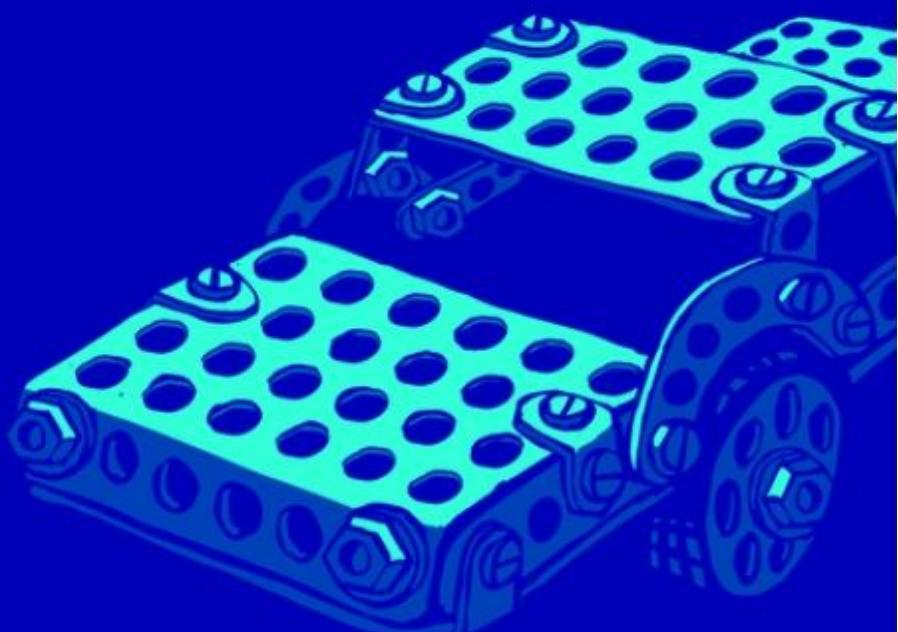


E.R. Alekseev, O.V. Chesnokova, T.V. Kucher



# Free Pascal and Lazarus

A textbook on programming

Библиотека Alt Linux

This page deliberately left blank.

In the series:

ALT Linux library

# **Free Pascal and Lazarus Programming Textbook**

E. R. Alekseev  
O. V. Chesnokova  
T. V. Kucher

Moscow  
ALT Linux; DMK-Press Publishers  
2010

UDC 004.432

BBK 22.1

A47

**Alekseev E.R., Chesnokova O.V., Kucher T.V.**

A47 Free Pascal and Lazarus: A Programming Textbook / E. R. Alekseev, O. V. Chesnokova, T. V. Kucher M.: ALTLinux; Publishing house DMK-Press, 2010. 440 p.: illustrated.(ALT Linux library).

**ISBN 978-5-94074-611-9**

Free Pascal is a free implementation of the Pascal programming language that is compatible with Borland Pascal and Object Pascal / Delphi, but with additional features. The Free Pascal compiler is a free cross-platform product implemented on Linux and Windows, and other operating systems. This book is a textbook on algorithms and programming, using Free Pascal. The reader will also be introduced to the principles of creating graphical user interface applications with Lazarus.

Each topic is accompanied by 25 exercise problems, which will make this textbook useful not only for those studying programming independently, but also for teachers in the education system.

The book's website is: <http://books.altlinux.ru/freepascal/>

This textbook is intended for teachers and students of junior colleges and universities, and the wider audience of readers who may be interested in programming.

**UDC 004.432**

**BBK 22.1**

**This book is available from:**

**The <<Alt Linux>> company: (495) 662-3883. E-mail: [zakaz@altlinux.ru](mailto:zakaz@altlinux.ru)**  
**Internet store: <http://shop.altlinux.ru>**

**From the publishers <<Alians-kniga>>:**

**Wholesale purchases: (495) 258-91-94, 258-91-95. E-mail: [books@aliens-kniga.ru](mailto:books@aliens-kniga.ru)**  
**Internet store: <http://www.aliens-kniga.ru/>**

The material in this book is distributed under the GNU FDL License. The book contains the following text on the first page after the cover: <<In the series: ALT Linux Library>>. The title is: <<Free Pascal and Lazarus: A Textbook on Programming>>. The book does not contain invariant sections. The authors of the sections are shown in the titles of the respective sections. ALT Linux is a trademark of ALT Linux. Linux is a trademark of Linus Torvalds. Other names in this textbook may be trademarks of their respective owners.

**ISBN 978-5-94074-611-9**

© Alekseev E.R., Chesnokova O.V.,

Kucher T.V., 2010

© ALT Linux, 2010

# Table of Contents

From the Editors.....	1
About free software for education.....	1
The “ALT Linux Library” Series.....	2
Introduction.....	3
Chapter 1. Development Tools.....	5
1.1 The Program Development Process.....	5
1.2 The Free Pascal Programming Environment.....	6
1.2.1 The Free Pascal Text Editor.....	9
1.2.2 Running a Program and Viewing Results.....	11
1.3 The Geany Development Environment.....	11
1.4 The Lazarus RAD Environment.....	14
1.4.1 Installing Lazarus on Linux.....	14
1.4.2 Installing Lazarus on Windows.....	15
1.4.3 The Lazarus Environment.....	17
1.4.4 The Lazarus Main Menu.....	18
1.4.5 The Application Form.....	22
1.4.6 The Lazarus Source Editor.....	22
1.4.7 The Component Palette.....	28
1.4.8 The Object Inspector.....	29
1.4.9 Your First Lazarus Program.....	30
1.4.10 A Useful Program.....	38
1.4.11 A Lazarus IDE Console Application.....	43
1.4.12 Data Input / Output Statements.....	46
Chapter 2. Introduction to Free Pascal.....	49
2.1 Lazarus Project Structure.....	49
2.2 The Console Application Structure.....	50
2.3 Language Elements.....	52
2.4 Free Pascal Data Types.....	52
2.4.1 The Character Type.....	53
2.4.2 The Integer Types.....	53
2.4.3 The Real Type.....	54

2.4.4 The DateTime Type.....	55
2.4.5 The Boolean Type.....	55
2.4.6 Creating New Types.....	55
2.4.7 The Enumeration Type.....	56
2.4.8 The Subrange Type.....	56
2.4.9 Structured Types.....	57
2.4.10 Pointers.....	59
2.5 Operators and Expressions.....	60
2.5.1 Arithmetic Operators.....	62
2.5.2 Relational Operators.....	64
2.5.3 Boolean Operators.....	64
2.5.4 Pointer Operators.....	64
2.6 Standard Mathematical Functions.....	65
2.7 Exercises.....	75
<b>Chapter 3. Flow Control.....</b>	<b>79</b>
3.1 Principal Algorithm Constructs.....	79
3.2 The Assignment Statement.....	81
3.3 The Compound Statement.....	81
3.4 Selection Structures.....	81
3.4.1 The If..Then..Else Selection Structure.....	81
3.4.2 The Case Selection Structure.....	98
3.4.3 Error Handling and Displaying Messages.....	101
3.5 Repetition Statements.....	105
3.5.1 The while..do Statement.....	105
3.5.2 The repeat..until Statement.....	107
3.5.3 The for..do Statement.....	108
3.5.4 Control Transfer Statements.....	110
3.5.5 Solving Problems Using Loops.....	111
3.5.6 Entering Data Using an InputBox.....	126
3.6 Exercises.....	135
3.6.1 Exercises (Branching).....	135
3.6.2 Exercises (Looping).....	138
<b>Chapter 4. Subroutines.....</b>	<b>141</b>
4.1 Local and Global Variables.....	141

4.2 Formal and Actual Parameters.....	142
4.3 Procedures.....	143
4.4 Functions.....	146
4.5 Solving Problems using Subroutines.....	151
4.6 Recursive Functions.....	168
4.7 Special Topics in Subroutines.....	172
4.7.1 Constant Parameters.....	172
4.7.2 Procedural Types.....	172
4.8 Developing Units.....	175
4.9 Exercises.....	178
Chapter 5. Arrays in Free Pascal.....	181
5.1 General Information about Arrays.....	181
5.2 Static Arrays.....	182
5.3 Array Operators.....	183
5.4 Element Input / Output.....	184
5.4.1 Input / Output in Console Applications.....	185
5.4.2 Input / Output in Graphical Applications.....	188
5.5 The Sum and Product of Elements.....	195
5.6 Finding the Maximum Element.....	196
5.7 Sorting Elements.....	197
5.7.1 Bubble Sort.....	198
5.7.2 Selection Sort.....	200
5.8 Removing an Element.....	202
5.9 Inserting an Element.....	205
5.10 Dynamic Arrays.....	207
5.11 Passing Arrays to Subroutines.....	208
5.12 Dynamic Memory Allocation.....	210
5.12.1 Dynamically Allocated Variables.....	210
5.12.2 Dynamically Allocated Arrays.....	213
5.13 Examples.....	215
5.14 Exercises.....	240

Chapter 6. Matrix Processing in Pascal.....	245
6.1 Element Input / Output.....	246
6.2 Matrix Algorithms.....	258
6.3 Dynamic Matrices.....	291
6.4 Exercises.....	293
Chapter 7. Files in Free Pascal.....	297
7.1 File Types in Free Pascal.....	297
7.2 Working with Typed Files.....	298
7.2.1 The AssignFile Procedure.....	298
7.2.2 The Reset and Rewrite Procedures.....	298
7.2.3 The CloseFile Procedure.....	299
7.2.4 The Rename Procedure.....	299
7.2.5 The Erase Procedure.....	299
7.2.6 The eof Function.....	299
7.2.7 Reading and Writing Data to File.....	300
7.2.8 The Filesize Function.....	309
7.2.9 The filepos Function.....	311
7.2.10 The seek Procedure.....	311
7.2.11 The truncate Procedure.....	311
7.3 Untyped Binary Files in Free Pascal.....	320
7.4 Text Files in Free Pascal.....	330
7.5 Exercises.....	335
Chapter 8. Strings and Records.....	339
8.1 Working with Text.....	339
8.2 Working with Records.....	343
8.3 Exercises (Strings).....	352
8.4 Exercises (Records).....	353
Chapter 9. Object Oriented Programming.....	359
9.1 Basic Concepts.....	359
9.2 Encapsulation.....	367
9.3 Inheritance and Polymorphism.....	371
9.4 Operator Overloading.....	381

9.5 Exercises.....	394
Chapter 10. Graphics in Lazarus.....	399
10.1 Drawing Tools in Lazarus.....	399
10.2 Plotting Graphs.....	408
10.3 Exercises.....	419
In Place of a Conclusion.....	423
About the Authors.....	425
About the Translator.....	427
Literature.....	429

## **Index of Tables**

Table 1.1: Example form properties.....	40
Table 1.2: Example label properties.....	40
Table 1.3: Example edit box properties.....	40
Table 1.4: Example button properties.....	40
Table 2.1: Integer data types.....	54
Table 2.2: Real data types.....	54
Table 2.3: Boolean data types.....	55
Table 2.4: Main operators in Free Pascal.....	61
Table 2.5: Boolean operations.....	64
Table 2.6: Standard math functions.....	66
Table 2.7: String processing functions.....	67
Table 2.8: FloatToStrF parameters.....	68
Table 2.9: Date and time functions.....	69
Table 2.10: Memory functions.....	69
Table 2.11: Captions for controls in Figure 2.7.....	70
Table 3.1: Number of roots in a cubic equation.....	92
Table 3.2: Message box type.....	101
Table 3.3: Dialog buttons.....	102
Table 3.4: Finding GCD for numbers A = 25 and B = 15.....	111
Table 3.5: Raising the number a to the power n.....	114
Table 3.6: Summing even numbers.....	117
Table 3.7: Number of divisors of the number n = 12.....	117
Table 3.8: Number of digits in a number.....	122
Table 3.9: Position of the current digit in a number m.....	124
Table 4.1: Form properties for Example 4.5.....	152
Table 5.1: Temperature values.....	181

Table 5.2: One-dimensional array of 7 real numbers.....	181
Table 5.3: Two-dimensional numeric array.....	182
Table 5.4: Properties of the Edit1 component in example.....	188
Table 5.5: Properties of the Button1 component in example.....	188
Table 5.6: Main properties of a component of type TStringGrid.....	192
Table 5.7: TStringGrid Properties to be used in the example.....	193
Table 5.8: Sorting array elements in ascending order.....	198
Table 5.9: Properties for labels, buttons, and text box in example.....	236
Table 5.10: Row table properties in example.....	236
Table 6.1: StringGrid1, StringGrid2 properties.....	250
Table 7.1: Form properties.....	304
Table 7.2: Label1 properties.....	304
Table 7.3: Label2 properties.....	304
Table 7.4: Memo1 properties.....	304
Table 7.5: Memo2 properties.....	304
Table 7.6: OpenDialog1 properties.....	304
Table 7.7: Button1 properties.....	304
Table 7.8: Button2 properties.....	305
Table 7.9: Input / Output error codes.....	332
Table 8.1: Population growth in cities.....	353
Table 8.2: Product details.....	354
Table 8.3: Information about schoolchildren.....	354
Table 8.4: Sales Details.....	354
Table 8.5: Employee information.....	355
Table 8.6: Tour Sales Information.....	355
Table 8.7: Employee details.....	356
Table 8.8: Academic employee details.....	356
Table 8.9: Book publication information.....	356
Table 8.10: Information about phone calls.....	357

Table 8.11: Instrument Information.....	357
Table 10.1: Color properties.....	400
Table 10.2: Charset values.....	401
Table 10.3: Line types.....	401
Table 10.4: Values for the Mode property.....	402

# From the Editors

## About free software for education

This book is not just a textbook on programming in the long-established tradition of teaching Pascal. This is the first Russian language edition dedicated to Free Pascal, the *free implementation* of this language, and to Lazarus, its free Integrated Development Environment. Like any free software, the Free Pascal compiler and Lazarus environment can be installed on any number of computers free of charge (no license fees), can be used without restrictions, and their source code is freely accessible for study and modification. Thus, they are ideal for teaching programming, because they require no payment from teachers and students and give them the opportunity to fully understand the subject (and even the compiler source code).

The benefits of free software for education was recognized in Russia at the highest level, and in the fall of 2007 the first open tender for the development and delivery of a free software educational software suite was advertised. The winner was the Armada group. That group included the ALT Linux company, which was in business since 2001 and was the leading Russian company in the development of free software and Linux distributions.

As part of the project, ALT Linux employees developed and tested a free software suite<sup>1</sup>, which was installed in more than 1000 schools in three pilot regions of Russia: the Republic of Tatarstan, Perm Territory and Tomsk Region. In addition to developing and debugging the implementation technology, technical support technology was also developed in these regions, which was very important for the success of the project. Despite the fact that there were only three pilot regions, any school in Russia could voluntarily join the project and receive a free software suite. Workshops were held in different parts of the country to spread word about the project and its benefits. By the end of 2008, 2151 schools were participating in the project.

In 2010, ALT Linux released Alt Linux 5.0 for Schools<sup>2</sup>, which was a new suite of educational software and an upgrade to the 2007-2008 free software suite. The suite included the ALT Linux operating system, all the software needed for preparing lessons and teaching information technology, a significant set of programs for mathematics, physics, drawing, astronomy and other subjects, educational and developmental games, and the standard set of office programs.

Alt Linux for Schools can be installed on almost any computer, as it includes versions designed for various hardware.



The Alt Linux for Schools 5.0 package includes Free Pascal and Lazarus and are ideal for mastering the material in this book and for teaching Free Pascal.

### The “ALT Linux Library” Series

The “ALT Linux library” is a series of books about free software for a wide variety of applications, published by ALT Linux and the DMK-Press publishing house. To date, the series has published the following books:

- *G.V. Kuryachy, K. A. Maslinsky* - Introduction to Linux: Course lectures. Textbook. Moscow, 2010.
- *V.B. Volkov* - Linux Junior: A book for the teacher. Moscow, 2010.
- *I.A. Khakhaev* - Free graphics editor GIMP: First Steps. Moscow, 2010.
- *I.A. Khakhaev et al.* - OpenOffice.org: Theory and Practice. Moscow, 2008.
- *E.R. Alekseev, E.A. Chesnokova, E.A. Rudchenko* - Scilab: Solving engineering and mathematical problems. Moscow, 2008.
- ALT Linux from the outside. ALT Linux from the inside. Moscow, 2006.

Material published in this series is distributed under free licenses, and full electronic versions can be downloaded from the internet for free. Detailed information about the series, about publications released or being readied for release, may be found on the project website<sup>3</sup>.

The editors of the series invite authors interested in writing books and articles on the use of free software to address a wide variety of tasks, for users ranging from the least sophisticated to professionals and specialists. More about publishing conditions can be found on the project website.

#### Endnotes:

---

1 [http://en.altlinux.org/Main\\_Page](http://en.altlinux.org/Main_Page)

2 At present (2020) the current version is Alt-Education 9.

3 <http://books.altlinux.org>

# Introduction

The authors of this book have long wanted to write a programming textbook that would be useful to users of various operating systems. Thanks to the ALT Linux company, we were able to do just that. Free Pascal was chosen as the programming language because we found that it was clear, logical and flexible, and it teaches good coding style. The open source Free Pascal compiler was implemented in many Linux distributions, and in Windows. The reader will also be introduced to the principles of creating graphical user interface applications in Lazarus.

Currently, there are many approaches to learning programming. In the opinion of the authors, one cannot learn programming in any language without learning to develop algorithms. One of the clearest ways of developing algorithms is through the use of flowcharts. The authors observed this over their many years of experience in teaching programming. How successful was our effort in writing a textbook on algorithms and programming will be left to the reader.

The authors hope that the reader has a basic knowledge of working with Linux or Windows and is familiar with high school mathematics.

The book consists of ten chapters.

In the first chapter, the reader will learn about the various ways of developing Free Pascal programs and will write his / her first programs.

The second chapter introduces the fundamental elements of the Free Pascal language (variables, expressions, statements). The simplest statements are described, including assignment and input/output. Program structure and examples of the simplest programs are introduced.

The third chapter is a key one for learning programming. It introduces a technique for developing algorithms using flowcharts. A large number of examples of flowcharts of algorithms and programs of various levels of complexity is provided. The authors recommend that the student carefully study all examples and solve the exercise problems in this chapter, before moving on to the later chapters.

In the fourth chapter, the reader will be introduced to sub-programs, using a large number of examples. The mechanism for passing data between subroutines is described. A section is devoted to recursive routines. The chapter closes with a discussion on creating your own units.

The fifth and sixth chapters are devoted to the study of algorithms for processing arrays and matrices. The reader will be introduced to the implementation of these algorithms in Free Pascal. These chapters, together with the third, are key to

understanding the principles of programming.

The seventh chapter introduces the reader to working with files in Free Pascal on Linux and Windows. Practical examples will show the mechanisms for random and sequential file access, and input/output error handling. Working with untyped and text files is described.

The eighth chapter addresses working with strings and records. Examples provided will allow the reader to understand the principles of working with tables in Free Pascal.

In the ninth chapter, the authors described the principles of object-oriented programming and their implementation in the Free Pascal language.

The tenth chapter discusses the graphics capabilities of Lazarus, and presents a detailed description of an algorithm for plotting continuous functions on the screen. The source code for displaying graphs of functions is provided, with detailed comments.

Each topic is accompanied by 25 exercise problems that will make the book useful not only for independent study, but also for teaching in educational institutions.

The authors thank the ALT Linux company, and Kirill Maslinsky in particular, for the opportunity to publish this book. The authors would also like to thank their families for their help and understanding.

Alekseev E.R., Chesnokova O.V., Kucher T.V.

Donetsk, January 2009

# Chapter 1. Development Tools

This chapter introduces programming with Free Pascal. Free Pascal was derived from the classic Pascal, which was developed in the late 60's of the twentieth century by Niklaus Wirth. Wirth developed it as an educational language for his students. Since then, Pascal has retained the simplicity and structure of the language developed by Wirth, while becoming a powerful programming tool. Modern Pascal can be used to perform simple calculations or develop programs for complex engineering and economic calculations.

## 1.1 The Program Development Process

Program development can be divided into the following stages:

- 1) Developing an algorithm for solving a problem. An *algorithm* is the series of actions necessary for the solution of a given problem.
- 2) Writing the program code. The *program source code* may be written in any programming language (in Free Pascal, for example) and entered into the computer, using a text editor.
- 3) Debugging the program. *Debugging a program* is the process of removing errors from the source code. Errors may be divided into syntactic and logical. If there are syntactic errors (including spelling errors) the program will not compile or run. Such errors are easier to correct. Logical errors are errors which allow the program to compile and run, but not as expected. Correcting logical errors is more complicated, and may require rewriting sections of the program, or even its entire algorithm.
- 4) Testing the program. *Testing the program* is the process of identifying errors in the way the program works.

The process of debugging and testing often requires running the program over and over. A program will run only after its source code in Pascal<sup>1</sup> is translated into *binary machine code*, to become an *executable file*. The process of translating the source code into machine code is called *translation*. All translators may be divided into two classes:

- *interpreters*, which are translators that translate each program token into machine code as soon as the token is entered into the editor;
- *compilers*, which translate the entire program as a whole, and if the translation of the entire program occurs without error, then the resulting binary code can be executed.

When a compiler is used as the translator, the process of translating the program code into machine code is known as compilation. The conversion of Pascal code to machine code requires a compiler<sup>2</sup>.

The main stages of compiling Pascal code include the following.

- 1) The compiler determines which external libraries<sup>3</sup> should be linked, parses the source code into its constituent elements, checks for syntactic errors and, if there is none, generates object code (on Windows a file with ".obj" extension, and on Linux a file with ".o" extension). The resulting binary file (object file) is not yet linked to any required external libraries.
- 2) At the second stage, the linker links the object file with the external libraries and generates the executable file. This stage is called *linking the program*. The executable code generated at the completion of this stage can now be run.

At this time there are several Pascal compilers, including the Borland Pascal compiler, Delphi, the open source cross-platform Free Pascal compiler.

## 1.2 The Free Pascal Programming Environment

Let us look at the *installation* of Free Pascal on Linux. On Linux, software may be installed using a package manager. Different Linux distributions use different package managers, such as Synaptic for example, which is used in ALT Linux. The Synaptic window is shown in Figure 1.1. In ALT Linux distributions for schools, Free Pascal and Lazarus are installed by default.

Note that before installing software in Linux, the list of software repositories (where the software is stored) must be installed and updated<sup>4</sup>.

To install Free Pascal using Synaptic, double-click on the **Search** button in the main window (see Figure 1.1) and enter **fpc** in the dialog that opens (see Figure 1.2). The Program Manager will display a list of software related to Free Pascal. Check **fpc-3.2.0 (Free Pascal - SDK-3.2.0 suite)**, using the correct version number instead of 3.2.0 in the Synaptic window, by right-clicking and selecting **Mark for Installation** in the context menu. Start the installation by clicking **Apply** in the Synaptic toolbar, after which the Program Manager will download and install the software.

The **fpc** meta-package includes the Free Pascal compiler **fpc** and the **fp-ide** Integrated Development Environment (IDE). Type **fp** in a Linux terminal window to run the IDE. Figure 1.3 shows the Free Pascal IDE window in Linux.

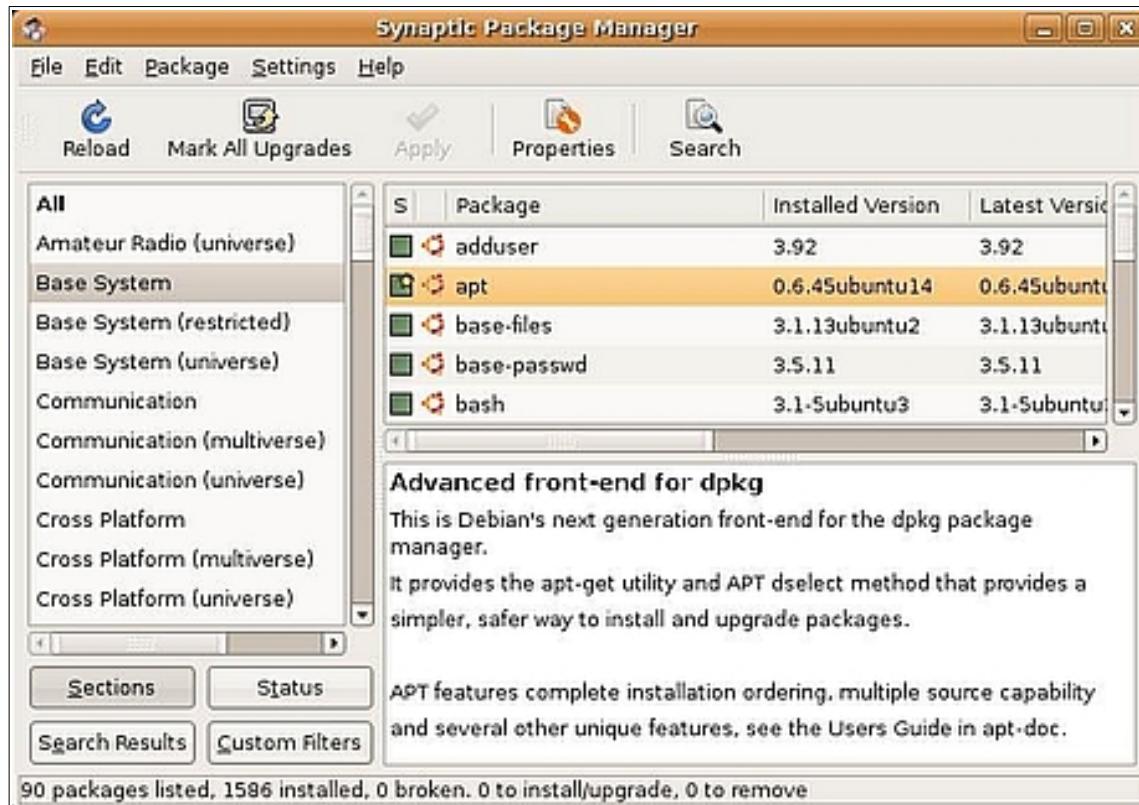


Figure 1.1: Synaptic Package Manager

To install Free Pascal on Windows, run the setup file downloaded from the download page<sup>5</sup>. The first dialog (after security checks) will remind you that you are about to install Free Pascal. Select **Next** to continue installing, or **Back** to return to the previous step. Press **Cancel** to stop the installation.

In the next window, you may change the destination for the Free Pascal installation. By default, the destination is the root directory of the C drive. To select a different destination, use the **Browse** button. This window also displays information about the amount space required for the installation.

In the next four windows, the user will be able to select the type of installation: **Full Installation**, **Minimum Installation**, or **Custom Installation**. Specify the name for the application in the main menu, select the file types supported by the environment, and start the Free Pascal installation by clicking the **Install** button. A progress bar will display progress, during which it is still possible to abort the installation.

Launching the Free Pascal programming environment in Windows can be done from the Start menu, by clicking **Free Pascal**. The window shown in Figure 1.4 will appear on the screen.

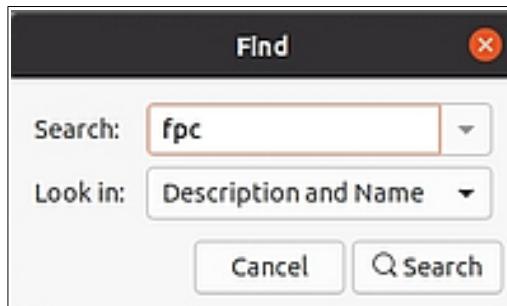


Figure 1.2: Using Synaptic to find fpc

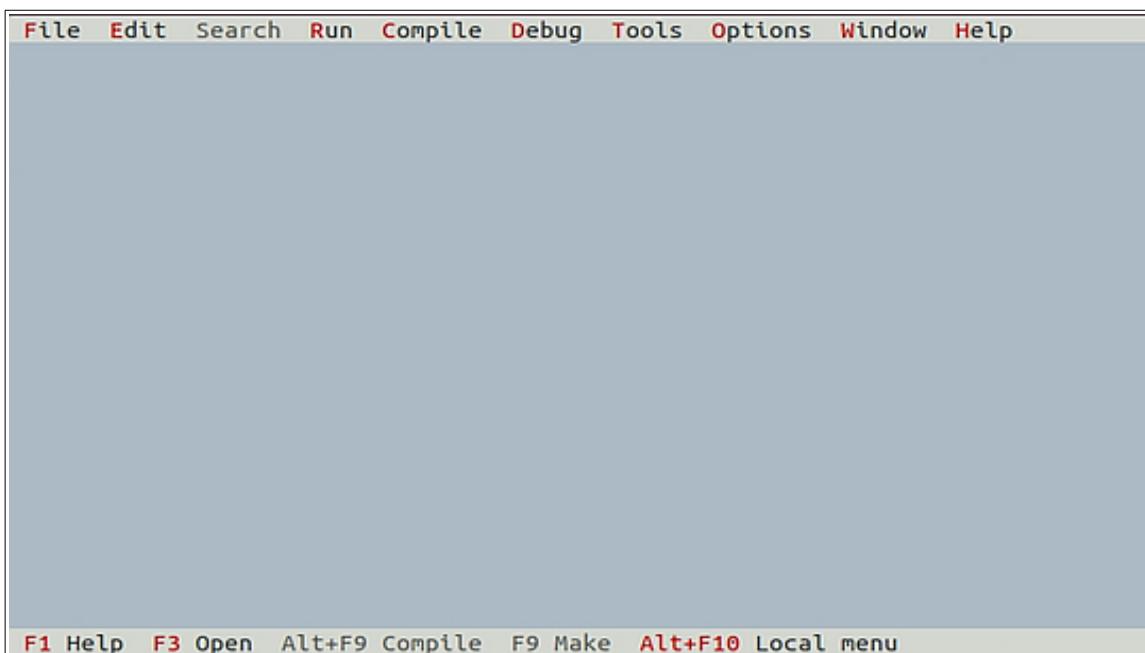


Figure 1.3: Free Pascal programming environment in Linux

The Free Pascal setup will install both the compiler and the programming environment.

The Free Pascal compiler may be run from the command prompt. To create an executable file from Pascal code, execute the command:

```
fpc name.pas
```

In this command, **fpc** is the Free Pascal compiler executable (simply speaking), and *name.pas* is the source code file. This will result in Linux creating an executable file named *name*, or Windows creating an executable file named *name.exe*.

When using the Free Pascal compiler, linking the program (running *make*) takes place automatically after compiling.

The work flow for using the Free Pascal compiler at the command line could be as



Figure 1.4: Free Pascal Compiler Window

follows: type the program source code in a text editor, run the compiler in a terminal window, correct syntax errors, then run the executable file. With this work flow, do not forget to save the corrected source code, otherwise the old version of the code will be compiled. Do not forget to save the corrected source code, otherwise the old version of the code will be compiled.

The IDE, however, can greatly simplify program development. The Free Pascal IDE includes a text editor, compiler and debugger. Let us examine them in more detail.

### 1.2.1 The Free Pascal Text Editor

The Free Pascal editor facilitates the creation and editing of source code. The editor goes into *edit mode* on opening an empty window (**File|New**) or loading a source file (**File|Open**), which is indicated by the small flashing *cursor* in the window. To switch back from editing mode to the main menu, press the **F10** key, or the **Esc** key to go back in general.

The Free Pascal editor has features that are typical of most text editors. Let us spend some time examining some of these features.

The main menu and function keys could be used for working with *text blocks* in Free Pascal. *Highlighting text* can be done using the **Shift** key and navigation (arrow) keys.

The **Edit** command in the main menu is for working with blocks of text:

- **Copy (Ctrl+Ins)** - copy selected text to the clipboard;
- **Cut (Shift+Del)** - cut selected text to the clipboard;
- **Paste (Shift+Ins)** - paste text from the clipboard;
- **Clear (Ctrl+Del)** - clear the buffer;
- **Select All** - select all text in the window;
- **Unselect** - cancel selection.

The **Copy** and **Cut** commands apply only to selected text. In addition, the **Edit** menu item contains the **Undo** and **Redo** commands, which can undo and redo completed actions. The key combinations for working with blocks are as follows (*no longer works!*):

- **Ctrl+K+B** - mark the beginning of the block;
- **Ctrl+K+K** - mark the end of the block;
- **Ctrl+K+T** - mark word to the left of cursor as a block;
- **Ctrl+K+Y** - erase block;
- **Ctrl+K+C** - copy block to cursor position;
- **Ctrl+K+V** - move block to cursor position;
- **Ctrl+K+W** - write block to a file;
- **Ctrl+K+R** - read a block from file;
- **Ctrl+K+P** - print block;
- **Ctrl+K+H** - unmark block; repeating **Ctrl+K+H** will delete the block.

*Working with files* in the Free Pascal environment is carried out using the **File** commands in the main menu, and function keys:

- **New** - open a window for creating a new program;
- **Open (F3)** - open a previously created file;
- **Save (F2)** - save the created file;
- **Save As** - to save the file under a different name;
- **Exit (Alt+X)** - exit the programming environment.

*When creating a new program*, it is assigned the standard name NONAMEOO.PAS by default.

The first time a file is saved, the user will be prompted to enter its name. From then on, the file will be saved under the same name. The **Save As** command is the same as saving the first time. If the file was not saved, you will be prompted to save changes in the file on attempting to exit the environment. *When opening a*

*previously created file*, select its name from the list of existing files.

The Free Pascal editor allows *working with multiple windows*. There are two ways to switch between windows:

- To switch to a window with a number from 1 to the 9, press the **Alt+i** key combination, where i is the window number (e.g. **Alt+5** will call of the fifth window);
- To display the list of windows on the screen, press the **Alt+0** key combination. A *list of active windows* will appear. Select the desired window and press **Enter**.

### 1.2.2 Running a Program and Viewing Results

After typing the program source code, it needs to be translated into machine code. To do this, call the compiler using the **Compile | Compile** command (or the keyboard shortcut **Alt+F9**). In the first stage, the compiler checks for syntax errors. If there are no syntax errors in the program, the screen will display the number of lines of translated code and the amount of available RAM.

If at any stage the compiler detects an error, then the cursor will point to the line where the error was found in the editor window. A short diagnostic message about the cause of the error will appear at the top of the editor.

To run the compiled program, execute the **Run | Run** command (key combination **Ctrl+F9**). A terminal window will appear and allow the user to interact with the program. When the program terminates, the Free Pascal environment screen will reappear.

In Windows, to view the results from the program, press the **Alt+F5** key combination. To return to the editor, press any key.

An alternative to the Free Pascal programming environment is the text-based editor Geany (<http://www.geany.org>), which can be used to develop programs in different programming languages. Developing programs using Geany is quite convenient.

## 1.3 The Geany Development Environment

The Geany development environment (sometimes referred to as a text editor) may be found in the repositories of most modern Linux<sup>6</sup> distributions. Install Geany on Linux in the normal way, by using Synaptic, for example.

To develop a Free Pascal program using Geany:

- 1) Create an application template in Pascal using the **File | New (with Template) | program.pas** file command. A window with a source code

template will appear (see Figure 1.5), in which the user enters the program code and saves it (see Figure 1.6).

- 2) To compile and run the program on completion, use the menu item **Build**. To compile the program, use the **Build | Compile (F8)** command. In this

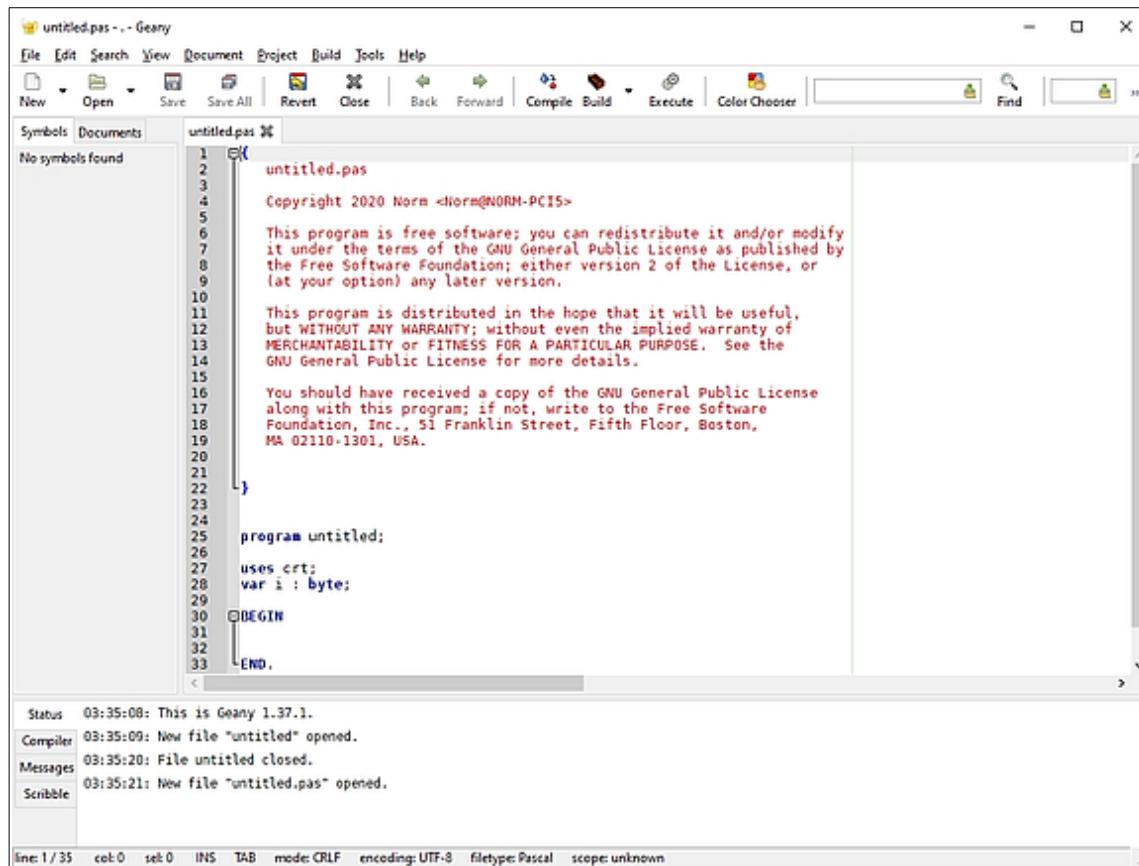


Figure 1.5: Geany window with Free Pascal program template

case, the executable file and a file with the program object code will be created. After compiling the program a detailed report about the compilation results will be shown below the code window (see Figure 1.6). This report should be studied carefully because it could help to quickly correct syntax errors. The error report (if there is an error) will indicate the line where the error was found, and provide a short description. Compilation reports may contain *errors* and *warnings*. Warnings are issues detected by the compiler, despite which, however, an executable file will still be generated.

- 3) To run the program, execute the **Build | Execute (F5)** command. After doing so, a terminal window will appear on the screen (see Figure 1.7), in which you can enter data and see the results of the program.

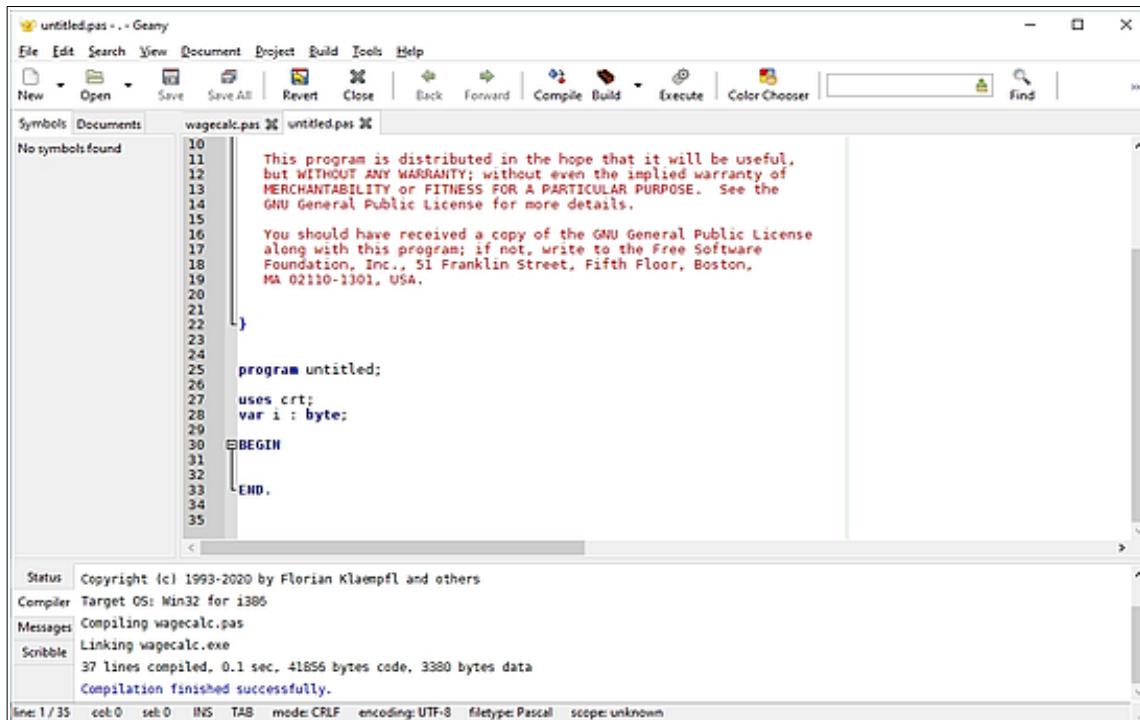


Figure 1.6: Geany window with Free Pascal code

In Geany, you can customize the commands for compiling, linking and running. To do this, use the **Build | Set Build Commands** command. The window for working with Pascal files is shown in Figure 1.8. When configuring the **Compile** and **Execute** lines, "%f" is the name of the file to be compiled, and "%e" is the filename without its extension.

```
*  
***  
*****  
*****
```

Figure 1.7: Terminal window showing program results

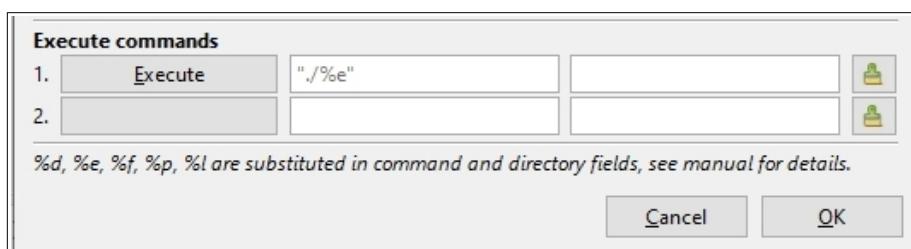


Figure 1.8: Set Free Pascal Build Arguments window

Choosing an environment for developing console programs in Free Pascal is up to

the user. The authors recommend that the reader use Geany<sup>7</sup> in Linux, although it is possible to write code using any common text editor (such as gedit, tea, kate, etc.), but compilation will have to be performed at the terminal. On Windows it makes sense to use fp-ide.

## 1.4 The Lazarus RAD Environment

Lazarus is a *Rapid Application Development (RAD) environment*, in which the programmer will be able to not just write code, but also see what programs with a Graphical User Interface (GUI) would look like.

The Lazarus RAD environment allows the user to build the program interface<sup>8</sup> from special controls that implement the necessary properties. The number of such controls is quite large. Each control contains ready-made program code and data necessary for the program to work, which saves the programmer from re-creating what was already created. This convenience significantly shortens the time to write a program. Also, the speed with which code is created in Lazarus is achieved because a significant amount of the code is generated automatically.

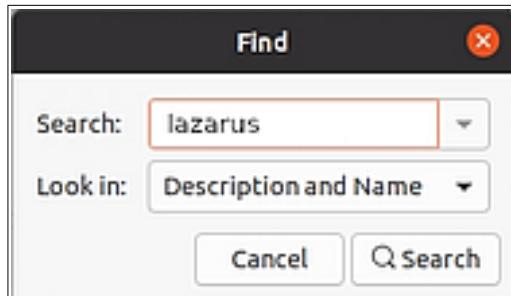


Figure 1.9: Synaptic search box

The Lazarus RAD environment contains a compiler, object-oriented programming environment and various technologies that facilitate and accelerate program creation.

### 1.4.1 Installing Lazarus on Linux

To install Lazarus, click the **Search** button in the Synaptic window (Figure 1.1). Enter the names of the programs needed (**lazarus**, **fp**, **fpc**, **fpc-source**) in the search window (Figure 1.9) and click the **Search** button. After the program manager finds Lazarus and Free Pascal, check **Lazarus**, **fp**, **fpc**, **fpc-ide** for installation (using the context menu or **Mark for Installation**) and start the installation by clicking the **Apply** button. Synaptic will then offer to install more packages that are required for Lazarus to work normally. After accepting these, the process of downloading the package files and installing Lazarus will begin. After installation, the program may

be launched (in Linux) using the **Development | Lazarus** menu command<sup>9</sup>.

You don't have to use the Synaptic package manager to install Lazarus. Instead, you may independently download all the necessary packages from the project site<sup>10</sup> and then manually install them. A detailed description of the manual installation process can be found on the internet<sup>11</sup>.

### 1.4.2 Installing Lazarus on Windows

Let us look at the details of installing Lazarus on Windows. To install, download the setup file from the download page<sup>12</sup>. The setup file opens the Setup Wizard which will guide you through the installation of Lazarus on your computer.

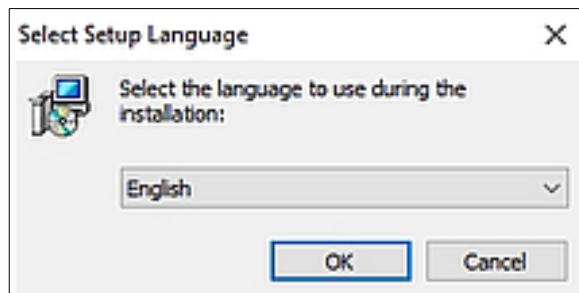


Figure 1.10: Language selection dialog



Figure 1.11: Starting the Lazarus installation process

After the Setup Wizard begins, a dialog box will appear (Figure 1.10) that allows the

user to choose the installation language. Pressing the **OK** button will bring up the next window. Pressing the **Cancel** button will abort the installation.

The next window (Figure 1.11) tells the user that the installation process is about to begin. Clicking the **Next** button will advance the installation to the next step in the Setup Wizard.

The next window (Figure 1.12) allows the user to select a destination location for the Lazarus installation, which will be installed at <C:\lazarus> by default. To select another destination, use the **Browse** button. To go back to the previous step at any point in the Setup Wizard, use the **Back** button.

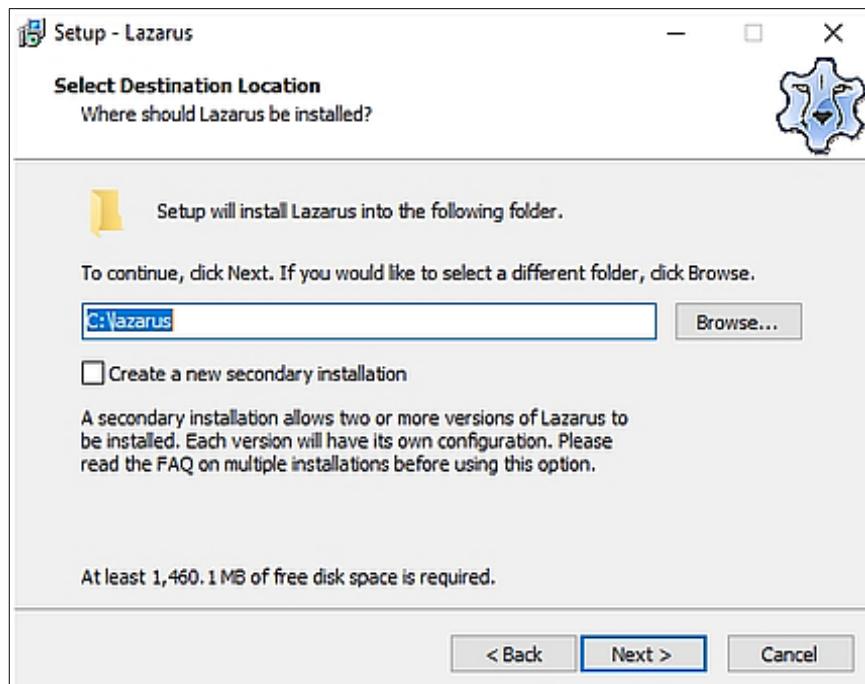


Figure 1.12: Installation destination dialog

The next step is to select components (Figure 1.13). Everything will be installed by default. You can skip the installation of a component by unchecking the box next to its name.

In subsequent windows in the Setup Wizard, the user may select the Start menu folder where the program shortcut will be placed,<sup>13</sup> and agree to have a Lazarus icon placed on the desktop<sup>14</sup>. Next, the Setup Wizard will tell you where Lazarus will be installed, the type of installation, which system components were selected, where the Lazarus shortcut will be created and if an icon will be created on the desktop. After that, the process will begin installing the application on the computer. A progress bar allows the user monitor the installation progress; the

installation may be aborted before it is complete by clicking the **Cancel** button. After the installation process is complete, the user is asked to click the **Finish** button. Lazarus may then be started from the main menu using the **Start | Lazarus | Lazarus** command.

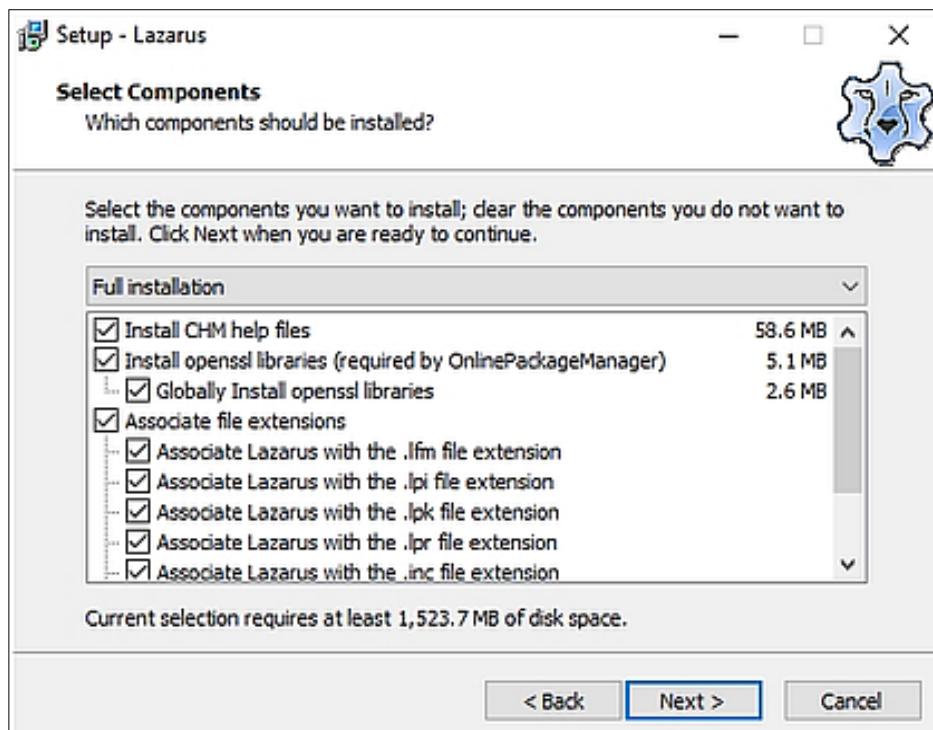


Figure 1.13: Component selection dialog

### 1.4.3 The Lazarus Environment

Figure 1.14 shows the window that appears after starting Lazarus. The upper part of this window contains the *main menu* and *toolbar*. The *Object Inspector* window is located on the left by default, and the *Source Editor* window is on the right.

The Form window, shown in Figure 1.15, could be accessed by minimizing or moving the Editor window.

Work on a program in a visual programming environment could be divided into two parts. The first is the creation of the program interface, and the second is the writing of the source code. The *Object Inspector* window and the Form window are necessary for creating the program interface, and the *Source Editor* is needed to work with source code. The collection of files from which the program is produced is called a *project*.

### 1.4.4 The Lazarus Main Menu

The Lazarus *main menu* contains the commands necessary for working with the Lazarus visual programming environment. Commands in the main menu may be accessed by left clicking.

The **File** menu is used for working with files in the Lazarus environment. The commands in this menu item can be divided into groups:

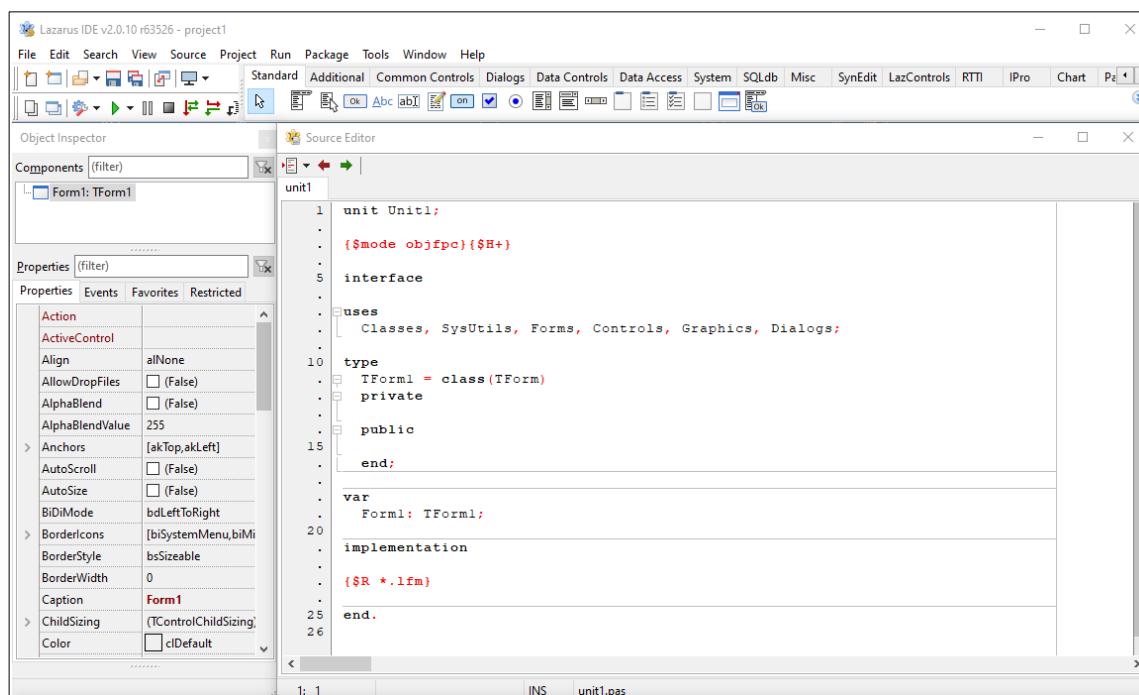


Figure 1.14: Lazarus visual programming environment

- creating new files – **New Unit**, **New Form**, **New...**;
- loading of previously created files - **Open...**, **Revert**, **Open Unit...**;
- saving files - **Save**, **Save As...**, **Save All**;
- closing files – **Close Page**, **Close All**;
- deleting files from the directory - **Clean directory...**;
- printing – **Print...**;
- restarting the environment - **Restart**;
- exit from environment - **Quit**.

Commands for *editing source code* can be found under the **Edit** menu. These commands are typical of most text editors:

- commands to undo or redo the last operation - **Undo, Redo**;
- commands for moving, copying and pasting selected text to the clipboard - **Cut, Copy, Paste**;
- commands for working with a selected block of text – **Indent Selection, Unindent Selection**;
- commands for changing the text case - **Uppercase Selection, Lowercase Selection**;
- commands for selecting a section of code are found under the **Select** menu items.

Commands specific to the Lazarus code editor are listed below.

The **Comment Selection** command adds two forward slashes at the beginning of each line of selected text, which turns the selected text into a comment,<sup>15</sup> while the **Uncomment Selection** command does the opposite.

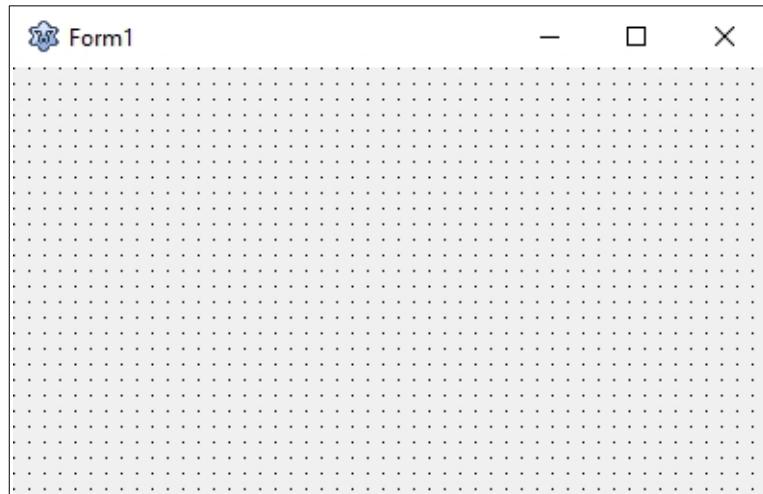


Figure 1.15: The empty form

The command **Enclose selection...** opens a dialog window (Figure 1.16), that allows the user to select the structure to enclose the selection.

The **Sort Selection...** command also opens a dialog box, in which you can set the sorting parameters for the selection.

The **Search** menu commands can also be divided into groups. The first group contains the *Find and Replace* commands, the second contains the *Jump* commands, and the third contains commands to work with *Bookmarks*. The fourth group contains *Find* commands for code blocks. Most of these commands are used in text editors, and the meaning of the rest should be clear from their names.

The **View** menu is used for *customizing the appearance of the programming environment*.

The first group of commands opens or activates the following windows:

- The **Object Inspector**, which allows you to define the external appearance and behavior of the selected object (for details see Section 1.4.8);
- The **Source Editor**, which allows you to create and edit source code (for details see Section 1.4.6);
- The **Project Inspector**, which contains general information about the project;
- The **FPDoc Editor**, which is a template editor;
- The **Code Browser**, which is the project explorer.

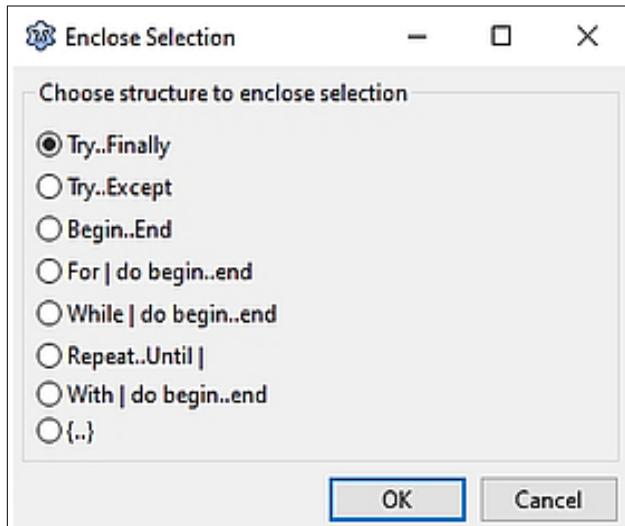


Figure 1.16: Enclosing selected text

The next group of commands under the **View** menu also opens dialog windows. These windows are for informational purposes only. The first command in this group is the **Toggle Form | Unit View** command. It switches between the Source Editor and the related Form window. The name of the **Unit Dependencies** command is self explanatory.

The **Debug Windows** command is used when debugging code. You can invoke **Debug Windows** to view breakpoints and the values of variables while the program is running.

The commands under the **Project** menu item are used to perform various project<sup>16</sup> operations:

- commands for creating a project - **New Project...** and **New Project from**

**File...;**

- commands for working with a previously created project - **Open Project...**, **Open Recent Project...**, **Close project**;
- commands for saving the project - **Save Project**, **Save Project As...**, **Publish Project...**;
- project management commands - **Project Inspector...**, **Project Options...**, etc.

Commands for starting and debugging a project may be found under the **Run** menu:

- **Build** - compile the program files;
- **Quick Compile** – compile only those program files that have changed;
- **Run** - run the project in debugger mode (including compilation, linking and execution);
- **Pause** - pause the program until any key is pressed.;
- **Step Into** – a step-by-step debugging mode that steps through procedures and functions;
- **Step Over** - a step-by-step debugging mode that does not step through procedures and functions;
- **Step over to Cursor** – in this debugging mode the program will run to the line on which the cursor is placed;
- **Stop** - aborts the program;
- **Reset Debugger** - resets all previously used debug settings and terminates the program;
- **Configure Build+Run File...** - set build options and run;
- **Evaluate|Modify...** - observe the value of a variable and / or find the value of an expression during program execution, and change the value of any variable, if necessary;
- **Add Watch...** - opens a window in which the values of specific variables and / or expressions can be watched and optionally modified while debugging a program;
- **Add Breakpoint** - set a breakpoint in the current line. The debugger will stop executing before the statement containing a breakpoint. Any number of breakpoints can be set in a program.

