function stepen(a:real;n:word):real;
begin
if n=0 then stepen:=1
else
if n<0 then
stepen:=1/stepen(a,-n)
else
stepen:=a*stepen(a,n-1);
end;
var
x:real;k:word;
begin
writeln('x='); readln(x);
writeln('k='); readln(k);
writeln(x:5:2,'^',k,'=',stepen(x,k):5:2);
end.
```

Пример 5.

Вычислить *n*-е число Фибоначчи.

Если нулевой элемент последовательности равен нулю, первый –

единице, а каждый последующий представляет собой сумму двух предыдущих, то это последовательность чисел Фибоначчи (0, 1, 1, 2, 3, 5,8, 13, 21, 34, ...).
Алгоритм рекурсивной функции fibonachi изображен на рис.3.

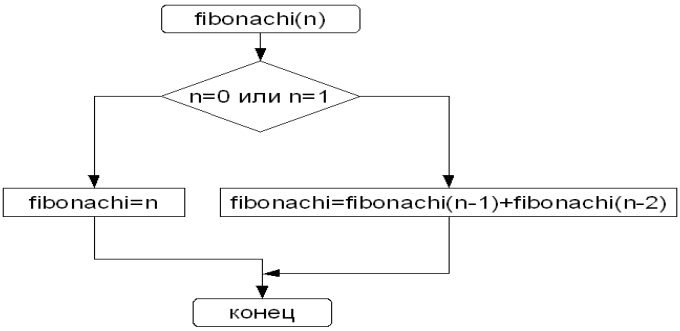


Рисунок 3. Рекурсивный алгоритм вычисления числа Фибоначчи

Текст подпрограммы:

```
function fibonachi(n:word):word;
begin
if (n=0)or(n=1) then fibonachi:=n
else fibonachi:=fibonachi(n-1)+fibonachi(n-2);
end;
var x:word;
begin
write('Введите номер числа Фибоначчи x=');
readln(x);
writeln(x,' -е число Фибоначчи = ',
fibonachi(x));
end.
```

Контрольные вопросы

1. Для чего предназначены функции?
2. Для чего предназначены процедуры?
3. Чем отличаются формальные и фактические параметры?
4. Опишите способы передачи параметров в подпрограммы и их особенности?
4. Что включает в себя заголовок подпрограммы?
5. Чем отличаются глобальные и локальные переменные?
6. Какая разница между процедурой и функцией?

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Основные источники:

1. Алексеев Е. Программирование на Free Pascal и Lazarus: курс / Е. Алексеев, О. Чеснокова, Т. Кучер. - 2-е изд., исправ. - Москва: Национальный Открытый Университет «ИНТУИТ», 2016. - 552 с.: ил. [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=429189.
2. Колокольникова А.И. Спецразделы информатики: основы алгоритмизации и программирования / А.И. Колокольникова. – Москва; Берлин: Директ-Медиа, 2019. – 424 с.: ил., табл. [Электронный ресурс]. – URL: http://biblioclub.ru/index.php?page=book&id=560695.
3. Костюкова Н.И. Комбинаторные алгоритмы для программистов / Н.И. Костюкова. – 2-е изд/, исправ/. – Москва : Национальный Открытый Университет «ИНТУИТ», 2016. – 217 с.: ил. – [Электронный ресурс]. URL: http://biblioclub.ru/index.php?page=book&id=429067.
4. Лубашева Т.В. Основы алгоритмизации и программирования: учебное пособие / Т.В. Лубашева, Б.А. Железко. - Минск: РИПО, 2016. - 378 с.: ил. - Библиогр. в кн. - ISBN 978-985-503-625-9; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=463632.
5. Семакин И.Г. Основы алгоритмизации и программирования. Практикум.- М.: Академия, 2016.-144 с.
6. ГОСТ 19.701–90 (ИСО 5807–85). Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.

Дополнительные источники:

1. Дроздов С.Н. Структуры и алгоритмы обработки данных: учебное пособие - Таганрог: Издательство Южного федерального университета, 2016. - 228 с.: схем., ил. - Библиогр. в кн. - ISBN 978-5-9275-2242-2; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=493032.

2. Комарова Е.С. Практикум по программированию на языке Паскаль: учебное пособие / Е.С. Комарова. - Москва; Берлин: Директ-Медиа, 2015. - Ч. 1. - 85 с. : ил., схем., табл. - Библиогр. в кн. - ISBN 978-5-4475-4914-5; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=425942.
3. Комарова Е.С. Практикум по программированию на языке Паскаль: учебное пособие / Е.С. Комарова. - Москва; Берлин: Директ-Медиа, 2015. - Ч. 2. - 123 с.: ил. - Библиогр. в кн. - ISBN 978-5-4475-4915-2; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=425943.
4. Митина О.А. Программирование / О.А. Митина, Т.Л. Борзунова; Москва: Альтаир: МГАВТ, 2015. – 61 с.: табл., ил. – [Электронный ресурс]. URL: http://biblioclub.ru/index.php?page=book&id=429764.
5. Нагаева И.А. Алгоритмизация и программирование. Практикум: / И.А. Нагаева, И.А. Кузнецов. – Москва; Берлин: Директ-Медиа, 2019. – 167 с.: ил., табл. – [Электронный ресурс]. URL: http://biblioclub.ru/index.php?page=book&id=570287.
6. Уйманова Н.А., Таспаева М.Г. Основы объектно-ориентированного программирования: практикум - Оренбург: ОГУ, 2017- 156 с.: ил. - Библиогр. в кн. - ISBN 978-5-7410-1993-1; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=485416.
7. Фленов М.Е. Библия Delphi. - СПб.: БХВ-Петербург, 2015. - 688с.
8. Фленов М.Е. Библия Delphi. - СПб.: БХВ-Петербург, 2015. - 688с.
9. Царёв Р.Ю. Алгоритмы и структуры данных (CDIO) / Р.Ю. Царёв, А.В. Прокопенко; – Красноярск: СФУ, 2016. – 204 с.: ил. – [Электронный ресурс]. URL: http://biblioclub.ru/index.php?page=book&id=497016.
10. Эйдлина Г.М., Милорадов К.А. Delphi: программирование в примерах и задачах. Практикум. – М.: ИНФРА-М, 2017.-119с.

Периодические издания:

1. Проблемы информатики. Издательство «Федеральное государственное бюджетное учреждение науки Институт вычислительной математики и математической геофизики Сибирского отделения Российской академии наук»;

2. Прикладная информатика: научно-практический журнал / гл. ред. А.А. Емельянов - Москва: Университет «Синергия» - ISSN 1993-8314; [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=495388;
3. Прикладная информатика: Университет «Синергия».

Интернет-ресурсы:

1. Федеральный центр информационно-образовательных услуг. Режим доступа: [http://fcior.edu.ru/05.05.2020]
2. Федеральные образовательные ресурсы». Режим доступа [http://www.edu.ru/ 05.05.2020]
3. Библиотека учебных курсов Microsoft. Режим доступа: [http://msdn.microsoft.com/ru-ru/gg638594 12.05.2020]
4. Программирование –Delphi/Pascal. Режим доступа [http://www.citforum.ru] 05.04.2020
5. Раздел Delphi и Pascal. Режим доступа [http://forum.developing.ru] 14.04.2020
6. Сайт Borland. Режим доступа [http://www.borland.ru] 05.04.2020
7. Borland Russian Community. Режим доступа [http://www.bdrc.ru] 15.04.2020