Commands under the **Package** menu are used for working with components<sup>17</sup>. Commands under the **Tools**<sup>18</sup> menu are used for configuring the programming environment.

The **Windows** menu is used for working with windows in Lazarus. The command names under this menu are the names of the windows, and selecting a name will activate the corresponding window.

The **Help** menu provides access to the online and built-in help systems that provide detailed *help* information for Free Pascal and Lazarus.

### 1.4.5 The Application Form

The Application Form (Figure 1.15) is the place where the program interface is constructed visually. Initially, this window contains only basic elements: the window title bar and buttons for maximizing, minimizing and closing the form. The window's working area is filled with a grid of points<sup>19</sup>. The programmer's task is to create the program interface by placing interface elements from the component palette on the form.

The Form Editor may be customized by setting options under the **Tools|Options...** menu (Figure 1.17).

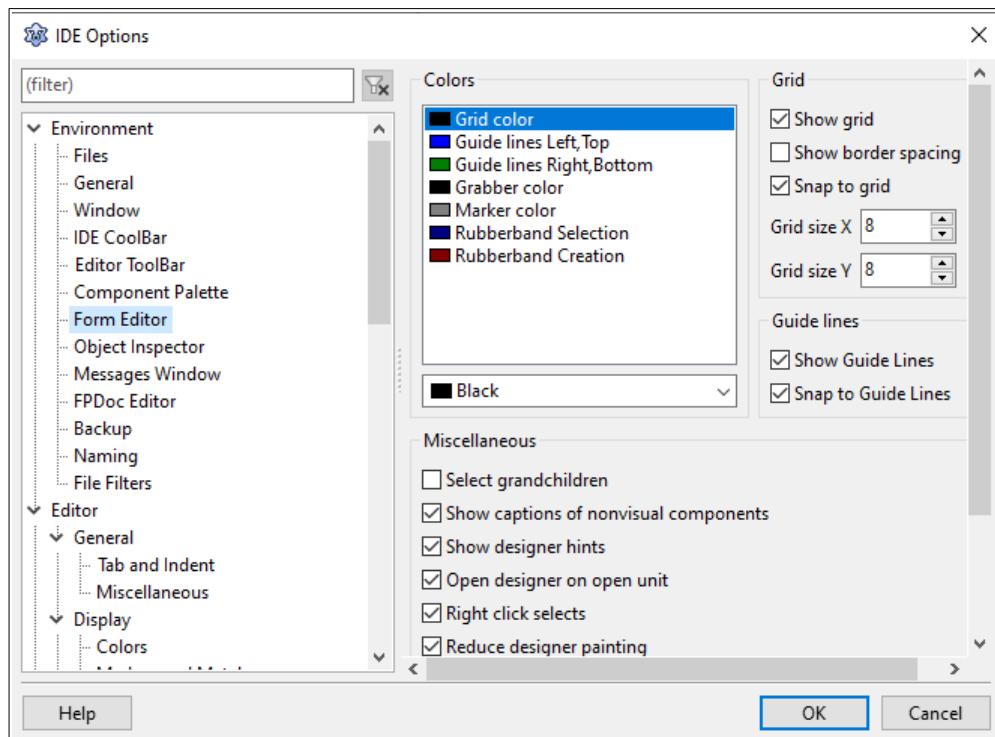


Figure 1.17: Customizing the Form Editor

### 1.4.6 The Lazarus Source Editor

The *Source Editor* (Figure 1.14) is closely related to the Application Form (Figure 1.18)

and they are shown together when a new project is created. By default, the Source Editor is shown in the foreground and covers the Form Editor. The **View | Toggle Form|Unit View** command, the **F12** key or the mouse may be used to switch between them.

The purpose of Source Editor is to create and edit source code, which must follow specific syntactic rules, and a specific algorithm. If the Form Editor determines the program's appearance, then the source code written in the Source Editor is responsible for its behavior.

Initially, the Source Editor contains skeleton code that could be compiled into a working application. This skeleton code appears in the Source Editor automatically, and the programmer augments it with additional code to create the program's functionality.

Note that when Lazarus starts, it automatically loads the *last project* worked on. This is because “**Open last project and packages at start**” was checked (Figure 1.19). This option may be found in the Environment settings under **Tools|Options**.

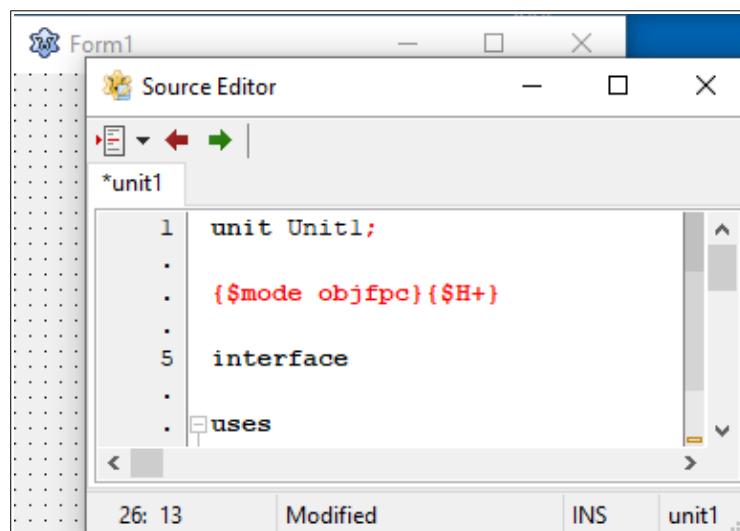


Figure 1.18: The Source Editor over the Form Editor

If you uncheck the box next to “**Open last project and packages at start**”, then Lazarus will create a *new project* on startup.

You can *customize the Source Editor* using the options under **Tools | Options... | Editor** (Figure 1.20). To change a setting, one has only to check or uncheck the box next to the setting and click **OK**.

The *source code font* may be changed in the settings under **Tools | Options... | Display** (Figure 1.21).

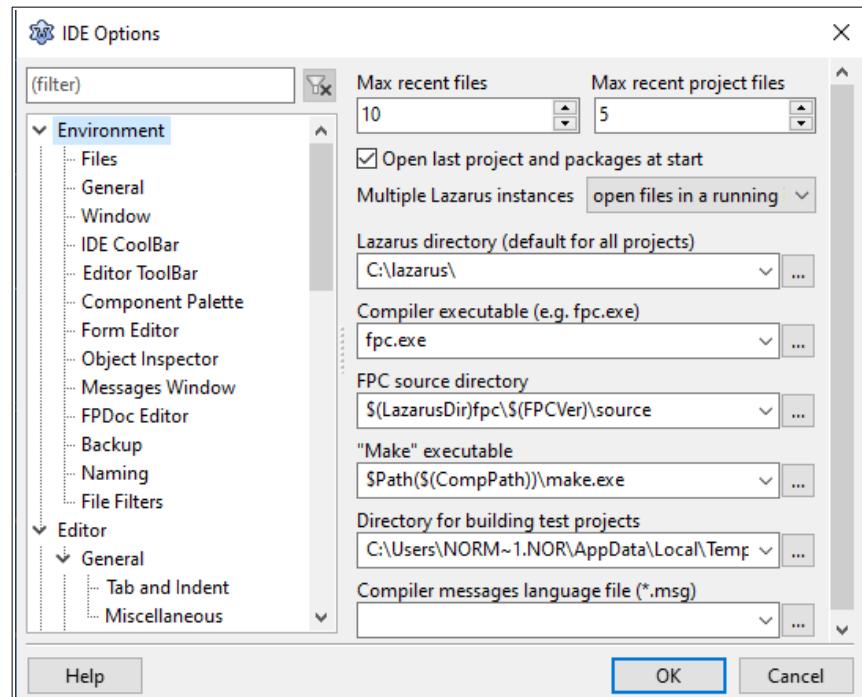


Figure 1.19: File settings dialog in Windows

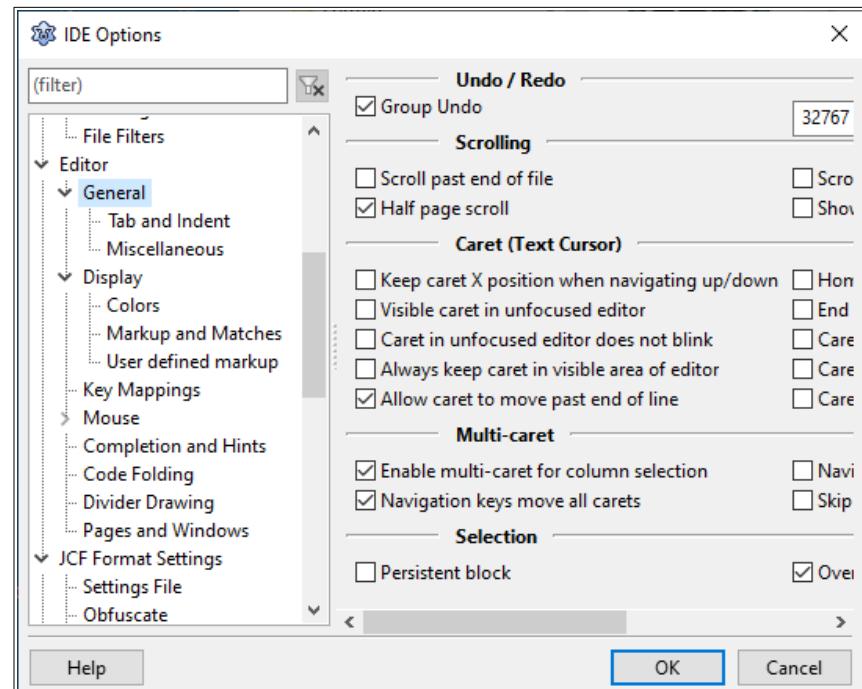


Figure 1.20: Source Editor settings dialog

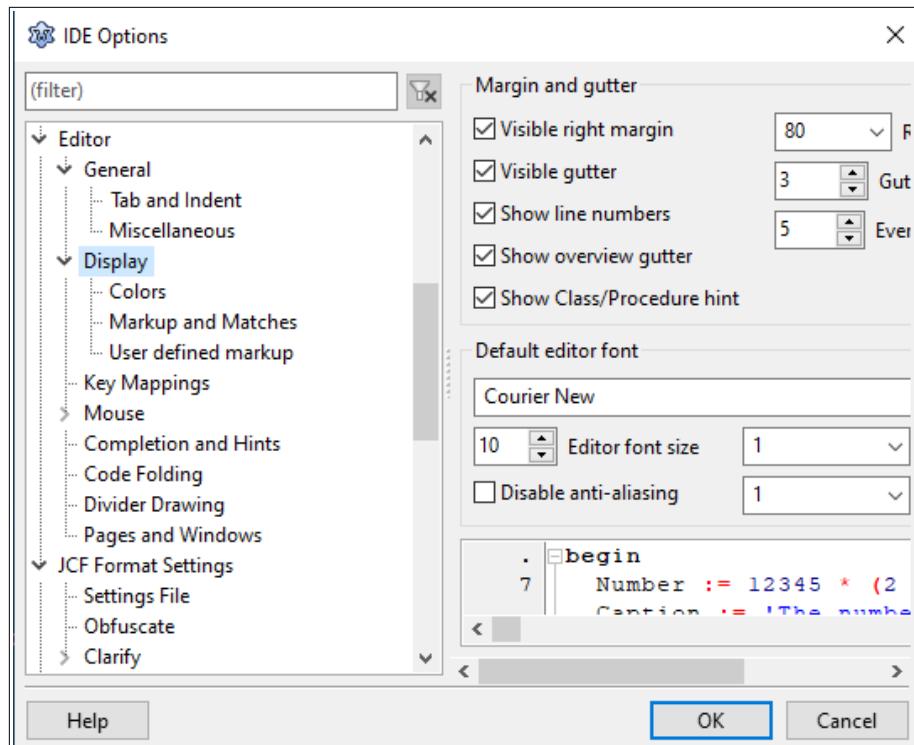


Figure 1.21: Source Editor Display settings dialog

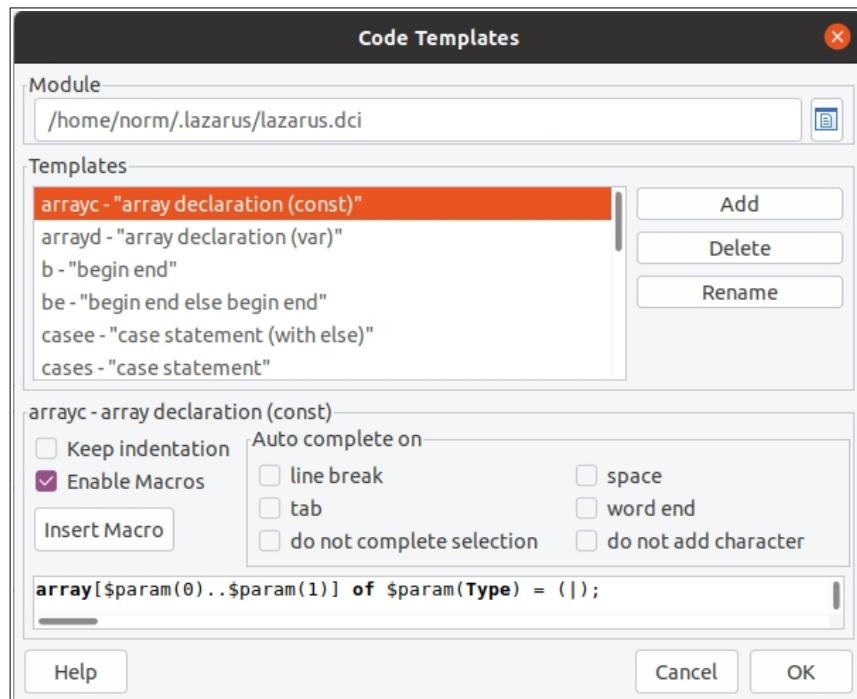


Figure 1.22: Code templates settings dialog in Linux



Figure 1.23: Code template creation dialog

Like many text editors, the Lazarus Source Editor was designed to *work with templates*. In this case, “template” refers to the automatic input of programming structures. For instance, if the user types the character “b” followed by a space, then the editor will display the begin... end structure on the screen. You can customize the code templates in the dialog (Figure 1.22), found under **Tools | Code Templates...**.

To activate a code template, select it (click on it with the mouse) in the Templates list and check the **Enable Macros** setting. Then in the group of check boxes under “**Auto complete on**”, check the “**space**” box. If “**word end**” was checked instead, then the template text would appear in the Source Editor after typing the character “b” and pressing the **Enter** key.

Templates can be added, removed and edited. Special buttons are provided in the **Code Templates** dialog to do this. At the bottom of the window, above the **OK** and **Cancel** buttons, there is a field for entering and editing templates with an active cursor. If you select any template from the list by clicking on it, then the corresponding structure will appear in the input field.

Clicking the **Add** button will bring up the dialog (Figure 1.23) for creating a template. In the **Token** field, enter the character or group of characters which will be associated with the new template. In the **Comment** field, provide a short description of the new template and click the **Add** button. The element and its comment will be added to the list of templates in the **Templates** window. To create the template itself, indicate which language structure will appear when you enter the “token” that you just created. To do this, select the newly created template and in the input field at the bottom of the Code Templates dialog, enter the appropriate text.

For example, let us create a code template for the repeat..until structure, which will be added to the text if the user types r. This will require the following actions:

- click **Add** in the **Code Templates** dialog (see Figure 1.22);

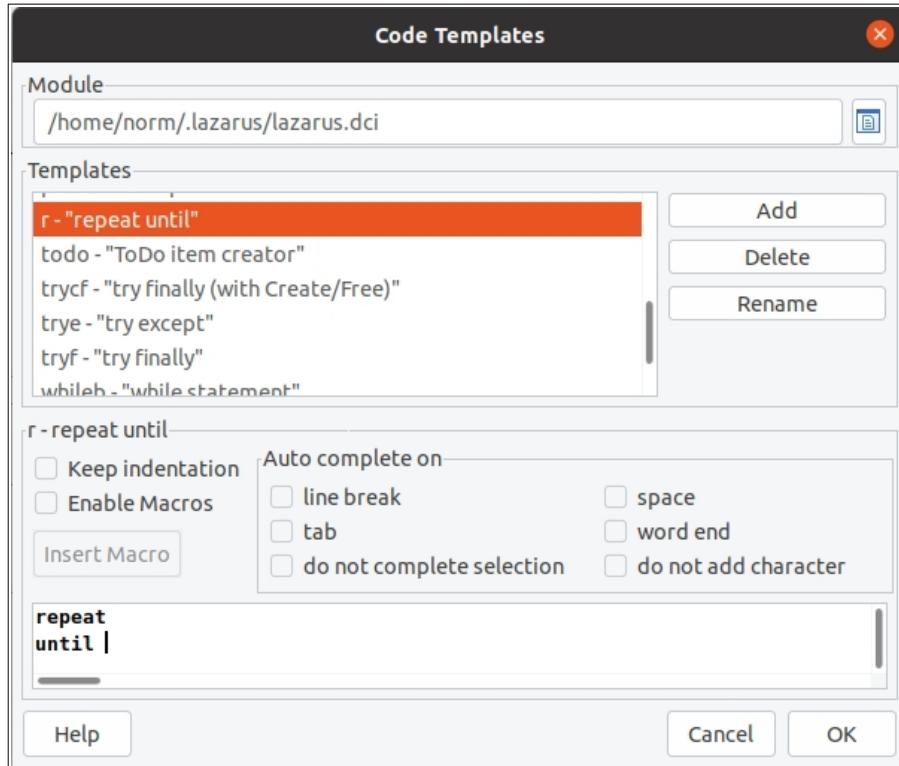


Figure 1.24: Creating a new template in Linux

- in the “**Add code template**” dialog that appears (Figure 1.23), in the **Token** field, enter “r”, and in the **Comment** field the phrase “repeat until”, and click the **OK** button;
- in the **Code Templates** dialog (Figure 1.24 or 1.25) select the line in the **Templates** list that says ‘r -“repeat until”’;
- enter the “repeat until” language structure in the input field at the bottom of the dialog;
- click the **OK** button in the Code Templates window.

To edit a template, select it in the Templates list and edit its corresponding text in the input box at the bottom of the dialog.

To remove a template from the list, select it and click **Delete**.

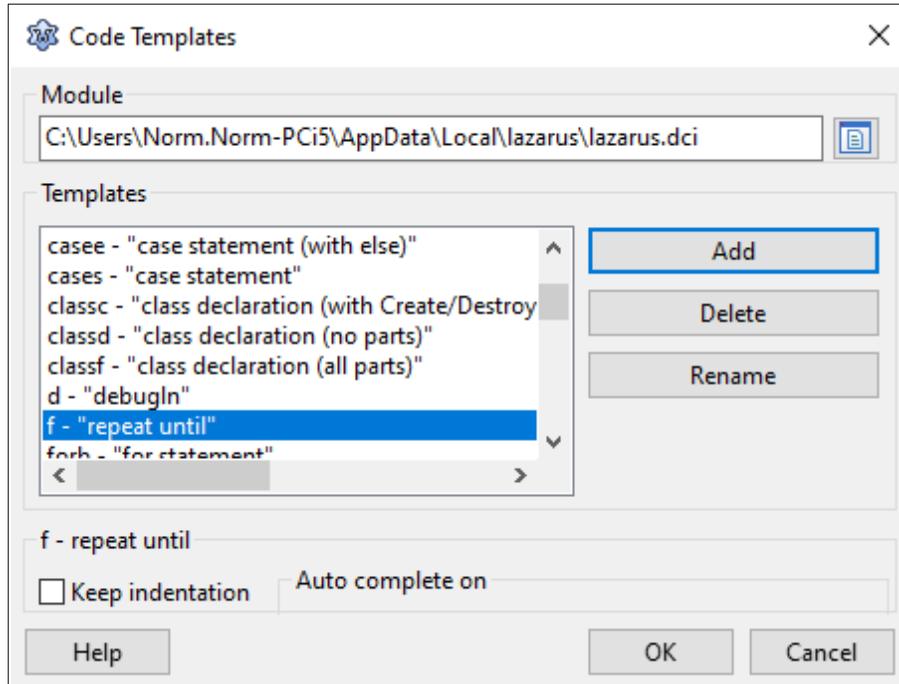


Figure 1.25: Creating a new template in Windows

### 1.4.7 The Component Palette

The *Component Palette* is located<sup>20</sup> under the main menu (Figure 1.14). It consists of a large number of tabs, which contain groups of associated components (Figure 1.26).

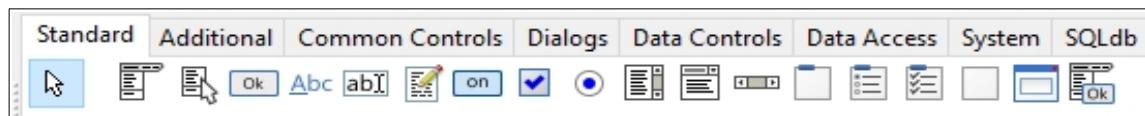


Figure 1.26: The Component Palette

A *component* is a functional interface element that has specific properties. The programmer creates the program's visual elements by placing components such as buttons, check boxes, edit boxes, etc, on a form.

The components in the palette are grouped by their functionality. After creating a project, the **Standard** tab containing the basic elements of windows opens by default. Other tabs can be opened by clicking on them.

Only two clicks are needed to place a component on a form:

- to select a component in the Component Palette;
- to place the component by its upper left corner at the desired position on the form.

### 1.4.8 The Object Inspector

The *Object Inspector* window is located on the left side of the Source Editor window. It contains information about the selected object. In Figure 1.27 the Object Inspector shows information about a newly created form.

The object inspector window has four tabs: **Properties**, **Events**, **Favorites** and **Restricted**, which are used to edit the properties of an object and describe the list of events to which the object will react. The Properties set reflects the external appearance of an object, and the *Events* set its behavior.

The Object Inspector's tabs contain grids. The left column contains the names of properties or events, and the right contains property values or the names of routines that handle events.

To select a property or event, click on the corresponding line.

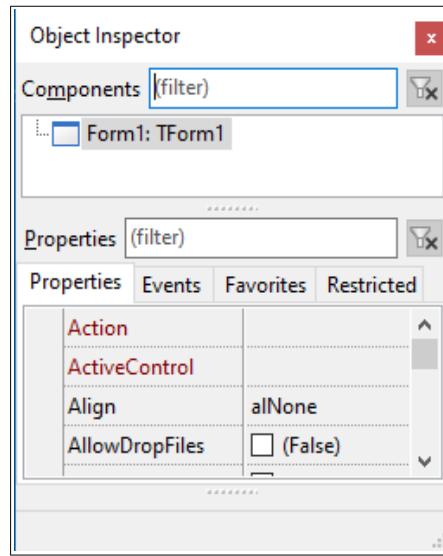


Figure 1.27: The Object inspector window

Properties shown in the grid may be simple or complex. Simple properties have a single value. For example, the Caption property has a character string value, the Height and Width properties each has a numerical value and the Enabled property has a logical value (**true** or **false**). Complex properties have multiple values. Take the Font property, for example. To the left of its name there is a "greater than" (">")

sign. This indicates that the property is complex, and clicking on the sign will reveal its full set of values.

The value of any property can be made editable with a simple mouse click. In some cases, a button with an ellipsis (...) or a button with an arrow pointing down may appear at the end of the edit box. Clicking on the ellipsis will open a dialog box for setting the values of complex properties (Font, for example). Clicking on the down arrow would cause a list of possible property values to drop down from the edit box. Select the appropriate value by clicking on it (Align, for example).

### 1.4.9 Your First Lazarus Program

Creating a Lazarus program occurs in two parts:

- creating its interface, or external appearance;
- writing the Free Pascal code, to make the interface elements work.

As we already know, there is the *Form Editor* to create the interface, and the *Source Editor* for writing the source code. These two editors are tightly integrated, so that placing a *component* in the Form Editor will automatically update the program code in the Source Editor.

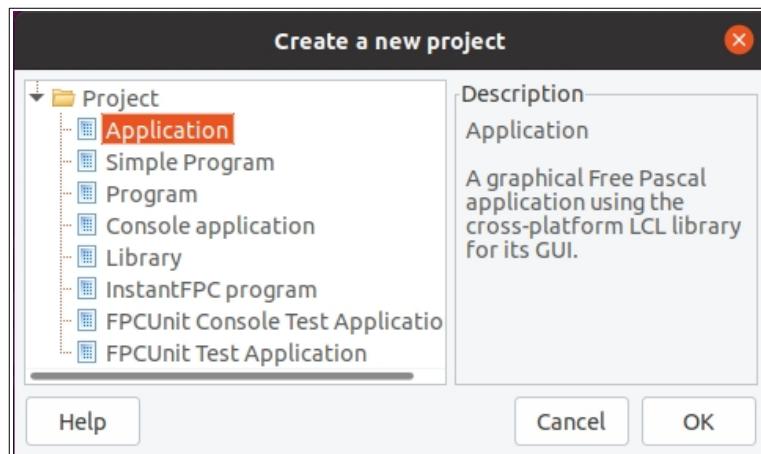


Figure 1.28: Creating a new project in Linux

Let us start our introduction to visual programming by creating a simple program with a button to be clicked, after which it will display a message.

Start by *creating a new project*. To do this, in the main menu execute the command **Project | New Project...**. In the dialog box that appears (Figure 1.28 or 1.29), select **"Application"** from the list and press the **OK** button. These actions will cause the *Form Editor* and *Source Editor* to appear.

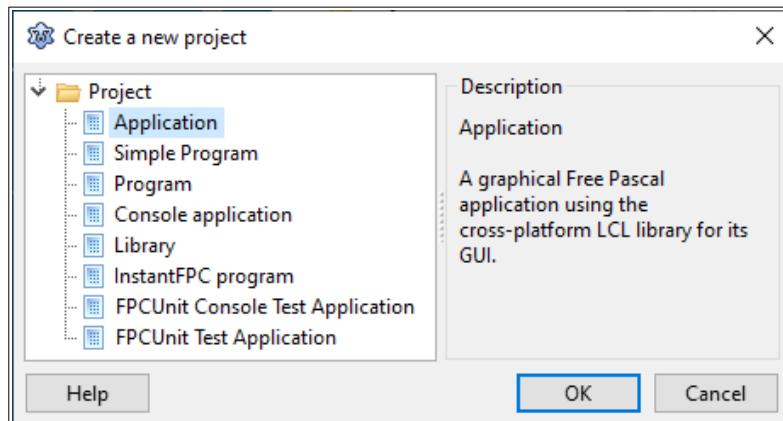


Figure 1.29: Creating a new project in Windows

Save the newly created project using the **Project | Save Project As...** menu command. The “**Save project**” dialog will open (Figure 1.32 or 1.33). Create a new folder called “Example1” using the “**New Folder**” button and save the project in it with the name “Example1.lpi”.

Next, save the project’s source code under its default name “Unit1.pas”, using the “**Save unit**” dialog, which appears automatically or by using the **File | Save As...** menu command (Figure 1.30 or 1.31).

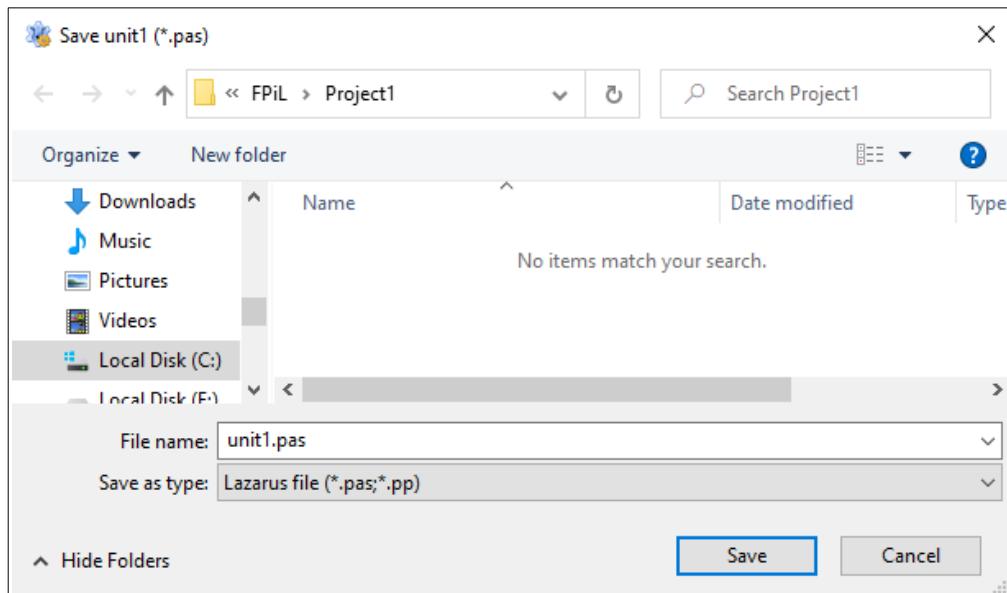


Figure 1.30: The Save unit dialog in Windows

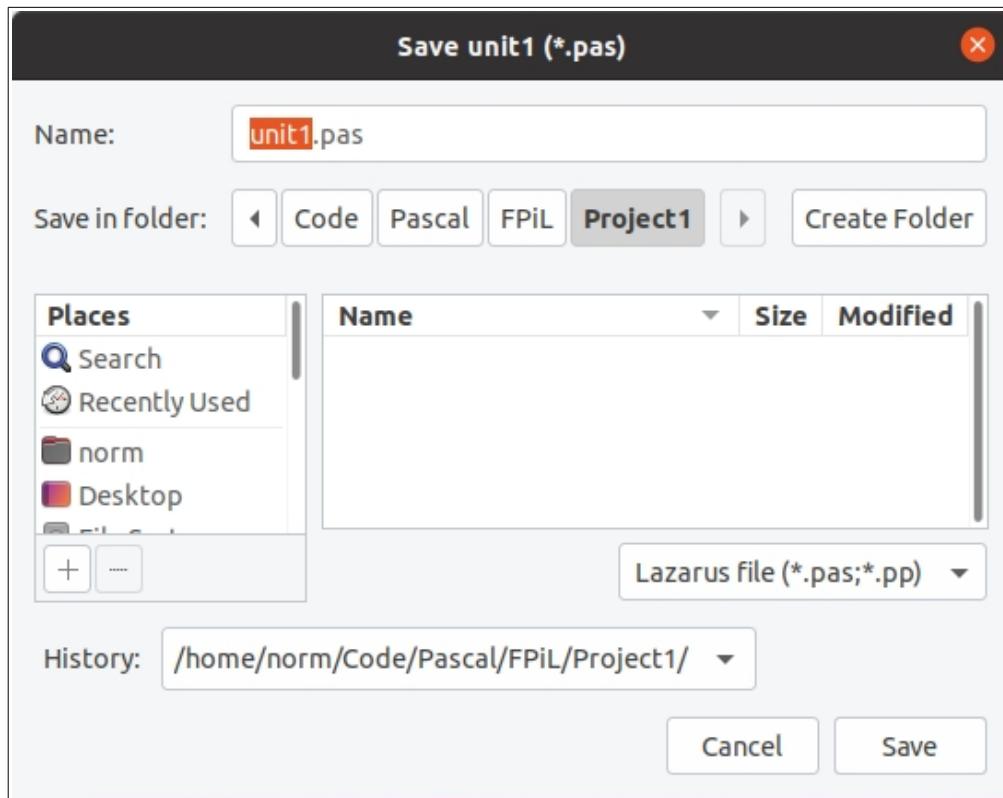


Figure 1.31: The Save unit dialog in Linux

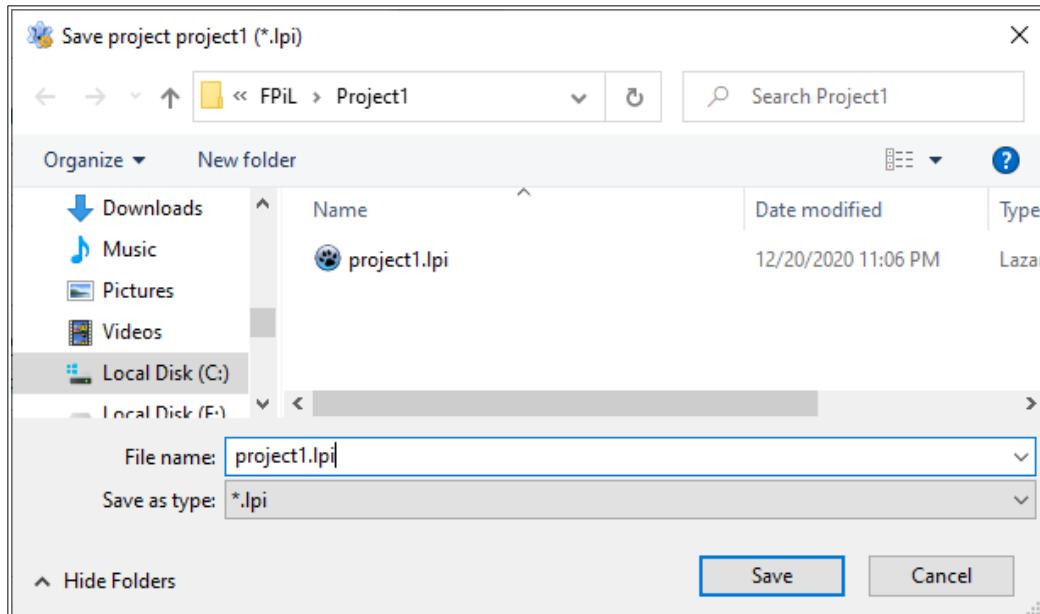


Figure 1.32: The Save project dialog in Windows

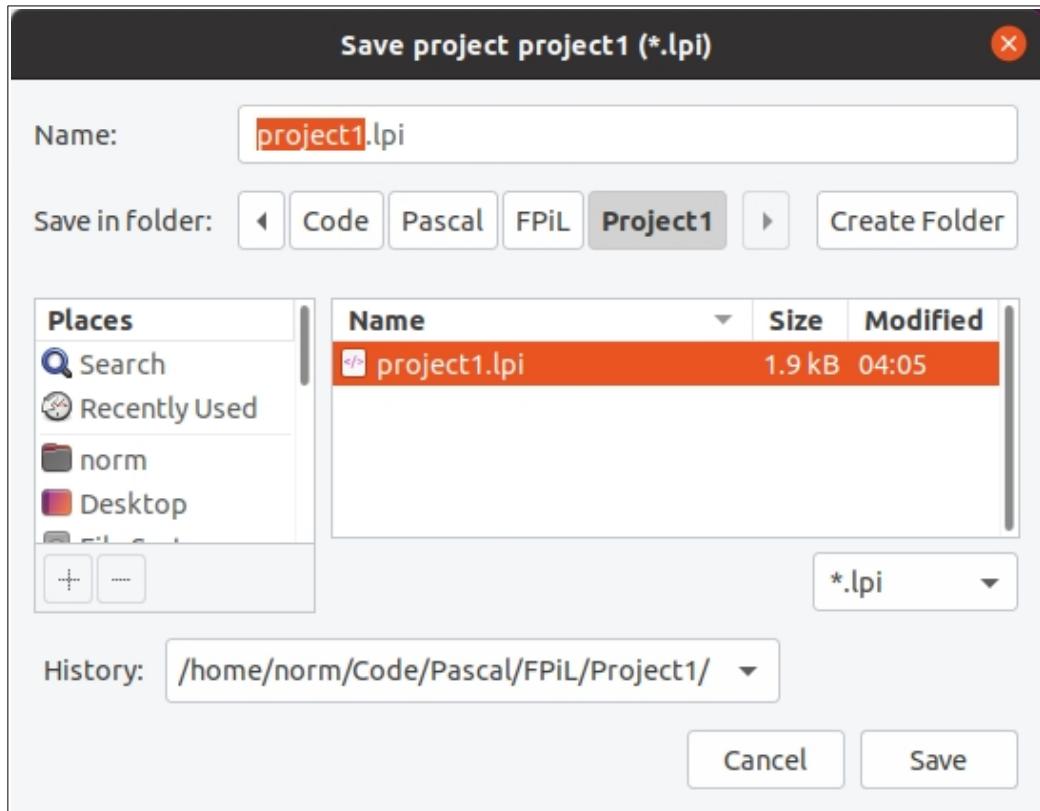


Figure 1.33: The Save project dialog in Linux

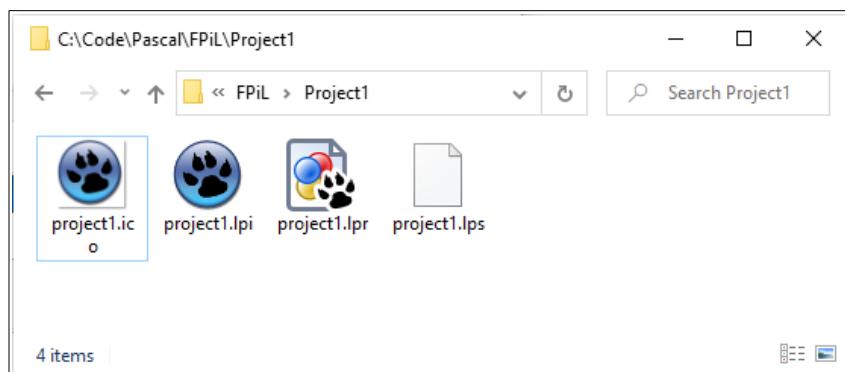


Figure 1.34: Project files on Windows (medium size icons)

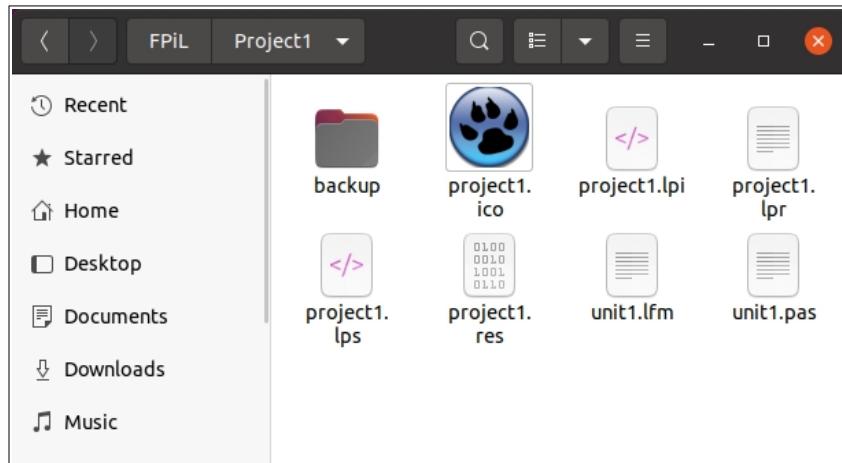


Figure 1.35: Project files on Linux

The actions above created and saved more than just two files. The Example1 folder (Figure 1.34 or 1.35) now has the source code text file **unit1.pas**, a file with information about Form1 **unit1.lfm**,<sup>21</sup> and the files **example1.lpr** and **example1.lpi**, which contain the software environment and project configuration settings.

Further changes will be saved by using the **Project | Save Project** menu command.

We can now start visual programming. The project has one form object, automatically named Form1. Let us change some of its properties using the Object Inspector.

Find Form1's **Caption** property in the Object Inspector. The form's Caption property is responsible for the text displayed in its Title Bar. It defaults to Form1. Change it to "EXAMPLE 1" (Figure 1.36). This change will be reflected immediately (after pressing Enter) in the form, where the title bar text will change to "EXAMPLE 1" (Figure 1.38). Similarly, you can resize forms by assigning new values to the Height and Width properties, but it is easier to do this with the mouse.

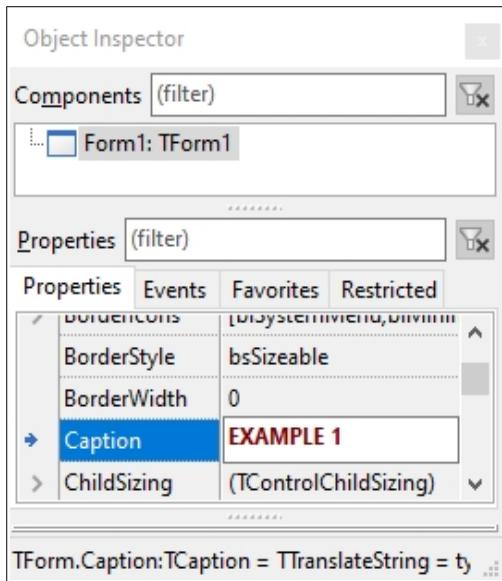


Figure 1.36: Setting the form Caption property

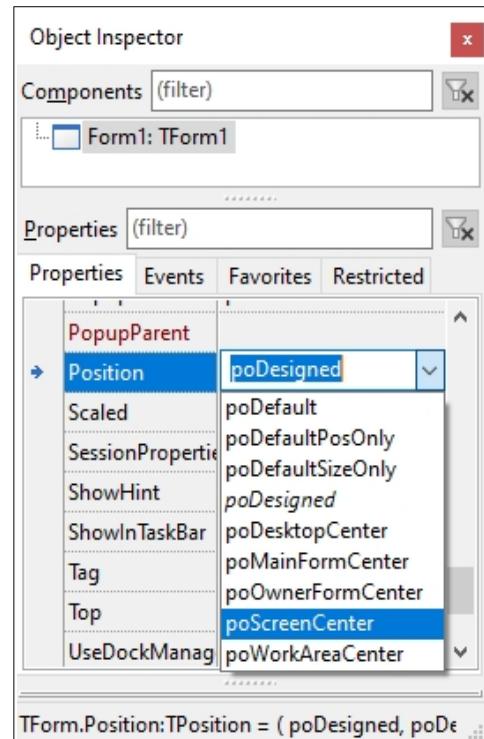


Figure 1.37: Setting the form Position property

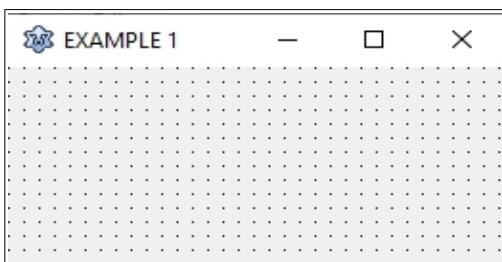


Figure 1.38: Setting the form title bar text

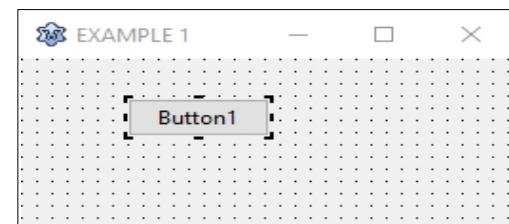


Figure 1.39: Placing a button on the form

Let us change another form property, the Position property (the position of the form on the screen). The values of this property are not entered manually, but are selected from a list (Figure 1.37). If the Position property is set to **poScreenCenter**, then the form will always open in the center of the screen.

Recall that we are writing a program about a button. It's time to put it on the form. To do this, find the TButton control in the Component Palette (Figure 1.26). To place a control on the form, you will need two mouse clicks: the first on the control, the second in the form window. The result is shown in Figure 1.3.

We now have two objects: Form1 and Button1. Recall that changes to property values in the Object Inspector affect the selected object. Therefore, select the Button1 object (by clicking on it, or in the top window) and change its Caption,

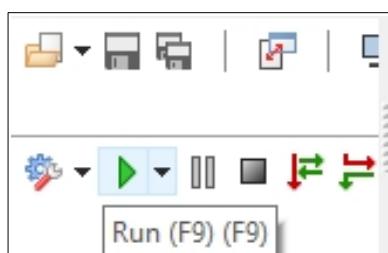


Figure 1.40: The Run button on the Lazarus toolbar

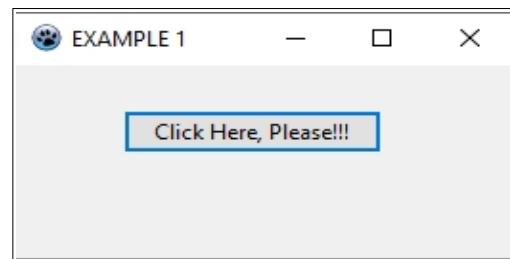


Figure 1.41: The working program

Height and Width properties. Set the Caption property to "Click here, Please!!!", the Width to 135 (pixels) and the Height to 25 (pixels). Save the project (**Project | Save project**, or **Ctrl-S**).

*Run the program* to see how it works. This can be done using the **Run | Run** command, the **F9** key or the **Run** button on the toolbar (Figure 1.40). The window and button should look like Figure 1.41.

Now click on the button and verify that nothing happens. This should not be surprising, because we have not yet written a single line of code. We have developed only the external appearance of the program, but not its functionality. Despite this, the Source Editor will contain the following code (*LResources* will be included under **uses** in Linux):

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, Forms, Controls, Graphics, Dialogs, StdCtrls;
type
  TForm1 =
    class(TForm)
      Button1: TButton;
    private
```

```

{private declarations}
public
{public declarations}
end;
var
  Form1: TForm1;
implementation
{$R *.lfm}
end.
```

The commands in the listing will be discussed later. Go back to the program and think about what the button should do. It should react to some event, such as a mouse click, for example. After starting the program, a window appears with a button in it, with the caption “Click Here, Please!!!”. Let us suppose that if clicked, it will delightfully exclaim “Hurray! IT WORKED!”.

To implement this idea, close the program by closing the window with the button in the usual way (by clicking on the cross in the upper right corner) to return to the edit mode. In the Object Inspector, select Button1 and switch to the **Events** tab. Select the OnClick event, which handles (processes) mouse clicks, and double-click in the field to the right of its name (Figure 1.42).

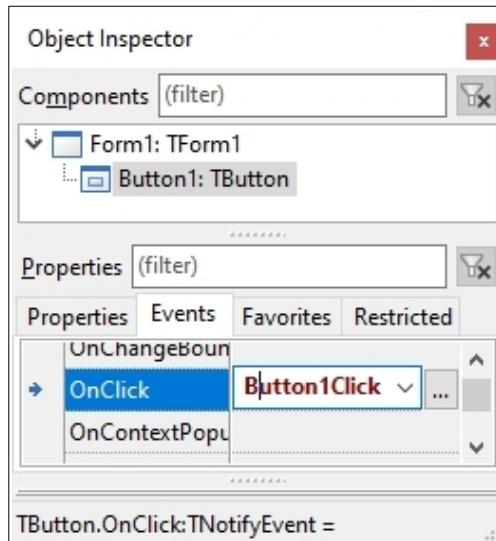


Figure 1.42: Selecting the Onclick event

After double-clicking in the OnClick field, the following code will appear in the Source Editor:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
end;
```

The code block shown above is called a *subroutine*. The purpose of this subroutine can be guessed from its name TForm1.Button1Click. It handles the Click event for the object **Button1** on **Form1**. The code between the words *begin* and *end* will be executed when the specified event (mouse click on Button1) occurs.

Click in the Source Editor between the words *begin* and *end* in the Button1Click subroutine and type:

```
Button1.Caption := 'Hurray! IT WORKED!';
```

This code will change the button property. This was achieved not by using the Object Inspector, but by using Pascal's assignment operator (:=). In other words, the code says "Assign (:=) to the Button1 Caption property the value 'Hurray! IT WORKED!'.

Since the assigned value is a string, it must be enclosed in single quotes. Note that Pascal statements MUST be terminated by a semicolon. The subroutine text in the Source Editor is now:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Caption := 'Hurray! IT WORKED!';
end;
```

Save the program, run it and verify that the button responds to a mouse click (Figure 1.43). You can close the project with the **Project | Close Project** menu command.

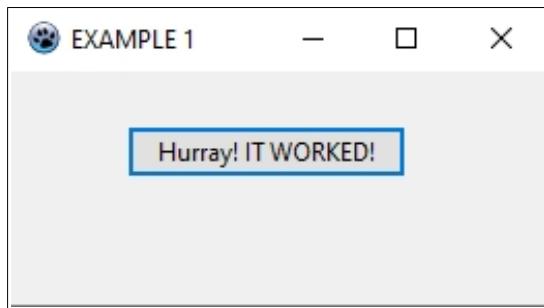


Figure 1.43: The finished program

## 1.4.10 A Useful Program

Our first program did nothing useful. Its main purpose was to introduce visual programming. The next program will also be simple, but somewhat useful. It will convert length expressed in feet and inches to meters. This problem may be

formulated as follows: convert feet and inches to meters, given that 1 foot = 12 inches = 0.3048 meter.

Thus, the program should allow the user to enter two numbers (feet and inches) and display the equivalent value in meters when a button is clicked.

Create a new project, using the menu command **Project | New Project...** and save it in a folder named Example2, using the **Project | Save project As...** menu command. Using the **Standard** tab on the Components Palette, place the following controls on the form: four **Label** objects (text controls that are used for displaying a short text string); two **Edit** objects (these are editable text boxes

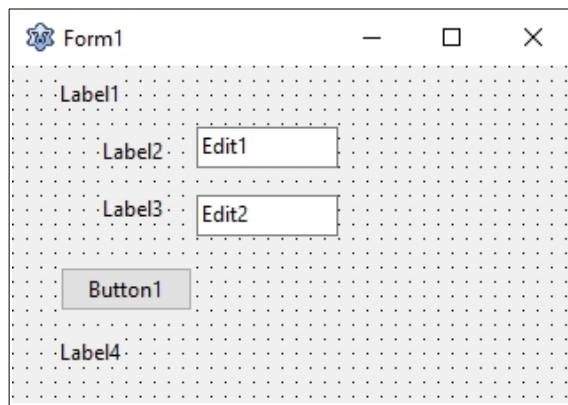


Figure 1.44: Form design

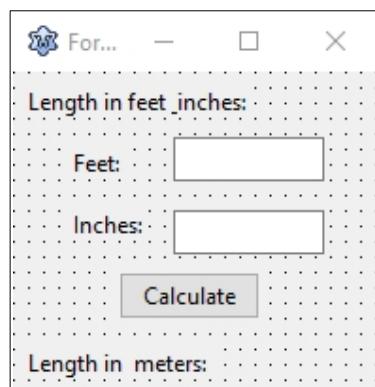


Figure 1.45: Changing form properties

for entering, displaying and editing a text string); one **Button** (this control is often used to execute some commands). The controls should be positioned on the form more or less as shown in Figure 1.44.

The first label will store the text “Length in feet & inches:”. Two labels, Label2 and Label3, will show exactly where to enter information, and Label4 will display the result. The length in feet and inches will be entered in the Edit1 and Edit2 edit boxes respectively. The computation to convert feet and inches to meters will occur

*Table 1.1: Example form properties*

<b>Property</b>	<b>Value</b>	<b>Property Description</b>
Caption	Convert Feet & Inches to Meters	Form header
Height	175	Form height
Width	280	Form width
Font.Name	Arial	Font name
Font.Size	10	Font size

*Table 1.2: Example label properties*

<b>Property</b>	<b>Label1</b>	<b>Label2</b>	<b>Label3</b>	<b>Label4</b>	<b>Property description</b>
Caption	Length in feet and inches:	Feet	Inches	Length in meters:	Control text
Height	16	16	16	16	Control height
Width	150	30	45	110	Control Width
Top	10	45	80	145	Top edge coordinate
Left	25	40	40	25	Left edge coordinate

*Table 1.3: Example edit box properties*

<b>Property</b>	<b>Edit1</b>	<b>Edit2</b>	<b>Property description</b>
Text	space	space	Text in the input field
Height	25	25	Control height
Width	135	135	Control width
Top	40	75	Top edge coordinate
Left	100	100	Left edge coordinate

*Table 1.4: Example button properties*

<b>Property</b>	<b>Value</b>	<b>Property description</b>
Caption	CALCULATE	Button text
Height	25	Control height
Left	115	Control Width
Top	110	Top edge coordinate
Width	100	Left edge coordinate

when Button1 is clicked and the result will be displayed in Label4. Change the properties of the form and the controls placed on it, per Tables 1.1 to 1.4 above. After all the changes are made, the form should look like Figure 1.45.

(Top and Left properties determine the position of a control relative to the top left corner of the form.)

The program interface is now ready. Let us deal with the computational part of the problem. To do this, create an OnClick event handler for Button1, which will handle a double-click on Button1 with the left mouse button. We will write a few statements between the words **begin** and **end** in the Button1 subroutine, which will result in the following code:

```
1.  procedure TForm1.Button1Click (Sender: TObject);
2.  var
3.    foot, inch: integer;
4.    meter: real;
5.  begin
6.    foot:= StrToInt(Edit1.Text);
7.    inch:= StrToInt(Edit2.Text);
8.    meter:= 0.3048*(foot + inch/12.0);
9.    Label4.Caption := 'Length in meters: ' + FloatToStr(meter);
10.   end;
```

Let us examine the program code line by line.

Line 1. *Subroutine header*. It consists of the keyword **procedure** and the subroutine name TForm1.Button1Click, and any arguments, in parentheses.

Line 2. Begins the *variable declaration* block.

Line 3. Declares “foot” and “inch” as integer variables. The computer interprets this statement as a command to allocate sufficient memory to store two integers<sup>22</sup>.

Line 4. Declares “meter” as a real variable.

Line 5. Begins a code block.

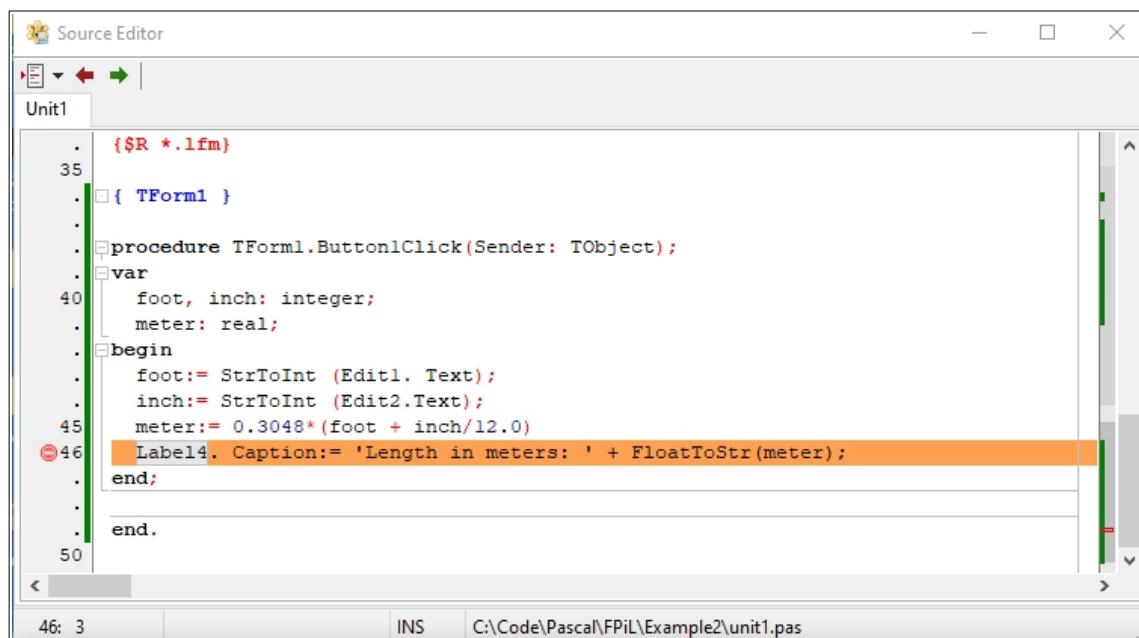
Line 6. The statement does the following. The text in the **Edit1** edit box **Text** property is read into memory. The computer recognizes this information as text, but the **StrToInt** function will convert the text to a whole number (integer). The resulting integer will be saved in memory under the name “foot”. The symbol **:=** denotes an *assignment statement*. For example, “a := 3.14” assigns the value “3.14” to the variable “a”.

Line 7. The text in the **Edit2** edit box is converted to an integer and assigned to the variable “inch”. The statements in Line 6 and Line 7 are performing *data input*

operations.

Line 8. The *value of the expression is calculated* and the result assigned to the variable "meter". Note that the multiplication sign is an asterisk (\*), the division sign is a slash (/), and the addition sign is a plus (+). When writing a mathematical expression in a programming language, all operations must be explicitly shown. You cannot, for example, omit the multiplication sign before the parentheses, as is customary in mathematics.

Line 9. Changes Label4 Caption property. The replacement text will consist of a combination of two text strings. The first is 'Length in meters: ' and the second is the variable "meter", converted to a text string. The + sign in this expression concatenates (joins) the text strings. The FloatToStr function converts the real variable "meter" to a text string. This line performed a data output operation.



The screenshot shows a Pascal source code editor window titled 'Source Editor'. The file is named 'Unit1.pas'. The code is as follows:

```

35   {$R *.lfm}
36
37   TForm1
38
39   procedure TForm1.Button1Click(Sender: TObject);
40   var
41     foot, inch: integer;
42     meter: real;
43   begin
44     foot:= StrToInt(Edit1.Text);
45     inch:= StrToInt(Edit2.Text);
46     meter:= 0.3048*(foot + inch/12.0)
47     Label4.Caption:= 'Length in meters: ' + FloatToStr(meter);
48   end;
49
50 end.

```

The line 'Label4.Caption:= ' + FloatToStr(meter);' is highlighted in orange, indicating a syntax error. The status bar at the bottom shows '46: 3' and 'C:\Code\Pascal\FPiL\Example2\unit1.pas'.

Figure 1.46: Error message

Line 10. Ends a code block.

Now our program is ready to be compiled and run. The compilation process translates the program's source code into binary machine code. To compile, execute the **Run | Quick Compile** menu command. During compilation, the code is checked for syntax errors, and the compilation process will stop if any is found. We intentionally made a mistake in Line 7 of the program code (omitted the semicolon at the end of the expression). The result of compiling a program with a syntax error is shown in Figure 1.46.

The line with the error is highlighted in red, and the Message<sup>23</sup> window under the editor window describes the error. Correct the error and compile the program again.

Execute the **Run** menu command or use the key combination **Ctrl+F9** to do this.

After linking, an executable file will be generated with the same name as the project.

Generally speaking, if the **Run | Quick Compile** menu command is skipped, and the **Run | Build** menu command is executed instead, then all the project files will be compiled and linked.

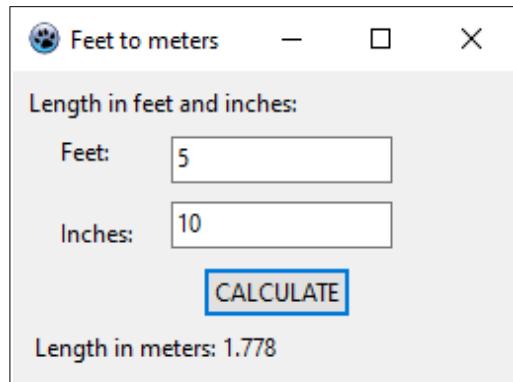


Figure 1.47: The program results

Run the program from the IDE (**Run | Run**) or by executing `./Example2` in Linux or `Example2.exe` in Windows. Figure 1.47 shows the finished program.

Note that errors may occur even after the program runs. For example, our program accepts integer input data. If you enter a fraction, the program will display a warning. In this case, *abort* the program with the **Run | Stop** menu command or (**Ctrl+F2**).

It would not be difficult to *return to this program* to improve it. You would need to execute the **Project | Open project...** menu command or (**Ctrl+F11**), go to the project folder and select the file named `Example2.lpi`.

#### 1.4.11 A Lazarus IDE Console Application

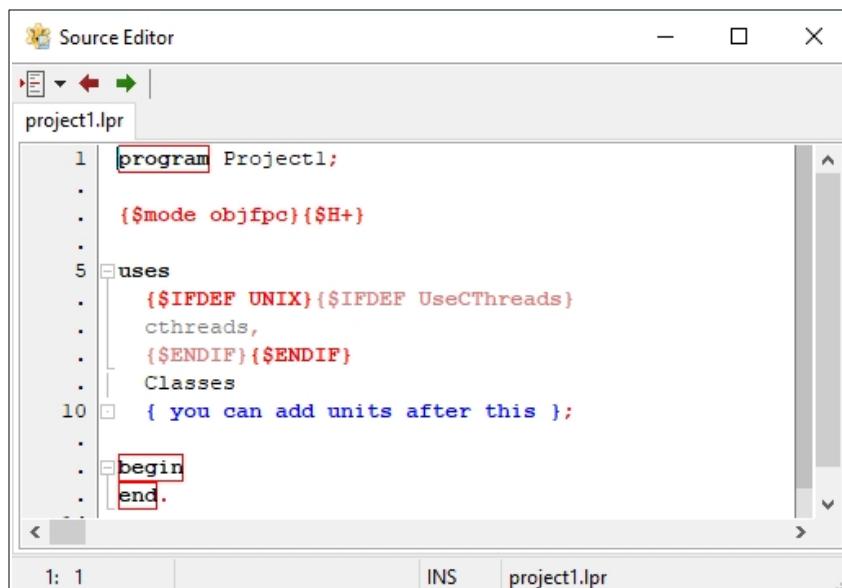
Lazarus can create not only *graphical applications* with a graphical interface, but also Free Pascal *console applications*. Lazarus has a "Program" project template for this.

To *create a console application*, run the **Project | New Project...** menu command, and in the dialog that appears (Figure 1.28 to 1.29) select "**Program**" click **OK**. The

Source Editor will appear on the screen (Figure 1.48), with the *skeleton structure* of a Free Pascal console program.

In general, a *Free Pascal program* consists of a program header, the declaration section and the body of the program itself.

The *program header* consists of the **program** keyword and the program name. In this case (Figure 1.48) it is Project1. That name was automatically assigned to the program, and can be changed.



The screenshot shows the Source Editor window with the title "Source Editor". The file being edited is "project1.lpr". The code editor displays the following Free Pascal code:

```
1 program Project1;
.
.
5 uses
.
.
.
10 begin
end.
```

The code is color-coded: "program" and "begin/end" keywords are in blue, while variable names and other identifiers are in black. Brackets and braces are in grey. Red text highlights the word "uses" and the conditional compilation directives "{\$IFDEF UNIX}{\$IFDEF UseCThreads}...{\$ENDIF}{\$ENDIF}".

Figure 1.48: Source Editor for a console application

The *declaration section* usually includes the declaration of the program's constants, types, variables, procedures and functions. In Figure 1.48 it can be seen that the declaration section begins from the word "**uses**". This is the section for including units. A *unit* is a special program that expands the capabilities of the programming language. In this case, the SysUtils unit is included, which allows you to work with files and directories, and the Classes unit, which works with components.

The declaration section is followed by the statement part of the program, or the program body. It starts with the keyword "**begin**" and ends with the keyword "**end**" and a period. The program body contains language statements for fulfilling the required task.

In addition, there may be comments in the program text. *Comments* are text enclosed in curly braces or starting with two forward slashes. This text is not part of the executable code, but is informational. For example, in our case, the text in curly brackets immediately after the declaration of the units informs the user that

custom code could be added from that point.

Let us look at an example. Assume that we need to *convert degrees to radians*. This problem is known from middle school mathematics and can be formulated as: to convert degrees to radians, multiply degrees by pi divided by 180, multiply minutes by pi divided (180 multiplied by 60) and add the results.

The code for this problem in a console application will look like:

```
program Example3;
{$mode objfpc} {$H+}
uses
  Classes
  {you can add units after this};
var
  deg, min: integer;
  rad: real;
begin
  write ('Enter number of degrees: ');
  readln (deg);
  write ('Enter number of minutes: ');
  readln (min);
  rad := deg * pi / 180 + min * pi / (180 * 60);
  writeln ('Radians =', rad:5:3);
end.
```

*Save, open, compile, link and run* the console application the same way as for a visual project. The results of our program will be as follows:

```
Enter the number of degrees: 165
Enter the number of minutes: 30
Radians = 2.8885199120506E + 000
```

It is easy to see that we added a declaration of variables to the automatically generated text (all variables used in the program must be declared):

```
var
  deg, min: integer;           // Declare two integer variables.
  rad: real;                  // Declare a real variable.
```

and the body of the program:

```
begin                      // Begin program body
  write('Enter number of degrees: '); // Prompt for degrees
  readln(deg);                 // Read in and save deg
  write('Enter the number of degrees: '); // Prompt for minutes
  readln(min);                // Read in and save the min variable
```

```
rad := deg*pi/180 + min*pi/(180 * 60);      // Calculate rad
writeln('Radians = ', rad);                  // Output rad
end.                                         // End program body
```

A simple text dialog between computer and user was used since the program had no graphical user interface. The data input (read) and output (write) statements were used for this.

### 1.4.12 Data Input / Output Statements

The **read** statement accepts *keyboard input*. It may have one of the following formats:

```
read (x1, x2,..., xn);
or
readln (x1, x2,..., xn);
```

where x1, x2,..., xn are a list of input variables. When entering real values, the integer and fraction parts of the number should be separated by a period.

When the program encounters a **read** statement, it stops until the user enters data. When entering numeric values, two numbers are considered separate if there is a space, tab or end-of-line character (Enter) between them. Press the **Enter** key after entering the last value.

The **readln** statement is like the **read** statement, except that the cursor will move to the next line after reading the last data value in the list. Remember that **Enter** will also move the cursor to a new line, regardless of how the data is read.

The **write** and **writeln** statements are used to *display information on screen*. In general, these statements have one of the following formats:

```
write (x1, x2,..., xn);
or
writeln (x1, x2,..., xn);
```

where x1, x2,..., xn are a list of output variables, constants, or expressions. If a list item contains text information, it must be written in single quotes.

The **write** and **writeln** statements display their output list sequentially. If the **writeln** statement is used, then the cursor will move to the next line after displaying information.

So, in our example, the statement “`write ('Enter the number of degrees :=');`” displays “Enter the number of degrees:” on the screen, which prompts the user to enter the value of the variable `deg`, while the “`readln (deg);`” statement will read the value of the variable “`deg`” into memory. The statement “`writeln ('Radians = ', rad);`”

displays two values on the screen: the string "Radians = " and the value of the variable rad.

As a rule, real data will be shown in *floating point format*:

#.#####E±##,

where # is a decimal digit from 0 to 9. To change to fixed-point format, multiply the number before E (the mantissa), by 10 raised to the value of the number after E (the exponent). For example,

0.35469000000E-01 = 0.35469000000\*10<sup>-1</sup> = 0.035469,  
-5.43286710000E+02 = -5.43286710000\*10<sup>2</sup> = -543.28671.

To display numbers in fixed-point format, use *formatted output*. To do this, the **write** or **writeln** statements should be written as:

**write** (identifier: output\_field\_width: fractional\_part);

where identifier is the name of the variable to be displayed; output\_field\_width is number of positions the entire number (integer part, point and fractional part), will occupy; fractional\_part is the number of positions required for the fractional part.

For example, the format of the program's output is:

```
Enter the number of degrees: 165
Enter the number of minutes: 30
Radians = 2.8885199120506E + 000
```

The statement

**writeln** ('Radians = ', rad);

will display the value of the variable rad in floating point format. If you use the statement

**writeln** ('Radians = ', rad:5:3);

then the result of the program will be displayed thus:

```
Enter the number of degrees: 165
Enter the number of minutes: 30
Radians = 2.889
```

where rad is in fixed-point format (the whole number takes five positions with three of them after the decimal point).

**Endnotes:**

- 
- 1 Or in any other language.
  - 2 Instead of the term “compiler”, computer literature may sometimes use the term “computer translator”.
  - 3 Pascal libraries store the object (binary) code of standard functions (such as  $\sin(x)$ ,  $\cos(x)$ , etc.) and procedures.
  - 4 The list of repositories is a list of official sites from which you can install software.
  - 5 <http://www.freepascal.org/download.html>
  - 6 There is a Geany version for Windows (<https://geany.org/download/releases>).
  - 7 This is the subjective advice of the authors.
  - 8 The interface is a dialog for information exchange.
  - 9 Lazarus may possibly be launched in other Linux distributions using a different command in the main menu.
  - 10 <https://lazarus-ide.org/index.php?page=downloads>
  - 11 <http://freepascal.ru/article//lazarus/20080316091540> (in Russian).
  - 12 <https://lazarus-ide.org/index.php?page=downloads>
  - 13 By default, a shortcut is created in the Start | Programs menu.
  - 14 Check the field next to the Create Icon command on the desktop.
  - 15 A comment is text in the code that is ignored by the compiler. Comments may be used as notes for explaining the code or for excluding code while debugging.
  - 16 16A single program in Lazarus is called a project.
  - 17 A component is ready-made program code that can be used when writing a program. The Package menu item is intended to expand the standard set for by adding components from other developers.
  - 18 The commands under the Tools menu are described in Section 1.4.6.
  - 19 The coordinate grid is displayed only when the program is created and *Show grid* is checked in Tools | Options | Environment | Form Editor (see Figure 1.17).
  - 20 You can enable (disable) the **Component Palette** by checking (unchecked) the “Palette is visible” check box in Environment | Component Palette dialog, under the **Tools | options...** menu.
  - 21 File names other than the default Unit1 may be used.
  - 22 Variables and their types will be discussed in more detail in the next chapter.
  - 23 If the Message window is not visible, it can be displayed using the View | Messages menu command.

## Chapter 2. Introduction to Free Pascal

In this chapter, the reader will be introduced to the project structure in the Lazarus environment and the main elements of the Free Pascal programming language: variables, constants, their types, basic operations and functions of the language.

### 2.1 Lazarus Project Structure

A *project* in Lazarus is the set of files used to build the a single *executable file*. In the simplest case, the list of project files includes:

- the Lazarus Project Information file (.lpi);
- the Lazarus Project file (.lpr);
- the Lazarus Resource file (.lrs);
- the Lazarus Form file (.lfm);
- the PAScal source code unit (.pas);

After *compiling* the program, a single executable file will be created from all project files.

A *program unit*, or just a unit, is a separately compiled unit, which is a set of data types, constants, variables, procedures and functions. A unit has the following structure:

```
unit unit_name; // Unit header.  
interface  
    // Declaration section.  
implementation  
    // Implementation section.  
end.           // End of the unit.
```

The *unit header* consists of the reserved word **unit** followed by the unit name and a semicolon. The declarations section, which begins with reserved word **interface**, declares the *program elements*, including types, classes, procedures and functions:

```
interface  
uses unit_list;  
type type_list;  
const constant_list;  
var variable_list;  
procedure procedure_name;  
...  
function function_name;
```

The **implementation** section contains *program code* that implements the mechanism that makes the declared program elements (the code for event handlers, procedures and functions created by the programmer) work. *Procedures and functions* in Lazarus are built on the modular principle<sup>1</sup>.

Lazarus also allows you to develop regular console applications like those created in the Free Pascal IDE (Integrated Development Environment) or Geany IDE, in addition to graphical applications. The authors strongly recommend that users begin learning programming by creating console applications. Thus, let us take a closer look at the structure of a console application.

## 2.2 The Console Application Structure

The structure of a console application looks like this:

```
program header;
uses unit1, unit2,..., unitn;
declaration section;
program body.
```

The program header consists of the reserved word *program*, the program name, which must comply with the rules for creating identifiers (see Section 2.3), and a semicolon, as for example:

```
program my_prog001;
```

The statement “**uses** unit1, unit2,..., unitn” is needed for including units which contain functions and procedures. To use the functions and procedures in a unit, it must be incorporated into the source code by using it in the **uses** statement.

The **declaration** section includes the following subsections:

```
section declaring constants;
type declaration section;
variable declaration section;
section declaring procedures and functions.
```

All variables, types, and constants must be declared before being used in a Free Pascal program. In ANSI Pascal, the order of the sections in a program must be strictly followed, but Free Pascal is less strict about this. A Free Pascal program may have several declaration sections for constants, variables, etc. The structure of the Free Pascal console program can be represented in more detail as follows:

```
program program_name;
uses unit1, unit2,..., unitn;
```

```
const constant_descriptions;
type type_description;
var variable_descriptions;
begin
  statements;
end.
```

The body of the program begins with the word *begin*, followed by Pascal statements implementing the algorithm to solve the problem at hand. A Pascal statement must end with a semicolon. Several statements, each ending with a semicolon, can be placed on the same line, or each statement could be placed on a separate line. Make a special note that the symbol ";" separates one statement from another. You could say that ";" is placed *between statements*. The program body ends with the reserved word **end**. Though several statements could be placed on the same line, it is strongly recommended that each statement be placed on a single line, and that multiple lines be used for complex statements.

Let us take a closer look at the structure of a program:

```
program program_name;
uses unit1, unit2,..., unitn;
const constant_descriptions;
type type_description;
var variable_descriptions;
begin
  statement_1;
  statement_2;
...
  statement_n
end.
```

Here is a simple example of Free Pascal program code:

```
program one;
const
  a = 7;
var
  b, c: real;
begin
  c := a+2;
  b := c-a*sin(a);
  writeln('c = ', c);
  writeln('b = ', b);
end.
```

## 2.3 Language Elements

A Free Pascal program may contain the following characters:

- Latin letters A, B, C,... , x, y, z;
- numbers 0, 1, 2,... , 9;
- special characters +, -, /, =, <,>, [,], (,), ;, :, {}, \$, #, \_, @, ', ^.

Keywords and identifiers consist of characters of the alphabet. *Keywords* are reserved words of the language that have a special meaning to the compiler and must be used only for their defined purpose (as language statements, data types, etc.). An *identifier* may consist of letters, numbers and the underscore character. The identifier starts with a letter or underscore character. Identifiers are used to name different parts (constants, variables, labels, data types, procedures, functions, units, classes) of the language. An identifier cannot contain a space. Uppercase and lowercase letters in names are not considered different, so that for ABC, abc, Abc are considered one and the same name, for example. Each name (identifier) must be unique and cannot be a keyword.

Comments can be used in the source code. If the text starts with two forward slashes “//”, or is enclosed between braces "{}" or placed between pairs characters (\* and \*), then the compiler will ignore it. For instance:

```
{A comment might look like this!}
(* Or like this. *)
// And if you use this method,
// then each comment must start
// with two forward slashes.
```

It is convenient to use comments for explaining the code or for temporarily excluding pieces of code while debugging.

## 2.4 Free Pascal Data Types

A program must process input data to solve the problem being addressed. Data is stored in the computer's memory and can be of many types, such as integers, real numbers, characters, strings, arrays, etc. The *data type* determines how numbers or characters are stored in memory. It determines the size of the memory location where the value will be stored, thus defining its maximum value or accuracy. The memory location is called a *variable*, which has a name (*identifier*), *type* and *value*. The name provides access to the memory location. During execution the value in the variable may change. Before use, every variable must be declared. The declaration of a variable in Free Pascal is done using the **var** keyword:

```
var variable_name: variable_type;
```

If several variables of the same type are declared, then the declaration may look like:

```
var variable_1, variable_2,..., variable_N: variable_type;
```

For example:

```
var
  ha: integer;    // An integer variable is declared.
  hb, c: real;    // Two real variables are declared.
```

A *constant* is a quantity that does not change its value during program execution. A constant is declared as follows:

```
const constant_name = value;
```

For example:

```
const
  h = 3;          // Integer constant.
  bk = -7.521;   // Real constant.
  c = 'abcde';   // A character constant.
```

## 2.4.1 The Character Type

*Character data* always occupies one byte in computer memory. This is because the character type needs only enough memory to store any of the 256 ASCII characters. The character type may be declared using the **char** keyword. For example:

```
var c: char;
```

The values of character variables and constants must be enclosed in single quotation marks, such as: 'a', 'b', '+', in source code.

## 2.4.2 The Integer Types

The *integer data types* can occupy one, two, four, or eight bytes in computer memory. The data value ranges of the integer types are shown in Table. 2.1 below.

Declaring integer variables may be done as follows:

```
var
  b: byte;
  i, j: integer;
  W: word;
  L_1, L_2: longint;
```

Table 2.1: Integer data types

Type	Range	Size, byte
Byte	0... 255	1
Word	0... 65535	2
Longword	0... 4294967295	4
ShortInt	-128... 127	1
Integer	Either Smallint or Longint	4
LongInt	-2147483648... 2147483647	4
Smallint	-32768... 32767	2
Int64	-2 <sup>63</sup> ... 2 <sup>63</sup>	8
Cardinal	Longword	4

### 2.4.3 The Real Type

The internal representation of a *real number* in computer memory is different from that of an integer. It is represented in exponential form  $mE\pm p$ , where  $m$  is the mantissa (integer or real number with decimal point) and  $p$  is the order (integer)<sup>2</sup>. To go from exponential form to the normal representation of a number<sup>3</sup>, multiply the mantissa by ten raised to the power  $p$ . For example:

$$-36.142E+2 = -36.142 \cdot 10^2 = -3614.2;$$

$$7.25E-5 = 7.25 \cdot 10^{-5} = 0.0000725.$$

A real number in Pascal can occupy from four to ten bytes. The ranges of values of real types are shown in Table. 2.2.

Table 2.2: Real data types

Type	Range	Significant digits	Size byte
Single	1.5E-45... 3.4E+38	7 - 8	4
Real	Platform dependent	???	4 or 8
Double	5.0E-324... 1.7E+308	15 - 16	8
Extended	1.9E-4932... 1.1E+4932	19 - 20	10
Comp	2E64+1... 2E63-1	19 - 20	8
Currency	-922337203685477.5808... 922337203685477.5807	19 - 20	8

Examples of declaring real variables:

```
var
  r1, r2: real;
  D: double;
```

## 2.4.4 The DateTime Type

The *DateTime* type TDateTime stores both date and time together. This type occupies eight bytes of computer memory, and is a double type, in which the date is stored in the integer part, and the time in the fraction part.

## 2.4.5 The Boolean Type

*Boolean data types* may only take two values: **true** or **false**. Only one Boolean data type was defined in standard Pascal. The Boolean data types defined in Free Pascal are shown in Table. 2.3.

Table 2.3: Boolean data types

Type	Size, byte	Ord(True)
Boolean	1	1
Boolean16	2	1
Boolean32	4	1
Boolean64	8	1
ByteBool	1	Any nonzero value
WordBool	2	Any nonzero value
LongBool	4	Any nonzero value
QWordBool	8	Any nonzero value

An example of a Boolean variable declaration:

```
var FL: boolean;
```

## 2.4.6 Creating New Types

Despite its rich set of built-in data types, Free Pascal provides a mechanism for creating new types. The **type** keyword is used for this:

```
type new_data_type = type_definition;
```

When a new data type is created, you can declare its variables thus:

```
var variable_list: new_data_type;
```

The use of this mechanism is shown in the following paragraphs.

## 2.4.7 The Enumeration Type

An *enumeration type* lists the values it would accept when it is declared:

```
var variable_name: (value_1, value_2 ,..., value_N);
```

This type can be useful if you need to declare a variable that accepts only a limited number of values. For instance:

```
var
  animal: (fox, rabbit);
  color: (yellow, blue, green);
```

Using enumeration types could make a program clearer:

```
type      // Create a new data type.
  seasons = (winter, spring, summer, autumn);
var       // Declare a variable of type seasons.
  SeasonsOfYear: seasons;
```

## 2.4.8 The Subrange Type

A *subrange type* is defined by specifying its limiting values from an ordinal (the host) type:

```
var
  variable_name: minimum_value.. maximum_value;
```

Note that in this statement, the two dots form a single symbol and a space is not allowed between them. Also, the left range limit should not exceed the right. For example:

```
var
  date: 1..31;
  symb: 'a'..'h';
```

Let us apply this mechanism for creating a new data type:

```
type
  // Create enumeration data type for the days of the week.
  Days_of_week = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  // Create a subrange type - Workdays.
  Workdays = Mon..Fri;
var       // Declare a Workdays variable.
  days: Workdays;
```

## 2.4.9 Structured Types

A *structured data type* may contain multiple elements in a single variable. In Free Pascal, structured types include arrays, strings, records, sets and files.

An *array* is a collection of data of the same type<sup>4</sup>. The number of elements in a static array is declared when the array is declared and it cannot change during program execution.

A static array is declared using the **array...of** keywords:

```
array_name: array [index_list] of datatype;
```

where:

- array\_name is any valid identifier;
- data\_type is any valid type.
- index\_list – the index range; the number of ranges defines the dimension of the array; ranges are separated from each other by a comma, and range limits, which are a subrange type, are separated from each other by the two-dot symbol:

[index1\_start..index1\_end, index2\_start..index2\_end,... ]

For example:

```
var
  // One-dimensional array of 10 integers.
  a: array[1..10] of byte;
  // Two-dimensional array of real numbers (3 rows, 4 columns).
  b: array [1..3, 1..4] of real;
```

Another way to declare an array is to create a new data type. For example:

```
type
  // Declare a three-dimensional array of integers as a new type.
  arr = array [0..4, 1..2, 3..5] of word;
var  // Declare a variable of the newly declared type.
      M: arr;
```

An *element in an array* may be accessed via its index, and in a multi-dimensional array via multiple indices:

array\_name [item\_index]

For example: a [5], b [2, 1], M [3, 2, 4].

A *string* is a series of characters. In Free Pascal, a string may be interpreted as a character array, with each character in the string having an index number, starting from one.

A string must be enclosed in single quotes when used in expressions. String variables may be declared as follows:

**variable\_name: string;**

or:

**variable\_name: string[string\_length];**

For example:

```
const S = 'SENTENCE';
var
  Str1: string;
  Str2: string[255];
  Sentence: string[100];
```

If the string length is not specified when it is declared then it may be up to 255 characters long. In the above example, the strings Str1 and Str2 are the same length.

A *record* is a data structure consisting of a fixed number of elements called record fields. Unlike arrays, record fields may contain different types of data. When declaring a record type, use the key words **record..end**:

**record\_name = record field\_list end;**

where record\_name is any valid identifier, and field\_list is the record fields. For example:

```
// Declare the record structure.
// Each record contains information about a student and
// consists of two fields: name and age.
type
student = record
  name: string;
  age: byte;
end;

var
  // Declare three objects of the declared record type:
  a, b, c: student;
```

A field in a record may be accessed using a compound name format: record\_.field\_name. For example:

```
a.name := 'Ivan Johnson';
a.age := 18;
b.name := a.name;
```

The *with* statement **with** record\_name **do** makes it easy to access record fields:

```
with a do
begin
  name := 'Peter Peters';
  age := 19;
end;
```

A *set* is a group of objects that are logically related to each other. The number of elements in the set can vary from 0 to 255. A set with no element is called an empty set. To declare a set, use the keywords **set of**:

```
set_name = set of base_type;
```

For example:

```
type
  TwoNumbers = set of 0..1;
var
  Num1, Num2, Num3: TwoNumbers;
```

A *file* is a named area of the computer's external memory. A file contains elements of the same type (any type, except files). The file size is not specified when it is declared and is limited only by the capacity of the disk on which it is stored.

In Free Pascal, a *typed file* may be declared as follows:

```
filename_variable = file of data_type;
```

*an untyped file*:

```
filename_variable = file;
```

and a text file:

```
filename_variable = TextFile;
```

For example:

```
var
  f1: file of byte;
  f2: file;
  f3: TextFile;
```

## 2.4.10 Pointers

As a rule, when processing a variable declaration, such as

```
variable_name: variable_type
```

the compiler automatically allocates memory for the variable, depending on its

type. The variable may be accessed through its name. Calls to the variable are replaced by the address of the memory location where the variable's value is stored. On termination of a routine in which a variable was declared, its memory is automatically freed.

The value of a variable can be accessed another way - by creating variables to store memory addresses. Such variables are called pointers.

A *pointer* is a variable whose value is the memory address where an object of a specific type is saved (another variable). A pointer must be declared, like any variable. When declaring a pointer, always include the object type to be saved at the memory address:

```
var variable_name: ^type;
```

Such pointers are called *typed*. For example:

```
var
  p: ^integer;
  // An integer variable will be stored,
  // at the address in the variable p.
  // In other words, p points to an integer.
```

In Free Pascal, you can declare a pointer that is not associated with any specific data type. To do this, use the **pointer** keyword:

```
var variable_name: pointer;
```

For example:

```
var
  x, y: pointer;
```

Such pointers are called *untagged* and are used for processing data of unknown structure and type, which may change during the program execution, that is, dynamically.

## 2.5 Operators and Expressions

An *expression* specifies the order of performing actions on data and consists of operands (constants, variables, function calls), parentheses and operators, such as  $a+b*\sin(\cos(x))$ . Table 2.4 presents the main operators in Free Pascal.

In complex expressions, the order in which operations are performed is determined by operator precedence, or priority. Free Pascal adopts the following *order of operator precedence*:

- 1) not.

2) \*, /, div, mod, and, shl, shr.

3) +, -, or, xor.

4) =, <>, >, <,> =, <=.

Using parentheses in expressions allows you to change the order of evaluation. Let us move on to a detailed examination of the basic operations of the language.

*Table 2.4: Main operators in Free Pascal*

Operator	Action	Types of Operands	Resulting Type
:=	assignment	any, but same type for both	same as operands
+	addition	integer/real	integer / real
+	string concatenation	string	string
-	subtraction	integer/real	integer / real
*	multiplication	integer/real	integer / real
/	division	integer/real	real
div	integer division	integer	integer
mod	remainder (from division)	integer	integer
not	arithmetic / Boolean negation	integer/Boolean	integer / Boolean
and	Arithmetic / Boolean AND	integer/Boolean	integer / Boolean
or	Arithmetic / Boolean OR	integer/Boolean	integer / Boolean
xor	Arithmetic / Boolean exclusive OR	integer/Boolean	integer / Boolean
shl	Shift left	integer	integer
shr	Shift right	integer	integer
in	Set membership test	set	Boolean
<	Less than	not structured	Boolean
>	More than	not structured	Boolean
<=	Less than or equal to	not structured	Boolean
>=	Greater than or equal to	not structured	Boolean
=	Equal to	not structured	Boolean
<>	Not equal to	not structured	Boolean

In complex expressions, the order in which operations are performed is determined by operator precedence, or priority. Free Pascal adopts the following *order of operator precedence*:

- 1) not.
- 2) \*, /, div, mod, and, shl, shr.
- 3) +, -, or, xor.
- 4) =, <>, >, <,>, =, <=.

Using parentheses in expressions allows you to change the order of evaluation. Let us move on to a detailed examination of the basic operations of the language.

### 2.5.1 Arithmetic Operators

Operators +, -, \*, / are called as *arithmetic operators*. Their purpose is well known.

*Integer arithmetic operators* operate only on integer operands and include:

- **div** - integer division, which returns the integer part of the quotient, while the fractional part is discarded). For example,  $17 \text{ div } 10 = 1$ ;
- **mod** – returns the remainder of a division operation, For example,  $17 \text{ mod } 10 = 7$ .

*Bitwise arithmetic operators* include the following operators: **and**, **or**, **xor**, **not**, **shl**, **shr**. In bitwise arithmetic, actions are performed on the binary form of integers.

*Arithmetic AND*<sup>5</sup>. Both operands are converted to binary, then Boolean bitwise multiplication is performed on the operands, per following rules:

$$\begin{array}{ll} 1 \text{ and } 1 & = 1 \\ 1 \text{ and } 0 & = 0 \\ 0 \text{ and } 1 & = 0 \\ 0 \text{ and } 0 & = 0 \end{array}$$

For example, if  $A = 14$  and  $B = 24$ , then their binary forms are:  $A == 000000000001110$  and  $B = 0000000000011000$ . The Boolean multiplication of A and B will result in  $000000000001000$ , or 8 in decimal notation (Figure 2.1). So  $A \& B = 14 \& 24 = 8$ .

<b>A =</b>	<b>0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0</b>	<b>= 14</b>
<b>B =</b>	<b>0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0</b>	<b>= 24</b>
<b>A and B =</b>	<b>0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0</b>	<b>= 8</b>

Figure 2.1: Boolean multiplication example

*Arithmetic OR*<sup>6</sup>. In this case, both operands are also converted to their binary form,

after which Boolean bitwise addition is performed on them, per the following rules:

$$\begin{aligned} 1 \text{ or } 1 &= 1 \\ 1 \text{ or } 0 &= 1 \\ 0 \text{ or } 1 &= 1 \\ 0 \text{ or } 0 &= 0 \end{aligned}$$

For example, the result of the Boolean addition of the numbers  $A = 14$  and  $B = 24$  will be  $A \text{ or } B = 30$  (Figure 2.2).

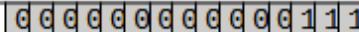
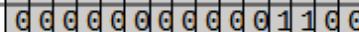
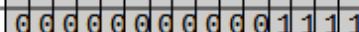
$A =$		$= 14$
$B =$		$= 24$
$A \text{ or } B =$		$= 30$

Figure 2.2: Boolean addition example

*Arithmetic exclusive OR (xor).* Both operands are converted to their binary form, after which a Boolean bitwise xor operation is performed on them, per the following rules:

$$\begin{aligned} 1 \text{ xor } 1 &= 0 \\ 1 \text{ xor } 0 &= 1 \\ 0 \text{ xor } 1 &= 1 \\ 0 \text{ xor } 0 &= 0 \end{aligned}$$

*Arithmetic negation (not).* This operator acts on a single operand. Using the **not** operator causes the bitwise inversion<sup>7</sup> of the binary form of a number (Figure 2.3).

$A =$		$= +13$
<b>not</b> $A =$		$= -14$

Figure 2.3: Arithmetic negation not A

*Left shift (M shl L).* The binary form of M is shifted to the left by L positions. Consider the operation  $15 \text{ shl } 3$ . The binary form of 15 is 1111. Shifting it 3 positions to the left will produce 1111000, which is 120 in the decimal system. So,  $15 \text{ shl } 3 = 120$  (Figure 2.4).

Note that shifting one bit to the left corresponds to multiplying by two, two bits to multiplying by four, and three bits to multiplying by eight. Thus, the operation  $M \text{ shl } L$  is equivalent to multiplying M by 2 to the power L.

*Right Shift (M shr L).* In this case, the binary form of the number M is shifted to the right by L positions, which is equivalent to integer division of the number M by 2 to the power L. For example,  $15 \text{ shr } 1 = 7$  (Figure 2.5),  $15 \text{ shr } 3 = 1$ .

```
000000000000111115  
◀ shift left 3 positions  
000000000011110000120
```

Figure 2.5: Left shift 15  $\text{shl } 3 = 120$

```
00000000000000111115  
Shift right 1 position ►  
000000000000001117
```

*Figure 2.4: Right shift 15 shr 1*

## 2.5.2 Relational Operators

*Relational operators* are applied to two operands and return a Boolean type as the result. There are seven such operators: `>`, `>=`, `<`, `<=`, `=`, `<>` and `in`. The result of a relational operation is a Boolean value of **true** or **false**.

The purpose of the `>`, `>=`, `<`, `<=`, `=`, `<>` operators is clear. The result of `2 > 3` is **false**, and `5 ≥ 4` is **true**. The result of `x ≤ y` depends on the specific values of `x` and `y`. Let us explain how the `in` operator works. The first operand of this operation is an expression, and the second is a **set** consisting of elements of the same type. The result of the operation is **true** if the left operand is a member of the set on the right.

### 2.5.3 Boolean Operators

Free Pascal defines the following *Boolean operators*: **or**, **and**, **xor**, **not**. Boolean operations are performed on the Boolean values **true** and **false**. Table 2.5 shows the results of Boolean operations.

*Table 2.5: Boolean operations*

Table 2: Boolean Operations					
A	B	Not A	A and B	A or B	A xor B
T	T	F	T	T	F
T	F	F	F	T	T
F	T	T	F	T	T
F	F	T	F	F	F

Boolean and arithmetic relational operators can be used in Boolean expressions.

## 2.5.4 Pointer Operators

When working with pointers, the address and dereferencing operators are used.

The *address operator* @ returns the address of its operand. For example:

```
var
  a: real;           // Declare a real variable a
  adr_a: ^real;     // Declare a pointer to a real type
...
```

---

```
adr_a := @a;           // Obtain the address of a and
// save it in the variable adr_a
```

The *dereferencing operator* `^` returns the value of the variable stored at the given address:

```
var
  a: real;           // Declare a real variable a
  adr_a: ^real;     // Declare pointer to type real
  ...
  a := adr_a^;      // The operator writes to the variable a
  // value stored at adr_a.
```

## 2.6 Standard Mathematical Functions

Free Pascal defines *standard mathematical functions*, which operate on arithmetical operands (Table 2.6).

Raising  $x$  to the power  $n$  could be a problem. If  $n$  is a positive integer, then you can multiply  $x$  by itself  $n$  times (which gives a more accurate result and is preferable) or use the formula<sup>9</sup>:

$$\begin{cases} x^n = e^{n \ln(x)}, & \text{for } x > 0, \\ x^n = -e^{n \ln|x|}, & \text{for } x < 0, \end{cases}$$

which may be programmed using standard language functions:

- **exp(n\*ln(x))** for positive  $x$ ;
- **-exp(n\*ln(abs(x)))** for negative  $x$ .

The same formula can be used to raise  $x$  to a fractional power  $n$ , where  $n$  is a proper fraction of the form  $k/l$ , and the denominator  $l$  is an odd number. If  $l$  is even, then finding the root of an even power may have restrictions if ( $x > 0$ ).

When raising the number  $x$  to a negative power, remember that

$$x^{-n} = 1/x^n.$$

Thus, when programming an expression that raises a number  $x$  to the power  $n$ , carefully analyze the values that  $x$  and  $n$  can take, since in some cases raising  $x$  to the power  $n$  is not possible.

*Functions for working with strings* are shown in Table. 2.7. Note that the `+` operator is like the `concat()` function.

*Table 2.6: Standard math functions*

<b>Name</b>	<b>Return Type</b>	<b>Argument Type</b>	<b>Action</b>
abs(x)	integer / real	integer / real	Absolute value of a number
sin(x)	real	real	sine
cos(x)	real	real	cosine
arctan(x)	real	real	Arc tangent
pi	no argument	real	The number $\pi$
exp(x)	real	real	exponent $e^x$
ln(x)	real	real	natural logarithm
sqr(x)	real	real	square of a number
sqrt(x)	real	real	square root of a number
int(x)	real	real	integer part of number
frac(x)	real	real	fraction part of number
round(x)	real	integer	rounding a number
trunc(x)	real	integer	fractional truncation numbers
random(n)	integer	integer	random number from 0 to n

*Functions defined in the Math unit<sup>8</sup>*

arcos(x)	real	real	Arc cosine
arcsin(x)	real	real	Arc sine
arccot(x)	real	real	Arc cotangent
arctan2(y, x)	real	real	Arc tangent y/x
cosecant(x)	real	real	cosecant
sec(x)	real	real	secant
cot(x)	real	real	cotangent
tan(x)	real	real	tangent
lnXP1(x)	real	real	natural logarithm of x+1
log10(x)	real	real	Logarithm of x to base 10
log2(x)	real	real	logarithm of x to base 2
logN(n, x)	real	real	Logarithm of x to base n

The purpose of the function FloatToStrF(V, F, P, D) is to produce a formatted string output from a real number. F is the format parameter and its values are shown in Table 2.8. P is the precision parameter which sets the number of digits in the entire output string and D is the number of digits after the decimal point.

*Table 2.7: String processing functions*

<b>Name</b>	<b>Argument Type</b>	<b>Return Type</b>	<b>Action</b>
<i>Working with strings</i>			
length(S)	string	integer	Length of string S
concat(S1,S2,...)	strings	string	Join S1, S2,...
copy(S, n, m)	string, integer, integer	string	Copy n characters of string S, starting from the m-th position
delete(S, n, m)	string, integer, integer	string	Delete n characters from string S, starting from the m-th position
insert(S, n, m)	string, integer, integer	string	Insert n characters into string S, starting at the m-th position
pos(S1,S2)	strings	integer	Position number from which begins occurrence of S2 in S1
chr(x)	integer	character	Returns the character with ASCII code x
ord(c)	character	integer	Returns the ASCII code for the character c
<i>Converting strings to other types</i>			
StrToDate( <i>S</i> )	string	date and time	Converts characters in string S to a date-and-time value
StrToFloat( <i>S</i> )	string	real	Converts characters in string S to a real value
StrToInt ( <i>S</i> )	string	integer	converts characters in string S to an integer
Val ( <i>S, X, Code</i> )	string	real	Converts string S into a real value which is saved in X, and sets Code = 0 if there were no errors
<i>Reverse conversion</i>			
DateTimeToStr( <i>V</i> )	date and time	string	Converts a date-and-time value to a string
FloatToStr( <i>V</i> )	real	string	Converts a real value to a string
IntToStr( <i>V</i> )	integer	string	Converts an integer to a string
FloatToStrF( <i>V, F, P, D</i> )	real	string	Converts a real number V to a string, using the format specified in F with parameters P and D

Table 2.8: *FloatToStrF* parameters

<b>Format</b>	<b>Appointment</b>
ffExponent	Scientific format. P is the mantissa width, less '.' and '-' (e.g. 5), D is the power width (2): 1.2345E+10.
ffFixed	Fixed point format. P is ignored. The fractional part is rounded to D digits (e.g. 3): 12.345.
ffGeneral	General format. Uses the shorter output of either ffFixed or ffExponent.
ffNumber	Same as ffFixed but with thousand separators inserted into output.
ffCurrency	Money format. Same as ffNumber, except P is ignored and the string ends with the monetary unit symbol.

The code below is an example of how *FloatToStrF* works:

```

var
  n: integer;
  m: real;
  St: string;
begin
  n := 5;
  m := 4.8;
  St := 'A. Johnson ';
  { To display the real number m, 4 positions are allocated,
    including one for the decimal point and two for two positions
    after it. The end-of-line character is chr(13). }

  Label1.Caption := 'Student ' + St + 'passed ' +
    IntToStr(n) + ' tests.' + chr(13) + 'His average score was ' +
    FloatToStrF (m, ffFixed, 4, 2);
end;

```

This will produce the following output:

Student A. Johnson passed 5 tests.  
His average score was 4.80.

Table 2.9 shows the functions for working with date and time.

If the size of a variable is unknown when it must be created, use dynamic memory<sup>10</sup>. In Free Pascal, *memory may be allocated* using the functions shown in Table 2.10.

Table 2.9: Date and time functions

Name	Argument Type	Return Type	Action
date	no argument	date-time	Returns the current date
now	no argument	date-time	Returns the current date and time
time	no argument	date-time	Returns the current time

Table 2.10: Memory functions

Name	Action
adr(x)	Returns the address of the argument x.
dispose(p)	Returns memory to the heap that was previously reserved for typed pointer p.
Freemen(p, size)	Returns memory to the heap that was previously reserved for the untyped pointer p.
GetMem(p, size)	Reserves a chunk of memory on the heap of size bytes and puts its address in untyped pointer p.
New(p)	Reserves a chunk of heap for saving a variable and places the address of its first byte in the typed pointer p.
SizeOf(x)	Returns the length in bytes of the internal representation of x.

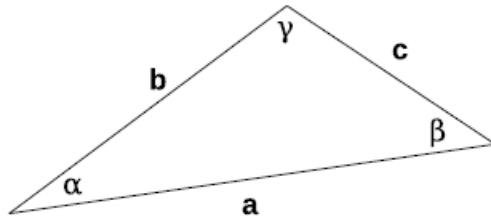


Figure 2.6: Illustration for Example 2.1

This chapter provides a detailed description of a few functions; more functions can be found in the Lazarus help system.

Consider the solution to this example using standard functions.

EXAMPLE 2.1. Find the area S, perimeter P and angles  $\alpha$ ,  $\beta$  and  $\gamma$  of a triangle (Figure 2.6) with sides of length a, b and c.

Before writing code, consider the mathematical formulae to solve this problem. To find the area of the triangle, use Heron's theorem:  $S = \sqrt{((r-a)*(r-b)*(r-c))}$ , where the semi-perimeter  $r = (a+b+c)/2$ . One angle may be found by using the cosine rule:

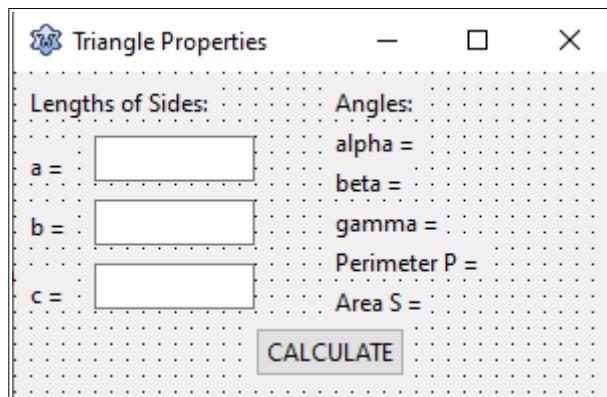


Figure 2.7: Form for Example 2.1

Table 2.11: Captions for controls in Figure 2.7

Control	Caption
Form1	Triangle Properties
Label1	Lengths of sides
Label2	a =
Label3	b =
Label4	c =
Label5	Angles
Label6	alpha =
Label7	beta =
Label8	gamma =
Label9	Perimeter P =
Label10	Area S =
Button1	CALCULATE

$\cos(\alpha) = (b^2 + c^2 - a^2) / (2 * b * c)$ , the second by the sine rule:  $\sin(\beta) = b/a * \sin(\alpha)$  and the third by the formula:  $y = \pi - (\alpha + \beta)$ .

The solution may be divided into the following steps:

- 1) Determination of the values of a, b and c (entering the values of a, b and c into the computer memory).
- 2) Calculate the values of S, P,  $\alpha$ ,  $\beta$  and  $y$  using the formulas discussed above.
- 3) Display the values of S, P,  $\alpha$ ,  $\beta$  and  $y$ .

Chapter 1 describes the steps for developing the interface. Try to develop it yourself. Place ten labels, three edit boxes and a button on the form. Change the captions of the form and labels to the values shown in Table 2.11. Enter a space for

the Text property of edit boxes **Edit1**, **Edit2**, **Edit3**. The form should now look like that shown in Figure 2.7.

So, the form is ready. The Lazarus Source Editor automatically writes the skeleton code for the unit, and names the main sections. Double click on the **Calculate** button to create the procedure TForm1.Button1Click in the **implementation** section:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

with its declaration in the **interface** section. This procedure is empty, of course. It is the programmer's job to add declarations and statements to the skeleton code. Commands added to the procedure, between the words *begin* and *end*, will be executed when the **Calculate** button is clicked. The TForm1.Button1Click procedure will look like this:

```
procedure TForm1.Button1Click(Sender: TObject);
{ Description of variables:
  a, b, c - lengths of sides of a triangle;
  alpha, beta, gamma - angles of the triangle;
  S is the area of the triangle;
  r is the semi-perimeter of the triangle
  All variables are real. }
var
  a, b, c, alpha, beta, gamma, S, r: real;
begin
{ Read Edit1, Edit2 and Edit3 and convert to real,
  using StrToFloat(x). Assign results to a, b and c. }
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  c := StrToFloat(Edit3.Text);
{ Calculate semi-perimeter r
  r := (a + b + c) / 2;
{ Calculate the area S, using the square root function
  sqrt(x). }
  S := sqrt(r*(r - a)*(r - b)*(r - c));
{ Calculate the angle alpha in radians, using acos(x) and
  sqr(x) - the square of x. }
  alpha := arccos((sqr(b)+sqr(c)-sqr(a))/(2*b*c));
{ Calculate beta in radians, using asin(x). }
  beta := arcsin(b/a*sin(alpha));
{ Calculate angle gamma in radians. The mathematical
  constant pi is generated by the function pi,
  which takes no arguments. }
  gamma := pi - (alpha + beta);
{ Convert from radians to degrees. }
```

```
alpha := alpha * 180 / pi;
beta := beta * 180 / pi;
gamma := gamma * 180 / pi;
{ To display the results of the calculations, use the
  string concatenation operator "+" and the
  FloatToStrF(x) function, which converts the real
  variable x to a string and displays it in the required
  format. In this case, the result is displayed
  using three positions, including the decimal point
  and with no place after the decimal point. The
  angles in radians are displayed in the
  form in Label objects. }
Label6.Caption := 'alpha = '+FloatToStrF(alpha, ffFixed,3,0);
Label7.Caption := 'beta = '+FloatToStrF(beta, ffFixed,3,0);
Label8.Caption := 'gamma = '+ FloatToStrF(gamma, ffFixed,3,0);
{ Use the FloatToStrF(x) function to produce formatted output.
  In this case, the entire number will be displayed using
  five places including the decimal point and two places
  after it.
  The values of the area and perimeter will be displayed
  on the form. }
Label9.Caption := 'Perimeter P = '+FloatToStrF(2*r, ffFixed,5,2);
Label10.Caption := 'Area S = '+FloatToStrF(S, ffFixed,5,2);
end;
```

Note that only ten statements were written specifically to solve the problem. The rest of the code in the Source Editor was created automatically. As a result, the code without comments looks like this:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls, Math;
type
  {TForm1}
TForm1 = class (TForm)
  Button1: TButton;
  Edit1: TEdit;
  Edit2: TEdit;
  Edit3: TEdit;
  Label1: TLabel;
  Label10: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Label4: TLabel;
  Label5: TLabel;
  Label6: TLabel;
  Label7: TLabel;
```

```
Label8: TLabel;
Label9: TLabel;
procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
var
  Form1: TForm1;

implementation
{$R *.lfm}
{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c, alpha, beta, gamma, S, r: real;
begin
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  c := StrToFloat(Edit3.Text);
  r := (a + b + c) / 2;
  S := sqrt(r*(r - a)*(r - b)*(r - c));
  alpha := arccos((sqr(b) + sqr(c) - sqr(a))/(2*b*c));
  beta := arcsin(b / a*sin(alpha));
  gamma := pi - (alpha + beta);
  alpha := alpha * 180 / pi;
  beta := beta * 180 / pi;
  gamma := gamma * 180 / pi;
  Label6.Caption := 'alpha = ' + FloatToStrF(alpha, ffFixed,3,0);
  Label7.Caption := 'beta = ' + FloatToStrF(beta, ffFixed,3,0);
  Label8.Caption := 'gamma = ' + FloatToStrF(gamma, ffFixed,3,0);
  Label9.Caption:= 'Perimeter P = '+FloatToStrF(2*r, ffFixed,5,2);
  Label10.Caption := 'Area S = ' + FloatToStrF(S, ffFixed,5,2);
end;
end.
```

Figure 2.8 shows the program displaying the results of a calculation, after clicking the **Calculate** button.

Now let us write a console application to solve the same problem. Start Geany, run the **File | New (with Template)** menu command and select the **program.pas** template. Enter the following program code in the editor window.

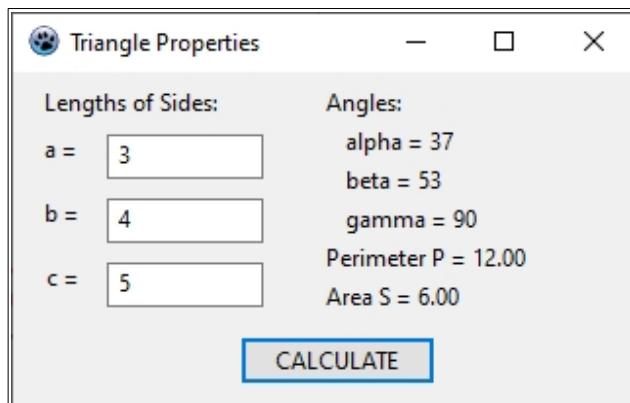


Figure 2.8: The program for Example 2.1

```

program pr3;
{ Include the math unit to access mathematical functions
  (Table 2.6) }
uses
  math;
var
  a, b, c, r, s, alpha, beta, gamma: real;
begin
  writeln('Enter the sides of the triangle: ');
  readln(a, b, c);
  // Calculate the semi-perimeter.
  r := (a + b + c) / 2;
  // Calculate the area, using sqrt(x), the square root function.
  s := sqrt(r * (r - a) * (r - b) * (r - c));
  { Calculate the value of the angle alpha in radians,
    using acos(x) and sqr(x), the square of x. }
  alpha := arccos((sqr(b) + sqr(c) - sqr(a)) / (2 *b *c));
  { Calculate the value of angle beta in radians, using asin(x). }
  beta := arcsin(b / a * sin(alpha));
  { Calculate the value of angle gamma in radians.
    The mathematical constant pi is generated
    by the function pi, which takes no arguments. }
  gamma := pi - (alpha + beta);
  // Convert from radians to degrees.
  alpha := alpha * 180 / pi;
  beta := beta * 180 / pi;
  gamma := gamma * 180 / pi;
  // Display the area and sides of the triangle.
  writeln('Area of triangle = ', S:6:2);
  writeln('alpha = ', alpha:6:2, ' beta = ', beta:6:2, ' gamma = ',
gamma:6:2);
end.

```

To compile the program in Geany, execute the **Build | Compile** menu command or **F8**. Execute the **Build | Execute** menu command or **F5** to run it. Figure 2.9 shows

the program's output.

```
Enter the sides of the triangle
3 4 5
Area of triangle = 6.00
alpha = 36.87 beta = 53.13 gamma = 90.00
```

## 2.7 Exercises

Develop a console application and a Lazarus graphical (GUI) program for each problem.

- 1) Find the hypotenuse and the angles of a right triangle if the lengths of its two legs are given.
- 2) Find the area of a right-angled triangle if the hypotenuse and the included angle  $\alpha$  are given.
- 3) Calculate the area  $S$  and perimeter  $P$  of a square if the length  $d$  of its diagonal is given.
- 4) Calculate the area  $S$  of a rectangle if its diagonal  $d$  and the angle  $\alpha$  between the diagonal and the longer side are given.
- 5) Find the angles  $\alpha$ ,  $\beta$  and  $\gamma$  of a triangle if the lengths  $a$ ,  $b$ ,  $c$  of its sides are given.
- 6) A solid body is a parallelepiped (each face is a parallelogram) of height  $h$ . The rectangle formed by its base has diagonals of length  $d$ , which intersect at an angle  $\beta$ . Find the volume  $V$  of the body and its surface area  $S$ .
- 7) The areas  $S$  and length  $a$  of a leg of a right-angled triangle are given. Find the length  $c$  of its hypotenuse, length  $b$  of its other leg and angles  $\alpha$  and  $\beta$ .
- 8) The area  $S$  of a square is given. Calculate the length  $a$  of its side, the length  $d$  of its diagonal and the area  $S_1$  of a circle circumscribed around the square.
- 9) The base of an isosceles triangle is  $c$  and the angle at its base is  $\alpha$ . Find its area  $S$  and the length  $a$  of its side.
- 10) Find the area and perimeter of rectangle ABCD with vertices  $A(x_1,y_1)$ ,  $B(x_2,y_2)$  and  $C(x_3,y_3)$ .
- 11) Calculate the area and perimeter of a right-angled triangle if the lengths of its two legs are given.
- 12) Find the perimeter of a right-angled triangle if the leg  $c$  and its opposite angle  $\alpha$  are given.

- 13) Calculate the area  $S$  of a circle inscribed in a square with diagonal  $d$ .
- 14) Calculate the area  $S$  of a rectangle with diagonals  $d$  and angle  $\alpha$  between them.
- 15) Calculate the perimeter of a right-angled triangle if its leg  $b$  and area  $S$  are known.
- 16) Find the area and perimeter of a square ABCD with vertices A( $x_1, y_1$ ) and C( $x_2, y_2$ ).
- 17) Find the hypotenuse  $c$ , second leg  $a$ , and the angles  $\alpha$  and  $\beta$  of a right-angled triangle with leg  $b$  and area  $S$ .
- 18) Find the area of an equilateral triangle if perimeter  $P$  is given.
- 19) Calculate the side  $a$ , diagonal  $d$  and area  $S$  of a square if perimeter  $P$  is given.
- 20) Find the area  $S$  and perimeter  $P$  of an isosceles triangle if base  $c$  and the height  $h$  are given.
- 21) Find the area and perimeter of triangle ABC with vertices A( $x_1, y_1$ ), B( $x_2, y_2$ ) and C( $x_3, y_3$ ).
- 22) A metal ingot is cylindrical with surface area  $S$ , height  $h$  and density  $\alpha$ . Calculate the mass  $m$  of the ingot.
- 23) The first term and the difference between terms of an arithmetic progression are given. Calculate the sum of the first  $n$  terms and the value of the  $n$ th term.
- 24) The first term and the denominator of a geometric progression are given. Calculate the sum of the first  $n$  terms and the value of the  $n$ th term.
- 25) A body falls from a height  $h$ . What is its velocity when it reaches the ground, and when does this occur?

---

Endnotes:

- 1 For details of procedures and functions, see Chapter 4.
- 2 Operations on numbers in exponential form are called floating point arithmetic, since the position of the decimal point changes depending on the order of the number.
- 3 A number in its common form is called a fixed-point number.
- 4 See Chapter 5 for more on arrays.
- 5 Conjunction.
- 6 Disjunction.
- 7 Bitwise inversion is the replacement of 1 with 0, and 0 with 1, in the binary form of a number.

- 8 These functions will work only if the Math unit is included, by placing the word “**Math**” after the word “**uses**” in the main program.
- 9 The formula is derived as follows: Take the logarithm of  $x^n$ , which gives  $n\ln(x)$ , then take the exponent of the latter.
- 10 Dynamic memory is an area of memory that is allocated during compilation for storing temporary data.

This page deliberately left blank.

## Chapter 3. Flow Control

This chapter describes a method for developing algorithms using flowcharts. It also describes the basic flow control statements, including: the conditional **if** statement, the **case** selection statement, and the **while..do**, **repeat..until** and **for..do** loop statements. Many examples of developing programs of varying complexity are provided.

### 3.1 Principal Algorithm Constructs

As a rule, the development of a program is preceded by the development of an algorithm.<sup>1</sup> An algorithm is a clear description of the series of operations necessary for obtaining the required result from a corresponding set of input data. A flowchart is one of several ways to represent an algorithm. When developing a flowchart, the steps for solving the problem are represented using geometric shapes. These shapes or symbols and are usually accompanied by text. The step execution order is shown by arrows connecting these shapes. Typical steps for solving problems are represented by the following geometric shapes:

- *Terminator symbol* (Figure 3.1). The text inside the shape: Start or End;



Figure 3.1: Terminator symbol

- *Input/Output symbol* (Figure 3.2). The text inside the symbol: Input (Output or Print) and a list of input (output) variables;



Figure 3.2: Data symbol

- *Process symbol* (Figure 3.3). Symbol contents include actions, computational operations or groups of operations;

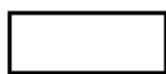


Figure 3.3: Process symbol

- *Decision symbol* (Figure 3.4). A logical condition is placed in the symbol. Evaluate the condition to determine which of the possible paths (branches) the computational process takes.

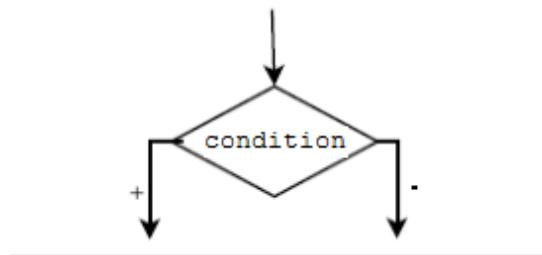


Figure 3.4: Conditional block

The partial flowcharts below illustrate the three main constructs for developing algorithms: sequential execution, branching and looping.

*Sequential execution* is a construct in which actions in two or more flowchart steps are processed consecutively (Figure 3.5).

*Branching* is the execution of one statement or another, depending on the result of processing a condition statement (Figure 3.6).

*Looping* is the repeated execution of a group of statements (Figure 3.7).

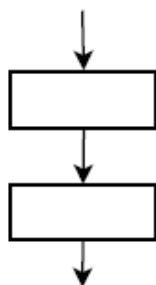


Figure 3.5: Sequential process

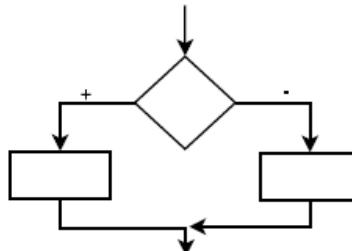


Figure 3.6: Branching

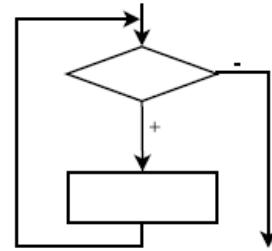


Figure 3.7: Looping

It is easy to see that each of the basic algorithm constructs has one entrance and one exit. This allows the nesting of constructs inside each other in any combination, and the development of algorithms for solving problems of any complexity.

The Free Pascal statements for the implementation of these basic algorithm constructs are discussed below.

## 3.2 The Assignment Statement

The Free Pascal *assignment statement* consists of two characters: a colon and an equals sign. The characters := are always written together. Spaces are allowed before the colon and after the equals sign. In general, the form of the assignment statement is:

variable\_name := value;

where "value" is an expression, variable, constant, or function. This statement executes as follows: The expression on the right side of the operator is evaluated first, after which its result is written into the memory area (variable) whose name is on the left side of the operator. For example, writing a := b means that variable "a" is assigned the value of expression "b".

Variable "a" and expression "b" must be the same type or compatible for assignment. That is, the type to which "b" belongs must fit within the type of variable "a".

An assignment statement, like any other statement in Free Pascal, must be separated from other statements by a semicolon.

## 3.3 The Compound Statement

A *compound statement* is a group of statements separated from each other by semicolons, starting with the reserved word **begin** and ending with the reserved word **end**:

```
begin
  statement_1;
...
  statement_n
end;
```

The compiler treats a compound statement as a single statement.

## 3.4 Selection Structures

*Branching*, one of the main algorithm constructs, is implemented in Free Pascal using two selection structures: **if** and **case**. They are both discussed below.

### 3.4.1 The If..Then..Else Selection Structure

When solving most problems, the order of calculations depends on certain conditions, such as the value of input data or intermediate results from the

previous program steps. To control the order of calculations using Boolean (logical) expressions, Free Pascal provides the **if..then..else** selection structure, with the general form:

```
if condition then statement_1 else statement_2;
```

where **if..then..else** are reserved words, *condition* is a Boolean expression<sup>2</sup>, and *statement\_1* and *statement\_2* are Free Pascal statements.

The operation of the selection structure proceeds as follows. First, the Boolean expression in *condition* is evaluated. If this returns **true** then *statement\_1* is executed. Otherwise, if the Boolean expression returns **false**, then *statement\_1* is ignored and control is transferred to *statement\_2*.

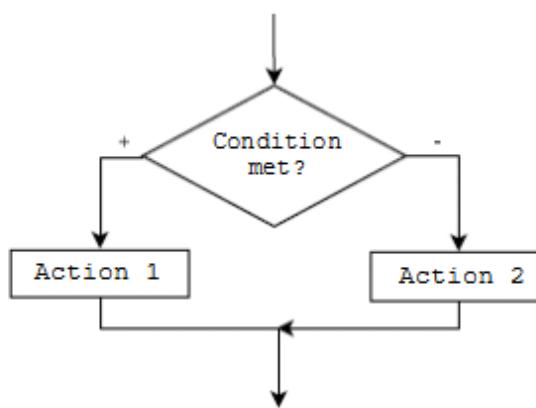


Figure 3.8: If..Then..Else selection structure

Figure 3.8 shows the algorithm implemented in the **if..then..else** selection structure.

For example, the following code compares the values of variables x and y:

```
write('x ='); readln(x);
write('y ='); readln(y);
if x = y then
  writeln('The value of x equals the value of y')
else
  writeln('The value of x is not equal to value of y');
```

If several statements must be processed after evaluating the condition, then a compound statement should be used:

```
if condition then
  begin
    statement_1;
    statement_2;
```

```
...
  statement_n;
end
else
begin
  statement_1;
  statement_2;
...
  statement_n;
end;
```

The alternative path **else** in the selection structure can be omitted if not needed:

**if** *condition* **then** *statement*;

or

```
if condition then
begin
  statement_1;
  statement_2;
...
  statement_n;
end;
```

In this “truncated” form, the selection structure works as follows: the statement (or compound statement) is either executed or bypassed, depending on the value of the Boolean expression in *condition*. Figure 3.9 shows the algorithm for this process.

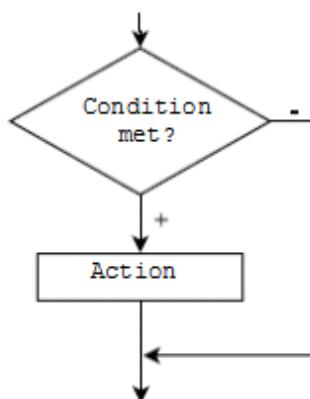


Figure 3.9: if selection structure without else

An example of this selection structure without an alternative **else** branch is given below:

```

write('x = '); readln(x);
write('y = '); readln(y);
c := 0;
{The value of the variable c changes
only if x is not equal to y. }
if (x <> y) then c := x + y;
{ Display the value of variable c on screen.
This statement will always be executed. }
writeln('The value of the variable c = ', c);

```

Selection structures may be nested within each other. When considering selection structures, the following rule applies: an **else** branch belongs to the closest **if** statement. For example, in the code:

```

if condition_1 then
  if condition_2 then
    statement_A
  else statement_B;

```

statement\_B refers to condition\_2, and in the construct

```

if condition_1 then
  begin
    if condition_2 then
      statement_A;
    end
  else statement_B;

```

it belongs to the **if** statement with condition\_1.

To compare variables in Boolean expressions, use the *relational operators* `:=`, `<>`, `<`, `>`, `<=`, `>=`. Complex Boolean expressions can be constructed using Boolean operators **and**, **or** and **not**. In Free Pascal, *relational operators have a lower priority than Boolean operators*, therefore the constituent parts of a complex Boolean expression should be enclosed in parentheses.

To check if the variable x belongs to the range [a..b], for example, the Boolean expression should look like:

**if** (x  $\geq$  a) **and** (x  $\leq$  b) **then...**

The code:

**if** x  $\geq$  a **and** x  $\leq$  b **then...**

is incorrect, since the expression will be evaluated as:

x  $\geq$  (a **and** x)  $\leq$  b.

Several examples<sup>3</sup> will show the use of the **if** statement.

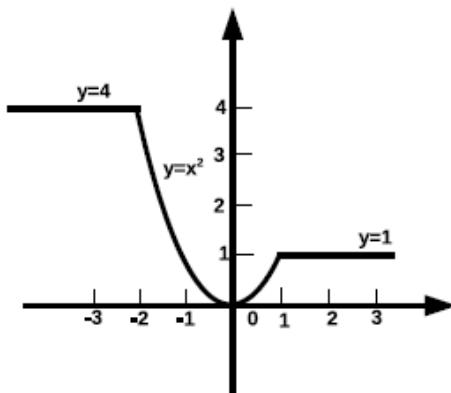


Figure 3.10: Graph for Example 3.1

EXAMPLE 3.1. Calculate  $y = f(x)$ , where  $x$  is a real number, for the function of the graph shown in Figure 3.10.

Analytically, the function in Figure 3.10, can be written thus:

$$y(x) = \begin{cases} 4, & x \leq -2, \\ x^2, & -2 < x < 1, \\ 1, & x \geq 1. \end{cases}$$

Let us compose a text-based algorithm for solving this problem:

- 1) Begin algorithm.
- 2) Input  $x$  (function argument).
- 3) If  $x \leq -2$ , go to Step 4. Otherwise, go to Step 5.
- 4) Calculate value of the function:  $y = 4$ . Go to Step 8.
- 5) If  $x \geq 1$ , go to Step 6. Otherwise go to Step 7.
- 6) Calculate value of the function:  $y = 1$ . Go to Step 8.
- 7) Calculation of the value of the function:  $y = x^2$ .
- 8) Display values of  $x$  and  $y$ .
- 9) End algorithm.

The flowchart for the algorithm described above is shown in Figure 3.11. As you can see, the flowchart is clearer and easier to understand than the text-based version.

From this point, flowcharts will be used almost always to describe the algorithms.

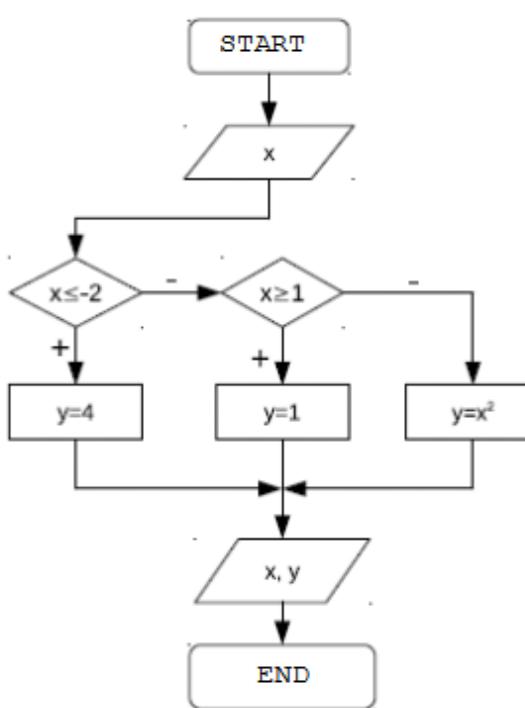


Figure 3.11: Algorithm for solving Example 3.1

The Free Pascal code should look like:

```

var
  x, y: real;
begin
  write('x = ');
  readln(x);
  if x <= -2 then
    y := 4
  else if x >= 1 then
    y := 1
  else
    y := sqr(x);
  writeln('x = ', x:5:2, '      y = ', y:5:2);
end.
  
```

Enter and run this program.

EXAMPLE 3.2. Determine if the point with coordinates  $(x; y)$ , where  $x$  and  $y$  are real numbers, lies on the hatched part of the plane shown in Figure 3.12.

As shown in Figure 3.12, the region is bounded by the lines  $x = -1$ ,  $x = 3$ ,  $y = -2$  and  $y = 4$ . Hence, the point with coordinates  $(x; y)$  will lie within this area if the following conditions are satisfied:  $x \geq -1$ ,  $x \leq 3$ ,  $y \geq -2$ , and  $y \leq 4$ . Otherwise, the point does not lie within the area.

The algorithm for solving this example is shown in Figure 3.13.

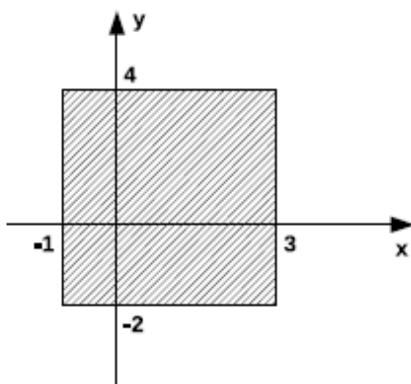


Figure 3.12: Diagram for Example 3.2

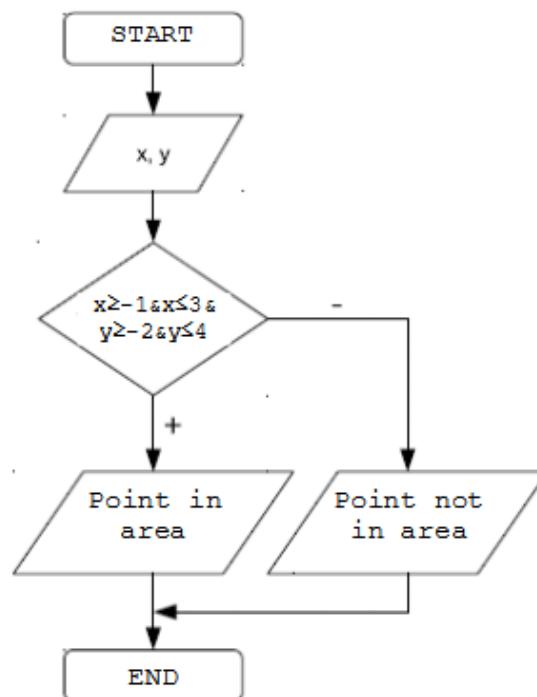


Figure 3.13: Algorithm for Example 3.2.

The code for Example 3.2 is given below:

```

var x, y: real;
begin
  write('x = '); readln(x);
  write('y = '); readln(y);
  if (x >= -1) and (x <= 3) and (y >= -2) and (y <= 4) then
    writeln('The point lies within the area.')
  else
    writeln('The point does not lie within the area.');
end.
  
```

**EXAMPLE 3.3.** Write a program to find the roots of the quadratic equation  $ax^2 + bx + c = 0$ , where the coefficients  $a$ ,  $b$  and  $c$  are real numbers.

The program will display the roots of the quadratic equation (real numbers  $x_1$  and  $x_2$ ) or a message stating that there are no roots.

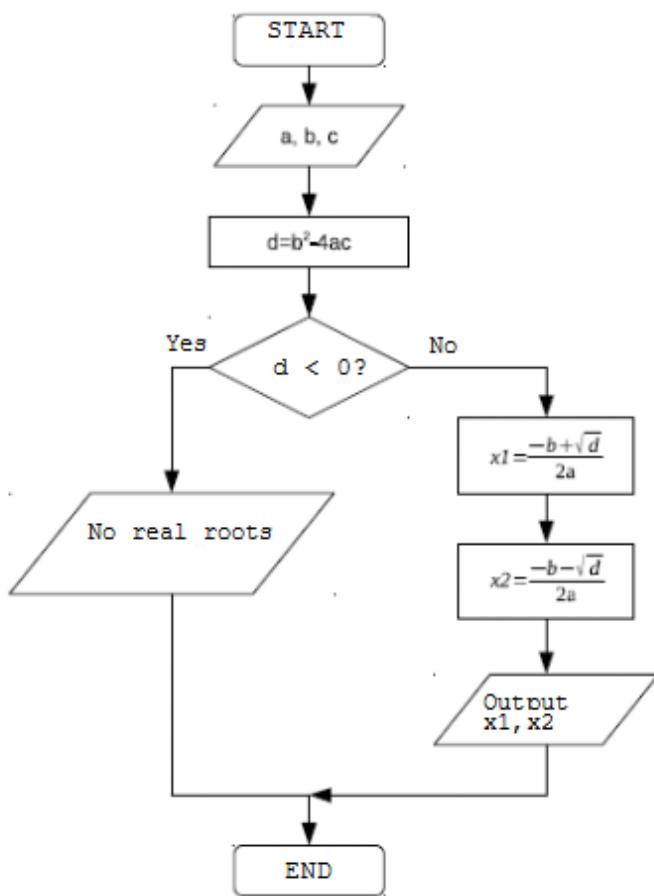


Figure 3.14: Algorithm for solving a quadratic equation

An auxiliary variable, a real variable  $d$ , will store the discriminant of the quadratic equation.

Let us compose a text-based algorithm for solving this problem.

- 1) Begin algorithm.
- 2) Enter  $a$ ,  $b$  and  $c$ .
- 3) Calculate the discriminant  $d$  using the equation  $d = b^2 - 4ac$ .

- 4) If  $d < 0$ , go to Step 5. Otherwise, go to Step 6.
- 5) Display message "There are no real roots" and go to Step 8.
- 6) Calculation roots  $x_1 = (-b + \sqrt{d})/(2a)$  and  $x_2 = (-b - \sqrt{d})/(2a)$ .
- 7) Display  $x_1$  and  $x_2$  on screen.
- 8) End algorithm.

The flowchart for this algorithm is shown in Figure 3.14. The code to implement this solution is given below:

```

var
  a, b, c, d, x1, x2: real; { Declare variables. }
begin
  writeln('Enter the coefficients a, b and c: ');
  readln(a, b, c);
  d := b*b-4*a*c;           { Calculate the discriminant. }
  if d < 0 then           { If the discriminant is negative, }
    writeln('No real roots')
  else
    begin
      { else calculate the roots x1, x2 }
      x1 := (-b+sqrt(d))/2/a;
      x2 := (-b-sqrt(d))/(2*a);
      {and display them on screen.}
      writeln('X1 = ', x1:6:3, '     X2 = ', x2:6:3)
    end
  end.

```

**EXAMPLE 3.4.** Write a program to find the real and complex roots of the quadratic equation  $ax^2 + bx + c = 0$ , if the coefficients  $a$ ,  $b$  and  $c$  are real numbers.

The program will display real numbers  $x_1$  and  $x_2$  if real roots exist or complex numbers  $x_1$  and  $x_2$  if no real roots exist.

An auxiliary variable, a real variable  $d$ , will be required to store the discriminant of the quadratic equation.

The following are possible steps in the solution:

- 1) Enter the coefficients  $a$ ,  $b$  and  $c$ .
- 2) Calculate discriminant  $d$  using the equation  $d = b^2 - 4ac$ .
- 3) Check the sign of the discriminant. If  $d \geq 0$ , compute the real roots using the following equations:

$$x_1 = (-b + \sqrt{d})/(2a) \text{ and } x_2 = (-b - \sqrt{d})/(2a)$$

and display them on the screen. If the discriminant is negative, display a message stating that there are no real roots, and compute the complex roots<sup>4</sup> using the following equations:

$$-b/(2 \cdot a) + i \cdot \sqrt{|d|}/(2 \cdot a) \text{ and } -b/(2 \cdot a) - i \cdot \sqrt{|d|}/(2 \cdot a).$$

Both complex roots have the same real parts, and the imaginary parts differ in sign. Therefore, it is possible to store the real part of the number  $-b/(2 \cdot a)$  in the variable  $x_1$ , the modulus of the imaginary part  $\sqrt{|d|}/(2 \cdot a)$  in the variable  $x_2$ , and output  $x_1 + i \cdot x_2$  and  $x_1 - i \cdot x_2$  as the roots.

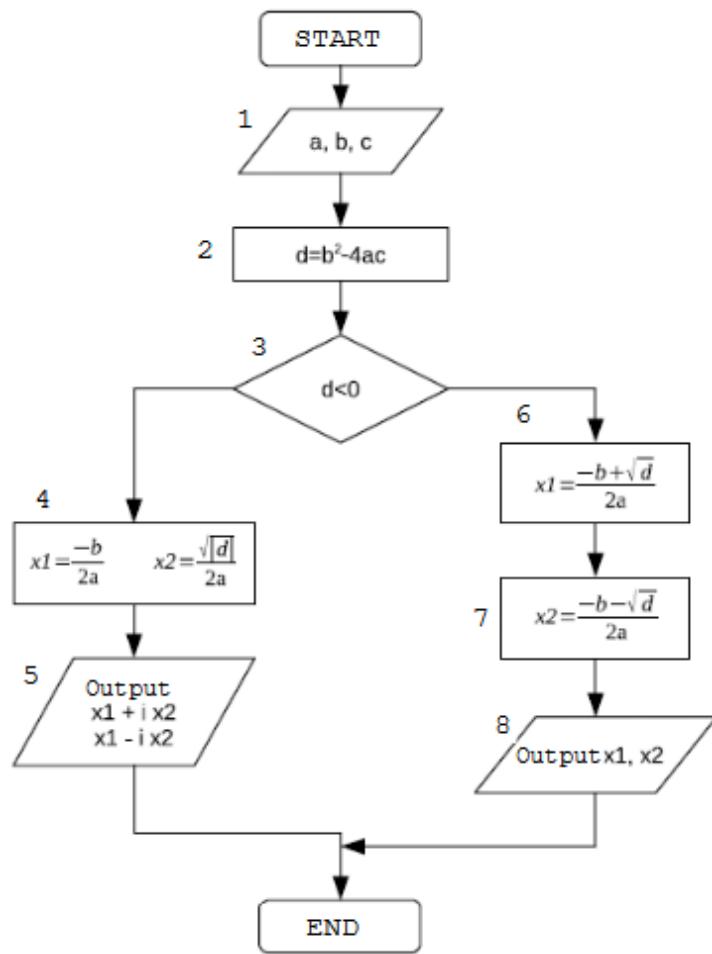


Figure 3.15: Algorithm for Example 3.4

The solution flowchart is shown in Figure 3.15. Block 1 inputs the coefficients of the quadratic equation. Block 2 calculates the discriminant and Block 3 checks its sign. If the discriminant is negative, then the roots are complex and are calculated in Block 4. The real part of the root is saved in the variable  $x_1$  and the imaginary

modulus in the variable x2. Block 5 displays the first root  $x_1 + ix_2$  and the second root  $x_1 - ix_2$ . If the discriminant is positive, then the real roots of the equation (Blocks 6 & 7) and displayed (Block 8).

The code for solving the Problem is shown below:

```

var a, b, c, d, x1,x2: real;
begin
  writeln('Enter the coefficients a, b and c: ');
  readln(a, b, c);
  d := b*b-4*a*c;
  if d < 0 then
    begin
      { If the discriminant is negative, display message "There
        are no real roots" and calculate the complex roots. }
      writeln('No real roots');
      x1 := -b/ (2*a);           {real part of complex roots}
      x2 := sqrt(abs(d))/(2*a);   {modulus of imaginary part}
      writeln('Complex roots of the equation: ',
        a:1:2, 'x^2 + ', b:1:2, 'x + ', c:1:2, '= 0');
      { Output complex roots as  $x_1 \pm ix_2$  }
      writeln(x1:1:2, '+i*(', x2:1:2, ')');
      writeln(x1:1:2, '-i*(', x2:1:2, ')');
    end
  else
    begin
      x1 := (-b + sqrt(d))/(2*a);  { or compute real roots x1,x2 }
      x2 := (-b - sqrt(d))/(2*a);
      { and display them on screen. }
      writeln('Real roots of the equation ',
        a:1:2, 'x^2 + ', b:1:2, 'x + ', c:1:2, '= 0');
      writeln('X1 = ', x1:1:2, 'X2 = ', x2:1:2);
    end
end.

```

EXAMPLE 3.5. Write a program to solve the cubic equation

$$ax^3 + bx^2 + cx + d = 0.$$

The general form of a cubic equation is:

$$ax^3 + bx^2 + cx + d = 0 \quad (3.1)$$

After dividing by a, Equation (3.1) takes the canonical form:

$$x^3 + rx^2 + sx + t = 0 \quad (3.2)$$

where  $r = b/a$ ,  $s = c/a$ ,  $t = d/a$ . In Equation (3.2) we can substitute  $x = y - r/3$  to obtain

a depressed cubic equation:

$$y^3 + py + q = 0, \quad (3.3)$$

where

$$p = (3s - r^2)/3, \quad q = 2r^3/27 - rs/3 + t.$$

The number of real roots in the depressed cubic equation (3.3) depends on the discriminant  $D = (p/3)^3 + (q/2)^2$  of the depressed cubic equation (Table 3.1).

*Table 3.1: Number of roots in a cubic equation*

Discriminant	Real roots	Complex roots
$D \geq 0$	1	2
$D < 0$	3	-

The roots of the depressed cubic equation can be calculated using Cardano's method:

$$\left. \begin{aligned} y_1 &= u + v \\ y_2 &= (-u + v)/2 + (u - v)/2 \cdot i \cdot \sqrt{3} \\ y_3 &= (-u + v)/2 - (u - v)/2 \cdot i \cdot \sqrt{3} \end{aligned} \right\} \quad (3.4)$$

where

$$u = (-q/2 + \sqrt{D})^{1/3}, \quad v = (-q/2 - \sqrt{D})^{1/3}.$$

With a negative discriminant, Equation (3.1) will have three real roots, which could be calculated using auxiliary complex values. To avoid this, use the formulas:

$$\left. \begin{aligned} y_1 &= 2 \cdot (\rho)^{1/3} \cos(\varphi/3), \\ y_2 &= 2 \cdot (\rho)^{1/3} \cos(\varphi/3 + 2\pi/3), \\ y_3 &= 2 \cdot (\rho)^{1/3} \cos(\varphi/3 + 4\pi/3), \end{aligned} \right\} \quad (3.5)$$

where

$$\rho = \sqrt{(-p^3/27)}, \quad \cos(\varphi) = -q/(2\rho).$$

If the depressed cubic equation (3.3) has a positive discriminant then the roots will be calculated using equations (3.4). Equations (3.5) will be used to calculate the roots if the discriminant is negative.

After calculating the roots of the depressed cubic equation (3.3) by using equations (3.4) or (3.5), use the equation

$$x_k = y_k - r/3, \quad k = 1, 2, 3 \dots$$

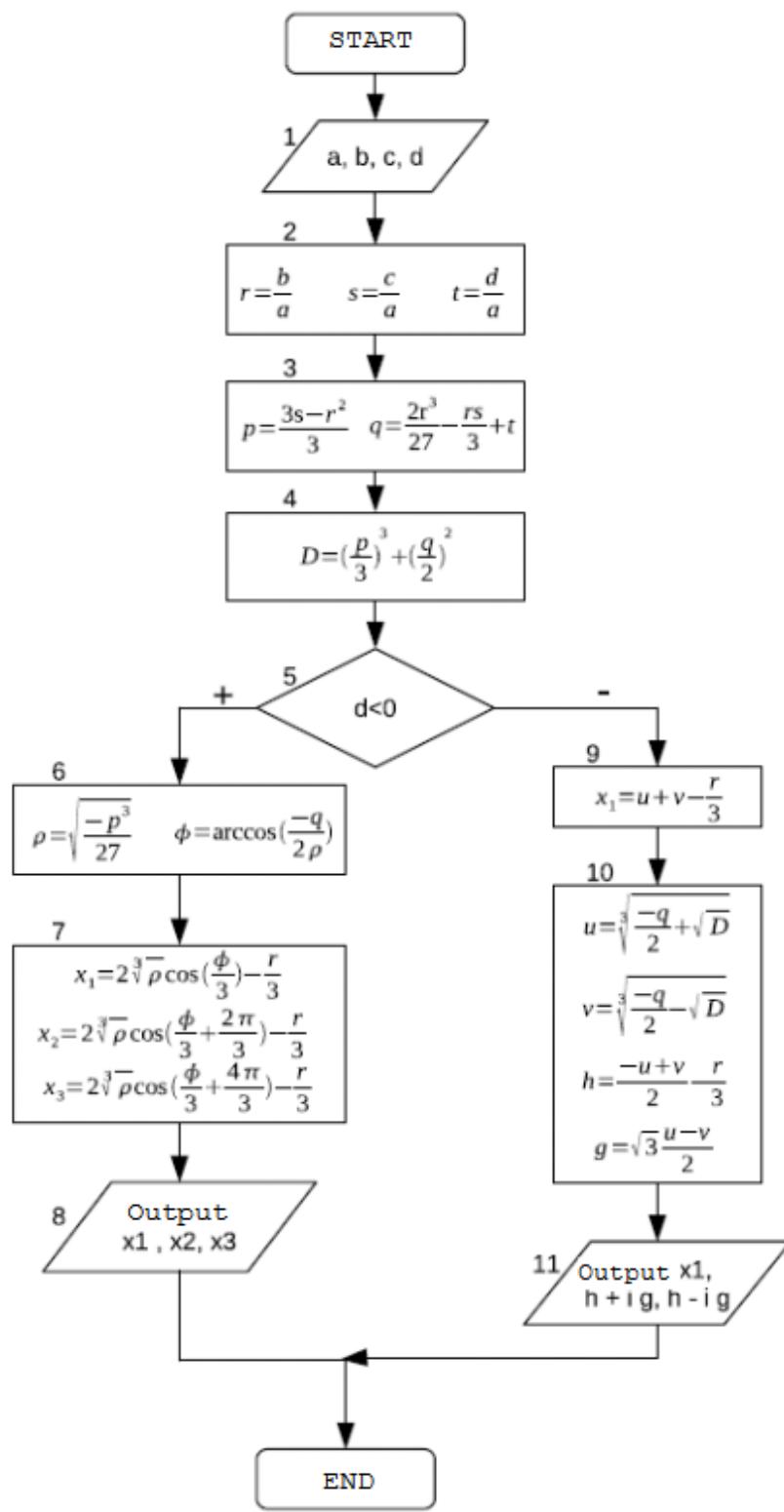


Figure 3.16: Algorithm for solving a cubic equation

to obtain the roots of the original cubic equation (see Equation 3.1).

The flowchart for solving a cubic equation is shown in Figure 3.16 above.

Flowchart description. Block 1 reads in the coefficients of the cubic equation. Blocks 2 and 3 calculate the coefficients of the canonical and depressed cubic equations. Block 4 calculates the discriminant. Block 5 checks the sign of the discriminant. If it is negative, then the roots are calculated using Equations (3.5) (Blocks 6 & 7). If it is positive, then the roots are calculated using Equations (3.4) (Block 9 & 10). Blocks 8 and 11 display results on the screen.

The commented code is given below<sup>5</sup>.

```

var
  a, b,c,d,r,s,t,p,q,ro,fi,x1,x2,x3,u,v,h,g: real;
begin
  // Enter the coefficients of the cubic equation.
  write('a = '); readln(a);
  write('b = '); readln(b);
  write('c = '); readln(c);
  write('d = '); readln(d);
  // Calculate coefficients of the canonical equation, using 3.2.
  r := b/a; s := c/a; t := d/a;
  // Calculate coefficients of the depressed cubic equation, by 3.3.
  p := (3*s-r*r)/3;
  q := 2*r*r*r/27 - r*s/3 + t;
  // Calculate the discriminant.
  D := (p/3)*sqr(p/3) + sqr(q/2);
  { Check the sign of the discriminant.
    The "then" branch implements Equations (3.5).
    The "else" branch implements Equations (3.4). }
  if d < 0 then
  begin
    ro := sqrt(-p*p*p/27);
    { The next two statements calculate the angle fi.
      First, calculate the cosine of the angle,
      Then calculate its inverse cos using arctan. }
    fi := -q/(2*ro);
    fi := pi/2-arctan(fi/sqrt(1-fi*fi));
    // Calculate the real roots of the equation x1, x2 and x3.
    x1 := 2*exp(1/3*ln(ro))*cos(fi/3)-r/3;
    x2 := 2*exp(1/3*ln(ro))*cos(fi/3+2*pi/3)-r/3;
    x3 := 2*exp(1/3*ln(ro))*cos(fi/3+4*pi/3)-r/3;
    writeln('x1 = ', x1:1:3, 'x2 = ', x2:1:3, 'x3 = ', x3:1:3);
  end
  else
  begin
    // Compute u and v and check the sign of radical expression.
    if -q/2+sqrt(d) > 0 then

```

```

u := exp(1/3*ln(-q/2 + sqrt(d)))
else
  if -q/2 + sqrt(d) < 0 then
    u := -exp(1/3*ln(abs(-q/2 + sqrt(d))))
  else
    u := 0;
    if -q/2 - sqrt(d) > 0 then
      v := exp(1/3*ln(-q/2 - sqrt(d)))
    else
      if -q/2 - sqrt(d) < 0 then
        v := -exp(1/3*ln(abs(-q/2 - sqrt(d))))
      else
        v := 0;
    // Calculate the real root of the cubic equation.
    X1 := u + v - r/3;
    // Calculate real and imaginary parts of complex roots.
    h := -(u + v)/2 - r/3;
    g := (u - v)/2*sqrt(3);
    writeln('x1 = ',x1:1:3, ' x2 = ', h:1:3, '+ i*', g:1:3,' x3 = ',
h:1:3, '-i*', g:1:3);
  end
end.

```

EXAMPLE 3.6. The coefficients of the bi-quadratic equation

$ax^4 + bx^2 + c = 0$  are a, b and c. Find all its real roots.

Input data: a, b, c. Output data: x1, x2, x3, x4.

To solve the bi-quadratic equation, substitute  $y = x^2$  to reduce it to the quadratic equation  $ay^2 + by + c = 0$ .

The algorithm for solving this problem is shown in Figure 3.17, and is described below:

- 1) Enter  $a$ ,  $b$  and  $c$  – the coefficients of bi-quadratic equation (Block 1).
- 2) Calculate the discriminant  $d$  (Block 2).
- 3) If  $d < 0$  (Block 3), display a message that there are no roots (Block 4). Otherwise, find roots  $y_1$  and  $y_2$  of the corresponding quadratic equation (Block 5).
- 4) If  $y_1 < 0$  and  $y_2 < 0$  (Block 6), display a message that there are no roots (Block 7).
- 5) If  $y_1 \geq 0$  and  $y_2 < 0$  (Block 8), then calculate the four roots by the equations  $\pm\sqrt{y_1}$ ,  $\pm\sqrt{y_2}$  (Block 9) and display the values of the roots (Block 10).

- 6) If conditions 4) and 5) are not met, then check the sign of  $y_1$ . If  $y_1 \geq 0$  (Block 11), then calculate two roots using the equation  $\pm\sqrt{y_1}$  (Block 12). Otherwise (if  $y_2 \geq 0$ ), calculate two roots using the equation  $\pm\sqrt{y_2}$  (Block 13).
- 7) Display of the calculated values of the roots (Block 14).

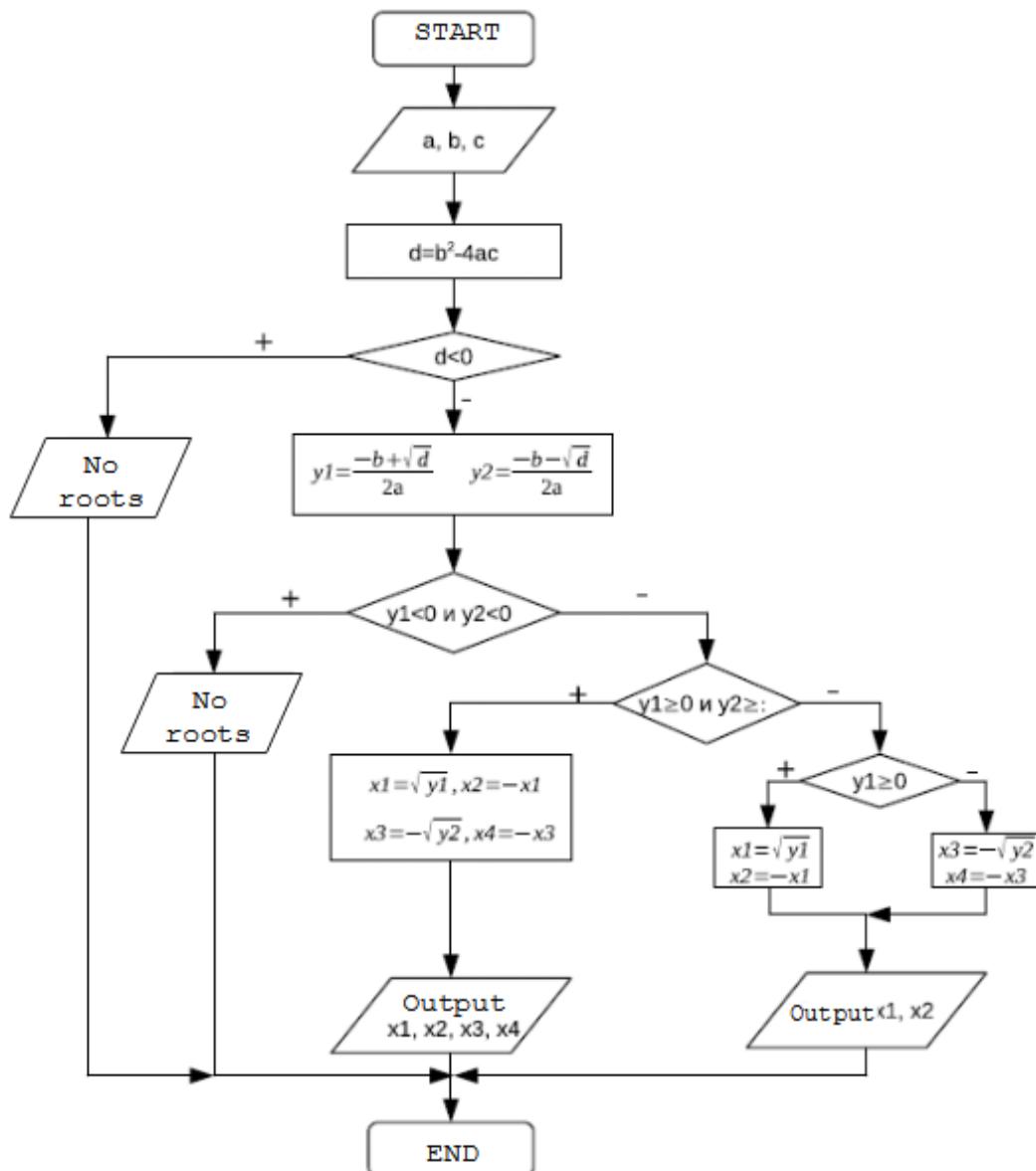


Figure 3.17: Algorithm for solving bi-quadratic equation

The Free Pascal code with comments is shown below:

{ Declaration of variables:

```
a, b, c - coefficients of the bi-quadratic equation,
d - discriminant,
x1, x2, x3, x4 - roots of the bi-quadratic equation,
y1, y2 are the roots of the quadratic equation ay^2+by+c=0. }

var
  a, b, c,d,x1,x2,x3,x4,y1,y2: real;
begin
  // Enter the coefficients of the equation.
  writeln('Enter the coefficients of the biquadratic equation');
  readln(a, b, c);
  D := b*b-4*a*c;                      { Calculate the discriminant. }
  if d < 0 then                         { If it is negative, }
    writeln ('No roots')                 { Display message 'No roots' }
  else                                     { If discriminant > = 0, }
begin
  // Calculate the roots of a quadratic equation.
  Y1 := (-b + sqrt(d))/(2*a);           { display message 'No roots'.}
  y2 := (-b - sqrt(d))/(2*a);
  // If both roots of a quadratic equation < 0,
  if (y1 < 0) and (y2 < 0) then
    writeln('No roots')                 { display message 'No roots'.}
    // If both roots of a quadratic equation > = 0,
  else if (y1 >= 0) and (y2 >= 0) then
begin
  // Calculate the four roots of the biquadratic equation.
  X1 := sqrt(y1);
  x2 := -x1;
  x3 := sqrt(y2);
  x4 := -x3;
  // Display the roots of the biquadratic equation on screen.
  writeln('X1 = ', x1:6:3, 'X2 = ', x2:6:3);
  writeln('X3 = ', x3:6:3, 'X4 = ', x4:6:3);
end
{ If both conditions below are not met:
  1. y1 < 0 AND y2 < 0,
  2. y1 >= 0 AND y2 >= 0,
  then check the condition y1 >= 0. }
else if (y1 >= 0) then
  //If it's true
begin
  x1 := sqrt(y1);
  x2 := -x1;
  writeln('X1 = ', x1:6:3, ' X2 = ', x2:6:3);
end
else
  // If the condition y1 >= 0 is false, then
begin
  x1 := sqrt(y2);
  x2 := -x1;
  writeln('X1 = ', x1:6:3, ' X2 ', x2:6:3);
end
```

```
    end  
end.
```

### 3.4.2 The Case Selection Structure

The **case** selection structure is used when a specific statement must be executed, depending on the value of a selector variable.

```
case selector of  
    value_1: statement_1;  
    value_2: statement_2;  
...  
    value_n: statement_n  
else  
    alternative_statement  
end;
```

The selector is a string or enumeration variable (including **integer**, **char** and **Boolean**), value\_1, value\_2,..., value\_n are specific values the selector could take, or an expression that determines which statement will be executed. The values in each selection (value-n) must be unique and can only appear in one selection. Overlapping values in selections will cause an error.

The case statement works as follows: The selector is evaluated and if its resulting value is in value\_n then statement\_n is executed. That is, if the selector is in value\_1, then statement\_1 is executed. If the selector value is in value\_2, then statement\_2 is executed, etc. If the selector value is not in any of value\_n, then the statement after keyword **else** is executed.

If the **else** branch is absent, then the statement would look like:

```
case selector of  
    value_1: statement_1;  
    value_2: statement_2;  
  
    value_N: statement_N;  
end;
```

In addition, a compound statement is allowed in a **case** statement. For example:

```
case selector of  
    value_1: begin statement_A; statement_B; end;  
    value_2: begin statement_C; statement_D; statement_E; end;  
...  
    value_n: statement_n;  
end;
```

The use of the multi-alternative selection structure is considered in the following

problems.

EXAMPLE 3.7. Print the name of the day of the week corresponding to the day of the month D, assuming that the first day of the week is Monday.

To solve the problem, use the **mod** operator, which calculates the remainder of the division of two numbers, and the condition that the first day of the week is Monday. If the remainder of the division (denote it by R) of D by seven is one, then this is Monday, two is Tuesday, three is Wednesday, etc. Thus, seven statements will be required, as shown in Figure 3.18.

The solution becomes simpler if the **case** statement is used:

```
var
  d: byte;
begin
  write('Enter number D = ');
  readln(D);
  case D mod 7 of
    { Calculate the remainder of D divided by 7,
      depending on the value read in, and
      print the name of the day of the week. }
    1: writeln('Monday');
    2: writeln('Tuesday');
    3: writeln('Wednesday');
    4: writeln('Thursday');
    5: writeln('Friday');
    6: writeln('Saturday');
    0: writeln('Sunday');
  end;
end.
```

There is no **else** branch in the proposed **case** statement. This is explained by the fact that the expression (D mod 7) can take only one of the following values: 1, 2, 3, 4, 5, 6, or 0.

EXAMPLE 3.8. Print the name of the season for the month m, where m = 1 for January, etc.

To solve this problem, four conditions should be checked. If m is 12, 1 or 2, it is winter. If m is 3, 4 or 5, it is spring. If m is 6, 7 or 8, it is summer, and if m is 9, 10 or 11, it is autumn. The possible values of m are 1 to 12, and if a number outside this

range is entered, an error message should be displayed. Use the **else** branch for this.

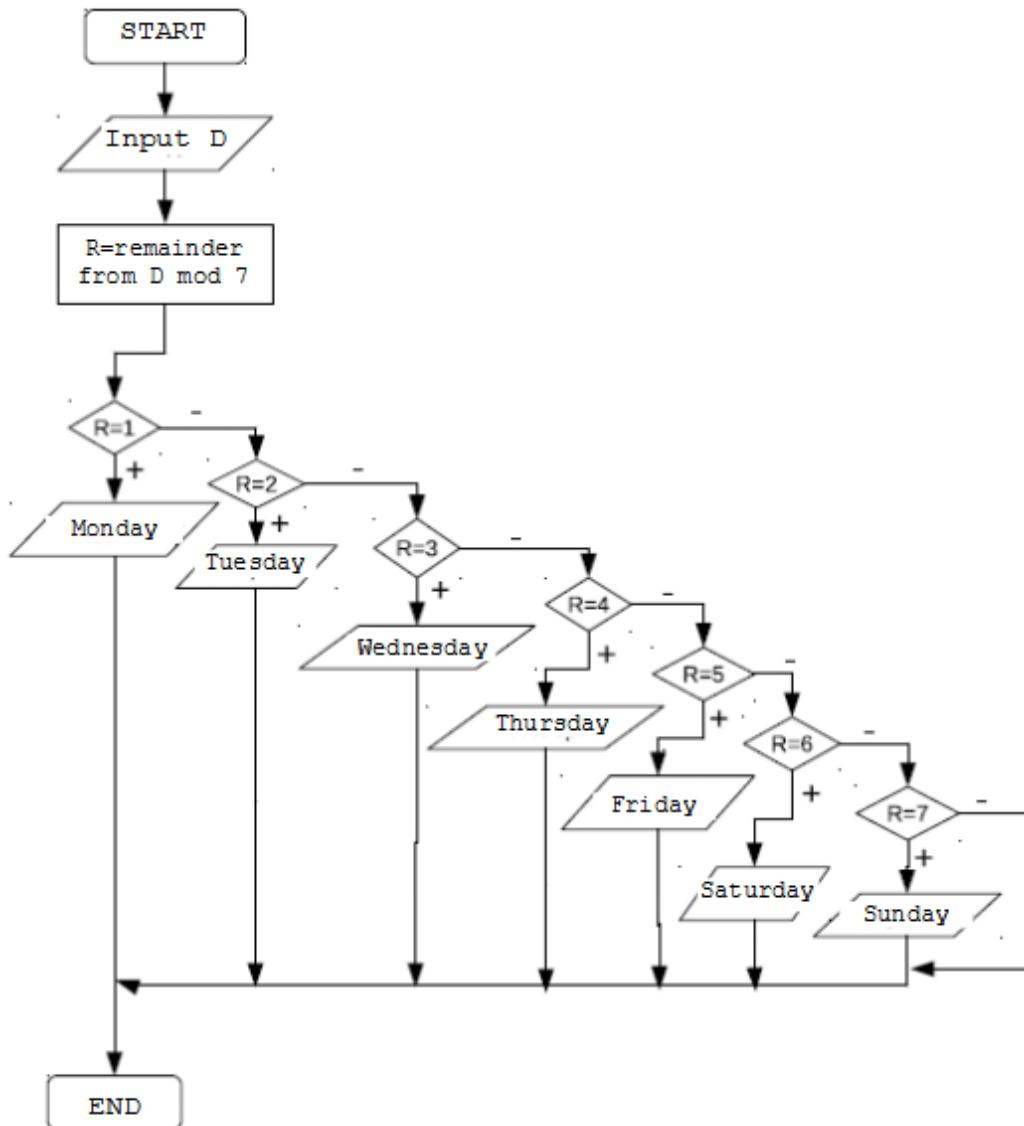


Figure 3.18: Algorithm for Example 3.7

```

var
  m: byte;
begin
  write('Enter the month number m =');
  readln(m);
  case m of
    // Print the season, depending on value of m.
    12, 1, 2: writeln('Winter');
    3..5:      writeln('Spring');
  end;
end.
  
```

---

```

6..8:      writeln('Summer');
9..11:     writeln('Autumn')
// If m is out of range, print error message.
else
  writeln('Error! Input out of range.');
end
end.

```

### 3.4.3 Error Handling and Displaying Messages

The fewer errors in a program, the better, and a very good one will have no errors whatsoever. This means that the programmer should not only thoroughly think over the program algorithm, but also predict possible user errors.

If the user makes a mistake when entering data, for example, he or she should be informed about it. The **MessageDlg** function, which displays a message in a separate window, can be used for this. In general, the function is written as follows:

`MessageDlg (message, message_type, [button_list], help);`

where

- message is the text to be displayed in the message window;
- message\_type determines the window look (Table 3.2);
- button\_list comprises constants (separated by commas), defining the types of the message window buttons (Table 3.3);
- help is the number of the help screen that will be displayed if F1 is pressed; zero if help is not available.

*Table 3.2: Message box type*

Parameter	Message Box Type
mtInformation	informational
mtWarning	warning
mtError	error message
mtConfirmation	confirmation request
mtCustom	custom

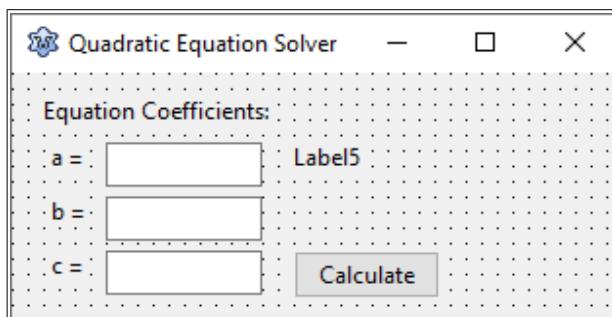
Let us return to the quadratic equation example (Example 3.3). We wrote an algorithm and a Free Pascal program to implement it but will now do the same in Lazarus. Create a new project<sup>6</sup> (Figure 3.19).

Place four labels (Label1, Label2, Label3, Label4) and three edit boxes (Edit1, Edit2, Edit3) on the form, to input the equation coefficients. Label5<sup>7</sup> will display the roots

of the equation or a message stating that there are no roots. All actions needed to calculate the roots of the quadratic equation will be executed when Button1 is pressed.

*Table 3.3: Dialog buttons*

Constant	Button
mbYes	Yes
mbNo	No
mbOk	OK
mbCancel	Cancellation
mbAbort	Abort
mbRetry	Repeat
mbIgnore	Skip
mbHelp	Help



*Figure 3.19: Quadratic equation form*

When entering data, the following errors may occur:

- an input field may contain a string that cannot be converted to a number;
- the value of the coefficient a may be zero.<sup>8</sup>

To prevent such mistakes, filter the data entered by the user. To do this, we will use the built-in procedure **Val(S, X, code)**, which converts string S to a number. The number type (**integer** or **real**) depends on the type of the variable X. If the conversion is successful, then code is set to zero, and the converted number is stored in X. Otherwise, code will contain the position in string S where the error occurred and nothing will be stored in X. The following is a fragment of the code with detailed comments:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```

var
  a, b, c, d, x1, x2: real;
  code1, code2, code3: integer;
begin
  { Enter the coefficients of the quadratic equation.
  Read string from Edit1 and convert to real.
  If successful, then code1 = 0 and the resulting
  number is written to the variable a. }
  val(Edit1.Text, a, code1);
  val(Edit2.Text, b, code2);
  val(Edit3.Text, c, code3);
  // If the conversions were successful, then
  if (code1 = 0) and (code2 = 0) and (code3 = 0) then
    // Check the first coefficient.
    // If the first coefficient is zero, then
    if a = 0 then
      // issue an appropriate message.
      MessageDlg('Please enter a non-zero value for
      a', mtInformation, [mbOk], 0)
    else // otherwise go to solving the quadratic equation
    begin
      d := b*b - 4*a*c;
      Label5.Visible := true;
      if d < 0 then
        Label5.Caption := 'The equation has' + chr(13) + 'no real
      roots.'
      else
        begin
          x1 := (- b + sqrt(d))/(2*a);
          x2 := (- b - sqrt(d))/(2*a);
          Label5.Caption := 'X1 = ' + FloatToStr(x1)+ chr(13) + ' X2 =
          ' + FloatToStr(x2);
        end;
    end
    else // Conversion failed. Issue a message.
      MessageDlg('Please enter a numeric value', mtInformation,
      [mbOk], 0);
  end;

```

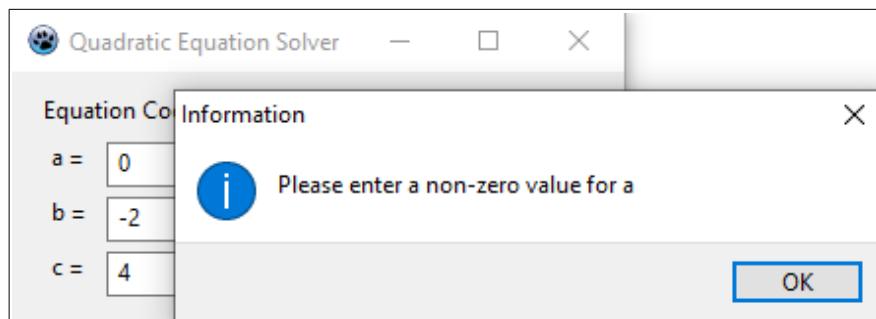


Figure 3.20: Handling input error - coefficient  $a = 0$

The results of the program are shown in Figures 3.20 to 3.23.

The input error handling method in this example is not the only one possible. You can filter the characters entered in the edit boxes. The set of characters in a number is small, and includes the digits from 0 to 9, the minus sign and comma. The Backspace (#8) is also needed, to allow the user to remove characters from the text box. These characters are placed in a set and, if the user attempts to enter a character that does not belong to the set, an error message will be displayed.

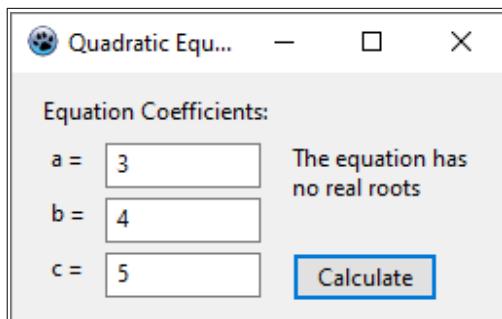


Figure 3.21: Quadratic equation

$$3x^2 + 4x + 5 = 0 \text{ has no real roots}$$

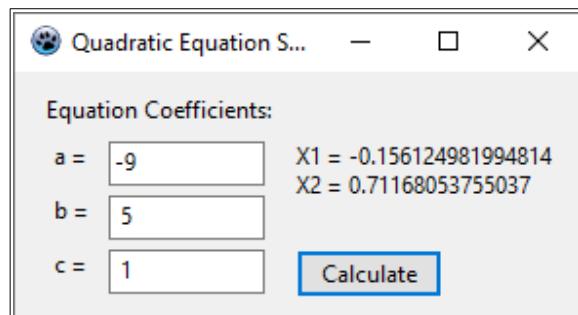


Figure 3.22: Roots found.

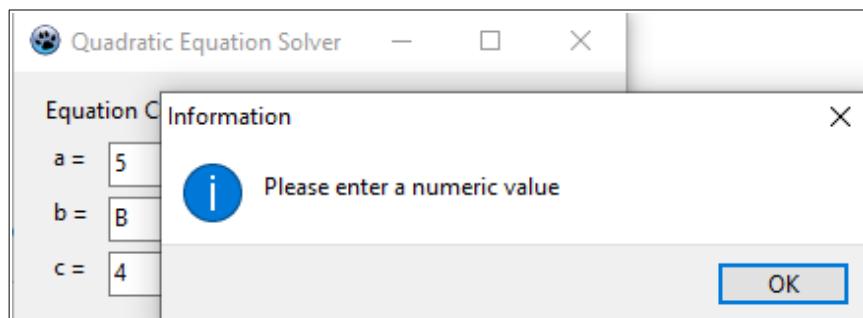


Figure 3.23: Handling input error - string entered

The ability to control character input is provided by the OnKeyPress event. If you select the Edit2 component in the object inspector tree and double-click next to the

OnKeyPress event in the Events tab, then a skeleton procedure for handling this event will be created automatically. The code and comments for this procedure are shown below:

```
//Control character entry in Edit2.  
procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);  
begin // If symbol not in set of valid characters, then  
  if not (Key in [#8, '.', '-', '0'.. '9']) then  
    begin // issue a message and  
      MessageDlg('Please enter a numeric value', mtInformation,  
      [mbOk], 0);  
      Abort; // interrupt the execution of the routine.  
    end;  
end;
```

## 3.5 Repetition Statements

A *cyclical process*, or loop, is a repetition of the same series of actions. The series of actions that is repeated in a loop is called *the loop body*. A single pass through the loop is called a *step*, or *iteration*<sup>9</sup>. The variables that change inside the loop and affect its termination are called *control variables*.

When writing looping algorithms, the following conditions must be met. First, the loop body must influence its control variables so that the loop can terminate. Secondly, the condition must consist of corrective expressions and values that are defined before the completion of the first iteration.

For the convenience of the programmer, Free Pascal provides three statements for implementing loops: **while**, **repeat..until** and **for**.

### 3.5.1 The while..do Statement

Figure 3.24 shows the flowchart of a *loop with a precondition*. The statement that implements this algorithm in Free Pascal is:

**while** expression **do** statement;

The expression must be a **Boolean** constant, a variable, or a **Boolean** expression.

The **while** statement works as follows. The value of the expression is calculated.

If it is **true**, the statement is executed. The loop terminates if the condition is **false**, after which control is transferred to the (first) statement after the loop body. The expression is evaluated before each iteration of the loop. If the expression is **false** when checked the first time, the loop will not be executed, not even once.

If more than one statement must be executed in the loop, use a *compound statement*:

```

while condition do
  begin
    statement_1;
    statement_2;
  ...
    statement_n;
  end;

```

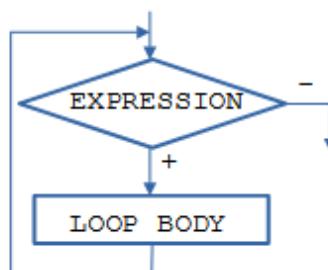


Figure 3.24: Loop structure algorithm with precondition

Consider the following example. Display the values of the function  $y = e^{\sin(x)} \cos(x)$  for segment  $[0; \pi]$  with a step of 0.1.

A loop with a precondition will be used:

```

var x, y: real;
begin
  { Assign a start value to the control variable. }
  x := 0;
  { Loop with precondition. }
  while x <= pi do
    { Loop while control variable x does not exceed pi }
    begin
      { Calculate y. }
      y := exp(sin(x))*cos(x);
      { Display x and y on screen. }

      writeln('x = ', x:3:1, ' y = ', y:6:4);
      { Change of control variable -go to the next value x. }
      x := x + 0.1;
    end; { End of loop. }
end.

```

Running this code displays a sequence of values of x and y:

```

x = 0; y = 1
x = 0.1; y = 1.0995
...

```

```
x = 3.1; y = -1.0415
```

### 3.5.2 The repeat..until Statement

If, in a loop with a precondition, the condition is checked before the loop body, then in a loop with a post condition, the condition is checked after the loop body (Figure 3.25). The statements in the loop body are executed first, then the condition is checked. If it is **false**, then the loop is executed again. Loop execution will end if the condition returns **true**.

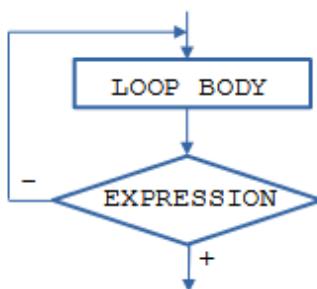


Figure 3.25: Loop structure algorithm with post condition

In Free Pascal, a loop with a post condition is implemented using the construct

```
repeat
  statement;
until expression;
```

or, if the body of the loop consists of more than one statement:

```
repeat
  statement_1;
  statement_2;
  ...
  statement_n;
until expression;
```

The expression must be **Boolean**.

The loop works as follows. At the beginning, the statements in the loop body are executed. The expression is then evaluated. If it is **false**, the loop body is executed again. Otherwise, the loop ends and control is transferred to the statement after the loop.

Thus, it is easy to see that a loop with a post condition will always be executed at least once, as opposed to a loop with a precondition, which may not be executed even once.

If we use a loop with a post condition to create a subroutine that displays the values of the function  $y = e^{\sin(x)} \cos(x)$  in the segment  $[0; \pi]$  with a step of 0.1, we will get:

```
var x, y: real;
begin
  { Assign an initial value to the control variable. }
  x := 0;
  { Loop with post condition. }
  repeat           {Begin loop}
    y := exp(sin(x))*cos(x);
    writeln('x = ', x, ' y = ', y);
    x := x + 0.1;   { Increment the control variable. }
  until x > pi;    { End loop }
  { when control variable exceeds final value. }
end.
```

### 3.5.3 The for..do Statement

The conditional loop statements are very flexible, but not very convenient for organizing strict loops that must be performed a specified number of times. The **for..do** loop statement is used in exactly such cases:

```
for control-variable := start-value to final_value do statement;
for control-variable := end-value downto start-value do statement;
```

where statement is a Pascal statement, control-variable is an integer, char, numeric or enumeration type, and start-value and end-value must be of the same type as control-variable.

The step width of the **for** loop is always constant and equal to the interval between the two closest values of the control variable type (with an integer value the step width of the loop is 1).

If the body of the loop consists of more than one statement, you must use a *compound statement*:

```
for control-variable := start-value to end-value do
  begin
    statement_1;
    statement_2;
  ...
    statement_n;
  end;
```

Let us describe the algorithm for the **for..do** loop (Figure 3.26).

- 1) Assign an initial value to the control variable.

- 2) If the value of the control variable exceeds the end value, then the loop terminates. Otherwise, Item 3 is performed.
- 3) Execute statement(s).
- 4) Change the value of the control variable by the corresponding step and go to Item 2.

This algorithm is a loop with a precondition.

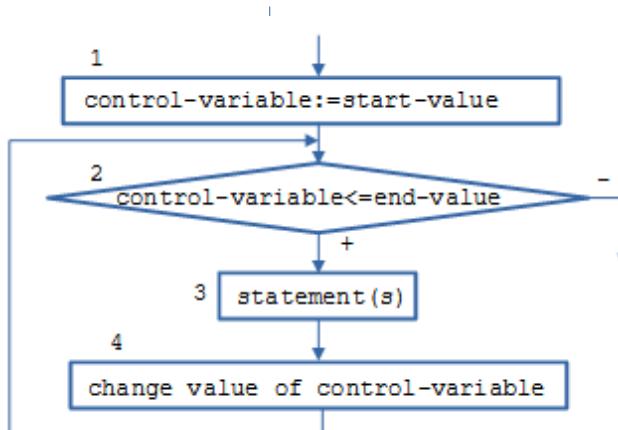


Figure 3.26: Algorithm for the for..do loop

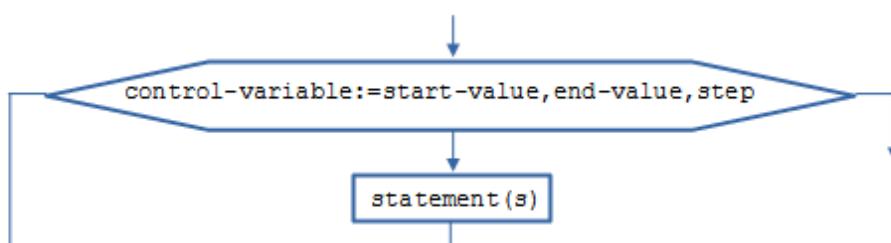


Figure 3.27: Representing a for..do loop in a flowchart

In the future, to avoid creating too cumbersome algorithms, in the flowcharts, the for loop will be depicted as shown in Figure 3.27<sup>10</sup>.

The code below demonstrates the use of the **for** loop:

```

var i: integer; c: char;
begin
  { Display numbers from 1 to 10. }
  for i := 1 to 10 do
    writeln(i);
  { Display numbers from 10 to -10. }
  for i := 10 downto -10 do
    writeln(i);

```

---

```

{ Display characters a through r. }
for c := 'a' to 'r' do
    writeln(c);
end;

```

Let us return to the problem of outputting the values of the function  $y = e^{\sin(x)} \cos(x)$  in the segment  $[0; \pi]$  with a step of 0.1. As you can see, the number of loop repetitions is not explicitly specified. However, this value can be easily calculated. Suppose the control variable  $x$  takes values in the range  $X_n$  to  $X_k$ , with a step width  $dx$ , then the number of repetitions of the loop body can be determined by the formula:

$$n = \text{round}(X_k - X_n)/dx + 1 \quad (3.6)$$

by rounding the result of the division to an integer. Therefore, the code to output the values of the function  $y = e^{\sin(x)} \cos(x)$  in the segment  $[0; \pi]$  with a step of 0.1 will be:

```

var i, n: integer; x, y: real;
begin
    { Number of repetitions }
    n := round((pi - 0) /0.1) + 1;
    x := 0;    { Initial value of the argument. }
    { Loop with a known number of repetitions }
    { i - control variable, changes from 1 to n with step 1. }
    for i := 1 to n do
        begin          { Begin loop. }
            y := exp(sin(x))*cos(x); { Calculate function value}
            { for the corresponding argument value. }
            writeln('x = ', x:4:2, ' y = ', y:6:4);
            x := x + 0.1;           { Calculate new argument value. }
        end;             { End loop. }
    end.

```

### 3.5.4 Control Transfer Statements

*Control transfer statements* forcefully change the order of execution. There are five such statements in Free Pascal: **goto**, **break**, **continue**, **exit** and **halt**.

The **goto** statement label, where the label is an ordinary identifier, is used for an unconditional jump, and control is transferred to the statement with the label.

label: statement;

Usually, the use of the **goto** statement complicates a program and makes it difficult to debug. Using this statement violates the principle of structural programming<sup>11</sup>, according to which all blocks in a program must have only one entry and one exit point. In most algorithms, the use of **goto** could be avoided. It will not be used in

this book, but the reader should be aware of its existence in Free Pascal.

The **break** and **continue** statements are used only inside loops. The **break** statement immediately exits from **repeat**, **while**, **for** loops, and control is transferred to the statement immediately following the loop. The **continue** statement starts a new loop iteration, even if the previous one was not completed.

The **exit** statement terminates the subroutine.

The **halt** statement terminates program execution.

### 3.5.5 Solving Problems Using Loops

Consider the use of loop statements through specific examples.

EXAMPLE 3.9. Find the greatest common divisor (GCD) of two natural numbers A and B.

*Input data:* A and B. *Output:* A, the GCD.

To solve the problem, we will use the Euclidean algorithm. At each step we will decrease the larger number by the value of the smaller until both values become equal, as shown in Table 3.4.

Table 3.4: Finding GCD for numbers A = 25 and B = 15.

Input data	First step	Second step	Third step	GCD(A, B)= 5
A = 25	A = 10	A = 10	A = 5	
B = 15	B = 15	B = 5	B = 5	

In the flowchart for solving the problem shown in Figure 3.28, a loop with a precondition will be used. The body of the loop will be looped through repeatedly until A is equal to B. Therefore, when creating a program a **while..do** loop will be used.

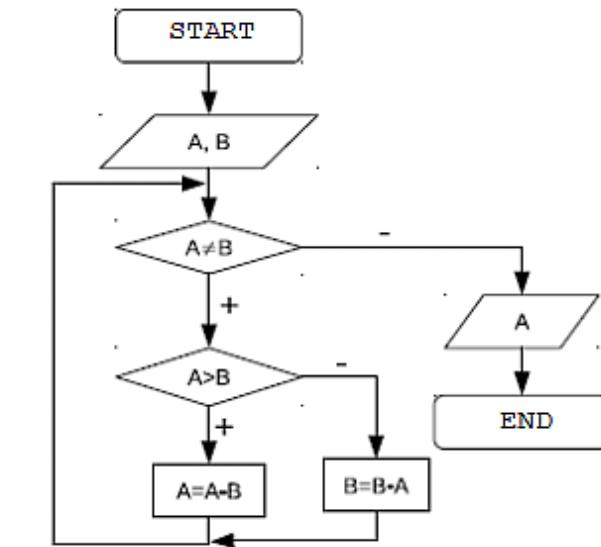


Figure 3.28: Algorithm for the greatest common divisor

A Free Pascal program to solve the problem is shown below:

```

var a, b: word;
begin
  writeln('Enter two natural (positive whole) numbers:');
  write('A = '); readln(a);
  write('B = '); readln(b);
  { If the numbers are not equal, execute the loop body. }
  while a <> b do
    { If > B, then decrease its value by B }
    if a > b then
      a := a - b
    { otherwise decrease the value of the number B by A. }
    else
      b := b - a;
    writeln('gcd = ', A);
end.
  
```

EXAMPLE 3.10. Compute the factorial of n ( $n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$ ).

*Input data:* n - an integer whose factorial is to be computed. *Output:* factorial - the product of whole numbers from 1 to n.

*Intermediate variables:* i - the control variable, an integer variable that sequentially takes values 2, 3, 4, etc, up to n.

The block flowchart is shown in Figure 3.29.

The user inputs the number n. The value of variable factorial, which will store the product of the sequence of numbers, is set to one. Control passes to a loop with control variable i, which is set initially to two. If the control variable is less than or equal to n, the statement in the loop body will be executed. This statement multiplies the old value stored in factorial by the current value of the control variable and stores the result in factorial. When the control variable i becomes greater than n, the loop ends, and the value currently stored in factorial is printed on screen.

Below is the Free Pascal code for calculating the factorial.

```
var
  factorial, n, i: integer;
begin
  write('n = '); readln(n);
  factorial := 1;
  for i := 2 to n do
    factorial := factorial*i;
  writeln(factorial);
end.
```

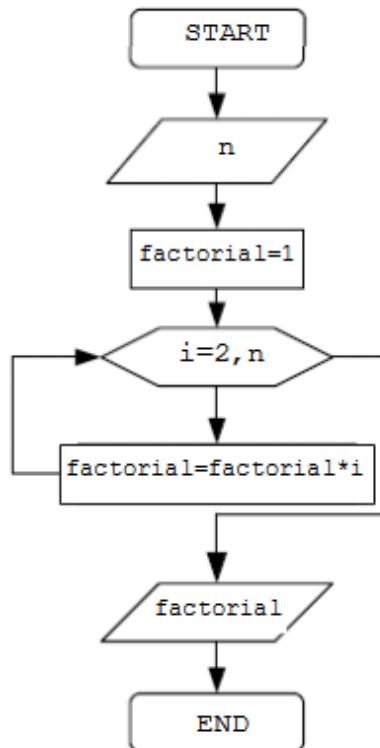


Figure 3.29: Factorial algorithm

EXAMPLE 3.11. Calculate  $a^n$ , where n is positive integer.

*Input data:* a - real number to be raised to the power n.

*Output:* p (real number) - the result of raising a real number a to a positive integer power n.

*Intermediate data:* i – the integer control variable that takes values from 1 to n, in steps of 1. Figure 3.30 shows the flowchart.

To obtain the integer power n of a real variable a, a should be multiplied by itself n times. The result will be stored in the real variable P. During the next iteration, the previous value of P will be read and multiplied by the base a, and the result will be written back to P. The loop will be executed n times.

Table 3.5 shows the protocol for the execution of the algorithm when raising the number 2 to the fifth power: a = 2, n = 5. Similar tables, filled by hand, are used to test all stages of a program.

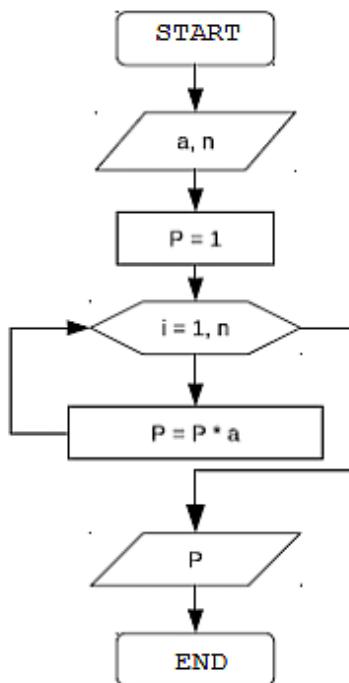


Figure 3.30: Raising a real number to an integer power

Table 3.5: Raising the number a to the power n

i		1	2	3	4	5
P	1	2	4	8	16	32

The following is the code to solve the problem.

```
var
  a, P: real;
  i, n: word;
begin
  write('Enter base a = ');
  readln(a);
  write('Enter exponent n = ');
  readln(n);
  P := 1;
  for i := 1 to n do
    P := P*a;
  writeln ('P = ', P:1:3);
end.
```

EXAMPLE 3.12. Calculate the sum of natural even numbers, not exceeding n.

*Input data:* n – integer.

*Output:* S - the sum of even numbers.

*Intermediate variables:* i – the integer control variable, which takes values 2, 4, 6, 8, etc.

When adding several numbers, the result should be accumulated in a variable, by reading the previous value of the sum from this variable and then adding the next term to it. When assigning the variable for accumulating the sum, care should be taken to ensure that old values stored in the variable or in its memory area do not affect the result of the addition. Before starting the loop, the value of the variable for accumulating the sum must be set to zero ( $s = 0$ ). The flowchart for solving this problem is shown in Figure 3.31.

Since the loop control variable  $i$  changes with a step of 2, the flowchart for solving this problem (Figure 3.31), uses a loop with a precondition, which is implemented in the program by using a **while..do** statement:

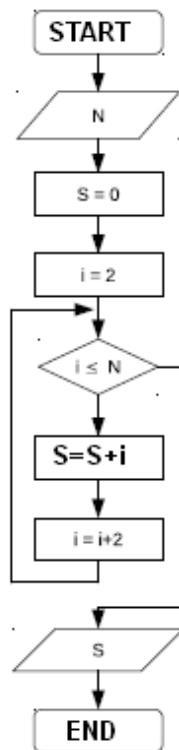


Figure 3.31: Sum of even natural numbers

```

var n, i, S: word;
begin
  write('n = ');
  readln(n);
  S := 0;
  i := 2;
  while i <= n do
  begin
    S := S + i;
    i := i + 2;
  end;
  writeln('S = ', S);
end.
  
```

The same problem can be solved differently using a **for..do** loop:

```

var n, i, S: word;
begin
  write('n = ');
  readln(n);
  S := 0;
  for i := 1 to n do
  { If the remainder of dividing the control variable by 2 is 0,
  then this number is even, therefore, it should be added }
  
```

```

if i mod 2 = 0 then
  S := S + i;
  writeln('S = ', S);
end.

```

Table 3.6 shows the results of testing the program for  $n = 7$ . It is easy to see that for odd values of the control variable, the value of the variable for accumulating the sum does not change.

*Table 3.6: Summing even numbers*

i	1	2	3	4	5	6	7
S	0	0	2	2	6	6	12

EXAMPLE 3.13. Given a natural number  $n$ . Define  $K$ , the number of divisors of this number that are less than  $n$  (for example, for  $n = 12$ , the divisors are 1, 2, 3, 4 and 6. Thus,  $K = 5$ ).

*Input data:*  $n$  - integer.

*Output:* - integer  $K$  - the number of divisors of  $n$ .

*Intermediate variables:*  $i$  - control variable, possible divisors of  $n$ .

In the flow chart shown in Figure 3.32, the following algorithm is implemented: The variable  $K$ , which will store the number of divisors of a given number, is initially set to zero ( $K = 0$ ), to avoid affecting the result ( $K = 0$ ). Next, a loop increments the control variable  $i$ , which serves as possible divisors of  $n$ . The control variable changes from 1 to  $N/2$  with a step of 1. If the given number is divisible by the control variable, then that  $i$  is a divisor of  $n$ , and the value of the variable  $K$  should be increased by one. The cycle must be repeated  $N/2$  times.

Table 3.7 shows the results of testing the algorithm when the given number  $n = 12$ .

*Table 3.7: Number of divisors of the number  $n = 12$*

i		1	2	3	4	5	6
K	0	1	2	3	4	4	5

The code corresponding to the algorithm is shown below:

```

var n, i, K: word;
begin
  write('n = ');
  readln(n);

```

```

K := 0;
for I := 1 to n div 2 do
  if n mod i = 0 then      { If n is divisible by i, then }
    K := K + 1;            { increment counter by one. }
writeln(' K = ', K);
end.

```

EXAMPLE 3.14. Determine if a natural number N is a prime number. A natural number is a prime number if it is divisible without a remainder only by one and itself. N = 13 is a prime number, since it is only divisible by 1 and 13. N = 12 is not a prime number, since it is divisible by 1,2, 3, 4, 6 and 12.

*Input data:* n - integer.

*Output:* message.

*Intermediate data:* i - control variable, possible divisors of n.

The algorithm for solving this problem (Figure 3.33) requires determining if numbers from 2 to n/2 are divisors of the number n. If there are no divisors then the number is a prime number.

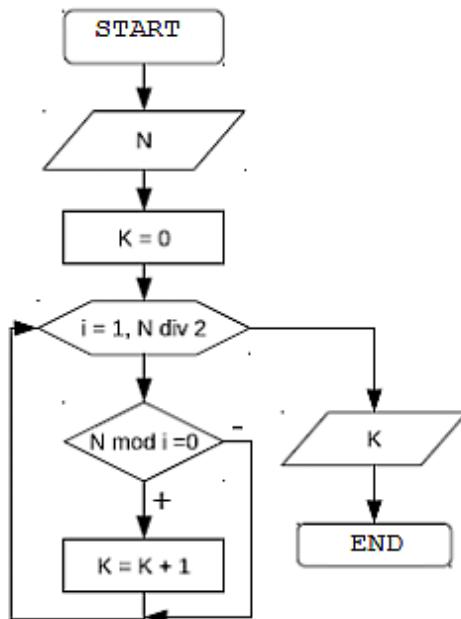


Figure 3.32: Divisors of a natural number

Assume n is a prime number (ok := **true**). Create a loop to cycle through the control

variable i, from 2 to n/2, while checking if n is divisible by i. If divisible, then n has at least one divisor and is not a prime number (ok := **false**). Exit the loop at once since it makes no sense to check for additional divisors.

The algorithm provides two exits from the loop. The first is natural and occurs when i exceeds n/2, while the second is early. After exiting the loop, if ok = **true**, then n is a prime number. Otherwise, n is not a prime number.

In Free Pascal, use **break** to exit early from a loop:

```
var
  n, i: integer;
  ok: boolean;
begin
  write('n = ');
  readln(n);
  ok := true;    { Assume the number is prime. }
  for i := 2 to n div 2 do
    { If there is at least one divisor, then }
    if n mod i = 0 then
      begin
        ok := false;
        { the number is not prime and }
        break;
      { exit early from the cycle. }
    end;
    { Checking the value of a Boolean parameter, and }
  if ok then
    { print the corresponding message. }
    writeln('The number ', n, 'is a prime number.')
  else
    writeln('The number ', n, 'is not a prime number.');
end.
```

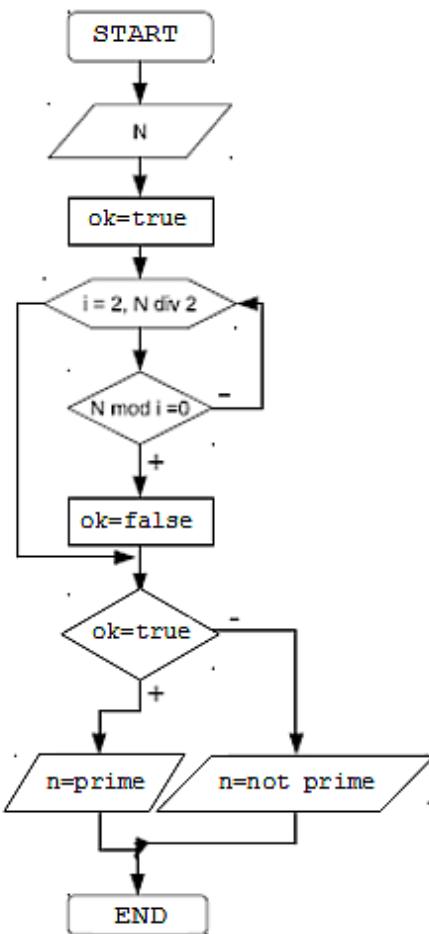


Figure 3.33: Finding a prime number

EXAMPLE 3.15. Determine the number of primes in the range from  $n$  to  $m$ , where  $n$  and  $m$  natural numbers, and  $n \leq m$ .

The algorithm for solving this problem is shown in Figure 3.34.

Note that the input is validated here. A loop with a post-condition will request valid input repeatedly, if  $n$  or  $m$  is not positive, or the value of  $n$  exceeds  $m$ , until the input data is valid. Then, each number in the specified range (the control variable  $i$  takes values from  $n$  to  $m$ ) is checked. If the number is a prime number, then the variable  $k$  is increased by one. Problem 3.14 defines prime numbers in detail.

The Free Pascal program to calculate the number of prime numbers in a given range is shown below:

```

var
  n, m, i, j, k: longint;
  ok: boolean;
begin
  repeat
    write ('n = ');
    readln (n);
    write ('m = ');
    readln (m);
  until (n > 0) and (m > 0) and (n < m);
  k := 0; { Number of primes. }
  for i := n to m do
  { The control variable i takes values from n to m. }
  begin { Find prime number. }
    ok := true;
    for j := 2 to i div 2 do
      if i mod j = 0 then
        begin
          ok := false;
          break;
        end;
      { If the number is prime, increase the k by 1. }
      if ok then k := k + 1;
    end;
    if k = 0 then
      writeln ('There are no prime numbers in the range.')
    else
      writeln ('Prime numbers in the range', k);
  end.

```

EXAMPLE 3.16. Find the number of digits in a natural number.

*Input data:* n - integer.

*Output:* kol - the number of digits in the number.

*Intermediate data:* m - a variable for temporarily storing the current value of n.

To find the number of digits in a number, you need to determine how many times the number can be divided by ten entirely. For instance, if n = 12345, then the number of digits kol = 5. The calculation results are summarized into Table 3.8. The algorithm to find the number of digits in a number is shown in Figure 3.35.

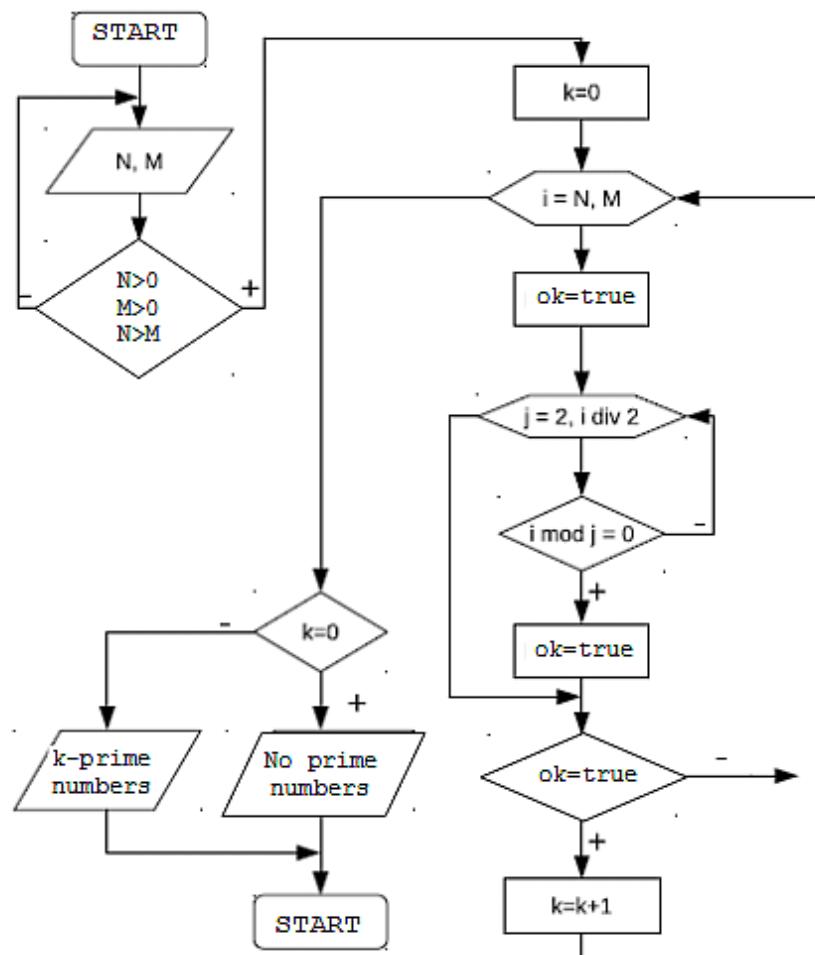


Figure 3.34: Finding the prime numbers in a given interval

Table 3.8: Number of digits in a number

kol	n
1	12345
2	12345 div 10 = 1234
3	1234 div 10 = 123
4	123 div 10 = 12
5	12 div 10 = 1
	1 div 10 = 0

The code to solve this problem can be written as follows:

```

var
  m, n: longint;
  kol: word;
begin
  { Verify if the input is positive (natural number). }
  repeat
    write('n = ');
    readln(n);
  until n > 0;
  m := n;           { Store value of variable N. }
  kol := 1;          { Let the number consist of one digit. }
  while m div 10 > 0 do
  { Execute the loop body while the number is divisible by 10. }
  begin
    kol := kol + 1;    { Counter for the number of digits. }
    m := m div 10;     { Change the number. }
  end;
  writeln('kol = ', kol);
end.

```

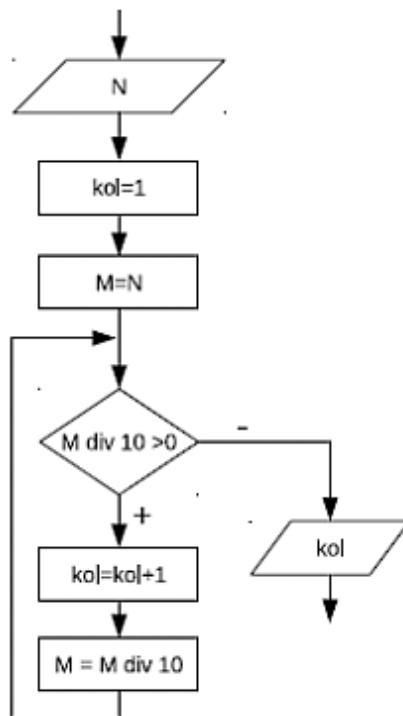


Figure 3.35: The number of digits in a number

EXAMPLE 3.17. Determine if a natural number  $N$  contains zeros, and their positions (for example, the number 1101111011 contains zeros in the third and eighth positions).

*Input data:* N - integer.

*Output:* pos - position of the digit in the number.

*Intermediate data:* i - control variable, m is a variable for temporarily storing the value n.

To determine the positions of digits in a number, the number of digits in the number<sup>12</sup> must be known. Thus, in the first part of the solution, the number of digits kol in the number must be found. Then, loop through the digits in the number. If a digit is zero, display on screen the position this digit occupies. The procedure for determining the positions of the digits in the number  $n = 120405$  is shown in Table 3.9.

The flowchart for solving this problem is shown in Figure 3.36.

*Table 3.9: Position of the current digit in a number m*

i	Number	Digit	Position
1	$120405 \text{ div } 10 = 12040$	$120405 \text{ mod } 10 = 5$	6
2	$12040 \text{ div } 10 = 1204$	$12040 \text{ mod } 10 = 0$	5
3	$1204 \text{ div } 10 = 120$	$1204 \text{ mod } 10 = 4$	4
4	$120 \text{ div } 10 = 12$	$120 \text{ mod } 10 = 0$	3
5	$12 \text{ div } 10 = 1$	$12 \text{ mod } 10 = 2$	2
6	$1 \text{ div } 10 = 0$	$1 \text{ mod } 10 = 1$	1

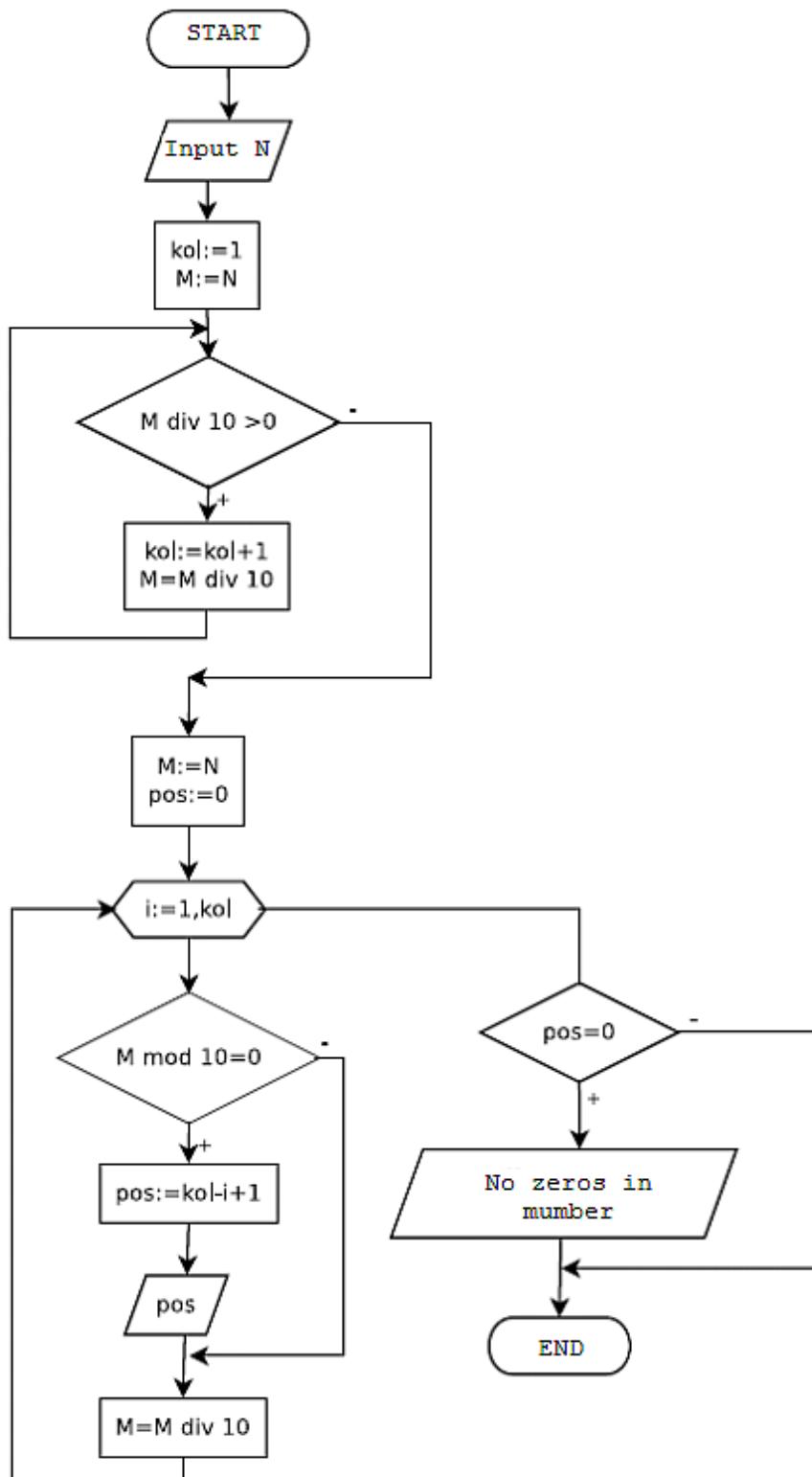


Figure 3.36: Algorithm for Example 3.17

The code to implement this algorithm is shown below:

```

var
  m, n: longint;
  i, pos, kol: word;
begin
  { Verify that the input is positive (natural number). }
  repeat
    write('n = ');
    readln(N);
  until n > 0;
  // Find kol - the number of digits.
  m := n;           { Store value of variable n. }
  kol := 1; { Assume the number has one digit. }
  while m div 10 > 0 do
  { Execute the body of the loop while n is divisible by 10. }
  begin
    kol := kol + 1;      { Counter for the number of digits. }
    m := m div 10;        { Change the number. }
  end;
  writeln('kol = ', kol);
  m := n;
  pos := 0; { Assume there are no zeros in the number. }
  for i := 1 to kol do
  begin
    { Select a digit from the number and compare it to zero. }
    if (m mod 10 = 0) then
      begin
        pos := kol - i + 1;      { Position of zero in the number. }
        writeln ('Zero at Position ', pos, '.');
      end;
    m := m div 10;        { Change the number. }
  end;
  if pos = 0 then writeln('The number does not contain 0.');
end.

```

### 3.5.6 Entering Data Using an InputBox

An *Input window* is a standard dialog that appears on screen by calling the **InputBox** function. In general, the code to call this function looks like this:

name := InputBox(window\_title, hint, value);

where

- window\_title is a string to set the title of the window;
- hint is an explanatory message;
- value is the default text that will appear in the input field when the window appears on the screen;

- name is a string variable, to store the current value in the input field.

After running the code fragment:

```
var
  S: string;
begin
  S := InputBox('BOX TITLE', 'Hint: enter input data', 'Given
value');
end;
```

the window shown in Figure 3.37 will appear.

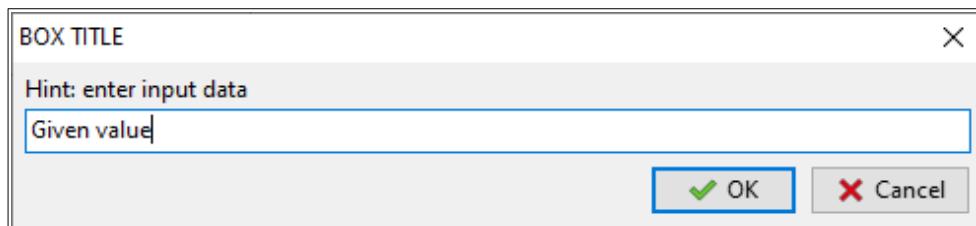


Figure 3.37: Input window

The user may edit the text in the input field. Clicking on the OK button will cause the line in the input field to be saved in the variable on the left of the assignment operator. In this case, the string 'Given value' will be saved in the variable S. Clicking the Cancel button will close the input window.

Since the InputBox function returns a string value, convert the string to numeric data, using the type conversion functions:

```
var
  S: string;
  degree, radian: real;
begin
  S := InputBox('Data input', 'Enter the angle in radians', '0.000');
  radian := StrToFloat(S);
  degree := radian * 180 / pi;
  MessageDlg ('Angle in degrees: '+ FloatToStr(degree),
    mtInformation, [mbOk], 0);
end;
```

You can use a dialog box when solving problems processing certain *number sequences*. Let us consider several such examples.

EXAMPLE 3.18. Determine the largest term in a sequence of N real numbers.

*Input data:* N is an **integer**; X is a **real** number, which defines the current term of the sequence.

*Output:* Max is a **real** number, sequence term with the highest value.

*Intermediate variables:* i is the control variable, number of input terms in that sequence.

The algorithm for finding the largest term in the sequence is as follows (Figure 3.38).

Create a variable Max, which will store the largest number in the sequence. Get the number of terms in the sequence and the first term in the sequence. Assume that the first term is the largest, and save it in Max. Then get the second term in the sequence and compare it to the value saved in Max. If the second term is greater than Max then save it to Max. Otherwise, do nothing. Then get the next term in the sequence and repeat the algorithm from the beginning. This way, Max will contain the largest term in the sequence after the entire sequence is processed<sup>13</sup>.

Place a label Label1 and button Button1 on a form (see Figure 3.39). Clicking on the button will execute the following procedure:

```
procedure TForm1.Button1Click (Sender: TObject);
var
  i, N: integer;
  Max, X: real;
  S: string;
begin
  S := InputBox ('Input', 'Number of terms in sequence: ',' 0 ');
  N := StrToInt(S);
  // Enter the first term in the sequence.
  S := InputBox('Term Input', 'Enter number: ',' 0 ');
  X := StrToFloat(S);
  Max := X; {Assume the first term is maximum Max = X.}
  for i := 2 to N do  { i=2, since first term already processed }
begin
  // Enter the next term in the sequence.
  S := InputBox ('Term Input', 'Enter number: ',' 0 ');
  X := StrToInt(S);
  if X > Max then Max := X;
end;
  MessageDlg ('Largest term: '+ FloatToStr(Max),      mtInformation,
[mbOk], 0);
end;
```

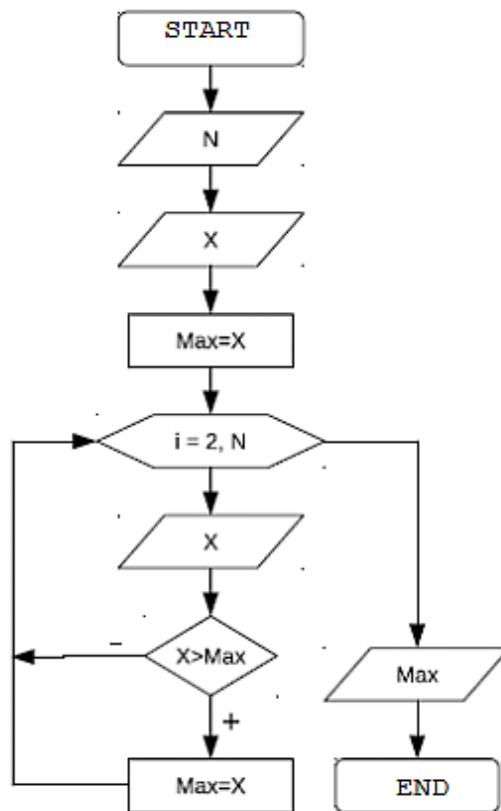


Figure 3.38: The largest number in a sequence

Dialog boxes in the program are shown in Figures 3.39 to 3.42.

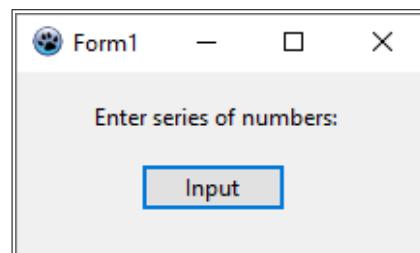


Figure 3.39: First dialog window  
for Example 3.18

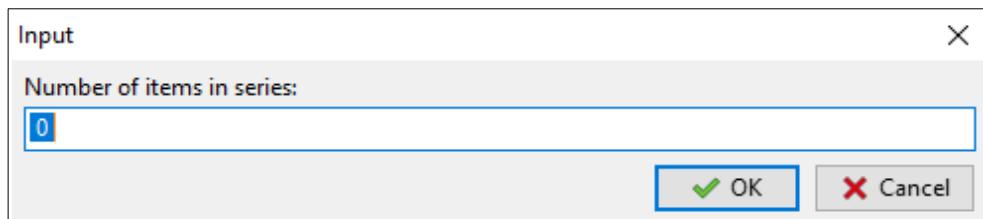


Figure 3.40: The second dialog box for Example 3.18

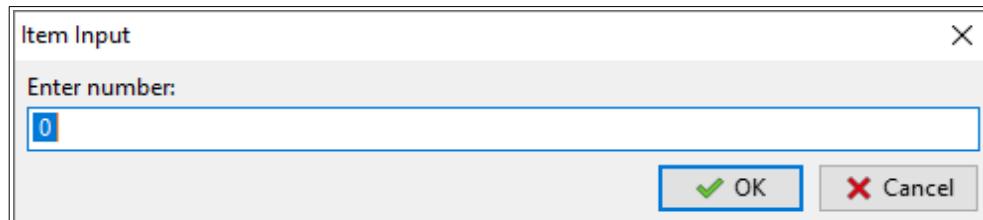


Figure 3.41: The third dialog box for Example 3.18

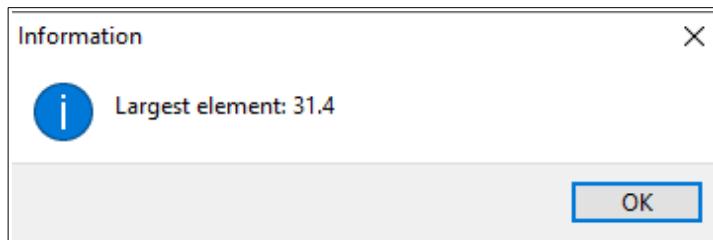


Figure 3.42: The result from Example 3.18

EXAMPLE 3.19. Find the smallest positive number in a sequence of integers ending with 0. If there is more than one<sup>14</sup>, determine how many there are.

The flowchart for solving this problem is shown in Figure 3.43.

Below is the routine, with detailed comments<sup>15</sup>. Run the routine by clicking on a button Button1 placed on a form.

```
procedure TForm1.Button1Click (Sender: TObject);
var
  N, k, min: integer;
  S: string;
begin
  // Enter the first term in the sequence.
  S := InputBox ('Input Sequence Term', ' Enter integer, or 0 to end
  ', '0');
  N := StrToInt(S);
  { Assume no positive numbers initially. Set K = 0. K will store
  the number of positive minimums. }
```

```
K := 0;
// While the number entered is not zero, execute loop body.
while N <> 0 do
begin
    // Check if the entered number is positive.
    if N > 0 then
begin
    { If N > 0 and K = 0, the 1st positive term.
      Assume N is the minimum. Set Min = N, K = 1. }
    if k = 0 then
begin
        k := 1;
        min := N;
end
    { If the term is not the first, compare to the saved minimum.
      If term is smaller, save it to Min and reset counter. }
    else
        if N < min then
begin
        min := N;
        k := 1;
end
    { If term equals the saved minimum, increase the number of
      minimums by 1. }
    else
        if N = min then k := k + 1;
end;           // End if N > 0 loop.
// Enter the next term in the sequence.
S := InputBox('Input sequence term',' Enter integer, or 0 to end
','0');
N := StrToInt(S);
end;           // End of if N <> 0 loop.

if k <> 0 then
{ If counter value is not zero, display minimum term and number of
  minimums. }
MessageDlg('MIN = ' + IntToStr(min) +'K = ' + IntToStr (k),
mtInformation, [mbOk], 0)
else
    // Otherwise state that there are no positive numbers.
    MessageDlg('No positive numbers',mtInformation, [mbOk], 0);
end;
```

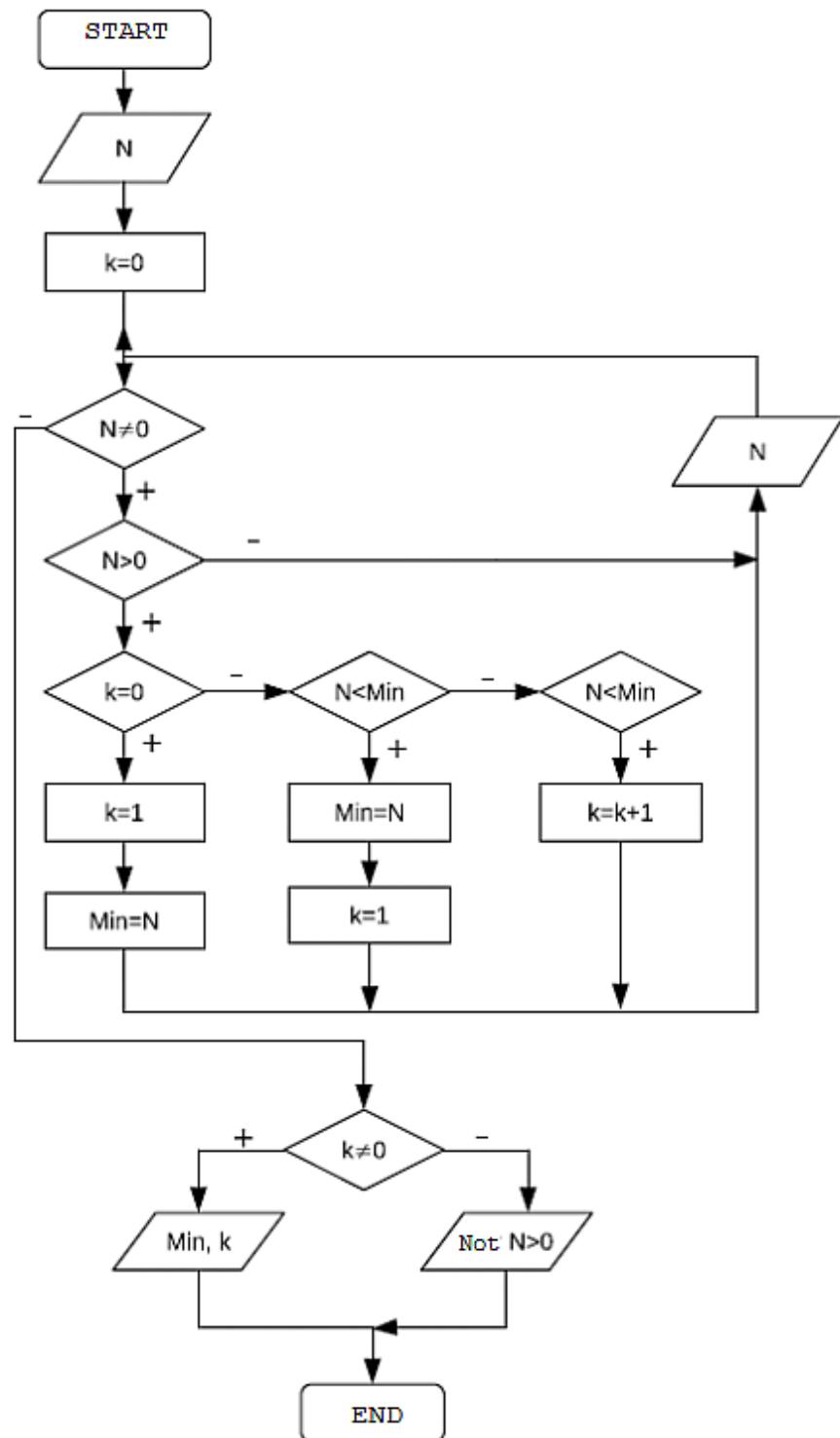


Figure 3.43: The minimum positive number in a sequence

EXAMPLE 3.20. Determine how many times a sequence of N arbitrary numbers changes sign.

To solve the problem, you need to multiply the terms of the sequence in pairs. If the product of a pair of numbers is negative, then these numbers have different signs.

Let k, the number of sign changes in the sequence, be 0.

Let the variable A store the current term in the sequence, and variable B the previous. Let N be the number of terms in the sequence. Create a loop, in which variable i changes from 1 to N. The following actions will be performed in the loop. Consider a term in the sequence (A). If this is the first term of the sequence ( $I = 1$ ), then there is nothing to compare it with, and the variable A is simply saved in the variable B ( $B := A$ ). If this is not the first term ( $i > 1$ ), then check the sign of the product  $A * B$  (current and previous sequence terms). If the product is negative, then the counter k increases by 1. After that, do not forget to save A in variable B. The algorithm flowchart is shown in Figure 3.44.

On a form, place two labels Label1 and Label2, an edit box Edit1 and a button Button1 (Figure 3.45).

The subroutine code that will be executed when the button is clicked is shown below.

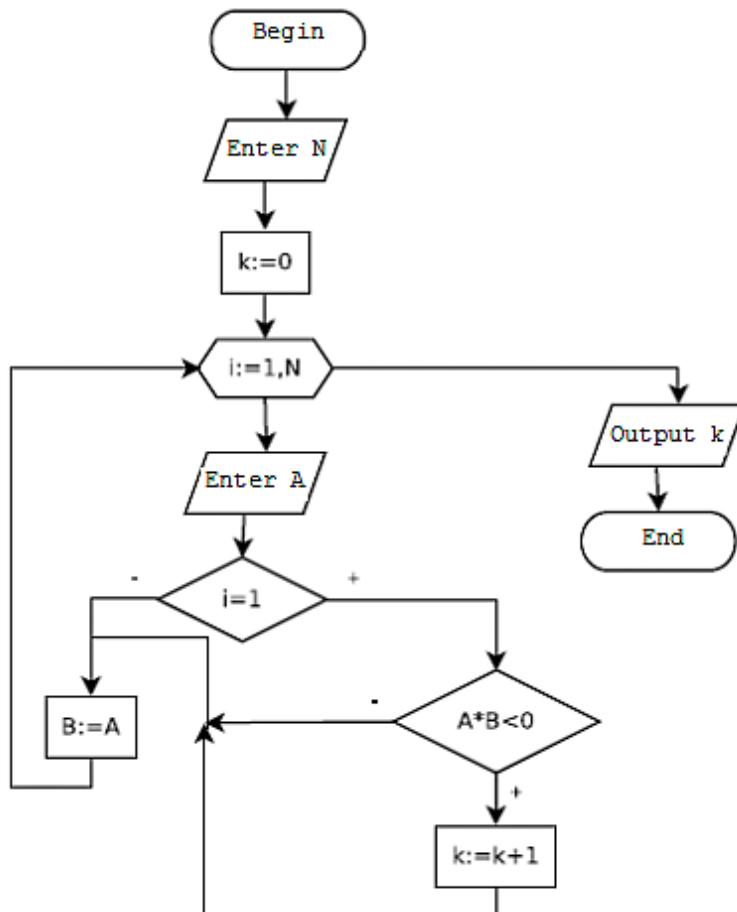


Figure 3.44: Algorithm for Example 3.20

The form has a title bar 'No of Sign Changes' with minimize, maximize, and close buttons. It contains a text input field 'Number of elements in series:' with the value '5'. Below it is a button 'Input Elements'.

Figure 3.45: The main form for Example 3.20

```

procedure TForm1.Button1Click (Sender: TObject);
var
  N, A, B, i, k: integer;
  S: string;
begin
  N := StrToInt(Edit1.Text);
  k := 0;
  B := 0;
  for i := 1 to N do
  begin
    S := InputBox('Input sequence term', 'Enter integer: ', '0');
    if S = '' then
      Continue;
    A := StrToInt(S);
    if B = 0 then
      B := A;
    if A < 0 then
      Inc(k);
    B := A;
  end;
  ShowMessage(IntToStr(k));
end;
  
```

```
A := StrToInt(S);
if (i <> 1) then
  if A*B < 0 then k := k + 1;
  B := A;
end;
MessageDlg('K = ' + IntToStr(k), mtInformation, [mbOk], 0);
end;
```

Dialogs used in the program are shown in Figure 3.46 and 3.47.

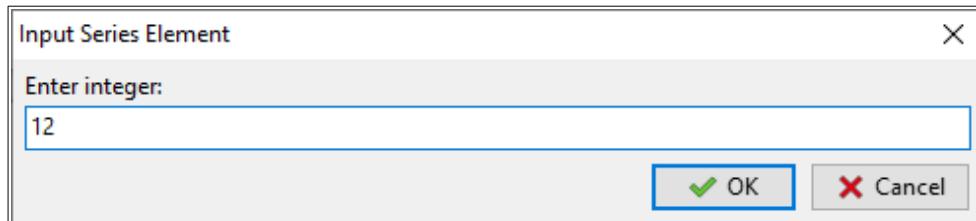


Figure 3.46: The second dialog box for Example 3.20



Figure 3.47: The final result displayed for Example 3.20

## 3.6 Exercises

Draw a flowchart for solving the problem and write a program.

### 3.6.1 Exercises (Branching)

Problems 1 to 8. Calculate  $f(a)$ , where  $a$  is a real number, for the function  $y = f(x)$  shown in the graphs in Figure 3.48 to Figure 3.55 below.

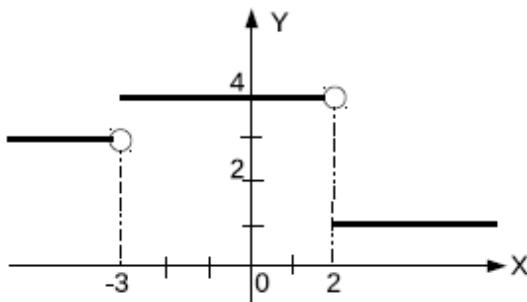


Figure 3.48: Problem 1

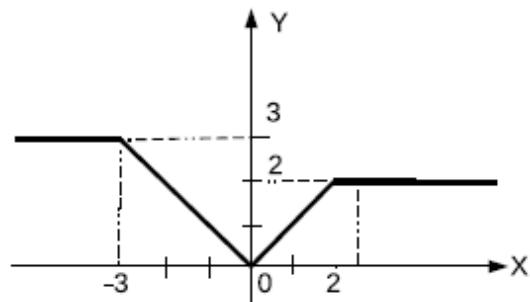


Figure 3.49: Problem 2

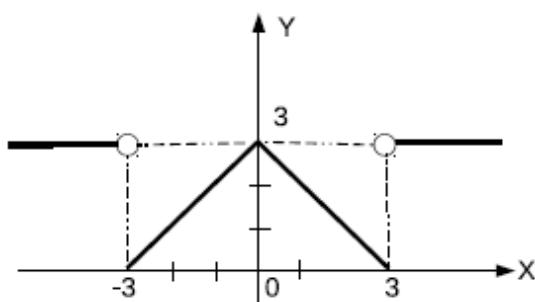


Figure 3.50: Problem 3

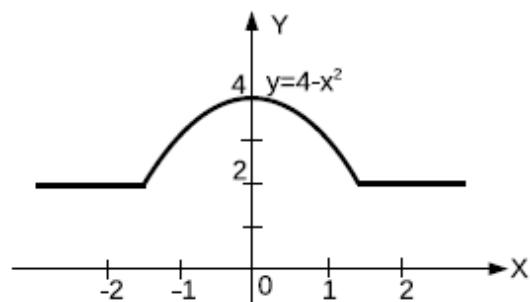


Figure 3.51: Problem 4

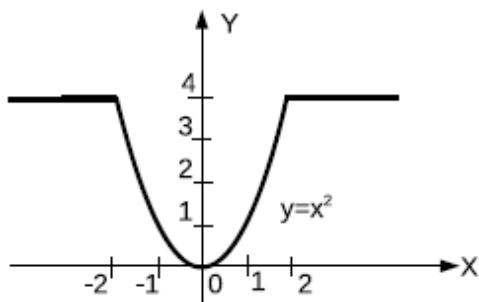


Figure 3.52: Problem 5

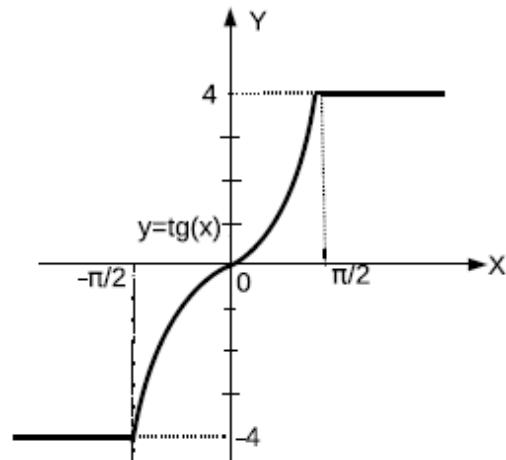


Figure 3.53: Problem 6

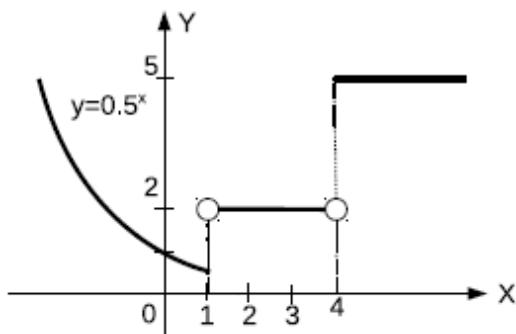


Figure 3.54: Problem 7

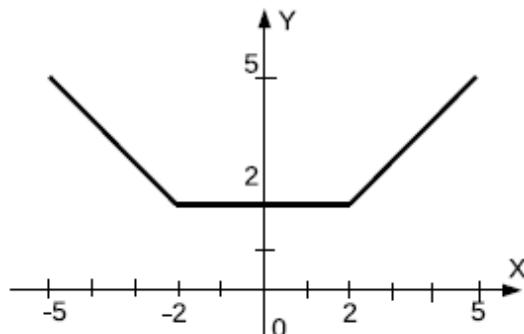


Figure 3.55: Problem 8

Problems 9 to 16. Determine if a point with coordinates  $(x, y)$ , where  $x$  and  $y$  are real numbers, lies within the hatched regions of the plane shown in Figure 3.56 to Figure 3.63.

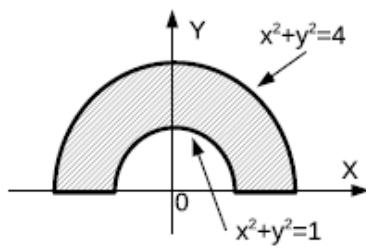


Figure 3.56: Problem 9

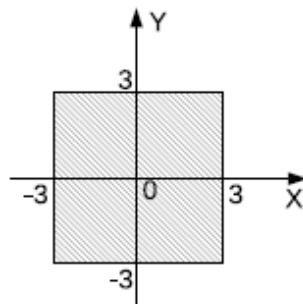


Figure 3.57: Problem 10

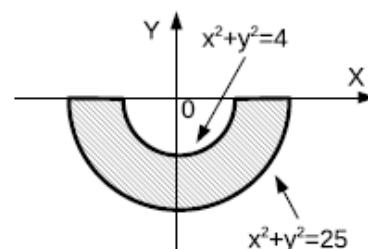


Figure 3.58: Problem 11

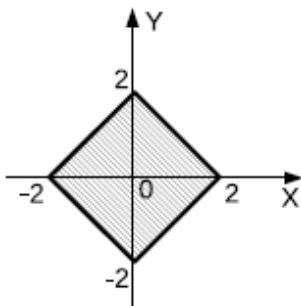


Figure 3.59: Problem 12

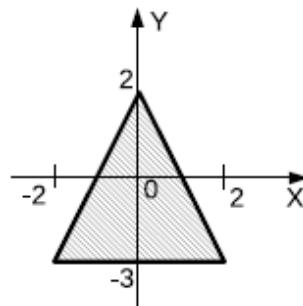


Figure 3.60: Problem 13

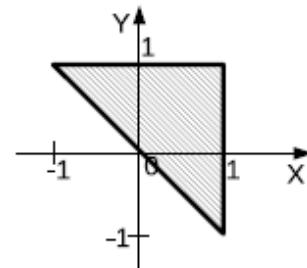


Figure 3.61: Problem 14

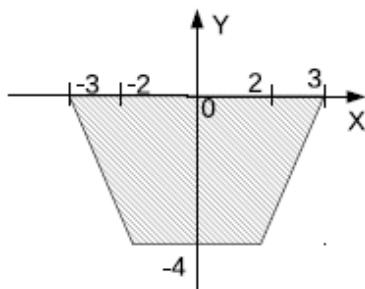


Figure 3.62: Problem 15

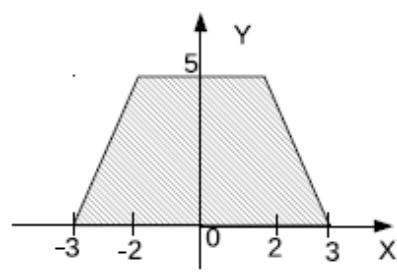


Figure 3.63: Problem 16

Solve the following problems:

- 17) Given: A circle with center at point  $O(x_0, y_0)$  and radius  $R_0$ , and point  $A(x_1, y_1)$ . Determine if  $A$  is inside a circle.
- 18) Determine if parabolas  $y = ax^2 + bx + c$  and  $y = dx^2 + mx + n$  intersect. Find the point of intersection if they intersect.
- 19) Determine if the lines  $y = ax^3 + bx^2 + cx + d$  and  $y = kx^3 + mx^2 + nx + p$  intersect. Find the point of intersection if they intersect.
- 20) Given: a circle with center at point  $O(x_0, y_0)$  and radius  $R_0$ . Find the points of intersection of the circle with the  $x$ -axis.
- 21) Given: a circle with center at point  $O(x_0, y_0)$  and radius  $R_0$ . Find the points of intersection of the circle with the  $y$ -axis.
- 22) Determine if the lines  $y = bx^2 + cx + d$  and  $y = kx + m$  intersect. Find the points of intersection if they intersect.
- 23) Given: a circle with center at point  $O(0,0)$  and radius  $R_0$ , and a straight line  $y = ax + b$ . Determine if the line and circle intersect. Find the points of intersection if they intersect.
- 24) Find the points of intersection of the line  $ax^2 + bx + c$  with the  $x$ -axis.
- 25) Determine if lines  $y = ax^4 + bx^3 + cx^2 + dx + f$  and  $y = bx^3 + mx^2 + dx + p$  intersect. Find the points of intersection if they intersect.

### 3.6.2 Exercises (Looping)

- 1) Calculate the sum of natural odd numbers not exceeding  $N$ .
- 2) Calculate the product of natural even numbers not exceeding  $N$ .
- 3) Determine the number of natural numbers that are multiples of three and

do not exceed N.

- 4) Given: the number  $n$ . Determine the value of the expression:

$$P = \frac{n!}{\sum_{i=1}^n i}$$

- 5) A sequence of nonzero numbers ends with 0. Calculate the sum of the positive terms of the sequence.
- 6) A sequence of nonzero numbers ends with 0. Determine how many times the sequence changes sign.
- 7) Calculate the sum of the negative terms in a sequence of arbitrary numbers.
- 8) Determine the number of zeros in a sequence of N arbitrary numbers.
- 9) A sequence of nonzero numbers ends with 0. Determine the largest number in the sequence.
- 10) Find all prime numbers not exceeding a positive integer P.
- 11) Determine if the number L is perfect. A perfect number L is a positive integer that is equal to the sum of all its positive divisors, but not the number itself. For example,  $6 = 1 + 2 + 3$  or  $28 = 1 + 2 + 4 + 7 + 14$ .
- 12) A sequence of nonzero numbers ends with 0. Determine the average value of the terms in the sequence.
- 13) Find the smallest positive number in a sequence of arbitrary numbers N.
- 14) Find the average value of the positive numbers in a sequence of arbitrary numbers N.
- 15) Calculate the percentage of positive and negative numbers in a sequence of nonzero numbers that ends with 0.
- 16) Calculate the percentage of positive, negative and zero terms in a sequence of arbitrary numbers N.
- 17) Find the number of perfect numbers (see Problem 11) in a sequence of positive integers that ends with 0.
- 18) Calculate the difference between the smallest and largest terms of a sequence of arbitrary numbers N.
- 19) Find all perfect numbers (see Problem 11) not exceeding natural number P.
- 20) Find The smallest number among the even terms of a sequence of N positive integers.
- 21) A sequence of positive numbers ends with 0. Determine if the terms in this sequence alternates in sign.

- 22) If P is a prime number, calculate P!.
- 23) Find the largest number in a sequence of N arbitrary numbers. Find the number of such numbers if there are several of them.
- 24) Find the number of even and odd divisors in the number P.
- 25) Determine if a sequence of N arbitrary numbers is strictly increasing (that is, each term is greater than the previous).

**Endnotes:**

---

- 1 "Algorithm" comes from *algorithmi*, *algorismus* - a Latin form of the name of the Persian mathematician Al-Khwarizmi.
- 2 A Boolean expression can take only one of two values: **true** or **false**.
- 3 For the problems in this chapter, little attention will be given to the program interface, so that the reader can focus on understanding algorithms and how to write them in Free Pascal.
- 4 Complex numbers are written in the form  $a+bi$ , where "a" is the real part of the complex number, "b" is the imaginary part, and "i" is  $\sqrt{-1}$ .
- 5 When calculating the values of u and v, the program checks the value of the radical expressions. If  $-q/2 \pm \sqrt{D} > 0$ , then  $u = (-q/2 + \sqrt{D})^{1/3}$  and  $v = (-q/2 - \sqrt{D})^{1/3}$ . If  $-q/2 \pm \sqrt{D} < 0$ , then  $u = (-q/2 + \sqrt{D})^{1/3}$  and  $v = (-q/2 - \sqrt{D})^{1/3}$ . Accordingly, if the radical expressions are zero then u and v will become zero.
- 6 See Chapter 1 for details on creating a project.
- 7 While building the form, make Label5 invisible by unchecking the Visible property (Label5.Visible := false) in the Object Inspector.
- 8 Division by zero could occur when calculating the roots.
- 9 The concept of iteration in mathematics and programming is somewhat different. In mathematics, iteration means the repetition of a mathematical operation using the result of a previous similar operation. In programming, iteration is a data processing technique in which actions are repeated several times, without having to call themselves (<http://en.wikipedia.org/wiki/Iteration>).
- 10 If the step width of the control variable is equal to one, it may be omitted in flowcharts.
- 11 Vladimir Andreevich Reutsky, who taught programming to the authors, used to say, "The use of **goto** is a sign of bad taste".
- 12 The algorithm for finding the number of digits in a number was discussed in the previous problem.
- 13 To find the smallest term in a sequence, assume that the first term is the smallest, save it in a variable Min, and then scan the subsequent terms for a smaller value. If found, save it in Min.
- 14 (Blank)
- 15 The algorithm for finding the maximum (minimum) terms in a sequence was explained in detail in Problem 3.18.

## Chapter 4. Subroutines

In programming practice, situations often arise when the same group of statements that attains a specific goal needs to be repeated in several places in the program. The concept of a subroutine was introduced to avoid wasting time when such repetition was needed.

A *subroutine* is a named, logically complete group of statements, which can be called and executed any number of times from different places in a program. There are two kinds of subroutines in Free Pascal: *procedures* and *functions*. The *main difference between a procedure and a function* is that a function always returns a specific type of value after execution of the statements making up its body. Thus, a function can be used directly in expressions, along with variables and constants.

### 4.1 Local and Global Variables

So, a subroutine is a named set of declarations and statements, fulfilling a specific task. Information passed to the subroutine for processing is called *parameters*. Before calling, or activating, a subroutine it must be declared in the declaration section of the calling program. A *subroutine declaration* includes its header and body. The *header* declares the name of the subroutine, and its parameters, if any, in parentheses. A function is also required to declare the type of the result it returns. The *body of the subroutine* follows the header and consists of variable declarations and executable statements.

A subroutine can contain declarations of other subroutines. Constants, variables and data types can be declared in the main program, and in subroutines, to varying degrees of nesting. Variables, constants and types declared in the main program before the declaring subroutines are known as *global variables*. They are accessible to all functions and procedures. Variables, constants and types declared in a subroutine are accessible only inside that subroutine and are known as *local variables*. Adhere to the following rules to correctly determine the scope of identifiers (variables):

- Each variable, constant or type must be declared before use.
- The scope of a variable, constant or type is that subroutine in which it was declared.
- The names of variables declared in a subroutine must be unique in that subroutine and cannot be the same as the name of the subroutine itself.
- Local and global variables with the same name are different variables. Accessing such a variable in a subroutine is interpreted as accessing the

- local variable (the global variable is not accessible).
- The variables that are accessible to a subroutine are those that are declared in it and before its declaration.

## 4.2 Formal and Actual Parameters

The exchange of information between called and calling functions is implemented using the parameter passing mechanism. Variables specified in the list in the function header are called *formal parameters*, or simply parameters of the subroutine. All variables in this list can be used inside the subroutine. The variables in the subroutine's call statement are *actual parameters*, or *arguments*.

The parameter passing mechanism ensures that data is exchanged correctly between actual and formal parameters, which allows the subroutine to process different data. There is a unique one-to-one relationship between the actual parameters in the calling statement and formal parameters in the subroutine header. *The number, types and position of the actual and formal parameters must match.*

Parameters are passed as follows. The expressions producing the data in the actual parameters are executed. Memory is allocated for the formal parameters, based on their types. The types of passed data are checked and a warning message is issued if they do not match the types of the corresponding formal parameters. If the number and types of formal and actual parameters are the same, then the mechanism for transferring data between actual and formal parameters will begin its work.

The formal parameters of a procedure can be divided into two classes: *value parameters* and *reference parameters*.

When passing data to a subroutine using *value parameters*, only the values of the actual parameters are passed, and the subroutine has no access to the actual parameters themselves.

When passing data, *variable parameters* replace<sup>1</sup> formal parameters, and therefore the subroutine has access to the data in the actual parameters. Any change to variable parameters in the subroutine will change the corresponding formal parameters. Consequently, *input data should be passed using value parameters, and data that should be modified by the subroutine should be passed using variable parameters.*

We will now move from a general theoretical discussion to the practical use of subroutines for solving problems. The study of subroutines will begin with procedures.

## 4.3 Procedures

A procedure declaration has the following form:

```
procedure procedure_name (formal_parameter_list);
label
  list_of_labels;
const
  list_of_constants;
type
  list_of_types;
var
  variable_list;
begin
  // The procedure body.
End;
```

The declaration begins with the *procedure header*, where **procedure** is a keyword, **procedure\_name** is any valid Free Pascal identifier, **formal\_parameter\_list** names of formal parameters and their types, separated by semicolons. Consider examples of procedure headers with value parameters:

**procedure** name\_1 (r: **real**; i: **integer**; c: **char**);

Parameters of the same type can be separated by commas:

**procedure** name\_2 (a, b: **real**; i, j, k: **integer**);

The list of formal parameters is optional and may be omitted:

**procedure** name\_3;

Reference parameters in the header must be preceded by the keyword **var**:

```
procedure name_4 (x, y: real; var z: real);
  // x, y are value parameters, and
  // z is a reference parameter.
```

The *procedure body* follows the header. It consists of a declaration section<sup>2</sup> (constants, types, variables, procedures and functions used in the procedure) and statements for implementing the procedure algorithm.

To call a procedure, use the calling convention:

```
procedure_name(list_actual_parameters);
```

The actual parameters in the calling statement list are separated by commas:

```
a := 5.3; k := 2; s := 'a';
name_1(a, k, s);
```

If the procedure has no formal parameters, then the parentheses may be omitted in the calling statement, like this:

```
name_3;
```

EXAMPLE 4.1. Find the real roots of a quadratic equation  $ax^2 + bx + c = 0$ .

The algorithm for solving this problem was described in detail in Problem 3.3 (Figure 3.14). However, that problem did not consider the input of incorrect coefficients. For example, if the user enters  $a = 0$ , then the quadratic equation will become linear. The algorithm for solving a linear equation is trivial:  $x = -c / b$  if  $b \neq 0$ . To avoid further complicating the already complicated algorithm for solving a quadratic equation, write it using procedures.

The commented code is shown below:

```
// Procedure for calculating real roots of a quadratic equation.
procedure roots(a, b, c: real; var x1,x2: real; var ok: boolean);
{ Input parameters: a, b, c - quadratic coefficients;
  Output parameters: x1, x2 - roots of the equation.
  ok - is false if there are no roots but true otherwise. }
var
  d: real;
begin
  d := b*b - 4*a*c;
  if d < 0 then
    ok := false
  else
    begin
      ok := true;
      x1 := (- b + sqrt (d)) / (2 * a);
      x2 := (- b - sqrt (d)) / (2 * a);
    end
end;           // End of subroutine

// Main program
var
  a_, b_, c_, x1_, x2_, x_: real; ok_: boolean;
begin
  write('a_ := '); readln(a_);
  write('b_ := '); readln(b_);
  write('c_ := '); readln(c_);
  if a_ = 0 then // If a = 0, then equation is not quadratic.
  begin
```

```

// Solve the linear equation bx + c = 0.
if b_ <> 0 then
begin
  x_ := - c_ / b_;
  writeln('x = ', x_);
end
else
  writeln('No roots');
end
else
// Solve of quadratic equation ax2+bx+c = 0.
begin
  // Call procedure.
  roots(a_, b_, c_, x1_, x2_, ok_);
  if ok_ then
    writeln('x1 = ', x1_:7:4, '   x2 = ', x2_:7:4)
  else
    writeln('No roots');
end;
end.

```

EXAMPLE 4.2. Find the largest and smallest digits in each term of a sequence of N positive integers.

To solve the problem, create a procedure max\_min that will produce two values: the minimum and maximum digits in a given number. The code is shown below:

{ The procedure returns max, the largest and min, the smallest digit in M. In the parameter list, M is a value parameter (input parameter), and max and min variable parameters (output parameters). }

```

procedure max_min (M: longint; var max: byte; var min: byte);
var
  i: byte;
begin
  i := 1;
  while M div 10 > 0 do
  begin
    if i = 1 then
    begin
      max := M mod 10;           // Assume the first digit is
      min := M mod 10;           // largest or
      i := i + 1;
    end;
    // Search for a digit greater than max or less than min.
  end;

```

```

if M mod 10 > max then max := M mod 10;
if M mod 10 < min then min := M mod 10;
M := M div 10;
end;
end;

// Main program
var
  X: longint;
  N, i, X_max, X_min: byte;
begin
  // The number of terms in the sequence.
  write('N = '); readln(N);
  for i := 1 to N do
  begin
    write('X = '); readln(X);      // term in sequence
    if X > 0 then                // If the term is positive, then
    begin
      max_min(X, X_max, X_min);      // call procedure
      // Print results.
      writeln('max = ', X_max, '  min = ', X_min);
    end;
  end;
end.

```

## 4.4 Functions

The *function declaration* also consists of a header and a body:

```

function function_name (formal_parameter_list):result_type;
label
  list_labels;
const
  list_constants;
type
  list_types;
var
  variable_list;
begin
  // Function body.
end;

```

The *function header* contains: the **function** keyword, a valid Free Pascal identifier - the `function_name`, formal parameter names and their types separated by semicolons - the `formal_parameter_list`, the function's return type (functions can return scalar **integer**, **real**, **Boolean**, symbolic values or reference types).

Examples of function declarations are shown below:

---

```
function fun_1(x: real): real;
function fun_2(a, b: integer): real;
```

The *body of a function* consists of the declaration section<sup>3</sup> (constants, types, variables, procedures and functions used in the procedure) and statements for implementing its algorithm. *The function body must always contain at least one statement that assigns a value to the function name.* In Delphi or objfpc mode it is allowed to assign the function result to the predefined variable *Result*. This helps if the function is renamed.

For example:

```
function fun_2(a, b: integer): real;
begin
    fun_2 := (a + b) / 2;
end;
```

*The function is called* by name with a list of actual parameters:

```
function_name(list_of_actual_parameters);
```

For example:

```
y := fun_1(1.28);
z := fun_1(1.28)/2 + fun_2(3, 8);
```

**EXAMPLE 4.3.** Find the arithmetic mean of the perfect numbers and geometric mean of the prime numbers in a sequence of N integers.

Recall that an integer is a prime number if it is evenly divisible only by itself and one. The algorithm for finding prime numbers was described in Problem 3.14 (Figure 3.33). This problem also includes perfect numbers. A perfect number is equal to the sum of all its divisors that are less than the number itself. An algorithm to find the divisors of a number was discussed in detail in Problem 3.13 (Figure 3.32).

To solve the problem, two functions are needed:

- **prime** returns true if a number is prime, or false otherwise;
- **perfect** returns true if a number is perfect, or false otherwise.

The commented code is shown below:

```
// Function to identify a prime number.
```

```
function prime(N: word): boolean;
var
  i: word;
begin
  prime := true;
  for i := 2 to N div 2 do
    if N mod i = 0 then
      begin
        prime := false;
        break;
      end;
  end;

// Function to identify a perfect number.
function perfect(N: word): boolean;
var
  i: word;
  S: word;
begin
  perfect := false;
  S := 0;
  for i := 1 to N div 2 do
    if N mod i = 0 then S := S + i;
    if S = N then perfect := true;
  end;

// Main program
var
  X: word;
  K, kol_p, kol_s, i: byte;
  Sum, Pro: real;
begin
  // Enter the number of terms in the sequence.
  write('K = '); readln(K);
  Sum := 0;                                // Accumulate sums.
  Pro := 1;                                 // Save product.
  kol_p := 0;                               // Prime number counter.
  kol_s := 0;                               // Perfect number counter.
  for i := 1 to K do
    begin
      // Enter a term in the sequence.
      write('X = '); readln(X);
      if prime(X) then // If the number is prime,
        begin
          Pro := Pro*X;           // Multiply
          kol_p := kol_p + 1;    // increment prime counter.
        end;
      if perfect(X) then // If the number is perfect,
        begin
          Sum := Sum + X;       // Accumulate sum
          kol_s := kol_s + 1;    // increment perfect counter.
        end;
    end;
end.
```

```

    end;
end;
if kol_s <> 0 then           // If perfect numbers were found,
begin
    Sum := Sum / kol_s;      // Arithmetic mean.
    writeln('The arithmetic mean of perfect numbers: ',Sum:5:2);
end
else                         // otherwise display message:
    writeln('There are no perfect numbers in the sequence.');
if kol_p <> 0 then           // If prime numbers were found,
begin
    Pro := exp(1 / kol_p * ln(Pro));    // Geometric mean.
    writeln('Geometric mean of prime numbers: ', Pro:5:2);
end
else                         // otherwise display message:
    writeln('There are no prime numbers in the sequence');
end.

```

**EXAMPLE 4.4.** Determine if a sequence of integers ending with 0 contains at least one numeric palindrome.

A numeric palindrome is a number that is symmetrical about its middle. For example, 123454321, 678876 are palindromes. To determine if a number is a palindrome, you need to compare the first and last digits, then the second and the penultimate, and so on. If at least one pair of numbers does not match, the number is not a palindrome.

To solve the problem, two functions are needed:

- **digits** finds the number of digits in a number (the algorithm was described in detail in Problem 3.16);
- **palindrome** returns true if the number passed to it is a palindrome.

The commented code is shown below:

```

// Function for calculating the number of digits in the number M.
function digits(M: longint): byte;
begin
    digits := 1;
    while M div 10 > 0 do
    begin
        digits := digits + 1;
        M := M div 10;
    end;
end;

```

```

{ This function returns true if the number M, consisting
of kol digits is a palindrome, and false, otherwise. }
function palindrome(M: longint; kol: byte): boolean;
var
  i: byte;
  j: longint;
begin
  j := 1;
// Raise the number 10 to the power kol-1.
  for i := 1 to kol - 1 do
    j := j * 10;
  palindrome := true;           // Assume number is palindrome.
  for i := 1 to kol div 2 do
begin
  // Identify most significant bit M div j (first digit).
  // Identify least significant bit of M mod 10 (last digit).
  // If the first and last digits do not match,
  if M div j <> M mod 10 then
begin
  palindrome := false; // number is not palindrome.
  break;               // Exit loop early.
end;
  // Change the number
  M := M - (M div j) * j;      // Remove first digit.
  M := M div 10;               // Remove last digit.
  j := j div 100;              // Decrease the bit depth.
end;
end;

// Main program.
var
  X: longint;
  ok: boolean;
begin
  write('X = '); readln(X); // Enter a term in the sequence.
  ok := false;             // Assume no palindromes.
  while X <> 0 do          // While X not zero,
begin
  if palindrome(X, digits(X)) then
begin
  ok := true;             // Found palindrome
  break;                  // Exit loop early.
end;
  // Enter the next term in the sequence.
  write('X = '); readln(X);
end;
if ok then
  writeln('sequence ','contains a palindrome.')
else
  writeln('sequence does not contain a palindrome.');
end.

```

## 4.5 Solving Problems using Subroutines

In this section, problems with simple algorithms are considered, while paying attention to their interface in Lazarus.

EXAMPLE 4.5. Create a program to automate the conversion of angular degrees to radians and vice versa, depending on user input. If the user enters the angle in radians, the answer will be in degrees and, if the angle is entered in degrees, the answer will be in radians.

For a mathematician, this task may not be difficult:

- To convert an angle in degrees to radians, multiply the number of degrees by  $\pi/180$ , the number of minutes by  $\pi/(180*60)$ , the number of seconds by  $\pi/(180*60*60)$ , and add the results;
- To convert an angle in radians to degrees, multiply the number of radians by  $180/\pi$ . Select the whole part from the resulting fraction to obtain degrees. Multiply the fractional part by 60 and take the whole part from the result to obtain minutes. Seconds are obtained similarly from the fractional part of minutes.

To convert an angle in degrees to radians, create the function

**function** deg2rad(deg, min, sec: **byte**): **real**;

and pass to it the integer values of degrees, minutes and seconds. The function returns the angle in radians.

To convert an angle in radians to degrees, create the procedure

**procedure** rad2deg(rad: **real**; **var** deg, min, sec: **byte**);

which takes a single input in radians and returns the angle in degrees, minutes and seconds.

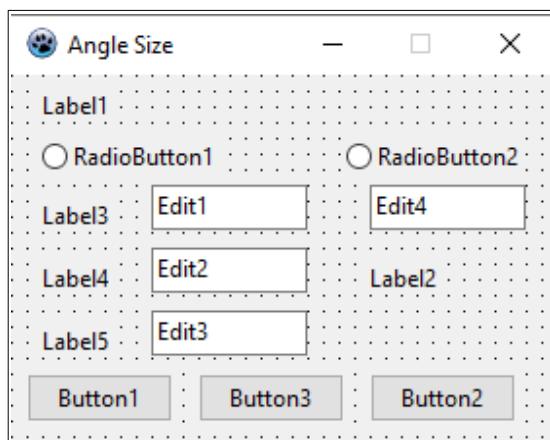
The program interface will be developed in Lazarus. Create a new project, set the form properties to those shown in Table. 4.1, and place components on the form as shown in Figure 4.1.

We are already familiar with the **Edit**, **Label** and **Button** components. The **RadioButton** component is a *switch* that is used to select one of several mutually exclusive options. At least two such components are usually placed on the form. They can have only two states, as defined by the **Checked** property. If one of the components has this property set to **true**, then the others are all set to **false**. In this

problem, we use two such components, **RadioButton1** and **RadioButton2**, to give the user a choice: if the first component is selected, degrees will be converted to radians

*Table 4.1: Form properties for Example 4.5*

Property	Value	Property Description
Caption	Angle size	Form header
Height	190	Form height
Width	280	Form width
BorderIcons.BiMaximize	false	Button to maximize window not available
BorderStyle	bsDialog	Dialog frame style, does not resize
Position	poScreenCenter	Window opens in screen center



*Figure 4.1: The form for Example 4.5*

and if the second is selected then the reverse will occur. Double clicking on the **RadioButton1** component will lead to creating the **TForm1.RadioButton1Click** procedure to handle (process) mouse click events on the radio button. The procedure code should include statements that will be executed if the user turns on or off the component.

We already know that the component properties can be changed while building the form, and directly in the program. If we double-click directly on the form but not on any component placed on it, Lazarus will automatically create the **TForm1.FormCreate** procedure, which handles events occurring when the form opens. This event is called **OnCreate** on the Object Inspector's Events tab. The **OnCreate** procedure runs just before the form is created. All actions associated with the form will be completed before the form appears on screen. All component

properties could be set before the form appears.

The buttons on the form will have the following functions:

- Button1 starts the conversion process, depending on the radio button settings;
- Button3 resets the form to its original appearance (before data input and output);
- Button2 closes the program.

The commented code is shown below:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class (TForm)
    Button1: TButton;
    Label1: TLabel;
    RadioButtonItem: TRadioButton;
    RadioButtonItem2: TRadioButton;
    Edit1: TEdit;
    Button2: TButton;
    Label2: TLabel;
    Edit2: TEdit;
    Edit3: TEdit;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit4: TEdit;
    Button3: TButton;
    procedure Button1Click (Sender: TObject);
    procedure RadioButtonItemClick (Sender: TObject);
    procedure RadioButtonItem2Click (Sender: TObject);
    procedure FormCreate (Sender: TObject);
    procedure Button2Click (Sender: TObject);
    procedure Button3Click (Sender: TObject);
  private
    {Private declarations}
  public
    {Public declarations}
  end;
  var
    Form1: TForm1;
implementation
  {$R *.dfm}

  // Click on the CALCULATE button.
```

```
procedure TForm1.Button1Click (Sender: TObject);
  function deg2rad(deg, min, sec: byte): real;
  { Convert degrees (deg, min, sec) to radians }
begin
  deg2rad := deg*pi/180 + min*pi/180/60 + sec*pi/180/60/60;
end;

procedure rad2deg(rad: real; var deg, min, sec: byte);
{ Convert radians to degrees (deg, min, sec) }
begin
  deg := trunc(rad*180/pi);
  min := trunc((rad*180/pi - deg)*60);
  sec := trunc(((rad*180/pi - deg)*60 - min)*60);
end;

var
  deg, min, sec: byte; // Angle in degrees.
  rad: real;           // Angle in radians.
  kod_g, kod_m, kod_s, kod_r: integer; // Input control.
begin
  if RadioButton1.Checked then      // If the first switch is on.
begin
  val(Edit1.Text, deg, kod_g);      // Enter degrees.
  val(Edit2.Text, min, kod_m);      // Enter minutes.
  val(Edit3.Text, sec, kod_s);      // Enter seconds.
  // If there was no error while entering, then
  if (kod_g = 0) and (kod_m = 0) and (kod_s = 0) then
begin
  Label2.Visible := true; // make the Label2 visible
  // and output results there.
  // Call the deg2rad function to convert degrees to radians.
  Label2.Caption := 'Angle value ' + chr(13)+ FloatToStrF
(deg2rad(deg, min, sec), ffFixed, 8, 6)+ ' radians';
end
  else // Otherwise, display error message when typing.
  MessageDlg ('Error while entering data!', MtWarning, [mbOk], 0);
end;
  if RadioButton2.Checked then // If the second switch is on.
begin
  val(Edit4.Text, rad, kod_r);    // Enter angle in radians.
  if (kod_r = 0) then // If there are no typing errors, then
begin
  Label2.Visible := true; // make the Label2 visible
  // and output results there.
  // Call rad2deg to convert radians to degrees.
  rad2deg(rad, deg, min, sec);
  Label2.Caption := 'Angle value ' + chr(13)+ IntToStr(deg) + 'd'
+ IntToStr(min) + 'm' + IntToStr(sec) + 's';
end
  else // Otherwise, display error message when typing.
  MessageDlg ('Error while entering data!', MtWarning, [mbOk], 0);
end;
```

```
    end;
end;

// Clicking the EXIT button (Button2).
procedure TForm1.Button2Click(Sender: TObject);
begin
  close;
end;

// Click on the CLEAR button.
procedure TForm1.Button3Click(Sender: TObject);
begin
  // Set the properties of the components to their initial state.
  Edit1.Text := '00';
  Edit2.Text := '00';
  Edit3.Text := '00';
  Edit4.Text := '00.000';
  Label1.Caption := 'Please enter a value';
  Label1.Font.Size := 10;
  Label3.Caption := 'Degrees';
  Label4.Caption := 'Minutes';
  Label5.Caption := 'Seconds';
  Button1.Caption := 'CALCULATE';
  Button2.Caption := 'EXIT';
  Button3.Caption := 'CLEAR';
  Edit4.Enabled := false;
  Label2.Visible := false;
  RadioButton1.Checked := true;
  RadioButton2.Checked := false;
end;

// Handle the form opening event.
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Setting the properties of the components.
  Edit1.Text := '00';
  Edit2.Text := '00';
  Edit3.Text := '00';
  Edit4.Text := '00.000';
  Label1.Caption := 'Please enter a value';
  Label1.Font.Size := 10;
  Label3.Caption := 'Degrees';
  Label4.Caption := 'Minutes';
  Label5.Caption := 'Seconds';
  Button1.Caption := 'CALCULATE';
  Button2.Caption := 'EXIT';
  Button3.Caption := 'CLEAR';
  Edit4.Enabled := false;
  Label2.Visible := false;
  RadioButton1.Checked := true;
  RadioButton2.Checked := false;
```

```
end;

// Handle the click event on RadioButton1.
procedure TForm1.RadioButton1Click (Sender: TObject);
begin
  if RadioButton1.Checked then
  begin
    Edit1.Enabled := true;
    Edit2.Enabled := true;
    Edit3.Enabled := true;
    Label5.Enabled := true;
    Label3.Enabled := true;
    Label4.Enabled := true;
    Edit4.Enabled := false;
  end;
end;

// Handle the click event on the RadioButton2.
procedure TForm1.RadioButton2Click (Sender: TObject);
begin
  if RadioButton2.Checked then
  begin
    Edit4.Enabled := true;
    Button1.Enabled := true;
    Edit1.Enabled := false;
    Edit2.Enabled := false;
    Edit3.Enabled := false;
    Label3.Enabled := false;
    Label4.Enabled := false;
    Label5.Enabled := false;
  end;
end;

end.
```

Program output screens are shown in Figure 4.2 and Figure 4.3, below.

Figure 4.2: Degrees to radians

Figure 4.3: Radians to degrees

**Problem 4.6.** Write a program to solve the equations:

- linear:  $ax + b = 0$ ;
- quadratic:  $ax^2 + bx + c = 0$ ;
- cubic:  $ax^3 + bx^2 + cx + d$ .

The solution of the linear equation is trivial:  $x = -b/a$ . The algorithms for solving the quadratic and cubic equations were considered in detail in Examples 3.4 and 3.5.

Create a new Lazarus project and configure the form properties as shown in Table 4.1, except for the Caption property, which will be assigned the value Solving Equations. Place the components on the form as shown in Figure 4.4.

Note that the form contains unfamiliar components. These include a CheckBox and a RadioGroup, which is group of radio buttons.

The CheckBox component allows the user to express a yes or no decision. Its Checked property (**true**, **false**) determines if the checkbox is checked or not. A form may contain several such components, but their individual state does not depend on the state of the others.

The *RadioGroup* component combines several radio buttons. Each radio button placed in it is added to the Items list and may be accessed via a number in the ItemIndex property. This component is empty when placed on a form. To add a radio button to it, Select it, go to the Object Inspector and select the Items property list editor. The lines typed in editor are used as hint labels to the right of the radio buttons, and their number determines the number of radio buttons in the group. In our case the list editor window will look like Figure 4.5.

After a *RadioGroup* is created, its ItemIndex property defaults to -1. This indicates

that are no radio buttons in the group. Set the ItemIndex property to the number of one of the items in the list, which is zero-based, either while building the form or programmatically, so one of the radio buttons will be checked when the component appears on screen.

The reader is already familiar with the other components on the form. Figure 4.6 to 4.8 shows the program in action. The user may choose the type of the equation to be solved, enter its coefficients and indicate which roots, real or complex (if possible), are required.

The commented code for the unit is shown below:

```
unit Unit1;
interface
uses
  Classes, SysUtils, Forms, Controls, Graphics, Dialogs, StdCtrls,
  ExtCtrls;
```

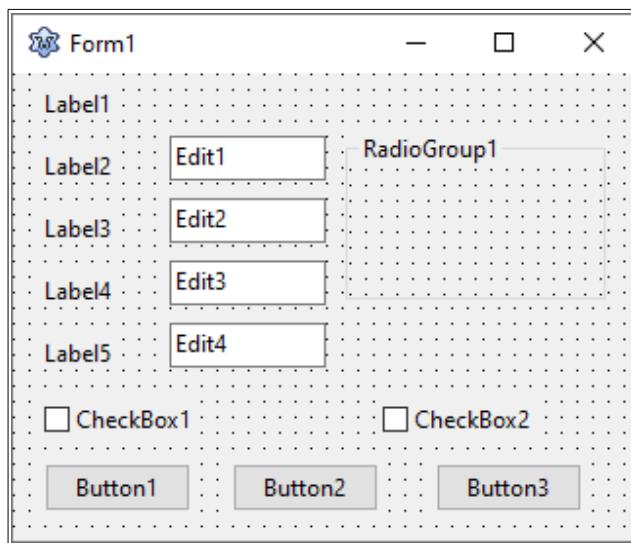


Figure 4.4: The form for Example 4.6

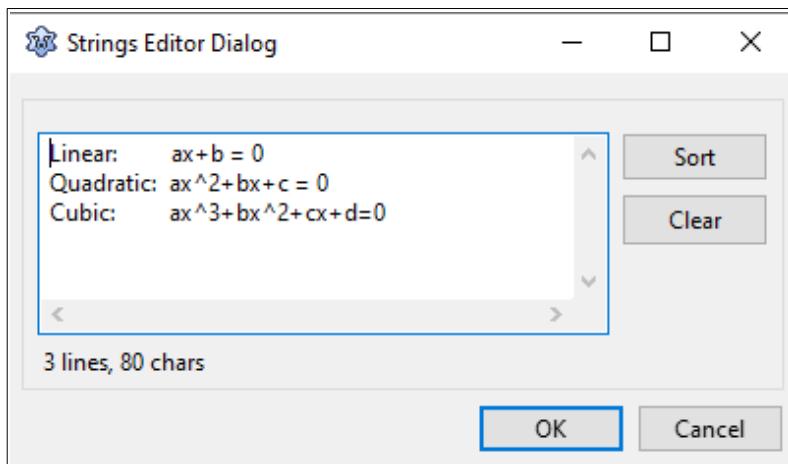


Figure 4.5: List editor window

```
type
  TForm1 = class (TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    Button1: TButton;
    Button2: TButton;
    RadioGroup1: TRadioGroup;
```

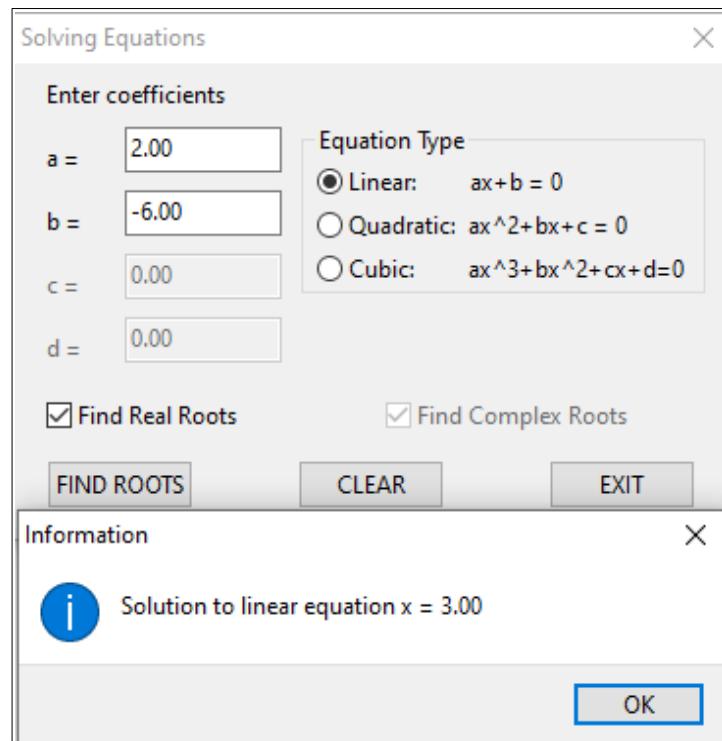


Figure 4.6: Solving a linear equation

```

Button3: TButton;
procedure FormCreate(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure RadioGroup1Click(Sender: TObject);
private
  {Private declarations}
public
  {Public declarations}
end;
var
  Form1: TForm1;
implementation
{$R * .dfm}

// Click on the FIND ROOTS button.
procedure TForm1.Button1Click(Sender: TObject);
procedure roots_1 (a, b: real; var x_: string);
{Linear equation.}
var
  x: real;
begin
  x := - b/a;
  x_ := FloatToStrF(x, ffFixed, 5, 2);
end;

```

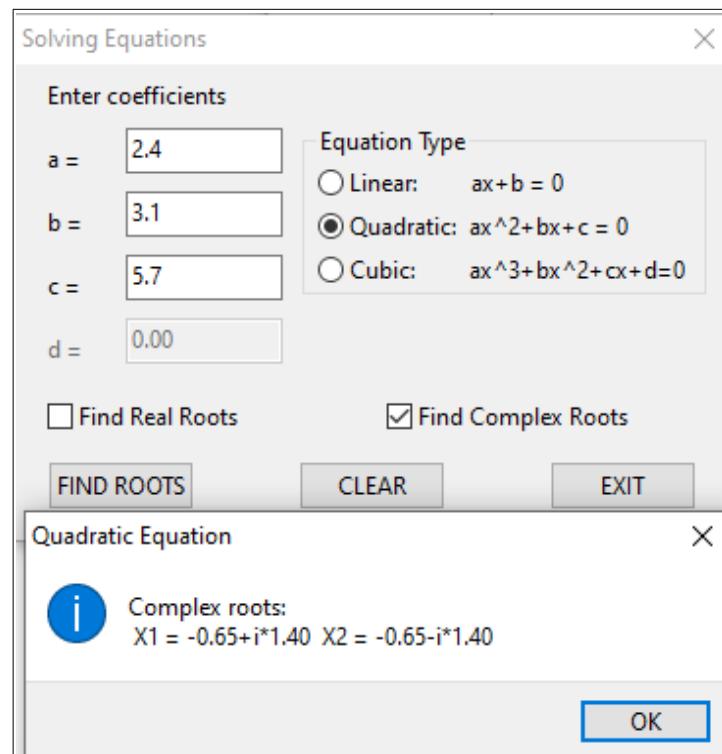


Figure 4.7: Solving a quadratic equation

```

procedure roots_2(a, b, c: real; var x1_,x2_: string; var ok:
byte);
{ Quadratic equation }
var
  d, x1, x2: real;
begin
  d := b*b - 4*a*c;
  if d < 0 then
    begin
      x1 := -b/(2*a);
      x2 := sqrt(abs(d))/(2*a);
      x1_ := FloatToStrF(x1,ffFixed,5,2)+ '+i*' + FloatToStrF
(x2,ffFixed,5,2);
      x2_ := FloatToStrF(x1,ffFixed,5,2)+ '-i*' + FloatToStrF
(x2,ffFixed,5,2);
      ok := 2;           // Complex roots.
    end
  else
    begin
      x1 := (-b + sqrt(d))/2/a;
      x2 := (-b - sqrt(d))/(2*a);
      x1_ := FloatToStrF(x1, ffFixed, 5, 2);
      x2_ := FloatToStrF(x2, ffFixed, 5, 2);
      ok := 1;           // Real roots.
    end
end.

```

```
    end
end;
```

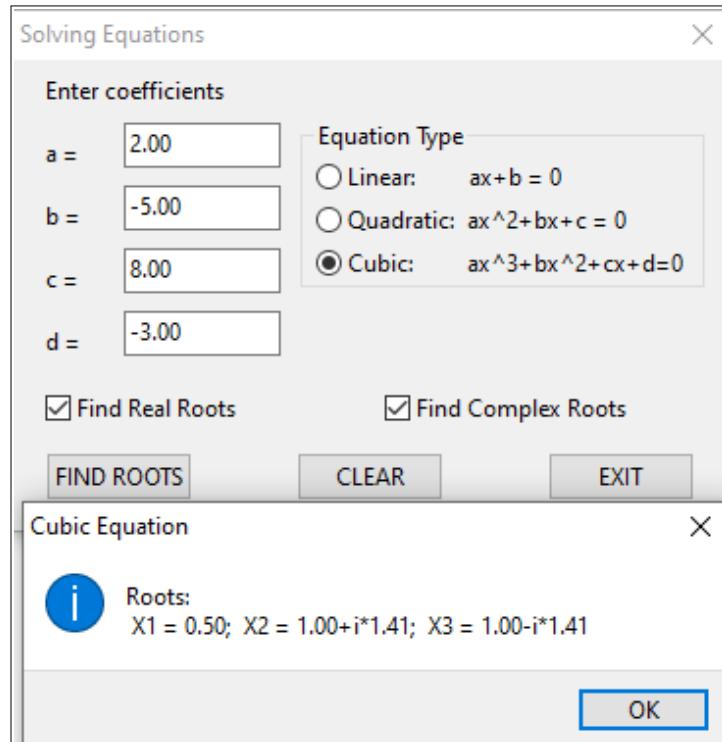


Figure 4.8: Solving a cubic equation

```
procedure roots_3 (a, b, c, d: real; var x1_, x2_, x3_: string;
var ok: byte);
{ Cubic equation }
var
  r, s, t, p, q, ro, fi, u, v, x1, x2, x3, h, g: real;
begin
  r := b/a;
  s := c/a;
  t := d/a; p := (3*s - r*r)/3; q := 2*r*r*r/27 - r*s/3 + t;
  d := (p/3)*sqr(p/3) + sqr(q/2);
  if d < 0 then
  begin
    ro := sqrt(-p*p*p/27);
    fi := -q/(2*ro);
    fi := pi/2 - arctan(fi/sqrt(1-fi*fi));
    // Calculate the real roots of the equation.
    x1 := 2*exp(1/3*ln(ro))*cos(fi/3) - r/3;
    x2 := 2*exp(1/3*ln(ro))*cos(fi/3 + 2*pi/3) - r/3;
    x3 := 2*exp(1/3*ln(ro))*cos(fi/3 + 4*pi/3) - r/3;
    x1_ := FloatToStrF (x1, ffFixed, 5, 2);
    x2_ := FloatToStrF (x2, ffFixed, 5, 2);
    x3_ := FloatToStrF (x3, ffFixed, 5, 2);
  end;
end;
```

```

ok := 1;                                // Real roots.
end
else
begin
  if -q/2 + sqrt(d) > 0 then
    u := exp(1/3*ln(-q/2 + sqrt(d)))
  else
    if -q/2 + sqrt(d) < 0 then
      u := -exp(1/3*ln(abs(-q/2 + sqrt(d))))
    else u := 0;
  if -q/2 - sqrt(d) > 0 then
    v := exp(1/3*ln(-q/2 - sqrt(d)))
  else
    if -q/2 - sqrt(d) < 0 then
      v := -exp(1/3*ln(abs(-q/2 - sqrt(d))))
    else v := 0;
  x1 := u + v - r/3;        // Calculate the real root.
// Compute complex roots.
h := -(u + v)/2 - r/3;
g := (u - v)/2*sqrt(3);
x1_ := FloatToStrF(x1,ffFixed,5,2);
x2_ := FloatToStrF(h, ffFixed,5,2) + 'i*' + FloatToStrF (g,
ffFixed, 5, 2);
x3_ := FloatToStrF (h, ffFixed, 5, 2)+ '-i*' + FloatToStrF (g,
ffFixed, 5, 2);
ok := 2;          // One real and two complex roots.
end
end;

// Handle the OnClick event for the FIND ROOTS button.
var
  a_, b_, c_, d_: real;
  kod_a, kod_b, kod_c, kod_d: integer;
  _x, _x1, _x2, _x3: string; _ok: byte;
begin
  case RadioGroup1.ItemIndex of
  0:                      // First radio button selected.
  begin
    // Input of initial data.
    val(Edit1.Text, a_, kod_a);
    val(Edit2.Text, b_, kod_b);
    // Input was successful.
    if (kod_a = 0) and (kod_b = 0) then
    begin
      if a_ <> 0 then           // First coefficient is not zero.
      begin
        // Solve linear equation.
        roots_1 (a_, b_, _x);
        // Display result.
        MessageDlg ('Solution to linear equation x = ' + _x,
mtInformation, [mbOk], 0);
      end;
    end;
  end;
end;

```

```

end
else                                // First coefficient is zero,
// display the corresponding message.
MessageDlg ('No roots!', mtInformation, [mbOk], 0);
end
else                                // Incorrect data entry.
MessageDlg ('Error while entering coefficients!', mtInformation,
[mbOk], 0);
end;
1:                               // Second radio button selected.
begin
// Input data.
val(Edit1.Text, a_, kod_a);
val(Edit2.Text, b_, kod_b);
val(Edit3.Text, c_, kod_c);
// Input was successful.
if (kod_a = 0) and (kod_b = 0) and (kod_c = 0) then
begin
  if a_ <> 0 then          // First coefficient not zero.
begin
  // Solve quadratic equation.
  roots_2(a_, b_, c_, _x1, _x2, _ok);
  // _ok contains information about the type of roots:
  // 1 - real, 2 - complex.
  // Both checkboxes are not checked.
  if (CheckBox1.Checked = false) and (CheckBox2.Checked = false)
then
  MessageDlg ('Select root type', mtInformation, [mbOk], 0);
  // Both checkboxes are checked.
  if CheckBox1.Checked and CheckBox2.Checked then
    MessageDlg ('Quadratic equation solution ' + chr(13) + ' X1
= ' + _x1 + ' X2 =' + _x2, mtInformation, [mbOk], 0);
    // First checkbox is checked, real roots required.
    if CheckBox1.Checked and (CheckBox2.Checked = false) and (_ok
= 1) then
      MessageDlg ('Real roots ','of quadratic equation' + chr(13) +
' X1 = ' + _x1 + ' X2 = ' + _x2, mtInformation, [mbOk], 0);
      // Second checkbox checked, but roots are real
      if (CheckBox1.Checked = false) and CheckBox2.Checked and (_ok
= 1) then
        MessageDlg ('Equation has only ','real roots.', mtInformation,
[mbOk], 0);
        // First checkbox checked, but roots are complex.
        if CheckBox1.Checked and (CheckBox2.Checked = false) and (_ok
= 2) then
          MessageDlg ('Equation has ','no real roots', mtInformation,
[mbOk], 0);
          // Second checkbox is checked, the roots are complex.
          if (CheckBox1.Checked = false) and CheckBox2.Checked and (_ok
= 2) then
            MessageDlg ('Complex roots of quadratic ','equation' + chr(13)

```

```
+ ' X1 = ' + _x1 + ' X2 = ' + _x2, mtInformation, [mbOk], 0);
    end
    else           // First coefficient is zero.
        MessageDlg ('The first coefficient should not be ','zero!', ,
mtInformation, [mbOk], 0);
    end           // Incorrect data entry.
    else
        MessageDlg ('Error while entering coefficients!', mtInformation,
[mbOk], 0);
end;

2:                      // Third radio button selected.
begin
    // Input data.
    val(Edit1.Text, a_, kod_a);
    val(Edit2.Text, b_, kod_b);
    val(Edit3.Text, c_, kod_c);
    val(Edit4.Text, d_, kod_d);
    // Input was successful.
    if (kod_a = 0) and (kod_b = 0) and (kod_c = 0) and (kod_d = 0)
then
begin
    if a_ <> 0 then           // First coefficient not zero.
begin
    // Solve cubic equation.
    // _ok contains information about the type of roots:
    // 1 - valid, 2 - one real and two complex.
    roots_3 (a_, b_, c_, d_, _x1, _x2, _x3, _ok);
    // Both checkboxes are not checked.
    if (CheckBox1.Checked = false) and (CheckBox2.Checked = false)
then
            MessageDlg ('Select required root type', mtInformation,
[mbOk], 0);
    // Both checkboxes are checked.
    if CheckBox1.Checked and CheckBox2.Checked then
        MessageDlg ('Roots of cubic ','equations' + chr(13) + ' X1 =
' + _x1 + ' X2 = ' + _x2 + ' X3 = ' + _x3, mtInformation, [mbOk], 0);
    // First checkbox checked. Roots are real.
    if CheckBox1.Checked and (CheckBox2.Checked = false) and (_ok =
1) then
        MessageDlg ('Real roots of ','cubic equation' + chr(13) + ' X1
= ' + _x1 + ' X2 = ' + _x2 + ' X3 = ' + _x3, mtInformation, [mbOk],
0);
    // First checkbox is checked, roots are complex.
    if CheckBox1.Checked and (CheckBox2.Checked = false) and (_ok =
2) then
        MessageDlg ('Real roots of ','cubic equation' + chr(13) + ' X1
= ' + _x1, mtInformation, [mbOk], 0);
    // Second checkbox checked, the roots are complex.
    if (CheckBox1.Checked = false) and CheckBox2.Checked and (_ok =
2) then
```

```
    MessageDlg ('Complex roots of ','cubic equations' + chr(13) +
' X1 =' + _x2 + ' X2 =' + _x3, mtInformation, [mbOk], 0);
    // Second checkbox is checked, but roots are real.
    if (CheckBox1.Checked = false) and CheckBox2.Checked and (_ok =
1) then
    MessageDlg ('Equation has only ','real roots.', mtInformation,
[mbOk], 0);
    end
    else
    MessageDlg ('The first coefficient should not be ','zero! ',
mtInformation, [mbOk], 0);
    end
    else                                // Incorrect data entry.
    MessageDlg ('Error while entering coefficients!', mtInformation,
[mbOk], 0);
    end;
end;
end;

// Click on the CLEAR button.
procedure TForm1.Button2Click(Sender: TObject);
begin
Label1.Caption := 'Enter coefficients';
Label2.Caption := 'a = ';
Label3.Caption := 'b = ';
Label4.Caption := 'c = ';
Label5.Caption := 'd = ';
Edit1.Text := '0.00';
Edit2.Text := '0.00';
Edit3.Text := '0.00';
Edit4.Text := '0.00';
Button1.Caption := 'FIND ROOTS';
Button2.Caption := 'CLEAR';
Button3.Caption := 'EXIT';
CheckBox1.Caption := 'Find Real Roots';
CheckBox2.Caption := 'Find Complex Roots';
CheckBox1.Checked := true;
Label4.Enabled := false;
Label5.Enabled := false;
Edit3.Enabled := false;
Edit4.Enabled := false;
CheckBox2.Enabled := false;
RadioGroup1.ItemIndex := 0;
end;

// Clicking the EXIT button.
procedure TForm1.Button3Click(Sender: TObject);
begin
close;
end;
```

```
// Form on create event.
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label1.Caption := 'Enter coefficients';
  Label2.Caption := 'a = ';
  Label3.Caption := 'b = ';
  Label4.Caption := 'c = ';
  Label5.Caption := 'd = ';
  Edit1.Text := '0.00';
  Edit2.Text := '0.00';
  Edit3.Text := '0.00';
  Edit4.Text := '0.00';
  Button1.Caption := 'FIND ROOTS';
  Button2.Caption := 'CLEAR';
  Button3.Caption := 'EXIT';
  CheckBox1.Caption := 'Find Real Roots';
  CheckBox2.Caption := 'Find Complex Roots';
  CheckBox1.Checked := true;
  Label4.Enabled := false;

  Label5.Enabled := false;
  Edit3.Enabled := false;
  Edit4.Enabled := false;
  CheckBox2.Enabled := false;
  RadioGroup1.ItemIndex := 0;
end;

// Select a radio button from the group.
procedure TForm1.RadioGroup1Click (Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:           // First radio button selected.
    begin
      Label2.Enabled := true;
      Label3.Enabled := true;
      Edit1.Enabled := true;
      Edit2.Enabled := true;
      Label2.Caption := 'a =';
      Label3.Caption := 'b =';
      Edit1.Text := '0.00';
      Edit2.Text := '0.00';
      Label4.Enabled := false;
      Label5.Enabled := false;
      Edit3.Enabled := false;
      Edit4.Enabled := false;
      CheckBox2.Enabled := false;
    end;
    1:           // Second radio button selected.
    begin
      Label2.Enabled := true;
      Label3.Enabled := true;
```

```
Label4.Enabled := true;
Edit1.Enabled := true;
Edit2.Enabled := true;
Edit3.Enabled := true;
Label2.Caption := 'a =';
Label3.Caption := 'b =';
Label4.Caption := 'c =';
Edit1.Text := '0.00';
Edit2.Text := '0.00';
Edit3.Text := '0.00';
Label5.Enabled := false;
Edit4.Enabled := false;
CheckBox2.Enabled := true;
end;

2:           // Third radio button selected.
begin
    Label2.Enabled := true;
    Label3.Enabled := true;
    Label4.Enabled := true;
    Label5.Enabled := true;
    Edit1.Enabled := true;
    Edit2.Enabled := true;
    Edit3.Enabled := true;
    Edit4.Enabled := true;
    Label2.Caption := 'a =';
    Label3.Caption := 'b =';
    Label4.Caption := 'c =';
    Label5.Caption := 'd =';
    Edit1.Text := '0.00';
    Edit2.Text := '0.00';
    Edit3.Text := '0.00';
    Edit4.Text := '0.00';
    CheckBox2.Enabled := true;
end;
end;
end;
end.
```

## 4.6 Recursive Functions

In programming, *recursion* refers to a subroutine that calls itself. Recursive functions are most often used for compact implementations of recursive algorithms. Classic recursive algorithms include raising a number to a positive integer power or calculating the factorial. On the other hand, any recursive algorithm can be implemented without recursion. The advantage of recursion is its compactness, but the disadvantage is the memory used for repeatedly calling itself and passing parameters. There is also a danger of memory overflow.

In a recursive function, it is necessary to provide some means for ending the recursive calls. Otherwise, the function will never terminate.

Let us look at examples of applying recursion.

**EXAMPLE 4.7.** Calculate the factorial of the number n.

The calculation of factorials was discussed in detail in Problem 3.10 (Figure 3.29). To solve this problem using recursion, create a factorial function, using the algorithm shown in Figure 4.9.

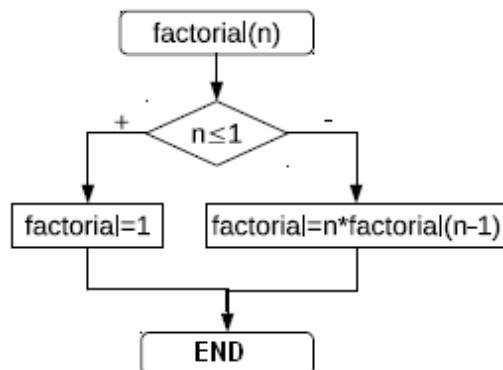


Figure 4.9: Recursive Factorial Computation Algorithm

The Free Pascal code for the recursion subroutine is shown below:

```

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

**EXAMPLE 4.8.** Calculate the n-th power of the number a, where n is an integer.

Raising a number to an integer power  $n$  is the same as multiplying this number by itself  $n$  times. This statement is true, however, only for positive values of  $n$ . If  $n$  is negative, then  $a^{-n} = 1/a^n$ . If  $n = 0$ , then  $a^0 = 1$ .

To solve the problem, create a recursive function `power`, using the algorithm shown in Figure 4.10.

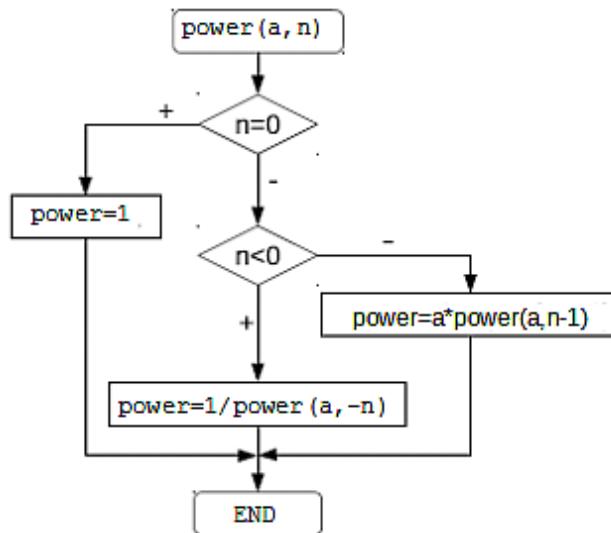


Figure 4.10: Raising a number to integer power

The code using recursion is shown below:

```

function power(a: real; n: word): real;
begin
  if n = 0 then
    power := 1
  else
    if n < 0 then
      power := 1/power(a, -n)
    else
      power := a*power(a, n-1);
end;

var
  x: real;
  k: word;
begin
  write('Base = ');
  readln(x);
  write('Power = ');
  readln(k);
  writeln(x:5:2, '^', k, ' = ', power(x, k): 5: 2);
end.
  
```

**EXAMPLE 4.9.** Calculate the n-th Fibonacci number.

If the zero term in a sequence is zero, the first - one, and each subsequent term is the sum of the two previous ones, then this a sequence of Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...).

Figure 4.11 shows the recursive Fibonacci function algorithm.

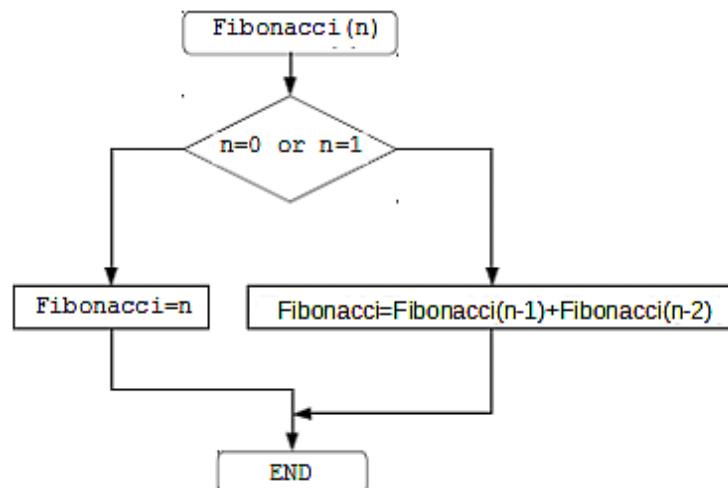


Figure 4.11: Fibonacci number recursive algorithm

The code for the subroutine is shown below:

```
function Fibonacci(n: word): word;
begin
  if (n = 0) or (n = 1) then
    Fibonacci := n
  else
    Fibonacci := Fibonacci(n - 1) + Fibonacci(n - 2);
end;

var
  x: word;
begin
  write('Enter an integer: ');
  readln(x);
  writeln('The ', x, '-th Fibonacci number is ', Fibonacci(x));
end.
```

## 4.7 Special Topics in Subroutines

Let us consider other ways of *passing parameters* to a subroutine.

### 4.7.1 Constant Parameters

A *constant parameter* is specified in the subroutine header like a value parameter, but with the reserved word **const** before it. This keyword applies up to the next semicolon. A constant parameter works like a variable parameter, but compiler will not allow the value of a constant parameter to be changed by the subroutine.

In Free Pascal, constant parameters, like variable parameters, can be untyped. The actual parameter can be a variable of any type, with the programmer being responsible for the correct use of the parameter.

### 4.7.2 Procedural Types

*Procedural types* were designed as a means of passing functions and procedures as actual parameters to other procedures and functions. To declare a procedural type, use a subroutine header without a name. There are two procedural types: *procedure parameters* and *function parameters*:

```
type
  procedure_type = procedure(formal_parameter_list);
  function_type = function(formal_parameter_list): type;
```

In addition, variables can be declared as procedural types:

```
var
  variable_name_1: procedure_type;
  variable_name_2: function_type;
```

For example:

```
type
  Fun1 = function(x, y: real): real;
  Fun2 = function: string;
  Proc1 = procedure(x, y: real; var c, z: real);
  Proc2 = procedure();
var
  F1, f2: Fun1;
  p1, p2: Proc1;
  arr: array [1..5] of Proc2;
```

Refer to procedural type variables by using their address<sup>4</sup>:

@variable\_name

The next problem will demonstrate the mechanism for passing subroutines as parameters.

EXAMPLE 4.10. Write a program that displays on screen a table of the values of functions  $f(x)$  and  $g(x)$ .

The calculation and display of the values is carried out using the OutFunc function. Its parameters are described below:

- a and b are the interval limits for changing the function arguments;
- n the number of points into which the interval (a, b) will be divided;
- ff is the function name.

```
{ Declare procedural type func. }
type
  func = function(x: real): real;
function OutFunc(a, b: real; N: word; ff: func): integer;
var
  x, y, hx: real;
begin
  dx := (b - a) / N;           { Change step of variable x. }
  x := a;
  while (x <= b) do
    begin
      y := ff(x);
      writeln('x = ', x:5:2, 'y = ', y:7:2);
      x := x + dx;
    end;
  OutFunc := 0;
end;

function f(x: real): real;
begin
  f := exp(sin(x))*cos(x);
end;

function g (x: real): real;
begin
  g := exp(cos(x))*sin(x);
end;

// Main program
begin
```

```

{ Call OutFunc with f as parameter. }
OutFunc (0, 1, 7, @f);
writeln;
{ Call MainFunc with g as parameter. }
OutFunc (0, 2, 8, @ g);
end.

```

*Function arrays* can be used as formal parameters in Free Pascal.

Let us modify Problem 4.10. The program shown below displays on screen a table of the values of several functions, using the OutFunc function. This function has the following parameters:

- interval (a, b);
- the number of points n in the interval (a, b);
- array of functions ff, to calculate table values;
- the number of functions m in the array ff.

```

type
  func = function (x: real): real;

function OutFunc (a, b: real; N: word; ff: array of func; m: word): integer;
var
  x, y, dx: real;
  i: integer;
begin
  dx := (b - a) / N;
  { Loop through all functions. }
  for i := 0 to m - 1 do
  begin
    x := a;
    while (x <= b) do
    begin
      { Calculate value of i-th function at point x. }
      y := ff[i](x);
      writeln('x =', x:5:2, ' y =', y:5:2);
      x := x + dx;
    end;
    writeln;
  end;
  OutFunc := 0;
end;

function f(x: real): real;
begin
  f := exp(sin(x))*cos(x);
end;

```

```

function g(x: real): real;
begin
  g := exp(cos(x))*sin(x);
end;

{ Declare an array of eight fff functions. }
var fff: array [1..8] of func;
begin
  fff[1] := @ f;           // Write real functions to fff array.
  fff[2] := @ g;
  {Procedure call.}
  OutFunc(0, 1, 7, fff, 2);
end.

```

Several subroutines can be combined into a unit and then used in other programs. Let us consider the process of creating a unit.

## 4.8 Developing Units

A *unit* is an autonomous software unit that includes various components: constants, variables, types, procedures and functions. We used units when developing graphical applications in Lazarus. Let us now look at how to create our own unit.

A unit has the following structure:

```

unit unit name;
interface
  // public interface
implementation
  // executable section
initialization
  // code to execute when the unit loads
finalization
  // code to execute when the program ends
end.

```

The unit header consists of the **unit** keyword followed by the unit name. The unit name must match the file name under which it is saved. For example, the *edit* unit must be saved as *edit.pas*.

The *interface section* begins with the **interface** keyword, followed by declarations of all the global objects in the unit: types, constants, variables and subroutines. These objects will be available to all units and software calling this unit.

The *implementation section* begins with the **implementation** keyword and implements the subroutines declared in the interface section. Local objects may also be declared here for local use. They will not be accessible to programs and

units calling this unit.

The *initialization section* contains statements that execute before other statements in the unit, usually to prepare the unit for its work. This section may be omitted.

The *finalizaion section* may be needed to deallocate memory allocated by code in the unit.

As an example, we will create a unit for working with complex numbers, which will contain subroutines that implement basic operations<sup>5</sup> with complex numbers:

- procedure sum(a, b: complex; **var** c: complex) will add two complex numbers a and b, and return the result via c;
- procedure diff(a, b: complex; **var** c: complex) will subtract two complex numbers a and b, and return the result via c;
- procedure mul(a, b: complex; **var** c: complex) will multiply two complex numbers a and b, and return the result via c;
- procedure divi(a, b: complex; **var** c: complex) will divide two complex numbers a and b, and return the result via c;

The unit will also contain the procedure output(a: complex), which will display a complex number on screen.

The code for the unit is shown below.

```
unit compl;
interface
  type
    complex = record
      x: real;
      y: real;
    end;
    procedure sum(a, b: complex; var c: complex);
    procedure diff(a, b: complex; var c: complex);
    procedure mul(a, b: complex; var c: complex);
    procedure divi(a, b: complex; var c: complex);
    procedure output(a: complex);
implementation

procedure sum (a, b: complex; var c: complex);
begin
  c.x := a.x + b.x;
  c.y := a.y + b.y;
end;

procedure diff(a, b: complex; var c: complex);
begin
  c.x := a.x - b.x;
  c.y := a.y - b.y;
```

```
end;

procedure mul(a, b: complex; var c: complex);
begin
  c.x := a.x * b.x - a.y * b.y;
  c.y := a.y * b.x + a.x * b.y;
end;

procedure divi(a, b: complex; var c: complex);
begin
  c.x := (a.x*b.x + a.y*b.y) / (b.x*b.x + b.y * b.y);
  c.y := (a.y*b.x - a.x*b.y) / (b.x*b.x + b.y * b.y);
end;

procedure output(a: complex);
begin
  if a.y >= 0 then writeln(a.x:1:3, '+', a.y:1:3, 'i')
  else writeln(a.x:1:3, '-', -a.y:1:3, 'i')
end;
end.
```

To compile the unit, launch Geany, enter the unit code, and save it under the name `compl.pas`. After that, execute the **Compile** command. After compiling, two files will be created in the folder where the unit is saved, one with a `.o` extension and the other with a `.ppu` extension.

To use the unit, you need the file with the `.o` extension. The compiled unit file with the `.o` extension should be either in the same folder as the program that will call it or where the compiler's standard units are located. It is now possible to write a program using this unit. The code for this console program is given below.

```
program test;
uses
  compl;
var
  g, h, e: complex;
begin
  writeln('Enter the first complex number:');
  read(dx, gay);
  output(g);
  writeln('Enter the second complex number:');
  read (h.x, h.y);
  output(h);
  sum(g, h, e);
  writeln('Sum of numbers');
  output(e);
  diff(g, h, e);
  writeln('Difference of numbers');
  output(e);
```

```
mul(g, h, e);
writeln ('Product of numbers');
output(e);
divi(g, h, e);
writeln('Quotient of numbers');
output(e);
end.
```

The program output is shown in Figure 4.12.

```
Enter the first complex number
3 4
3.000 + 4.000i
Enter the second complex number
8 -9
3.000 + 4.000i
Sum of numbers
11.000-5.000i
Difference of numbers
-5.000 + 13.000i
Product of numbers
60.000 + 5.000i
Quotient of numbers
-0.083 + 0.407i
```

Figure 4.12: Operations with complex numbers

## 4.9 Exercises

Write a program using procedures and functions.

- 1) Determine if a sequence of integers ending with 0 contains at least one number, for which the sum of its digits is equal to the number of digits. To solve this problem, write a procedure that returns the sum and number of digits in a number.
- 2) Determine if a sequence of integers ending with 0 contains at least one perfect number. Create a function to identify a perfect number.
- 3) Determine if the sequence of N positive numbers contains at least one prime number. Write a function to identify a prime number.
- 4) Count the number of numeric palindromes in a sequence of N positive integers. Write a function to identify a numeric palindrome.
- 5) Count the number of perfect numbers in a sequence of N positive integers. Write a function to identify a perfect number.

- 6) Determine which number, in a sequence of positive integers ending with 0, has the most divisors. Use a function to find the number of divisors in a number.
- 7) Determine which number, in a sequence of positive integers ending with 0, has the most digits. Use a function to count the number of digits in a number.
- 8) Display on screen the values of the function  $f(x) = x - 2e^x$  and its first derivative  $f'(x)$ , in the range -5 to 5. Write functions to calculate the values for  $f(x)$  and  $f'(x)$ .
- 9) Display on screen the first m Fibonacci numbers. Use a function to calculate the n-th Fibonacci number.
- 10) Find the number with the least number of digits in a sequence of N positive integers. Use a function to determine the number of digits in a number.
- 11) Calculate and print the value of the factorial for all positive terms in a sequence of N integers. Use a function to calculate the factorial of a number.
- 12) Display on screen all the numbers in a sequence of positive numbers ending with 0 that are not prime numbers, and their divisors. Use a function to identify prime numbers.
- 13) Display on screen all perfect numbers, and their divisors, in a sequence of N integers. Use a function to identify perfect numbers.
- 14) Find the arithmetic mean of the prime numbers in a sequence of positive integers ending with 0. Use a function to identify prime numbers.
- 15) Find the number containing the most zeros in a sequence of N positive integers. Write a function to count the zeros in a number.
- 16) Find the sum of all palindromes in a sequence of N positive integers. Write a function to identify a numeric palindrome.
- 17) Count the number of terms that contain a zero in a sequence of positive integers ending with 0. Write a procedure that returns true if a number contains a zero, and false otherwise.
- 18) Find the number of zeros and ones for each number in a sequence of positive integers ending with 0. Write a procedure that returns the number of zeros and ones in a number.
- 19) For each term in a sequence of N integers find the average value of its digits. Write a function to calculate the average of the digits in a number.
- 20) Find the number of digits and the smallest digit for each number in a sequence of positive integers ending with 0. Write a procedure that, for each of a

given number, returns two parameters: the number of digits in it and the smallest digit.

- 21) Display the number of digits and the number divisors for each term in a sequence of N integers. Write a procedure to find both numbers.
- 22) Write each number, in a sequence of positive integers ending with 0, in reverse order. For example, 12345 -> 54321. Write a function to reverse numbers.
- 23) For each term in a sequence of positive integers ending with 0, display on the screen the number of digits in the number and its largest digit. Write a procedure that returns the number of digits and the largest digit in a number.
- 24) Calculate the sum of the digits of each prime number in a sequence of N positive integers. Write a procedure that determines if a number is prime and calculates the sum of its digits. If the number is not a prime number, then the procedure should display an appropriate message.
- 25) For each number in a sequence of positive integers ending with 0, determine the sum and quantity of the digits in the number. Write a procedure to calculate the sum and quantity of the digits in a number.

#### Endnotes:

---

1 <sup>T</sup>he addresses of the actual parameters are passed to the subroutine.

2 The declaration section in a procedure may be omitted if not needed.

3 The declaration section in a function may be omitted if not needed.

4 Address operator (See Section 2.5.4).

5 The sum of two numbers  $a+bi$  and  $c+di$  is  $(a+c)+(b+d)i$ . The difference between these numbers is  $(a-c)+(b-d)i$ . The product of these numbers is  $(ac-bd)+(bc+ad)i$ , and the quotient is  $(ab+bd)/(c^2+d^2)+(bc+ad)i/(c^2+d^2)$ .

# Chapter 5. Arrays in Free Pascal

## 5.1 General Information about Arrays

In the previous chapters, we looked at tasks that used scalar variables. However, when processing data of the same type (integers, strings, dates, etc.) it may be more convenient to use arrays. For example, you can use an array to store temperatures for a year. So, instead of creating 365 variables, such as temperature1, temperature2, temperature3, ... ... temperature365, to store the temperature for each day, you can use a single array called temperature, in which each value will correspond to the array element index (see Table 5.1).

*Table 5.1: Temperature values*

Element no.	1	2	3	4	...	364	365
Temperature	-1.5	-3	-6.7	1	...	2	-3

Thus, the following definition can be given.

A *static array* is a structured data type consisting of a fixed number of elements of the same type. The initial size of a *dynamic array* is zero when it is declared.

The array shown in Table 5.2, has 7 elements, and each element stores a **real** number. The elements in the array are numbered from 1 to 7. Such an array is just a list of the same type of data and is called a simple, or one-dimensional array. To access the data stored in a specific array element, specify the name of the array and the element number, called its index.

To store data in table form in rows and columns, use multidimensional arrays.

*Table 5.2: One-dimensional array of 7 real numbers*

Array elements						
1	2	3	4	5	6	7
-1.5	-3.913	13.672	-1.56	45.89	4.008	-3.61

Table 5.3 shows an example of an array consisting of three rows and four columns. This is a two-dimensional array (matrix). By convention, the rows are shown in the first dimension, and the columns in the second. To access data stored in this array, specify the array name and two indices: the first matches the row number, and the second the column number of the required item.

Table 5.3: Two-dimensional numeric array

		Column number			
		1	2	3	4
Row number	1	6.3	4.3	-1.34	5.02
	2	1.1	4.7	8.12	8.50
	3	-2.4	-6.2	11.23	8.18

After a general acquaintance with the concept of an array, we will discuss working with arrays in Free Pascal.

## 5.2 Static Arrays

The keywords **array of** are used to declare arrays. There are two ways to declare static arrays.

In the first, create a new data type and then declare variables of the new type. In this case, the format of the type statement is:

```
type
  type_name = array [index_type] of component_type;
```

Use an enumeration type for index\_type. The component\_type is any previously declared data type, for example:

```
type
  arr = array [0..12] of real;
  // An array data type containing 13 elements, from 0 to 12.
  dabc = array [-3..6] of integer;
  // Data type dabc containing 10 elements, from -3 to 6.
var
  x, y: arr;
  z: dabc;
```

A static array may also be declared as follows, without creating a new type:

```
var variable: array [index_type] of variable_type;
```

For example:

```
var
  z, x: array [1..25] of word;
  // Arrays z and x contain 25 word elements, from 1 to 25.
  g: array [-3..7] of real;
  // Array g contains 11 real elements, from -3 to 7.
```

Predefined constants may be used to declare a static array:

---

```

const
  n = 10;
  m = 12;
var
  a: array [1..n] of real;
  b: array [0..m] of byte;

```

*Constants must be declared before use!*

A two-dimensional array (matrix) can be declared using a one-dimensional array as its base type:

```

type
  onedim = array [1..200] of real;
  twodim = array [1..300] of onedim;
var
  ab: twodim;

```

The same structure can be obtained, using a different notation:

```

type
  twodim = array [1..300, 1..200] of real;
var
  ab: twodim;
or
var
  ab: array [1..300, 1..200] of real;

```

The three methods each resulted in the declaration of a matrix of real numbers, consisting of 300 rows and 200 columns.

Similarly, three-dimensional arrays, or arrays of even more dimensions, could be created:

```

type
  abc = array [1..4, 0..6, -7..8, 3..11] of real;
var
  b: abc;

```

## 5.3 Array Operators

To work with an array as a single unit, use the name of the array only (without the index in square brackets). To access an array element, specify the array name and the element index in square brackets, such as x[1], y[5], c[25], A[8], for example.

Free Pascal defines an assignment operator for arrays with identical structures

(with the same size and data type).

For example, if the arrays C and D are declared as

```
var C, D: array [0..30] of real;
```

then you the following statement is valid:

```
C := D;
```

This operation will immediately assign all the values in array D to the elements of array C, with corresponding indices.

Any other array operation must be carried out one element at a time, using a loop to sequentially process the array elements, starting with the first element, then the second, third, ..., and n-th (see Figure 5.1 and 5.2). A for..do loop is convenient for processing an array element by element.

## 5.4 Element Input / Output

Free Pascal does not have any special means for input / output operations on an entire array at once, and such operations should therefore be performed element by element. For an array input operation, enter the 1st, 2nd and 3rd, etc. array elements sequentially, and do the same for an output operation. Hence, a standard loop for processing an array is needed for array input and output (see Figure 5.3 and 5.4). To refer to an array element, specify the array name and the element index in square brackets, such as X[5], b[2], etc, for example.

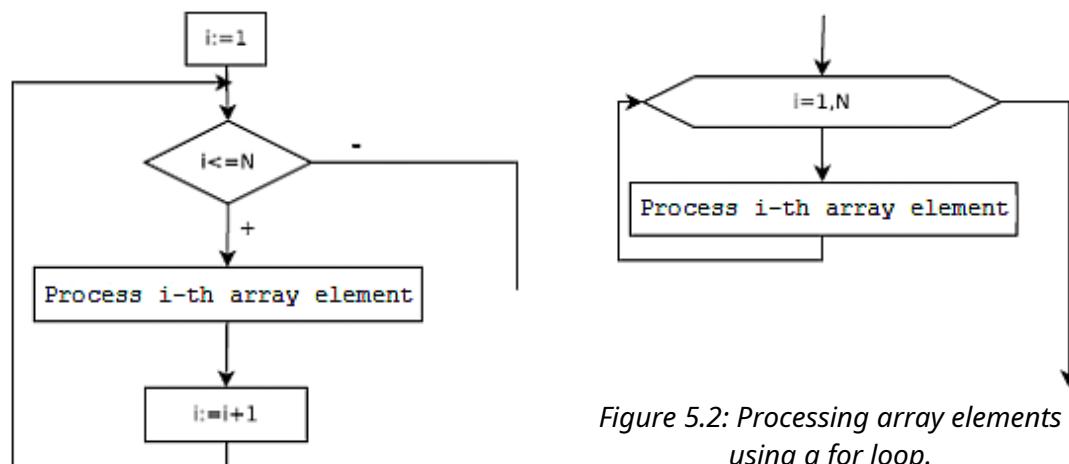


Figure 5.1: Processing array elements using a precondition loop

Figure 5.2: Processing array elements using a for loop.

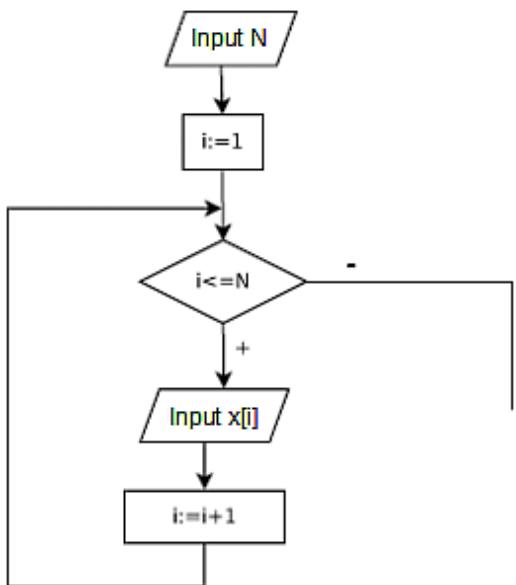


Figure 5.3: Processing array elements using a precondition loop

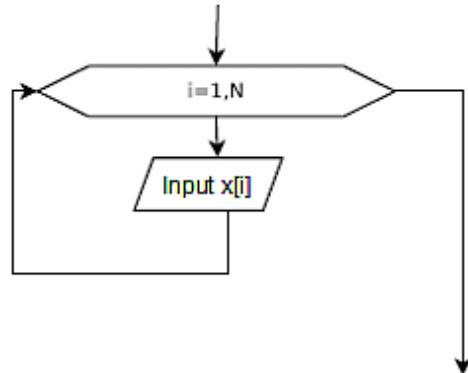


Figure 5.4: Array input using a modification block.

### 5.4.1 Input / Output in Console Applications

Consider the implementation of these algorithms in a console application.

```

// Input the elements of array X using a while loop.
var
  x: array [1..100] of real;
  i, n: integer;
begin
  write('Enter the size of the array (100 max): ');
  readln(N);
  i := 1;
  while (i <= N) do
  begin
    write('x(', i, ') = ');
    readln(x[i]);
    i := i + 1
  end;
end.

// Input the elements of array X using a for loop.
var
  x: array [1..100] of real;
  i, n: integer;
begin
  write('Enter the size of the array (100 max): ');
  readln(N);

```

```
for i := 1 to N do
begin
  write('x(', i, ') = ');
  readln(x[i]);
end;
end.
```

The **for..do** loop is more convenient to use to process the entire array and will therefore be used for processing array input / output operations from here on.

Array output is processed like array input, except that an array element output block will replace the array element input block.

Consider several options for outputting to screen an array of real numbers  $a = (1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8)$ , to understand how they differ from each other, and to decide which option would be most convenient in a specific situation.

```
// Option 1
for i := 1 to n do write (a[i]:3:1);
```

The output to screen from Option 1 is shown in Figure 5.5. Note that there is no space between the elements on screen.



```
1.12.23.34.45.56.67.78.8
```

Figure 5.5: Option 1 output to screen



```
1.10 2.20 3.30 4.40 5.50 6.60 7.70 8.80
```

Figure 5.6: Option 2 output to screen



```
1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8
```

Figure 5.7: Option 3 output to screen

```
// Option 2
for i := 1 to n do write (a[i]:6:2);
```

The output to screen from Option 2 is shown in Figure 5.6.

```
// Option 3
for i := 1 to n do write (a[i]:3:1, '');
```

The output to screen from Option 3 is shown in Figure 5.7.

```
// Option 4
writeln('Array A');
for i := 1 to n do
  writeln(a[i]:6:2);
```

The output to screen from Option 4 is shown in Figure 5.8.

```
// Option 5
for i:= 1 to n do write('a (' , i , ')=' , a[i]:3:1, ' ');
```

The output to screen from Option 5 is shown in Figure 5.9.

```
Array A
1.10
2.20
3.30
4.40
5.50
6.60
7.70
8.80
```

Figure 5.8: Option 4 output to screen

```
a(1)=1.1  a(2)=2.2  a(3)=3.3  a(4)=4.4  a(5)=5.5  a(6)=6.6
a(7)=7.7  a(8)=8.8
```

Figure 5.9: Option 5 output to screen

```
a(1) = 1.10
a(2) = 2.20
a(3) = 3.30
a(4) = 4.40
a(5) = 5.50
a(6) = 6.60
a(7) = 7.70
```

Figure 5.10: Option 6 output to screen

```
// Option 6
```

```
for i := 1 to n do writeln('a (' , i , ') =' , a[i]:6:2);
```

The output to screen from Option 6 is shown in Figure 5.10.

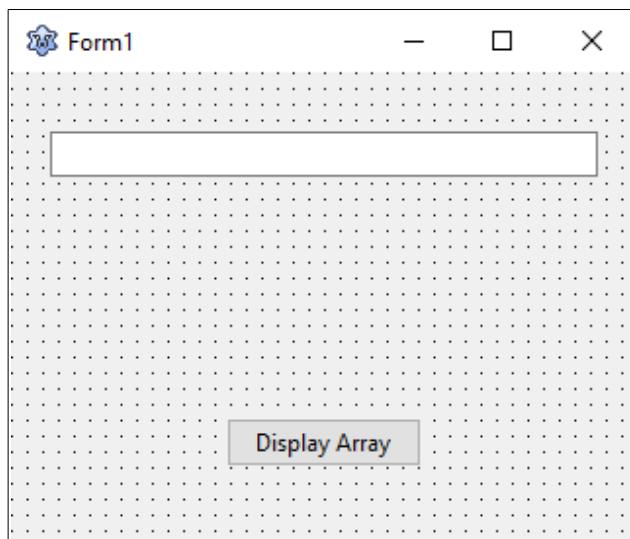


Figure 5.11: Displaying an array of real numbers

Table 5.4: Properties of the Edit1 component in example

Property	Name	Text	Height	Left	Top	Width	ReadOnly
Value	Edit1	''	23	20	30	280	True

Table 5.5: Properties of the Button1 component in example

Property	Name	Caption	Height	Left	Top	Width
Value	Button1	Display Array	25	110	175	100

## 5.4.2 Input / Output in Graphical Applications

Consider possible ways performing array input / output in graphical applications. A standard TEdit component could be used to display an array.

Consider these possible ways, using as an example the standard task of displaying an array of real numbers, such as  $a = (1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8)$ .

Place a button (a TButton component) and a TEdit component on a form (see Figure 5.11).

Table 5.4 and 5.5 shows the TEdit and TButton properties for this example.

In this application, clicking on the Array Output button, will display the array *a* in Edit1 edit box. The array output algorithm is as follows. Each number is converted to a string using the FloatToStr function, and the resulting string is appended to the text already in the edit box. The commented code for the Button1Click event handler is shown below.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  // Source array a.
  a: array [1..8] of real = (1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8);
  i: integer;
  // The variable n = number of array elements (real numbers)
  n: integer = 8;
  S: string = '';
begin
  Edit1.Text := '';
  // for loop processes array elements sequentially
  for i := 1 to n do
    // Add string from converted element a[i] to Edit1 text
    Edit1.Text := Edit1.Text + FloatToStr(a[i]) + ' ';
end;
```

After clicking the Output Array button, the application window should look like the one shown in Figure 5.12.

The simplest way in Lazarus to input an array would be by using an InputBox function. As an example, consider an application in which array input will occur when a button is clicked. Place a button on a form. The commented code for Button1Click event handler is given below.

```
procedure TForm1.Button1Click (Sender: TObject);
var
  i, n: byte;
  X: array [1..20] of real;
begin
  // Get number of array elements.
  n := StrToInt(InputBox('Input Array Elements', 'n = ', '7'));
  for i := 1 to n do          // Element by element input.
    // Enter the next array element.
    X[i] := StrToFloat(InputBox ('Input Array Elements','Enter Item '
+ IntToStr(i) + ':', '0, 00'));
end;
```

Clicking on the button will cause a window for entering the array size to appear on screen (see Figure 5.13).

After entering the array size correctly, dialog boxes for entering each element, like

that shown in Figure 5.14, will appear in succession.

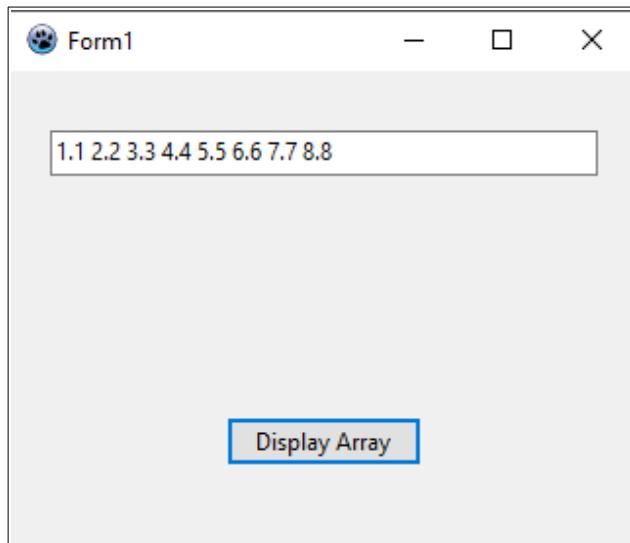


Figure 5.12: Displaying an array in an edit box

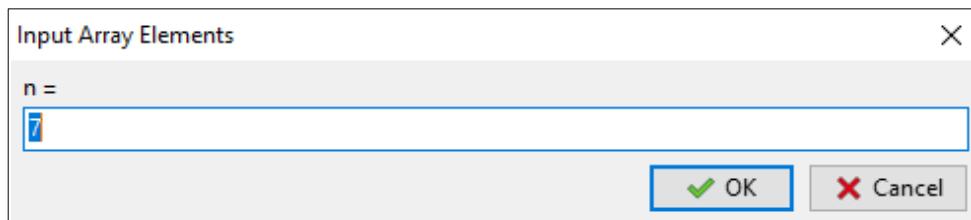


Figure 5.13: Entering the array size

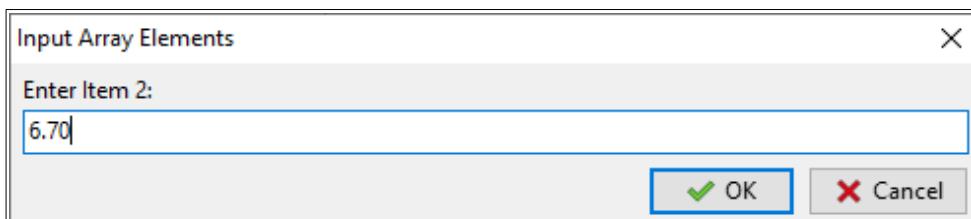


Figure 5.14: Entering the second array element

The MessageDlg function may be used to display an array:

```
for i := 1 to n do
  MessageDlg('X[' + IntToStr(i) + '] = ' + FloatToStr(X[i]),
  mtInformation, [mbOk], 0);
```

which will open a separate window for each element (see Figure 5.15).

To allow the user to view all the elements of an array at the same time, create a

string from them and then display the string on a form using a label or a message box, for example.

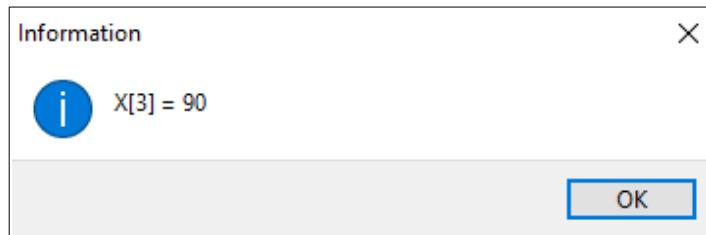


Figure 5.15: Displaying the third array element

```

var
  i, n: byte; X: array [1..20] of real; S: string;
begin
  S := '';
  for i := 1 to n do
    { Convert the array elements to strings and add spaces to
      form a string of array elements separated by spaces. }
    S := S + FloatToStrF (X [i], ffFixed, 5, 2) + ' ';
    // The resulting string can be displayed as a label
  Label2.Caption := S;
  { The resulting string may also be displayed as a message
    using a MessageDlg function (S, mtInformation, [mbOk], 0); }

```

The most versatile Input / Output method for both one-dimensional and two-dimensional arrays is the TStringGrid component. Let us examine this component in detail.

Figure 5.16 shows an application form containing a TStringGrid component<sup>1</sup>. The cells in rows of the grid will allow both the reading and editing of data. Thus, this component can perform both the input and output of array data.

The main properties of this component are shown in Table 5.6.

Note that the indices in the TStringGrid Cells property use a different order [col, row] from the indices in a matrix [row, col].

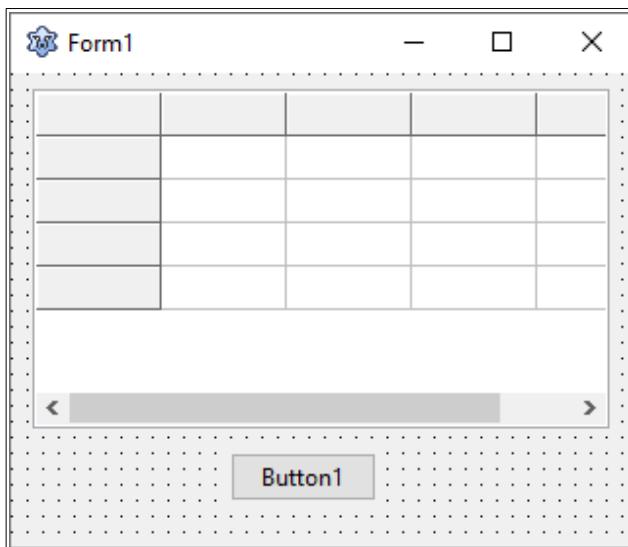


Figure 5.16: Form with grid component

Table 5.6: Main properties of a component of type TStringGrid

Property	Description
Name	Component name
ColCount	Number of columns in grid
RowCount	Number of rows in grid
Cells	A two-dimensional array that stores the grid contents. A grid cell at column number col and row number row, is identified by Cells[col, row]. Rows in this component are numbered from 0 to RowCount-1, and columns from 0 to ColCount-1.
FixedCols	The number of fixed columns from the left, which are highlighted in color and which remain in place during horizontal scrolling.
Fixed Rows	The number of fixed rows from the top, which are highlighted in color and which remain in place during vertical scrolling.
ScrollBars	This parameter determines if scroll bars will be displayed, and has the following possible values: <ul style="list-style-type: none"> <li>• <b>ssNone</b> - no scrollbars present (in this case the user can move about the grid only with the cursor);</li> <li>• <b>ssHorizontal</b>, <b>ssVertical</b>, or <b>ssBoth</b> - horizontal, vertical or both scroll bars are present;</li> <li>• <b>ssAutoHorizontal</b>, <b>ssAutoVertical</b>, or <b>ssAuto</b> - horizontal, vertical, or both scroll bars will appear</li> </ul>

---

	as needed.
Options.goEditing	Boolean variable that determines if the user can ( <b>true</b> ) or cannot ( <b>false</b> ) edit cell content
Options.goTab	Boolean variable that allows ( <b>true</b> ) or disallows ( <b>false</b> ) using the Tab key to move from cell to cell in the grid
DefaultColWidth	Default column width
DefaultRowHeight	Default row height
GridLineWidth	The width of the dividing lines between the cells in the grid
Left	Distance from grid to left border of form
Top	Distance from grid to top of form
DefaultColWidth	Default column width
DefaultRowHeight	Default row height
Height	Height of TStringGrid component
Width	Width of TStringGrid component
Font	The font in which cell content is displayed

*Table 5.7: TStringGrid Properties to be used in the example*

Property	StringGrid1	StringGrid2	Description
ColCount	8	8	Number of columns
RowCount	1	1	Number of rows
FixedCols	0	0	Number of fixed columns from left
FixedRows	0	0	Number of fixed rows from top
Options.goEditing	True	False	Boolean variable that determines if cell content may be edited by the user
Left	10	10	Distance from grid to the left form border
Top	40	110	Distance from grid to the top border
Height	24	24	Component height
Width	540	540	Component Width

Consider using this component for array input / output in an example application that will input eight real numbers into an array and then output them in reverse order.

Place two labels, two TStringGrid components and a button on a form. The StringGrid1 and StringGrid2 properties may be set to the values in Table. 5.7.

The array I / O application form window will be like that shown in Figure 5.17.

In the first table we will enter the elements of the array, in the second we will transform called array. Clicking the Run button will bring up the following subroutine:

```
procedure TForm1.Button1Click (Sender: TObject);
var
  n, i: integer;
  a: array [0..7] of real;
begin
  for i := 0 to 7 do          // Input array.
    { Read element from StringGrid1 cell, convert to number
      and assign to an array element. }
    a[i] := StrToFloat(StringGrid1.Cells[i,0]);
  for i := 0 to 7 do          // Output array.
    // Convert array element to string. Place in StringGrid2 cell
    StringGrid2.Cells[i,0] := FloatToStrF(a[7-i],ffFixed,5,2);
end;
```

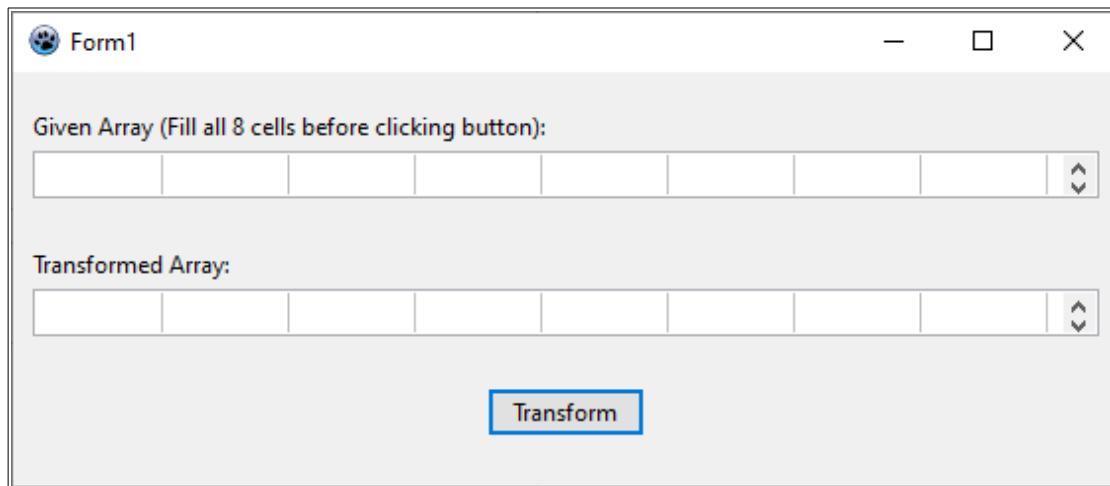


Figure 5.17: Array input / output form (empty)

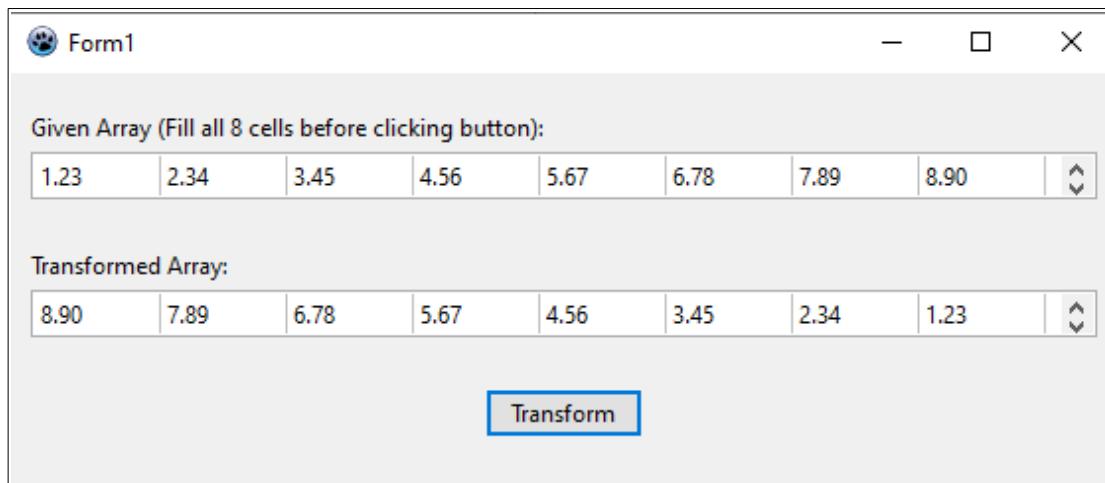


Figure 5.18: Array input / output form

When the program is launched, an application window like that shown in Figure 5.17 will appear. The user enters the original array and clicks the Transform button, after which the application window will look like that shown in Figure 5.18.

Section 5.4 covered various input / output methods for both console and graphical applications. From here on, either will be used, depending on which is more convenient for solving a specific problem.

We will now consider the main algorithms for processing one-dimensional arrays, many of which are like the algorithms for working with sequences (calculating the sum, product, searching for elements that meet certain criteria, subsets, etc.). The difference is that an array allows simultaneous access to all its elements and thus facilitates more complex actions (for example, sorting array elements, deleting and inserting elements, etc).

## 5.5 The Sum and Product of Elements

The algorithms for finding the sum and product of array elements are like the algorithms for finding the sum and product of the terms of a sequence.

To find the sum  $S$  of the elements of an array  $X$ , containing  $n$  elements, set  $S := 0$ , and then sequentially add the elements of array  $X$  to it. The algorithm flowchart for calculating the sum is shown in Figure 5.19.

The corresponding code for this algorithm will look like:

```
s := 0;
for i := 1 to n do
  s := s + x [i];
writeln('s = ', s:7:3);
```

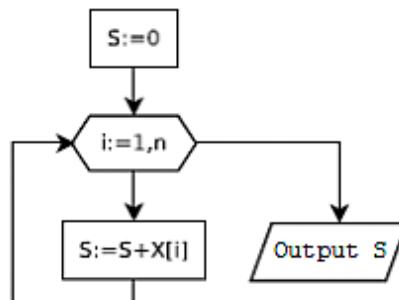


Figure 5.19: The sum of array elements

Let us now find the product P of the elements of the array X. The solution reduces to finding the value of P, by initially setting  $P := 1$  and then multiplying it sequentially by the value of the i-th element of the array. The algorithm flowchart is shown in Figure 5.20.

The corresponding code will look like this:

```

p := 1;
for i := 1 to n do
  p := p*x[i];
writeln('P = ', P:7:3);
  
```

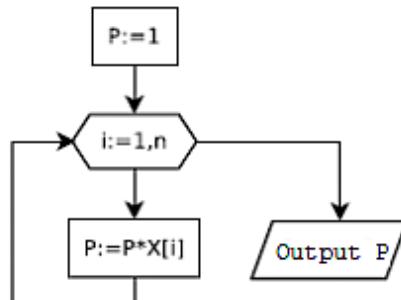


Figure 5.20: Finding the product of array elements

## 5.6 Finding the Maximum Element

Consider the problem of finding the maximum element (`Max`) and its index (`Nmax`) in an array X consisting of n elements.

The algorithm for solving the problem is as follows. Assume the first element array is the maximum, and assign its value to `Max`, and its index (the number 1) to `Nmax`. Then compare all the elements, starting with the second, with the maximum, using a loop. If the current element is larger than the current maximum value, then

assign it (the current element) to Max, and the current index i to Nmax. The process of finding the maximum element in an array is shown in the flowchart in Figure 5.21.

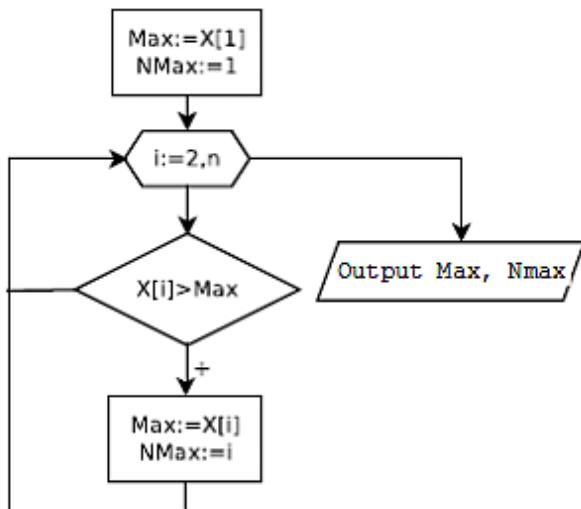


Figure 5.21: Finding the maximum element and its index

The corresponding code looks like:

```

Max := X [1];
Nmax := 1;
for i := 2 to n do
  if X [I] > Max then
    begin
      Max := X [i];
      Nmax := i;
    end;
  write('Max = ', Max:1:3, 'Nmax = ', Nmax);
  
```

The algorithm for finding the minimum element in an array will differ from the above only by replacing Max with Min and Nmax with Nmin, and by changing  $>$  in the if statement to  $<$ .

## 5.7 Sorting Elements

Sorting is the process of arranging array elements in ascending or descending order, by value. For example, an array X with n elements will be sorted in ascending order by value, if

$$X[1] \leq X[2] \leq \dots \leq X[n],$$

and in descending order if

$$X[1] \geq X[2] \geq \dots \geq X[n].$$

Many sorting algorithms are based on the concept that two elements should be rearranged so that they are correctly positioned relative to each other after rearranging. When sorting in ascending order, the element with the lower index cannot be larger than the element with a large index<sup>2</sup> after rearranging. Let us look at a few sorting algorithms.

### 5.7.1 Bubble Sort

The most well-known sorting method for sorting arrays is the bubble sort method. Its popularity is due to its easy-to-remember name<sup>3</sup> and the simple algorithm. Bubble sorting is based on comparing successive array elements and swapping them when needed, using a loop. Let us consider in greater detail a bubble sort algorithm to sort in ascending order, for example.

Compare the first element of the array with the second, and if the first is larger than the second, then swap them. Then compare the second with the third, and if the second is larger than the third, then swap them too. Next, compare the third and the fourth, and if the third is larger than the fourth, then swap them too. After three comparisons, the largest element is Element 4.

If we continue to compare adjacent elements: the fourth with the fifth, the fifth with the sixth, etc., up to the nth – 1 with the nth elements, the largest element will end up in the last (nth) place after these comparisons.

*Table 5.8: Sorting array elements in ascending order*

Item number	1	2	3	4	5
Source array	7	3	5	4	2
First scan	3	5	4	2	7
Second scan	3	4	2	5	7
Third scan	3	2	4	5	7
Fourth scan	2	3	4	5	7

If this algorithm is now repeated from the first to nth - 1 elements (the nth element is already completely sorted), the second largest element will end up in the nth - 1 place. This is repeated until the entire array is sorted.

Table 5.8 shows in detail the process of sorting elements in an array. It is easy to see that for sorting an array consisting of n elements, you need to scan it n - 1 times, each time decreasing the scan range by one element. The flowchart of the algorithm described is shown in Figure 5.22. To swap two elements in an array

(block 4 – the last figure), a buffer b is used, which temporarily stores the value of the element to be replaced. Below is the code for a console application to sort an array in ascending order using the bubble sort method.

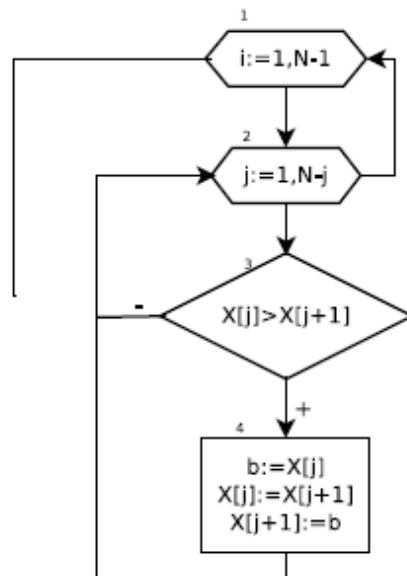


Figure 5.22: Bubble sort in ascending order

```

program sort_array;
var
  i, j, n: byte;
  X: array [1..100] of real;
  b: real;
begin
  write('Enter the size of the array: ');
  readln(n);
  for i := 1 to n do
  begin
    write('X[', i, '] = ');
    readln(X[i]);
  end;
  writeln('Array X:');
  for i := 1 to n do
    write(x[i]:5:2, ' ');
  writeln;
  for j := 1 to n - 1 do          {Number of scans}
    for i := 1 to n - j do        {Scan range}
      if X[i] > X[i+1] then      {If current greater than next}
      begin                        {swap them.}
        b := X[i];                {Store current element.}
        X[i] := X[i+1];            {Replace current with next}
        X[i+1] := b;                {Replace next with b.}
      end;

```

```
writeln('Sorted array:');
for i := 1 to n do
  write(X[i]:5:2, ' ');
writeln;
end.
```

Figure 5.23 shows the output from this program. To sort the array elements in descending order by value, replace the  $>$  sign with  $<$  (see block 3 in Figure 5.22) when comparing elements. Consider the following sorting algorithm.

```
Enter the size of the array
5
X[1] = 7
X[2] = 3
X[3] = 5
X[4] = 4
X[5] = 2
array X
7.00 3.00 5.00 4.00 2.00
ordered array
2.00 3.00 4.00 5.00 7.00
```

Figure 5.23: Program output for sorting array in ascending order

### 5.7.2 Selection Sort

The Selection Sort method could be used to sort the elements of an array in ascending (or descending) order, by finding the index of maximum (or minimum) element. The selection sort algorithm flowchart is shown in Figure 5.24.

Find the largest element in the array (blocks 2 to 5) and swap it with the last element (block 6). After this, the largest element will migrate to its proper place. Repeat these actions (blocks 2 to 6), reducing the number of elements to be scanned by one (block 7) until there is only one element left to be scanned (block 8). Because the number of elements is being reduced by 1 in each step, copy N into K at the beginning of the algorithm (block 1) before K is decreased, so as not to lose the size of the array (N).

When sorting an array in descending order, the minimum element should be moved. To do this, change the  $>$  sign to a  $<$  sign in Block 4 in the algorithm (Figure 5.24).

The code for sorting an array in ascending order using the selection sort method is shown below.

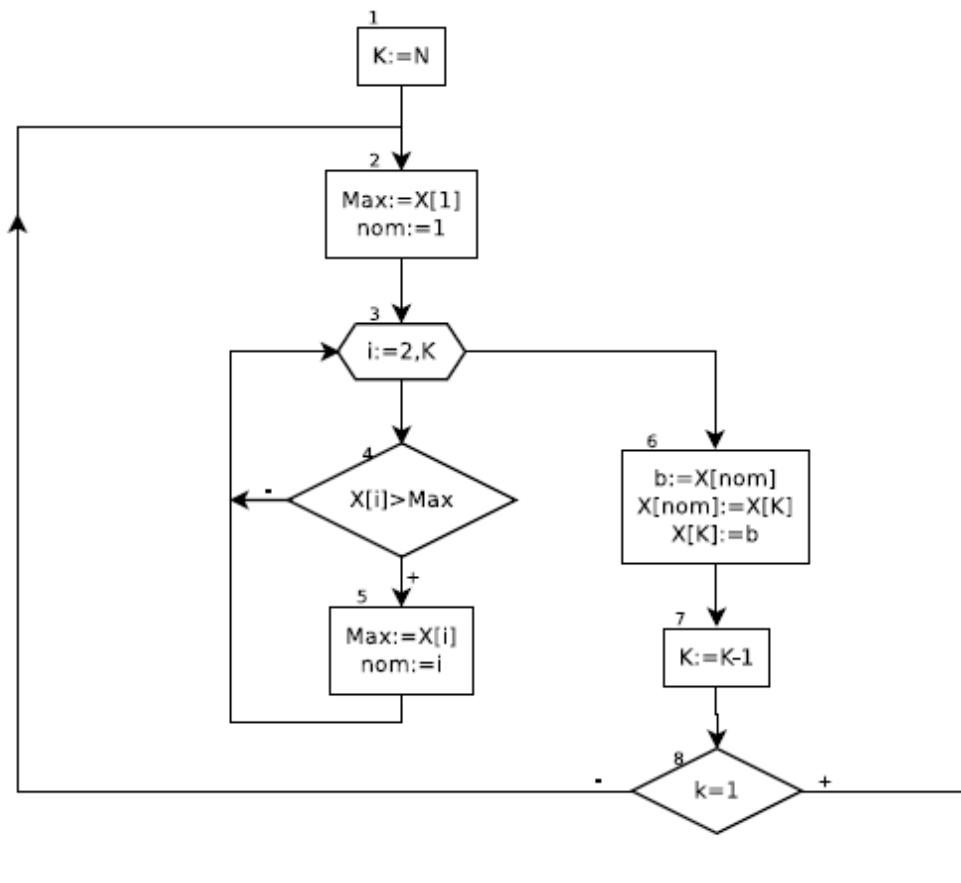


Figure 5.24: Sorting array in ascending order by selecting the largest element

```

k := n;
repeat
  max := x[1];
  nom := 1;
  for i := 2 to k do
    if max < X[i] then
      begin
        max := X[i];
        nom := i;
      end;
    b := x[nom];
    x[nom] := x[k];
    x[k] := b;
    k := k - 1;
until k = 1;
  
```

The next important algorithm for processing arrays is the algorithm for removing an element from an array.

## 5.8 Removing an Element

Let us start our acquaintance with the algorithm for removing an array element with the following simple task. Say we need to remove the third element from the array X, containing 6 elements. The algorithm for removing the third element is to overwrite the third element with the fourth, the fourth with the fifth and the fifth with the sixth.

```
X[3] := X[4];
X[4] := X[5];
X[5] := X[6];
```

Thus, all elements from the third to the fifth must be moved one place to the left - the i-th element must be overwritten by the (i-th + 1). The algorithm flowchart is shown in Figure 5.25.

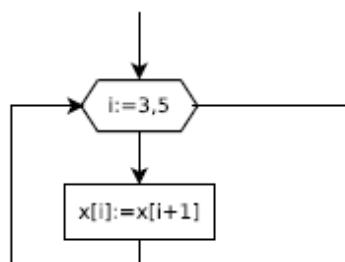


Figure 5.25: Removing 3rd element  
of array

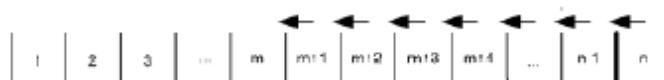


Figure 5.26: Removing an element from an array

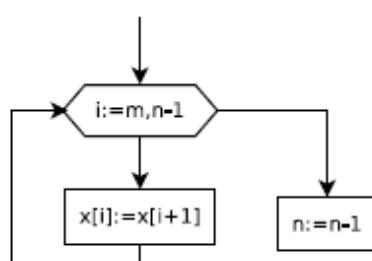


Figure 5.27: Removing m-th  
element from array of n  
elements

Now let us consider a more general problem: the removal of the m-th element from an array X containing n elements. To do this, just overwrite the m-th element with the (m-th + 1) element, the (m-th + 1) element with the (m-th + 2), etc., and the (n-th - 1) element with the n-th. The process of removing an element from an array is shown in Figure 5.26.

The algorithm for removing the m-th element from an array X containing n elements is shown in Figure 5.27.

After removing an element<sup>4</sup> from the array, the number of elements in the array will decrease by one and the index of some of the elements will change. If an element is removed, then the one after it will take its place and looping to it (by increasing the index by one) will not be necessary. The next element will move up by itself because of the deletion.

Consider the following problem as an example.

EXAMPLE 5.1. Remove negative elements from X (n) array.

The algorithm for solving this problem is quite simple. Loop over the elements of the array, and if an element is negative, delete it by shifting all subsequent elements one place to the left. After removing the element there is no need to continue immediately to the next one for further processing. It will move to the current position by itself. The flowchart for solving Problem 5.1 is shown in Figure 5.28.

Below is the commented code.

```
program sort_array;
  var i, n, j: byte;
  X: array [1..100] of real;
begin
  write('Enter the size of the array: ');
  readln(n);
  { Input array. }
  for i := 1 to n do
  begin
    write('X[', i, '] = ');
    readln(X[i]);
  end;
  writeln('Array X');           { Output source array }
  for i := 1 to n do
    write(x[i]:5:2, ' ');
  writeln;
  i := 1;                      { Begin processing array }
```

```
while (i <= n) do
{ If next array element X[i] is negative, then }
  if x[i] < 0 then
    begin
      {delete the element of the array with index i.}
      for j := i to n - 1 do
        x[j] := x[j+1]; { Reduce the size of the array. }
    { There is no need to move to the next element of the array. }
    n := n - 1;
  end
  else
{ If the element was not removed, then go to the next element of the
array. }
  i := i+1; { End processing array }
  writeln('Modified array'); { Output transformed array }
  for i := 1 to n do
    write(X[i]:5:2, ' ');
  writeln;
end.
```

The output from the program is shown in Figure 5.29.

```
Enter the size of the array
7
X[1] = 1
X[2] = -2
X[3] = 3
X[4] = -4
X[5] = 5
X[6] = -6
X[7] = 7
Array X
1.00 -2.00 3.00 -4.00 5.00 -6.00 7.00
Modified array
1.00 3.00 5.00 7.00
```

Figure 5.28: The results of the program for Example 5.1

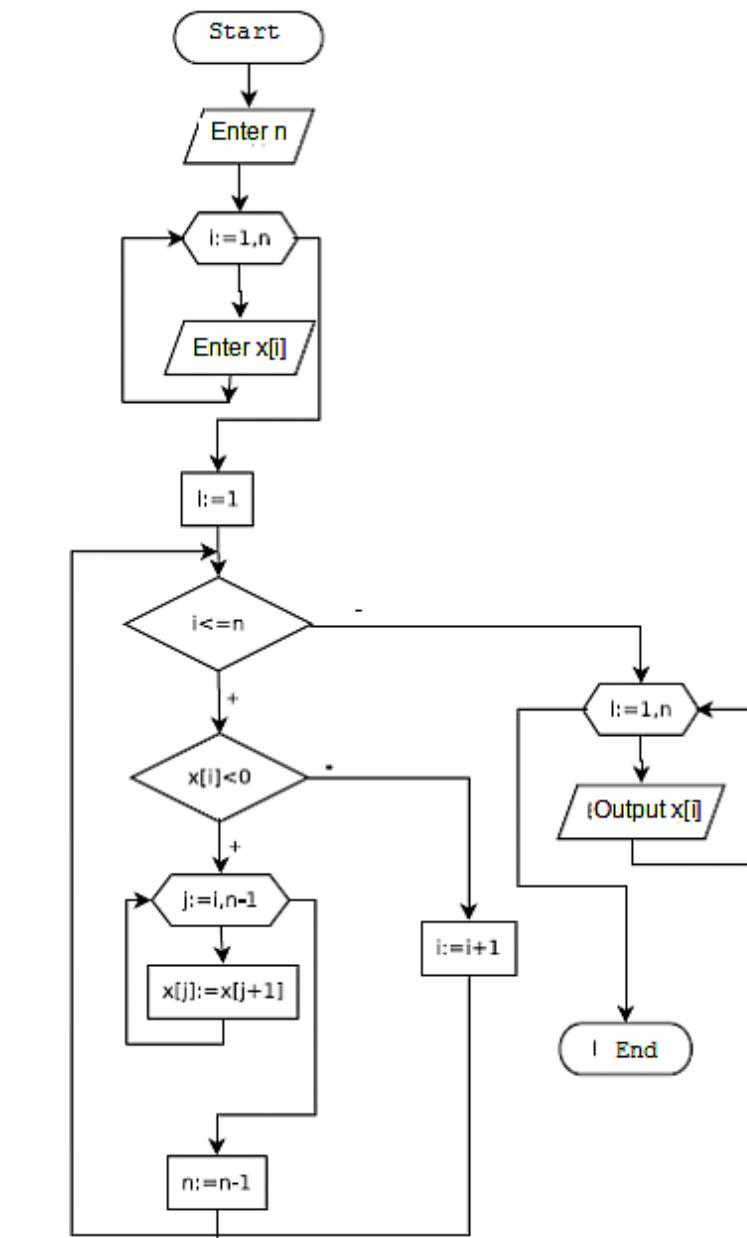


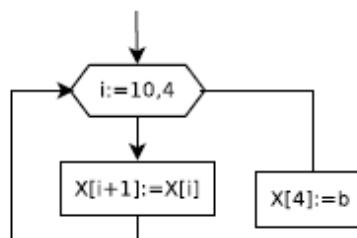
Figure 5.29: Flowchart for solving Example 5.1

## 5.9 Inserting an Element

Consider a simple task: insert the number b into the array X(10), between the third and fourth elements.

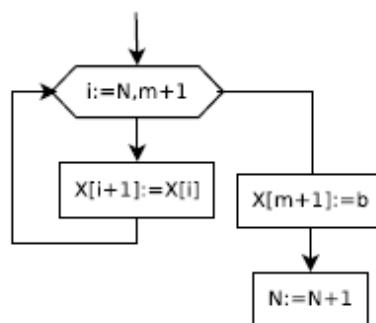
To solve this problem, all the elements of the array, starting with the fourth, should shift to the right by one. Then the element b is saved in the fourth element of the

array, by setting  $X[4] := b$ . To avoid losing the data in the adjacent elements, the tenth element should be moved to the right first, then the ninth, the eighth, etc, all the way to the fourth. The flowchart for the insertion algorithm is shown in Figure 5.30.



*Figure 5.30: Insert number b between 3rd and 4th array elements*

For the general case, the flowchart for inserting the number  $b$  into the array  $X(N)$ , between the elements with numbers  $m$  and  $m+1$  is shown in Figure 5.31.



*Figure 5.31: Insert number b into array X*

Below is the code that implements this algorithm<sup>5</sup>.

```

var
  i, n, m: byte;
  X: array [1..100] of real;
  b: real;
begin
  write('Array size n = ');
  readln(n);
  for i := 1 to n do
  begin
    write('X[', i, '] = ');
    readln(X[i]);
  end;

```

---

```
writeln('Array X:');
for i := 1 to n do
  write(x[i]:5:2, ' ');
writeln;
write('Insert after element: m = ');
readln(m);
write('Data to insert: b = ');
readln(b);
for i := n downto m + 1 do
  x[i + 1] := x[i];
x[m + 1] := b;
n := n + 1;
writeln('Modified array:');
for i := 1 to n do
  write(x[i]:5:2, ' ');
writeln;
end.
```

## 5.10 Dynamic Arrays

Dynamic arrays, which can be sized or resized during program execution, have an initial size of zero when declared. For example:

```
var
  array_1: array of real;           //1-dimensional array
  array_2: array of array of char; //2-dimensional array
  array_3: array of array of byte; //3-dimensional array
```

Allocating memory on the heap and specifying index limits for each dimension in dynamic arrays is carried out during program execution using the `SetLength` function:

**SetLength**(array\_name, index\_boundary\_list);

Memory allocated by `SetLength` is freed automatically, but to free the allocated memory explicitly, execute the statement:

array\_name := **NIL**;

The *lower bound* of a dynamic array (the minimum index) is always zero. The *upper bound* (the maximum index) is returned by the standard function:

**high**(array\_name)

*Dynamic arrays* can be used in normal array processing in Free Pascal. Consider, for example, the use of a *dynamic array* for the simplest of tasks, such as finding the sum of array elements.

```
program dynarray;
var
```

---

```

x: array of real;    // Dynamic array declaration
s: real;
i, n: integer;
begin
  write('Array size n = ');
  readln(n);
  SetLength(x, n);      // Allocate heap memory for n real values
  for i := 0 to high(x) do
    read(x[i]);
  s := 0;
  for i := 0 to high(x) do
    s := s + x[i];
  writeln('Sum = ', s:7:3);
  x := NIL;            // Free memory (not necessary).
End.

```

## 5.11 Passing Arrays to Subroutines

Let us see how to pass arrays to a subroutine. As discussed previously (in Chapter 4), when declaring variables in the formal parameter list of a subroutine, their names and types must be specified. However, the type of any parameter in the list can only be a standard or a previously declared type. Therefore, to pass an array to a subroutine, you must declare its type<sup>6</sup> before declaring the procedure:

```

type
  array_type = array [index_list] of type;
procedure procedure_name (array_name: array_type);

```

For example:

```

type
  vector = array [1..10] of byte;
  matrix = array [1..3, 1..3] of real;
procedure proc(A: matrix; b: vector; var x: vector);

```

Passing a fixed-length string, which is an array<sup>7</sup>, to a subroutine should be done similarly. For example:

```

type
  string_5 = string[5];
  string_10 = string[10];
function fun(Str: string_5): string_10;

```

Passing a regular string variable to a subroutine is simpler:

```

function fun2(S: string): string;

```

Arrays<sup>8</sup> can be passed to a subroutine using an *open array parameter*, which is a formal array parameter with a specified type but unspecified size. The actual parameter passed to the subroutine may be a static or dynamic array. A procedure may be declared with an open array parameter as shown below:

```
procedure procedure_name(public_array_name: array of type);
```

Passing an open array to a subroutine makes it possible to process one-dimensional arrays of arbitrary length:

```
{ This procedure is intended to output messages to the screen
  with the values of the elements of a one-dimensional array.
  The subroutine parameter is an open array of integers. }
```

```
procedure outputArray(X: array of integer);
var
  i: byte;
begin
  // The indices of the open array are from 0 to high(X).
  for i := 0 to high(X) do
    //Display the message: X[item_number] = item_value.
    writeln('X[', i, '] = ', X[i]);
end;

var
  A: array [1..10] of integer;
  C: array of integer;
  i: byte;
begin
  // Create a one-dimensional array A of 10 elements.
  for i := 1 to 10 do A[i] := 2*i + 1;
  // Allocate memory for 3 integer values:
  SetLength(C, 3);
  // Create a one-dimensional array C of 3 elements.
  for i := 0 to 2 do C[i] := 1 - 2*i;
  // Call subroutine.
  outputArray(A);
  outputArray(C);
end.
```

Without using open arrays, the outputArray procedure would have had to be written thus:

```
// Type description: an integer array, indices from 0 to 10.
type
  arr = array [0..10] of integer;
{ The procedure is intended to output messages to the screen
  with the values of the elements of a one-dimensional array. }
```

---

```

procedure outputArray(X: arr; nN, nK: byte);
{ subroutine parameters:
  1. An array of integers X.
  2. The lower bound for the nN index.
  3. The upper bound for the nK index. }
var
  i: byte;
begin
  // The array indices are from nN to nK.
  for i := nN to nK do
    writeln('X[' , i, '] =' , X[i]);
end;

```

## 5.12 Dynamic Memory Allocation

All static variables declared in programs up to this point were placed in one continuous area of working memory, called a data segment. To work with large arrays, so-called dynamic memory can be used, which is allotted to the program after it starts. The amount of dynamic memory available can vary over a wide range and is limited by the available computer memory, by default.

Placing data in dynamic memory should occur spontaneously during program execution. With dynamic placement, the amount of data to be stored does not need to be known beforehand. In addition, data cannot be accessed by name, as with static variables.

The computer working memory is a collection of elementary memory cells for storing information in bytes, each of which is numbered. These numbers are called addresses, and they allow access to any byte of memory.

### 5.12.1 Dynamically Allocated Variables

Free Pascal has a flexible memory management facility, using pointers. A *pointer* is a variable that contains the address of a byte of memory.

As a rule, a pointer is associated with some data type. In such a case it is called *typed*. It is declared using the ^ sign, which is placed before the corresponding type. For example:

```

type
  arr = array [1..2500] of real;
var
  a: ^integer;
  b, c: ^real;
  d: ^arr;

```

In Free Pascal, you can declare a pointer without associating it to a specific data

type. The standard pointer type is used for this. For example:

```
var  
  p, c, h: pointer;
```

Pointers of this kind are called *untyped*. Since untyped pointers are not associated with a specific type, it is convenient to use them to dynamically allocate memory for data which may have types and structures that change during program execution.

Since values stored in pointers are memory addresses, one might think that the value in a pointer can be copied to any other. In fact, this operation can be performed only between pointers associated with the same data types. For example:

```
var  
  p1, p2: ^integer; p3: ^real; pp: pointer;
```

In this case, the assignment `p1 := p2;` valid, while `p1 := p3;` is not, because `p1` and `p3` point to different data types. This limitation does not apply to untyped pointers, so you can write `pp := p3; p1 := pp;` and achieve the desired result.

Dynamic memory in Pascal is a continuous array of bytes called the “heap”. Physically, the heap is located outside the memory area occupied by the body of the program.

The beginning of the heap is stored in the standard variable **HeapOrg** and the end in the variable **HeapEnd**. The current boundary of unallocated dynamic memory is stored in the **HeapPtr** pointer.

Memory for a dynamic variable is allocated using the **new** procedure, which takes a typed pointer as its parameter. When **new** is called, the pointer acquires the value of the dynamic address, starting at which data can be stored. For example:

```
var  
  i, j: ^integer;  
  r: ^real;  
begin  
  new(i);  
  new(r);  
  new(j);
```

After executing the first statement, pointer `i` takes the value that the heap pointer `HeapPtr` had before this. `HeapPtr` itself increases its value by two since the size of the internal representation of the **integer** type associated with pointer `i` is 4 bytes. The `new(r)` statement causes another offset of the `HeapPtr` pointer, but by 8 bytes, which is the size of the internal representation of the **real** type. A similar process

will occur for a variable of any other type. After a pointer points to a specific physical byte of memory, any value of the corresponding type could be stored at this address. A pointer followed immediately by a ^ sign, without any space in between, could be used for this. For example:

```
i^ := 4 + 3;  
j^ := 17;  
r^ := 2 * pi;
```

Thus, the value pointed to by the pointer, that is, the actual data stored on the heap, is indicated by the ^ sign, which is placed immediately after pointer. If this sign is absent after the pointer, then it stores the address where the data is stored. Dynamically allocated data (but not their address!) can be used as constants and variables of the corresponding type anywhere it is allowed, for example:

```
r^ := sqr(r^) + sin(r^ + i^) - 2.3
```

This statement is not allowed:

```
r := sqr(r^) + i^;
```

since the pointer r cannot be assigned a value of a real type. Similarly, the statement below is not allowed:

```
r^ := sqr(r);
```

since the pointer r stores an address (as opposed to the data stored at this address), which cannot be squared. The assignment r^ := i will also be erroneous, since the real data that r^ points to, cannot store a pointer value (address).

Dynamic memory may not only be taken from the heap, but also returned to it. To do this, use the **dispose(p)** procedure, where p is a pointer, to return to the heap the memory associated with the pointer p, without changing the value of the pointer.

When working with pointers and dynamic memory, the programmer is responsible for the correct usage of the new and dispose procedures and for work with addresses and dynamic variables, since the compiler will not catch related errors. Such errors can cause the computer to freeze, or worse!

Another possibility is to free an entire chunk of the heap. To do this, the current value of HeapPtr is stored in a pointer using the **mark** procedure, before starting heap allocation. After this, a chunk of the heap can be freed at any time, starting from the address stored by the **mark** routine, to the end of the heap. The **release** procedure is used for this.

The mark routine remembers the current heap address in HeapPtr (using mark(ptr), where ptr is a pointer of any type that will store the current value of

HeapPtr). Procedure `release(ptr)`, where `ptr` is a pointer type, frees the section of the heap from the address stored in the pointer to the end of the heap.

## 5.12.2 Dynamically Allocated Arrays

The `getmem` and `freemem` procedures can be used to work with pointers of any type. The `getmem (p, size)` procedure, where `p` is a pointer and `size` (of type word) is the amount in bytes of memory to be the allocated dynamically, reserves the required amount of memory on the heap, starting at the pointer.

The `freemem (p, size)` procedure, where `p` is a pointer, and `size` (of type word) is the amount in bytes of memory to be released back to the heap at the pointer. If the procedure is applied to memory that has already been released, an error will occur.

Like dynamic arrays, *dynamically allocated arrays* are arrays of variable length, for which memory can be allocated (and changed) during program execution, each time the program is launched, or in different parts of the program. Use `x^[i]` to refer to the `i`-th element of a dynamic array.

Let us see how *dynamically allocated arrays* work.

EXAMPLE 5.2. Find the maximum and minimum elements in the array `X(n)`.

First, consider the solution using static arrays.

```
// Solution using a static array
program statarray;
var
  x: array [1..2] of real;
  i, n: integer;
  max, min: real;
begin
  write('Enter array size (2 max): ');
  readln(n);
  for i := 1 to n do
  begin
    write('x[',i,'] = ');
    readln(x[i]);
  end;
  max := x[1];
  min := x[1];
  for i := 2 to n do
  begin
    if x[i] > max then max := x[i];
    if x[i] < min then min := x[i];
  end;
end.
```

---

```

end;
writeln('max = ', max:1:4);
writeln('minimum = ', min:1:4);
end.

```

Now let us look at the solution using pointers. Memory will be allocated using the **new** and **dispose** procedures (dynalaray1 program) or **getmem** and **freemem** (dynalaray2 program).

*Note that the solution using getmem is the only one that will accept more than 2 elements (the size of the base array) and still function correctly.*

```

// Solution using new and dispose
program dynalaray1;
type
  arr = array [1..2] of real;
var
  x: ^arr;
  i, n: integer;
  max, min: real;
begin
  { Allocate memory for dynamically allocated array. }
  new(x);
  write('Enter array size (2 max): ');
  readln(n);
  for i := 1 to n do
  begin
    write('x('', i, '') = ');
    readln(x^[i]);
  end;
  max := x^[1];
  min := x^[1];
  for i := 2 to n do
  begin
    if x ^ [i] > max then max := x ^ [i];
    if x ^ [i] < min then min := x ^ [i];
  end;
  writeln('maximum = ', max:1:4);
  writeln('minimum = ', min:1:4);
  dispose(x);   { Free memory. }
end.

// Solution using getmem and freemem
program dynalaray2;
type
  arr = array [1..2] of real;
var
  x: ^arr;
  i, n: integer;

```

```

max, min: real;
begin
  write('Enter array size: ');
  readln(n);
  { Allocate memory for n array elements. }
  getmem(x, n*sizeof(real));
  for i := 1 to n do
    begin
      write('x(', i, ') = ');
      readln(x^[i]);
    end;
  max := x^[1];
  min := x^[1];
  for i := 2 to n do
    begin
      if x^[i]> max then max := x^[i];
      if x^[i] <min then min := x^[i];
    end;
  writeln('maximum =', max:1:4);
  writeln('minimum =', min:1:4);
  freemem(x, n*sizeof(real));    { Free memory. }
end.

```

When working with *dynamically allocated variables*, observe the following order of work:

- 1) Declare pointers.
- 2) Allocate memory for array (using the **new** or **getmem** function).
- 3) Process the dynamically allocated variable.
- 4) Free memory (using **dispose** or **freemem** function).

## 5.13 Examples

**EXAMPLE 5.3.** Copy all negative elements in array A to array B if array A contains k integers.

The solution to the problem is as follows. Iterate through the elements of array A sequentially. If a negative element is found then save it in array B. Figure 5.32 shows that the first negative element occurs in array A at index 3, the second and third at indices 5 and 6 respectively, and the fourth at index 8. The indices of these elements in array B will be 1, 2, 3 and 4.

An additional variable will be needed to keep track of the indices of the elements as they are added to array B. The flowchart in Figure 5.33 shows that this variable is

$m$ , which stores the index of each element added to array B.

Initially, array B contains no element, and therefore  $m = 0$  (block 2).

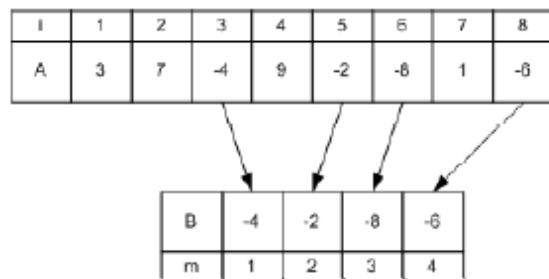


Figure 5.32: Creating array B from negative elements of array A

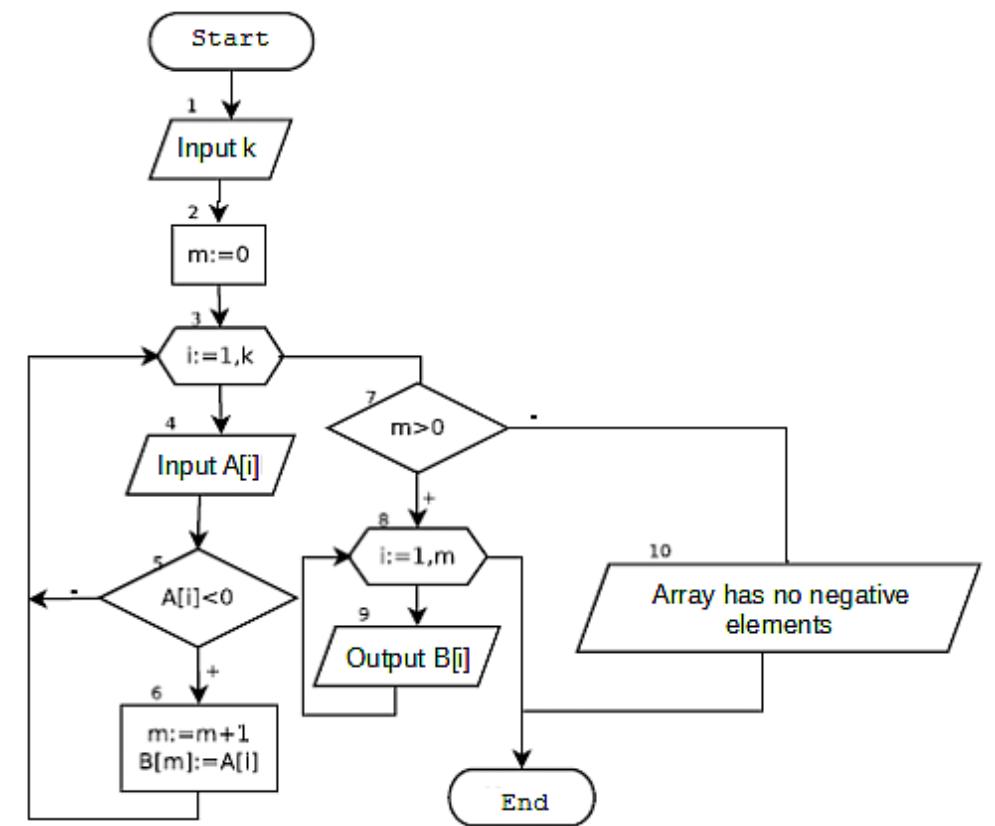


Figure 5.33: Creating array B from negative elements of array A

In the loop (block 5), we sequentially iterate through all elements of A. If an element in array A with a negative value is encountered, then the variable  $m$  is increased by one, and the value of the element in array A is saved in array B at index  $m$  (block 6). In block 7, we verify if array A contained negative elements and if elements were added to array B. As a result of this algorithm,  $m$  elements

containing negative values were added to array B.

The code below implements the algorithm described above.

```

var
  a, b: array [1..200] of integer;
  k, m, i: byte;
begin
  write('Enter number of elements in array A: ');
  readln(k);
  m := 0;
  for i := 1 to k do
  begin
    write('A[', i, '] = ');
    readln(A[i]);
    if A[i] < 0 then
    begin
      m := m + 1;
      B[m] := A[i];
    end;
  end;
  if m > 0 then
    for i := 1 to m do
      write(B[i], ' ')
  else
    write('No negative values in array A!');
end.

```

EXAMPLE 5.4. Given: array  $y$ , containing  $n$  integers. Create an array  $z$  with the negative numbers from array  $y$  at the beginning, then the positive numbers and finally the zeros.

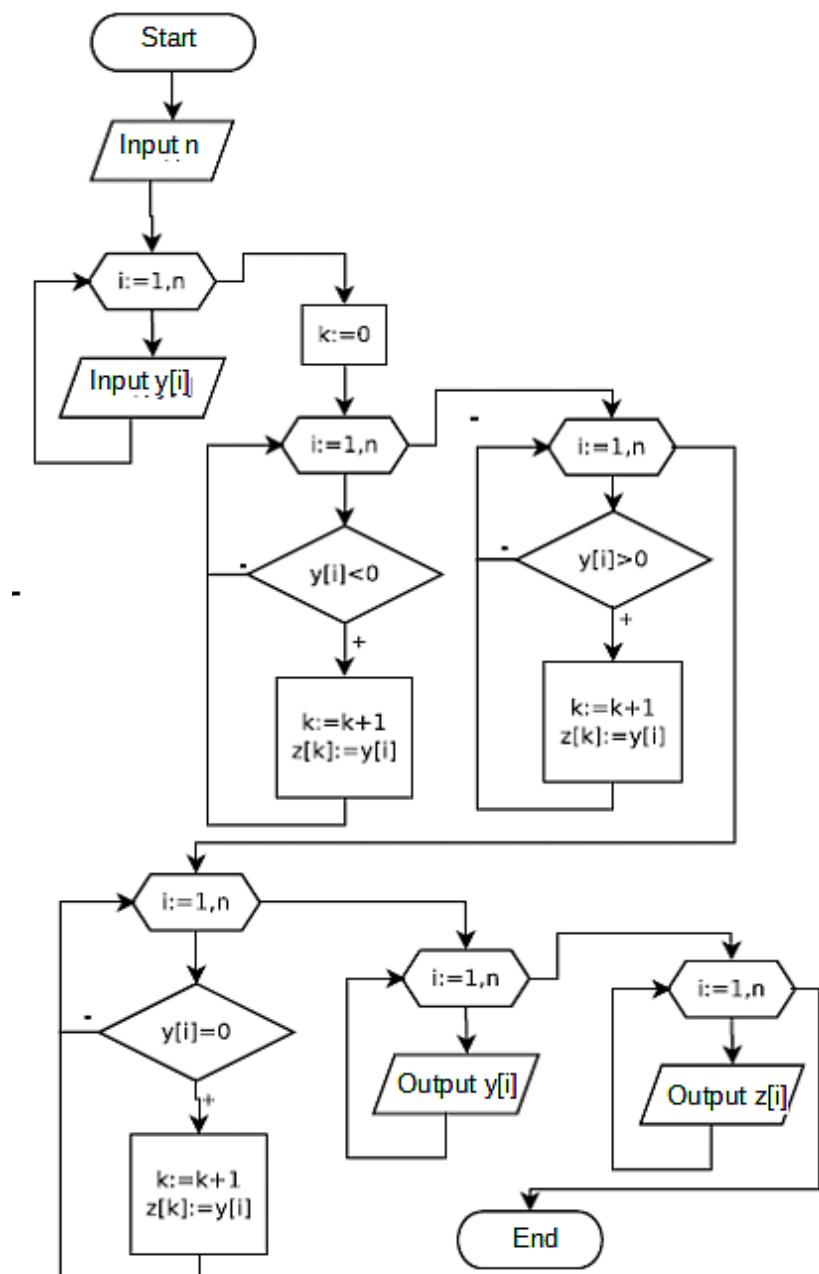
The algorithm for solving this problem is based on the algorithm for copying elements satisfying a specific condition from one array to another, which was discussed in detail in the previous problem. The flowchart for the solution of Problem 5.4 is shown in Figure 5.34.

The commented code is shown below.

```

program mas_four;
var
  y, z: array [1..50] of integer;
  i, k, n: integer;
begin
  write('Enter number of elements in array y (n <= 50): ');
  readln(n);

```



*Figure 5.34: Flowchart for Example 5.4*

```
for i := 1 to n do           // Input array y.  
begin  
    write('y[', i, '] = ');  
    readln(y[i]);  
end;  
k := 0;  
// Copy negative numbers from array y to array z.
```

```

for i := 1 to n do
  if y[i] < 0 then
    begin
      k := k + 1;
      z[k] := y[i];
    end;
  // Copy positive numbers from array y to array z.
  for i := 1 to n do
    if y[i] > 0 then
      begin
        k := k + 1;
        z[k] := y[i];
      end;
    // Copy zeros from array y to array z.
    for i := 1 to n do
      if y[i] = 0 then
        begin
          k := k + 1;
          z[k] := y[i];
        end;
      // Print array y.
      writeln('Array y:');
      for i := 1 to n do
        write(y[i], ' ');
      writeln;
      // Print array z.
      writeln('Array z:');
      for i := 1 to n do
        write(z[i], ' ');
    end.

```

EXAMPLE 5.5. Rewrite the elements in array X in reverse order.

The algorithm for solving the problem is as follows: we swap the 1st and n-th elements, then the 2nd and n-th - 1 elements, and so on until we get to the middle of the array. The element with index i should be swapped with the element n+1-i. The flowchart for swapping elements in an array is shown in Figure 5.35.

The code for solving Problem 5.5 using a console application is shown below.

```

program mas_five;
type
  arr = array [1..100] of real;
var
  x: arr;
  i, n: integer;

```

```
b: real;
```

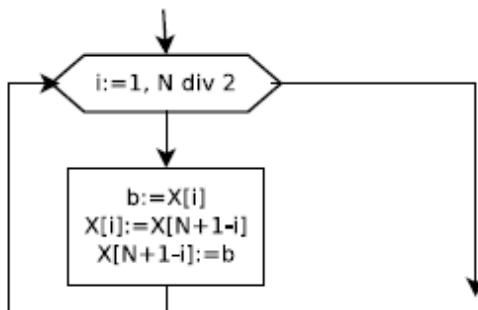


Figure 5.35: Part of flowchart for Example 5.5

```

begin
  // Enter the size of the array.
  write('Enter the size of the array: ');
  readln(n);
  // Input array.
  for i := 1 to n do
  begin
    write('x[' , i , '] = ');
    readln(x[i]);
  end;
  { Loop over first half of the array, and swap 1st element with
    n-th, 2nd with (n-1), ... i-th c (n + 1-i) -m }
  for i := 1 to n div 2 do
  begin
    b := x[n + 1 - i];
    x[n + 1 - i] := x [i];
    x[i] := b;
  end;
  // Output the reversed array.
  writeln('Reversed array:');
  for i := 1 to n do
    write(x[i]:1:2, ' ');
end.

```

EXAMPLE 5.6. Remove the first four zero elements in array X, consisting of n elements.

Initially, the number of zero elements is zero ( $k = 0$ ). Loop through the array elements. If a zero element is encountered, then the number of zero elements is increased by 1 ( $k := k + 1$ ). If the number of zero elements is less than or equal to 4, then remove the next zero element. If a fifth zero element ( $k > 4$ ) is found, then exit

early from the loop (further processing of the array is not needed).

The flowchart is shown in Figure 5.36.

The commented code is shown below.

```

const
  n = 20;
var
  X: array [1..n] of integer;
  k, i, j: integer;
begin
  for i := 1 to n do
  begin
    readln(X[i]);
    write('X[',i,'] = ');
  end;
  k := 0; { Number of zero elements. }
  j := 1; { Element index in array X. }
  while j <= n do {Loop until end of array.}
  begin
    if x[j] = 0 then { If element is zero, then }
    begin
      k := k + 1; { count it }
      if k > 4 then { If k exceeds 4 }
        break { then exit loop early. }
      else { else, remove j-th element.}
        for i := j to n - k do
          X[i] := X[i + 1];
    end
    { If a nonzero element encountered, go to next. }
    else
      j := j + 1; {If the element is nonzero.}
  end;
  { Print the modified array. }
  for i := 1 to n - k do
    write(X[i], ' ');
end.

```

**EXAMPLE 5.7.** Find the sum of the prime numbers in an integer array C(N).

The idea of the algorithm is as follows. Initially, the sum is equal to 0. Loop through the elements. If an element is a prime number, then add it to the sum. To determine if a number is prime, write the prime function. The flowchart for this function is shown in Figure 5.37.

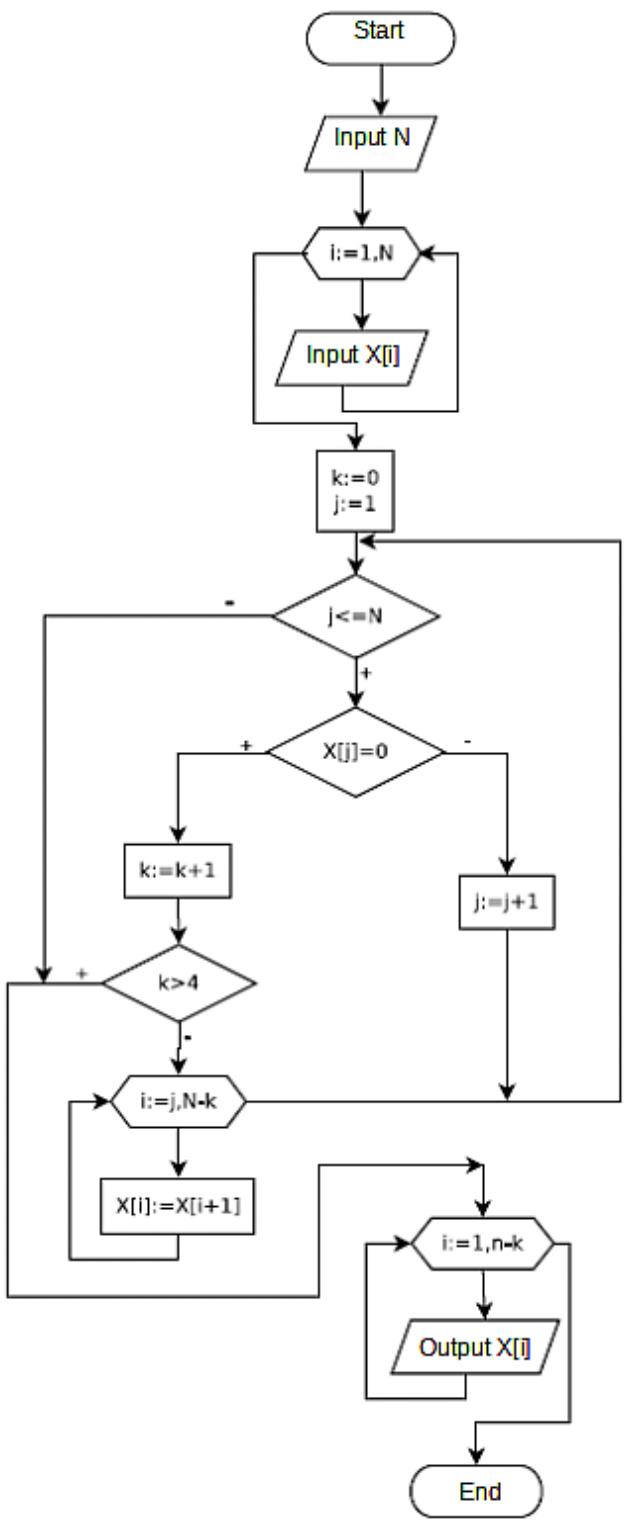


Figure 5.36: Algorithm for Example 5.6

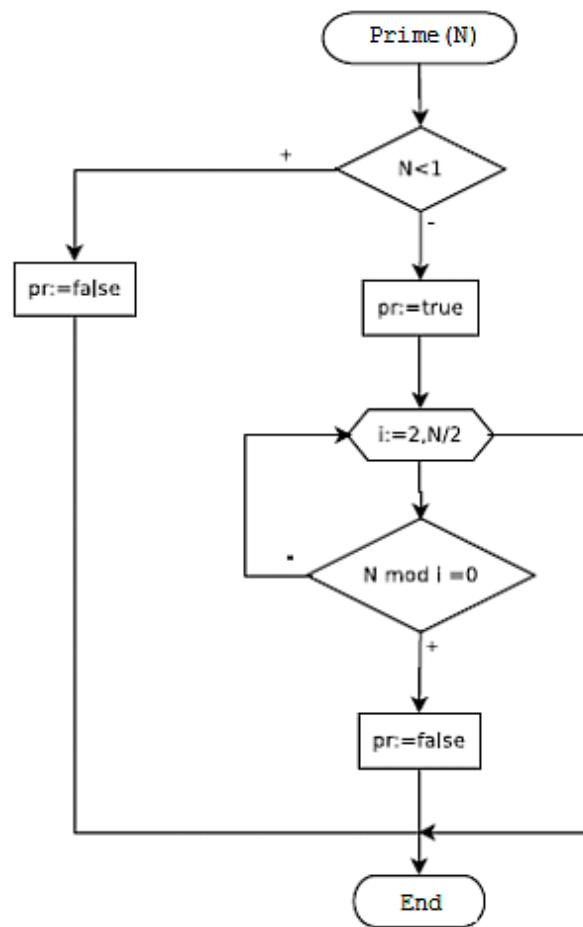


Figure 5.37: Prime number function flowchart

The prime function header is:

```
function prime(N: integer): boolean;
```

The function returns **true** if N is prime. Otherwise, the function returns **false**.

The flowchart for solving Example 5.7 is shown in Figure 5.38.

Below is the commented code that implements this algorithm.

```

program mas7;

function prime(N: integer): boolean;
var
  i: integer;
  ok: boolean;

```

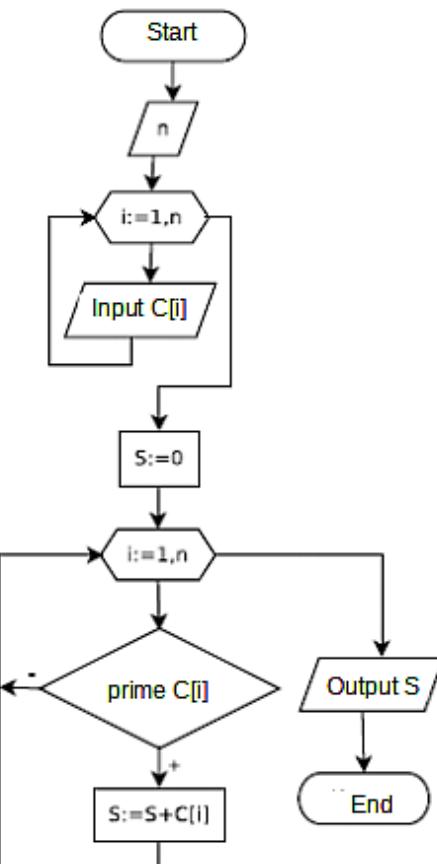


Figure 5.38: Flowchart for Example 5.7

```

begin
  if N < 1 then
    ok := false
  else
    begin
      ok := true;           { Assume N is prime. }
      for i := 2 to N div 2 do
        if (N mod i = 0) then { If divisor other than 1 or N }
          begin
            ok := false;     { then return false }
            break;           { and exit loop early. }
          end;
    end;
    prime := ok;
  end;

var
  c: array [1..50] of integer;
  i, n: byte;
  
```

```

S: word;
begin
  write('Enter the number of elements in array C: ');
  readln(n);
  for i := 1 to n do
    begin
      write('Enter C[', i, ']: ');
      readln(C[i]);
    end;
  S := 0;
  for i := 1 to n do
    if prime(C[i]) then { If the number is prime }
      S := S + C[i]; { add it to S. }
  writeln; { Display the final sum. }
  writeln('Sum of prime numbers in array is: ', S);
end.

```

EXAMPLE 5.8. Determine if a given array has a sequence of elements with alternating signs (Figure 5.39). If so, output to screen the number of such sequences.

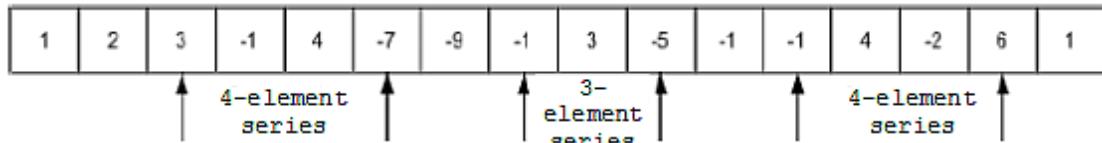


Figure 5.39: Array with 3 sequences of alternating signs

The idea of the algorithm is as follows. Initially, the number of sequences (*kol*) is equal to zero, and the length of a sequence (*k*) is equal to one<sup>9</sup>. Looping sequentially from 1 to *n*-1, adjacent elements are compared (first and second, second and third, ..., and penultimate and last).

If the product of adjacent elements is negative, the number of elements in the sequence<sup>10</sup> will be increased by 1. If the product is not negative, then there two possibilities: either the end of a sequence of alternating elements has been “cut off”, or a pair of elements of the same sign has been found.

Determining which one it is can be done by comparing *k* with one. If *k* > 1, then the end of a sequence of alternating elements broke off and *kol* must be increased by 1. After the break in the sequence, the number of elements in it is set equal to one (*k* = 1).

After exiting from the loop check if there is a sequence of characters with alternating signs at the end of the array. If there is such a sequence ( $k > 1$ ), then the number of sequences (kol) should be increased by 1 again. To finish, display kol, the number of sequences with alternating signs. The flowchart for solving Problem 5.8 is shown in Figure 5.40.

Below is the code for a console application in Free Pascal.

```

var
  x: array [1..50] of real;
  n, i, k, kol: integer;
begin
  write('Number of elements in array: ');
  readln(n);
  for i := 1 to n do
  begin
    write('x[',i,'] = ');
    readln(x[i]);
  end;
  k := 1;           { k=1, since min sequence = 2 elements. }
  kol := 0;         { Sequence counter }
  for i := 1 to n - 1 do
  if x[i]*x[i+1] < 0 then { If product negative, signs differ }
    k := k + 1           { Increase sequence element counter. }
  else
  begin
    { If sequence broken, increment sequence counter }
    if k > 1 then kol := kol + 1;
    k := 1;             { Reset sequence element counter. }
  end;
  if k > 1 then { Check for sequence at end of array. }
  kol := kol + 1; { If so, increment counter. }
  if kol > 0 then
    write('Number of alternating sequences = ', kol)
  else
    write('No alternating sequence.')
end.

```

Next, we will consider a slightly more difficult problem about sequences.

**EXAMPLE 5.9.** Find the longest sequence of ones in a given array.

Save the length of the longest sequence in *max* and the index of its last element in *kon\_max*.

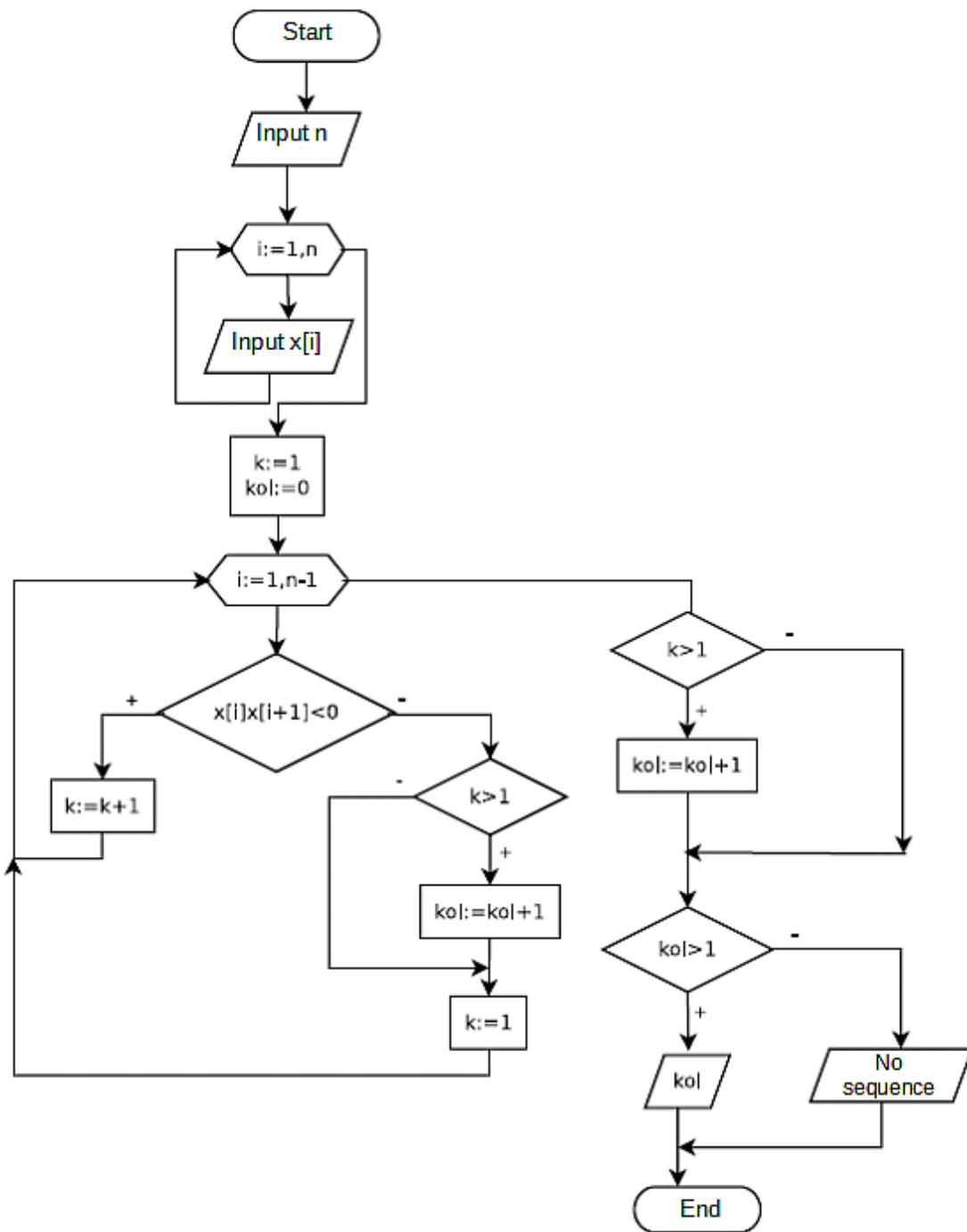


Figure 5.40: Flowchart for Example 5.8

This problem is like the previous one, except that it must be established that not only has the sequence ended, but the sequence itself must also be saved. A sequence can be characterized by two of three parameters: the first element of the

sequence, the last element of the sequence, and the length of the sequence. The last element of the sequence (*kon*) and its length (*k*) will be used because the sequence will be defined by when it terminates.

The algorithm for solving this problem is as follows. First, set the number of sequences (*kol*) and the sequence length (*k*) to zero. Looping through the elements, if the current element is 1, then increment the sequence<sup>11</sup> length. If the current element is not 1, then there are two possible cases: a sequence of ones has ended, or an element that is not 1 was found. The correct case can be determined by comparing *k* with unity. If *k* > 1, then a sequence of ones just ended, and the number of sequences (*kol*) must be increased by 1. Also, save the end of the sequence (*kon := i - 1*) and the length of the sequence (*length := k*).

After that, check the number of sequences. If this is the first sequence (*kol = 1*), then assume it is the longest, save the length *k* of the current sequence in *max*, and save *kon* (the last element in the current sequence) in *kon\_max*. If this is not the first sequence (*kol > 1*), then compare the length *k* of the current sequence with the length *max* of the current longest sequence. And if *k > max*, then we assume that the current sequence is the longest (*max := k*; *kon\_max := kon*). If the current element is not equal to one, set the number of elements in the sequence to zero (*k := 0*).

After exiting the loop, check if there was a sequence of ones at the end of the loop. If there was a sequence was at the end, then process it the same manner as a sequence in the loop.

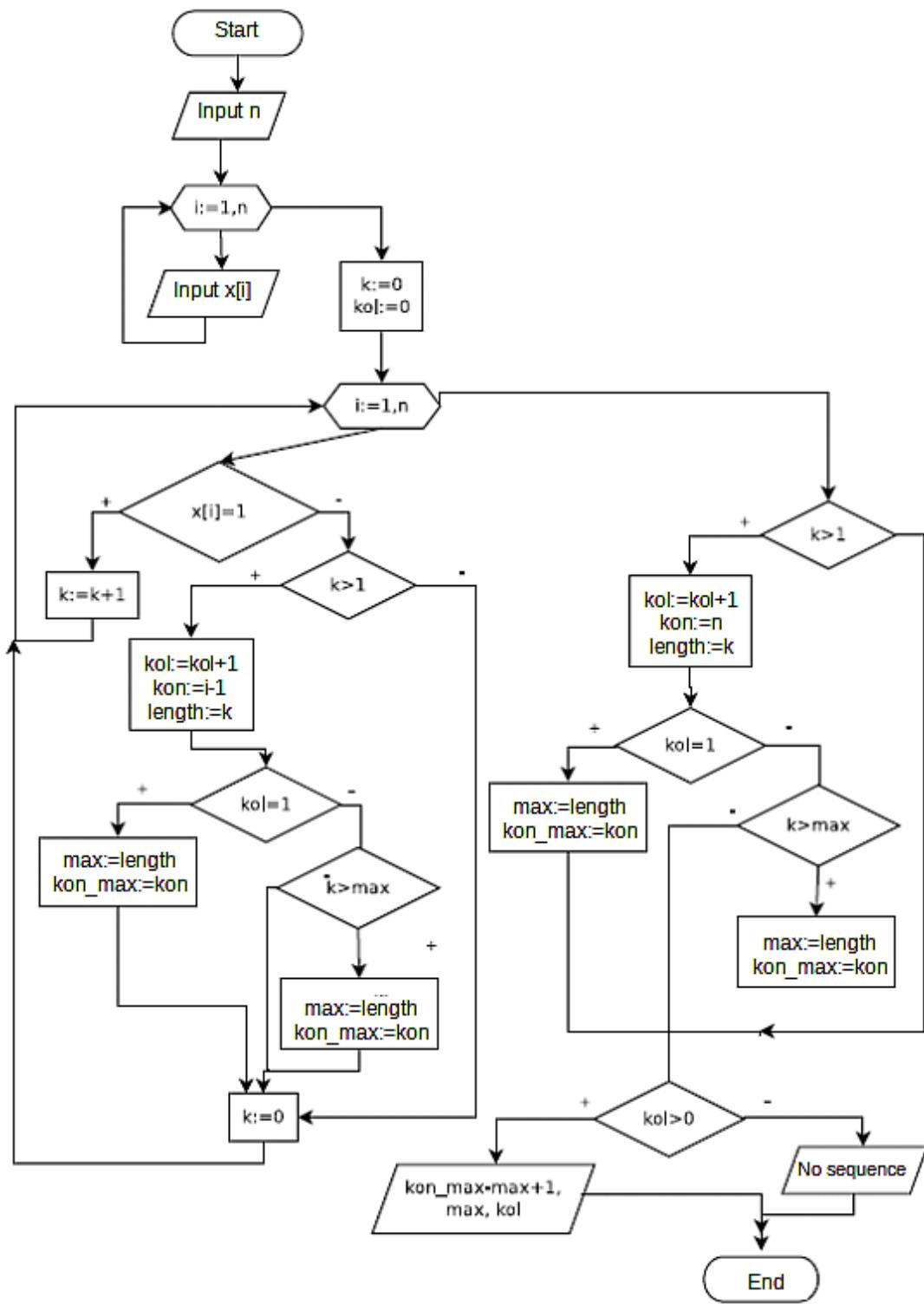
The flowchart for solving the problem is shown in Figure 5.41.

The code for solving the problem is shown below.

```

const
  T = 1;                                { Target sequence element }
var x: array [1..50] of integer;
n, i, k, kol, kon, max, kon_max, length: integer;
begin
  write('Enter array size: ');
  readln(n);
  writeln('Array X:'); { Input array }
  for i := 1 to n do
  begin
    write('X[',i,'] = ');
    readln(x[i]);
  end;
  k := 0;                                { Sequence length. }
  kol := 0;                                { Number of sequences. }
  for i := 1 to n do
    if x[i] = T then                      { Loop through array }

```



*Figure 5.41: Flowchart for Example 5.9*

```

k := k + 1           { increment sequence length. }
else                { If current element not 1, then }
begin
  if k > 1 then      { if ongoing sequence just ended }
    begin
      kol := kol + 1;
      kon := i - 1;
      length := k;
      if kol = 1 then
        begin
          max := length;
          kon_max := kon;
        end
      else
        if k > max then
          begin
            max := length;
            kon_max := kon;
          end;
    end;
    k := 0;             { Reset sequence length to 0. }
  end;    { If there is a sequence at the end of the array }
if k > 1 then
begin
  kol := kol + 1;      { increment number of sequences, }
  kon := n;              { save end index of sequence }
  length := k;           { and save sequence length. }
  //Process as in loop.
  if kol = 1 then
    begin
      max := length;
      kon_max := kon;
    end
  else
    if k > max then
      begin
        max := length;
        kon_max := kon;
      end;
end;
// Display results
if kol > 0 then      { If a sequence was found }
begin
  writeln('Number of sequence of ', T, ' = ', kol);
  writeln('The longest sequence starts at index ',
         kon_max - max + 1, ', and ends at index ', kon_max, '.');
  writeln('Its length is ', max);
end
else
  writeln('No sequence of ', T, ' found.');
end.

```

EXAMPLE 5.10. Translate all elements of an array of real numbers to a base-p number system.

Before solving the problem, let us figure out the algorithm to convert a real number from decimal to another number system. This algorithm can be divided into the following stages:

- 1) Separate the number into integer and fractional parts.
- 2) Convert integer part of number to other number system.
- 3) Convert fractional part of number to other number system.
- 4) Combine integer and fractional parts in other number system.

### **Algorithm to convert integer to another number system**

Divide the integer by the base of the new number system. This will produce a quotient and a remainder. The remainder of the division will be the least significant (rightmost) digit of the converted number. It must be multiplied by 10 to the zero power, to correct its order.

If the quotient is not zero, then continue dividing. The new remainder will become the next digit (from the right) of the converted number, but it must be multiplied by 10 to the first power, etc, to correct its order. Continue dividing until the quotient becomes 0.

A characteristic of this algorithm is that the converted digits are produced in reverse order, from the least to the most significant (rightmost to leftmost) digit. They can then be rearranged into the correct order by multiplying them by powers of 10, and thus converted to the new number system, all in a single pass.

### **Algorithm to convert fractional part to another number system**

Multiply the fractional part of the number by the base of the new number system. The most significant (leftmost) bit of the converted fractional part will be the integer part of the resulting product, multiplied by  $10^{-1}$ . Multiply the fractional part again by the base of the new number system. The next digit will be the integer part of the product, multiplied by  $10^{-2}$ . Multiply the fractional part again by the base of the new number system, until the required number of decimal places is obtained.

The flowchart for the function to convert a real number N from the decimal system to another system is shown in Figure 5.42.

Note how raising 10 to the required power is implemented in the flowchart and in the function. Powers of 10 are used when converting the integer part of the original number, starting with 0. The result of raising 10 to the power, starting with 1, is saved in the variable q, which is then multiplied by 10 in each iteration of the loop. When converting the fractional part of the number, negative powers of 10 are needed:  $10^{-1}$ ,  $10^{-2}$ , .... Thus, when converting the fractional part of the number, the variable q is set = 0.1, which is divided by 10 in each loop iteration.

The commented code for solving Problem 5.10 is shown below.

```

{ Function to convert a real number N to a base-p number.
Input parameters:
N - real number to be converted,
p - new base number (integer),
kvo - number of digits in fractional part of converted number. }

function convert(N: real; P: word; kvo: word): real;
var
  i, N1, ost: word;
  s1, N2, r, s2: real;
  q: real;
begin
  if N < 0 then      { If N is negative, convert recursively. }
    r := - convert(abs(N), P, kvo)
  else
    begin
      N1 := trunc(N);           { Extract integer part N1 of N. }
      N2 := frac(N);           { Extract fractional part N2 of N. }
      s1 := 0;                  { Set converted integer part = 0. }
      s2 := 0;                  { Set converted fractional part = 0. }

      // Convert integer part
      q := 1;                  { Store  $10^i$ , starting with i=0. }
      while (N1 >> 0) do       { Loop until N1 = 0. }
      begin
        ost := N1 mod P; { Remainder=next digit from right }
        s1 := s1 + ost * q; { Multiply by  $10^i$  to reverse order. }
        N1 := N1 div P;
        q := q * 10;         { Raise q by  $10^1$ . }
      end;

      // Convert fractional part
      q := 0.1;              { Store  $10^i$ , starting with i=-1. }
      for i := 1 to kvo do
      begin
        N2 := N2 * P;          { Multiply fractional part by 10. }
        s2 := s2+trunc(N2)*q; { Next digit = integer part of N2*P. }
      end;
    end;
end;

```

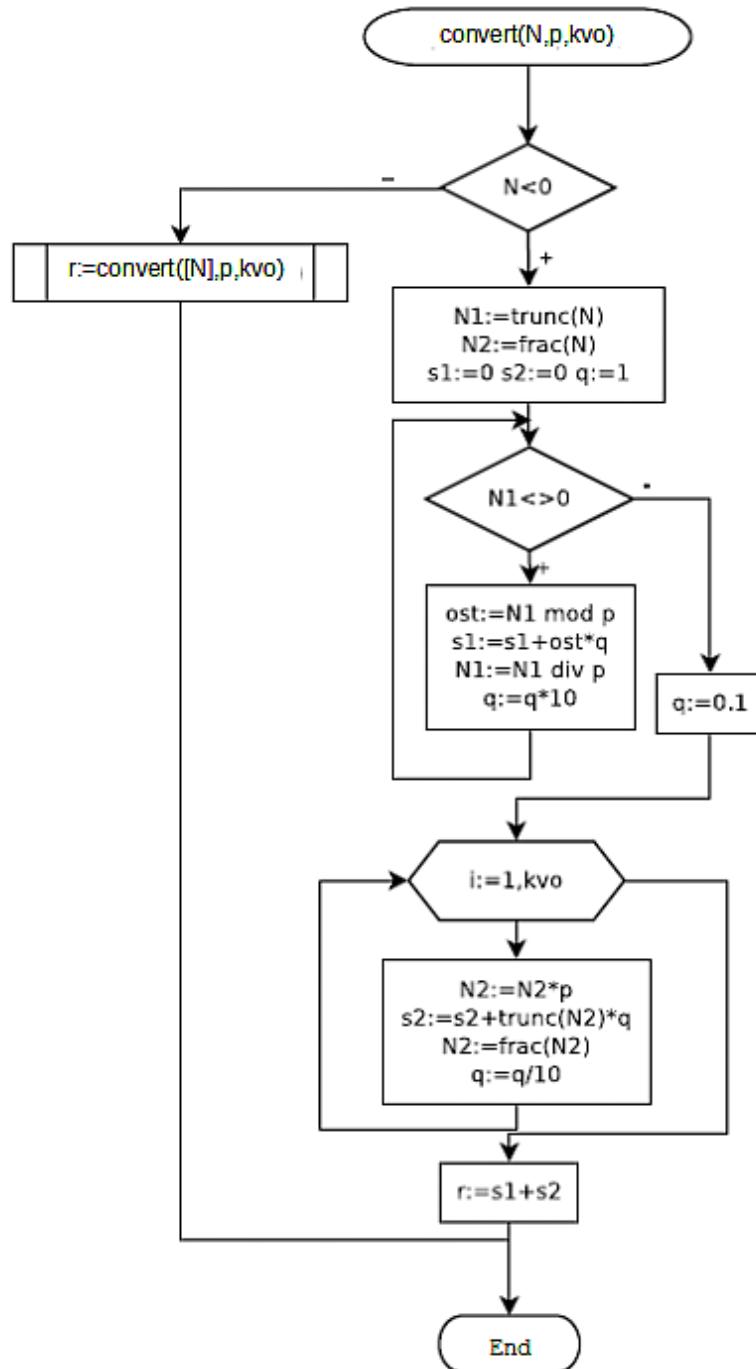


Figure 5.42: Convert real number to base-p number system

```

N2 := frac(N2);
q := q / 10;
end;
r := s1 + s2;      { Add parts in base-p system. }
  
```

```
end;
convert = r;
end;

// Main program
var
  C: array [1..100] of real;
  p, i, n, kvo: word;
begin
  write('Enter size of array: ');
  readln(n);
  writeln('Enter array C'); { Input array. }
  for i := 1 to n do
    begin
      write('C[',i,'] = ');
      read(C[i]);
    end;
  write('Enter the new base: ');
  readln(p);
  write('Enter number of decimal places: ');
  readln(kvo);
  for i := 1 to n do { Convert array to new base. }
    c[i] := convert(C[i], p, kvo);
  writeln('Converted array C:'); { Display converted array. }
  for i := 1 to n do
    write(C[i]:1:kvo, ' ');
end.
```

```
Enter size of array: 8
Enter array C:
C[1] = 5.7
C[2] = 9.905
C[3] = -78.946
C[4] = 76.894
C[5] = -11.1111
C[6] = 11.11
C[7] = 0.89654
C[8] = -0.1324
Enter the new base: 5
Enter number of decimal places: 5

Converted array C:
10.32222 14.42303 -303.43311 301.42133 -21.02342 21.02333
0.42201 -0.03123
```

Figure 5.43: Console program output for Example 5.10

EXAMPLE 5.11. From an array Y of positive integers, remove the last 3 numbers which, when in octal form, have digits that form a decreasing sequence.

To solve this problem, a function that checks if the digits of the octal form of a number form a decreasing sequence, is needed.

The title of this function will be:

**function octal(N: word): boolean;**

The function's input (N) is a decimal integer. The function returns true if the digits of the octal form of N form a descending sequence, and false otherwise. When developing an algorithm for this problem, remember that when converting decimal numbers to octal form, the converted digits will be produced in reverse order. Thus, the function should return true when the converted digits form an increasing sequence.

The commented code for the function is shown below.

```
function octal(N: word): boolean;
var
  ok: boolean;
  digit, digit_st: word;
  i: integer;
begin
  i := 0;
  { Assume the digits in converted N form a decreasing sequence. }
  ok := true;
  while N <> 0 do          { while N is not 0, }
    begin
      digit := N mod 8;      { get next digit from right }
      N := N div 8;
      i := i + 1;
      if i > 1 then          { If digit is not first }
        { Remember: converted digits are produced in reverse order. }
        if digit <= digit_st then
          begin
            ok := false;
            break;
          end;
        digit_st := digit;
    end;
  octal := ok;
end;
```

The algorithm for solving the problem is as follows. Loop over all the numbers in

the array in reverse order. Check if the digits of the octal form of the current element form a decreasing sequence. If they do, then the quantity of such numbers ( $k$ ) is increased by 1. If  $k \leq 3$ , then delete the current array element.

Create a graphical application to solve Problem 5.11. Place the following components on a form: three buttons, three labels, a text box and two string grids. Arrange them as shown in Figure 5.44.

The component properties are shown in Tables 5.9 and 5.10.

*Table 5.9: Properties for labels, buttons, and text box in example*

Name	Caption (Text)	Width	Visible	Left	Top
Label1	Enter array size:	81	true	16	14
Label2	Source array:	68	false	16	45
Label3	Modified array:	80	false	16	120
Edit1	7	40	true	112	8
Button1	OK	75	true	176	8
Button2	Remove numbers	110	false	16	200
Button3	Exit	75	false	176	200

*Table 5.10: Row table properties in example*

Name	ColCount	RowCount	Visible	FixedCols	FixedRows	Options.goEditing
StringGrid1	7	1	false	0	0	true
StringGrid2	7	1	false	0	0	false

When the application starts, Label1, Edit1 (the text box for entering the size of the array) and Button1 are visible.

When the OK button (Button1) is clicked, the size is read from the Edit1 text box and the following components become visible: two other buttons, Label2 and StringGrid1 for entering the array elements. Label1, Edit1 (the text box for entering the size of the array size) and Button1 become invisible. After clicking on the OK button (Button1), the application window should look like that shown in Figure 5.45.

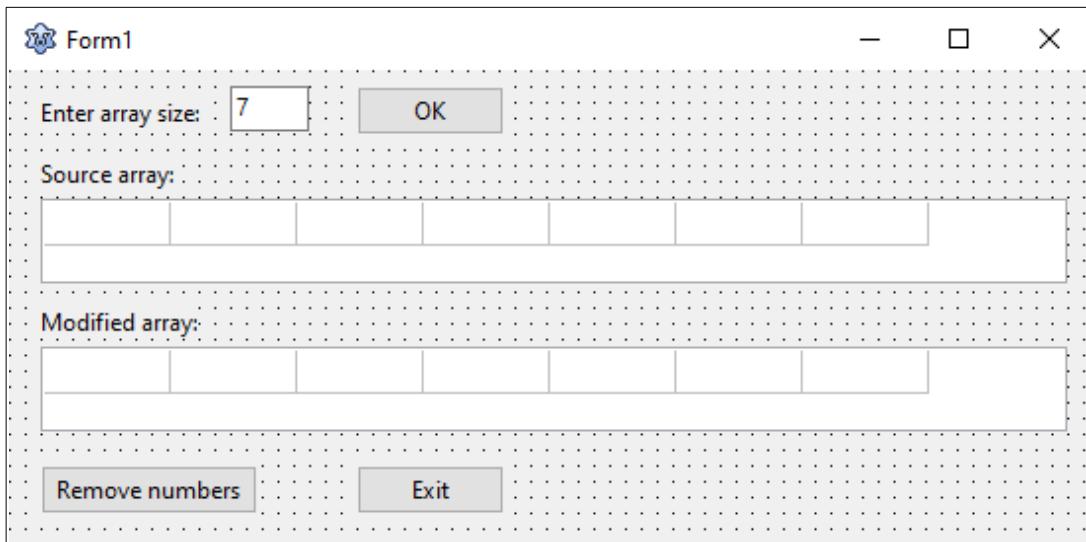


Figure 5.44: Form for Example 5.11

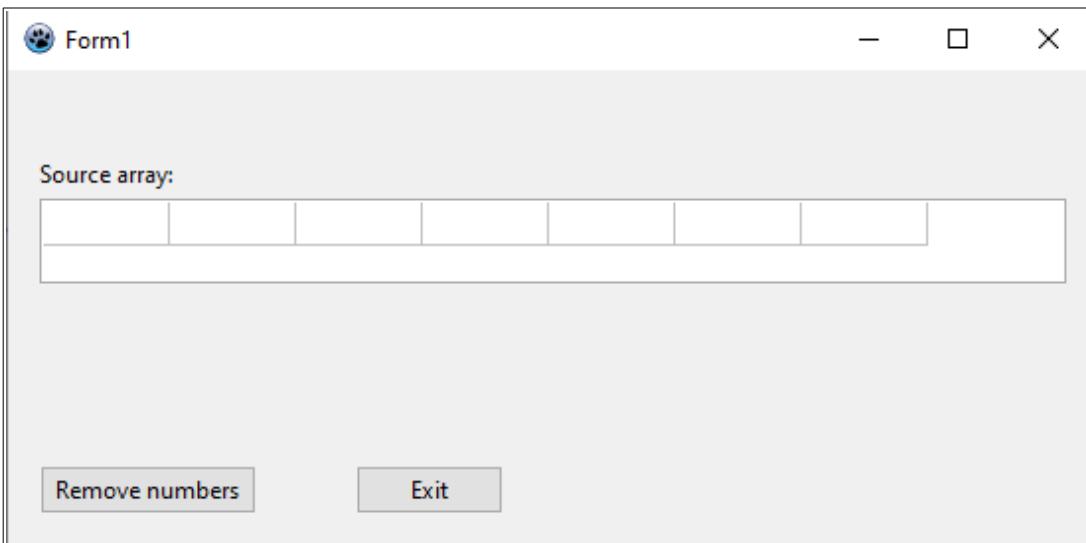


Figure 5.45: Application window after clicking the OK button

When the **Remove numbers** button is clicked, the following actions will occur:

- the array is read from StringGrid1;
- the deletion from the array of the last 3 numbers, which in their octal form, have digits that form a decreasing sequence;
- Label3 and StringGrid2 become visible for displaying the elements of the modified array.

The unit code with essential comments are shown below:

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls, Grids;
{Form declaration.}
type
{TForm1}
TForm1 = class(TForm)
  Button1: TButton;
  Button2: TButton;
  Button3: TButton;
  Edit1: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  StringGrid1: TStringGrid;
  StringGrid2: TStringGrid;
  procedure Button1Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
  procedure Button3Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
type
arr = array [1..100] of word;
var
  Form1: TForm1;
  N: word;
  X: arr;
implementation
{TForm1}
{ Function octal checks if digits of the octal form of N, form a
decreasing sequence. }

function octal(N: word): boolean;
var
  ok: boolean;
  digit, digito: word;
  i: word;
begin
  i := 0;
  digito := 0;
  { Assume digits in octal form of N form a decreasing sequence. }
  ok := true;
  while N <> 0 do          { While N is not 0, }
begin
  digit := N mod 8;         { get next digit from right }
```

```

N := N div 8;
i := i + 1;
if i > 1 then      { If the digit is not the first }
{ Note: converted digits are produced in reverse order.
  Check for opposite condition while digits are produced. }
  if digit <= digito then    { If sequence decreasing }
begin
  ok := false;
  break;
end;
digit_st := digit;
end;
octal := ok;
end;

// Function to remove element with index m from array X(N).
procedure remove(var X: arr; m: word; var N: word);
var
  i: word;
begin
  for i := m to N - 1 do
    x[i] := x[i + 1];
  N := N - 1;
end;

// Handler for clicking the OK button.
procedure TForm1.Button1Click(Sender: TObject);
begin
  N := StrToInt(Edit1.Text); { Read array size. }
  Label1.Visible := false; { Array size label invisible }
  Edit1.Visible := false; { Edit box invisible }
  Button1.Visible := false; { OK button invisible }
  Label2.Visible := true; { Source array label visible }
  StringGrid1.Visible := true; { Source grid visible }
  StringGrid1.ColCount := N; { Set number of columns. }
  Button2.Visible := true; { Delete numbers button visible }
  Button3.Visible := true; { Exit button visible }
end;

// Event handler for "Remove numbers" button
procedure TForm1.Button2Click (Sender: TObject);
var
  k, i: word;
begin
  for i := 0 to N - 1 do { Read array from string grid. }
    X[i + 1] := StrToInt(StringGrid1.Cells[i, 0]);
  k := 0; { Counter of decreasing sequences. }
  for i := N - 1 downto 0 do { Loop thru array in reverse. }
    if octal(x[i]) then
begin
  k := k + 1; { Increment counter. }
  if k <= 3 then { Delete up to 3 numbers. }

```

```
    remove(x, i, N);
end;
Label3.Visible := true;           { Exit button visible }
StringGrid2.Visible := True;      { Output string grid visible }
StringGrid2.ColCount := N; { Set number of columns }
for i := 0 to N - 1 do          { Output the modified array. }
  StringGrid2.Cells[i, 0] := IntToStr(X[i]);
end;

{ Exit button handler }
procedure TForm1.Button3Click (Sender: TObject);
begin
  Close;
end;

end.
```

Figure 5.46 shows the application window of the program for Example 5.11.

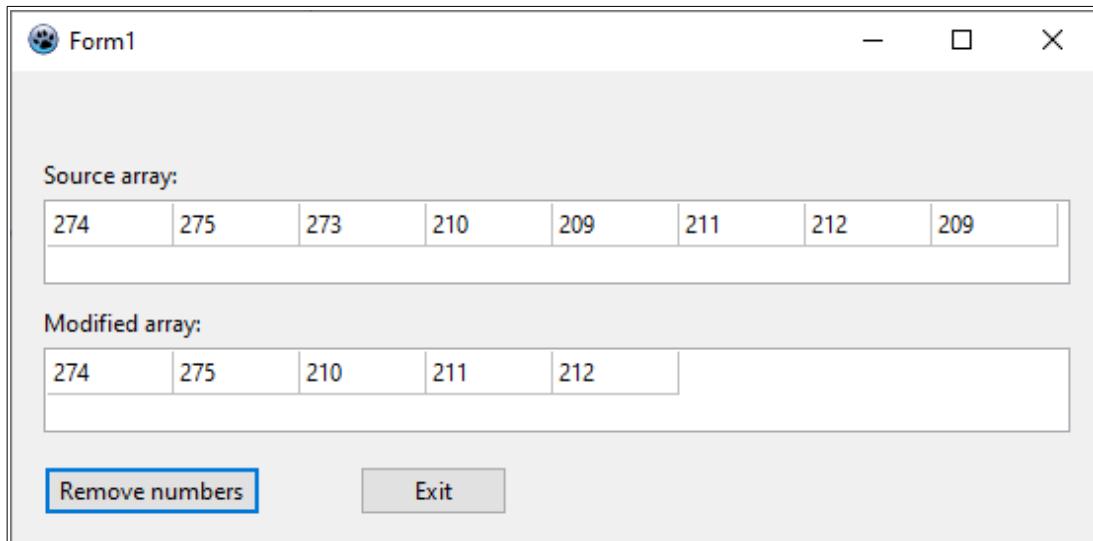


Figure 5.46: Window with results for Example 5.11

This section on processing arrays ends with this example, and with a suggestion that the reader independently solve the problems below.

## 5.14 Exercises

- 1) Copy the positive elements in array X to a row in array Y. Compute the sum of the elements in array X and the product of the elements in array Y. Remove elements from array Y that are between the largest and smallest elements.

- 2) Create array B by copying into it the elements in array A with odd indices. Calculate the arithmetic mean of the elements in array B and remove the largest, smallest and fifth elements from it.
- 3) Copy the first five positive elements and the last two prime number elements in an array of integers X, to array Y. Find the maximum negative element in array X.
- 4) Copy the elements in array X satisfying the condition  $1 \leq x_i \leq 2$ , to a row in array Y. Swap the maximum and minimum elements in array Y.
- 5) Copy the elements in an array of integers X to array Y, in reverse order. Find the number of even, odd and zero elements of array Y.
- 6) Find the maximum and minimum elements among the positive odd elements in the integer array X. Remove all zero elements from the array.
- 7) Copy the elements of integer array  $X = (x_1, x_2, \dots, x_{12})$  to array  $Y = (y_1, y_2, \dots, y_{12})$ , while moving the elements of array X three places to the right. Three elements at the end of array X are moved to the beginning:  $(y_1, y_2, \dots, y_{12}) = (x_{10}, x_{11}, x_{12}, x_1, x_2, \dots, x_9)$ . Find the index of the largest prime number and smallest positive element in arrays X and Y.
- 8) Copy the elements of array  $X = (x_1, x_2, \dots, x_{15})$  while moving its elements four places to the left, to array  $Y = (y_1, y_2, \dots, y_{15})$ . Four elements at the beginning of X are moved to the end:  $(y_1, y_2, \dots, y_{15}) = (x_5, x_6, \dots, x_{15}, x_1, x_2, x_3, x_4)$ . Swap the smallest and largest elements in array Y.
- 9) Find the number of elements that are less than the arithmetic mean in array X. Remove the positive elements located between the maximum and minimum elements.
- 10) Calculate the arithmetic mean of the elements of the array X located between its minimum and maximum values. If the minimum element is located before the maximum, then then rearrange the elements between them in increasing order.
- 11) Determine if a given array contains a group of elements arranged in increasing order by value. If yes, then determine the quantity such groups.
- 12) Find the smallest sequence of odd numbers in a given array of integers.
- 13) Remove all prime numbers that occur before the maximum value in an array of integers.
- 14) Remove the penultimate group of elements with alternating signs from an array.
- 15) Find the number of the perfect numbers in an array of positive integers X.

Remove the last two negative numbers from the array. Create array Y and put in it the index of elements in array X that are prime numbers.

- 16) Copy the positive elements of an array of integers X to array Y, in reverse order. Calculate the percentage of even, odd and zero elements in array Y. Convert elements in array Y to the binary system.
- 17) Find the maximum and minimum elements among the positive even elements in the integer array X. Remove from array X the perfect numbers located after the maximum.
- 18) X and Y are arrays of real numbers. Create an array Z and put in it the positive elements from X and Y, converted the base-7 number system. Find the index of the maximum and minimum elements in array Z.
- 19) Copy the positive even elements from integer arrays X and Y to array Z. Swap the minimum and maximum elements in array Z. Convert the elements in array Z to the base-4 number system.
- 20) Remove all numbers greater than the average of the prime numbers in integer array X.
- 21) The arrays X and Y of real numbers contain the coordinates of points in a plane. Find the two closest points.
- 22) Determine if a given array of real numbers contains a group of elements in decreasing order by value. If yes, then find the smallest such group.
- 23) In a given array of integers, find the largest sequence of even elements.
- 24) Remove from an array of integers all elements that do not contain the digit zero when converted to a base-5 number system.
- 25) Create array C from arrays A and B of real numbers, by copying to it elements of arrays A and B which do not contain the digit seven, when converted to the base-8 number system.

---

#### Endnotes:

- 1 The TStringGrid component may be found on the Additional tab in the Component Palette.
- 2 When sorting in descending order, the element with the lower index cannot be smaller than the element with a higher index, after rearranging.
- 3 The name of the algorithm comes from the similarity with the movement of bubbles in the reservoir of water, as each bubble finds its own level.
- 4 And shift part of the array one element to the left.
- 5 When declaring array X, size it for inserting an additional element.
- 6 See Section 2.4.9 for the array data type and its declaration. Working with arrays is described in

detail in this chapter.

- 7 See Section 2.4.9 for the string data type and its declaration.
- 8 See Sections 2.4.9 and 5.2 for declaring static arrays.
- 9 This is because the minimum number of elements in a sequence is 2.
- 10 Number of consecutive elements with alternating signs.
- 11 Number of consecutive ones.

This page deliberately left blank.

## Chapter 6. Matrix Processing in Pascal

A matrix is a two-dimensional array, each element of which has two indices: the row number and the column number. You can declare a two-dimensional array (matrix) this way:

**name: array [index1\_begin..index1\_end, index2\_begin..index2\_end] of type;**

where

- *type* defines the type of array elements,
- *name* is the matrix name,
- *index1\_begin..index1\_end* is the range of row numbers,
- *index2\_begin..index2\_end* is the range of column numbers of the matrix.

For example,

**var h: array [0..11, 1..10] of integer;**

This declares a matrix of integers *h*, consisting of twelve rows and ten columns (rows are numbered from 0 to 11, and columns from 1 to 10).

There is another way to declare matrices, by creating a new data type:

```
type
  new_type = array [index1_begin..index1_end] of type;
var
  name: array [index2_begin..index2_end] of new_type;
or
type
  new_type = array [range_list] of type;
var
  name: new_type;
```

For example:

```
type
  arr = array [1..30] of integer;
  matrix = array [0..15, 0..13] of real;
var
  a, b: array [1..10] of arr;
  c: matrix;
```

In this case, matrices *a* and *b* have 10 rows and 30 columns, and matrix *c* has 16 rows and 14 columns.

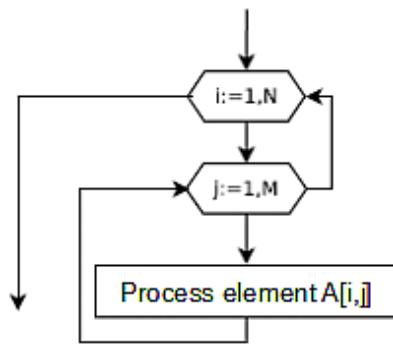


Figure 6.1: Row-by-row matrix processing

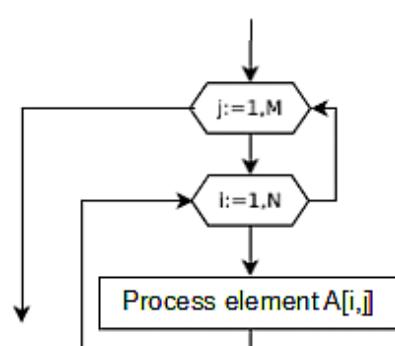


Figure 6.2: Column-by-column matrix processing

To refer to an element of a matrix, specify its name, and in square brackets separated by commas, its row and column numbers:

`name[row_num, column_number]`

or

`name[line_num][column_num]`

For example,  $h[2,4]^1$  is the element of matrix  $h$  located in the row numbered two and column numbered four.

To process all the elements of the matrix, two loops must be used. To process the matrix row by row, the outer loop should iterate sequentially over the rows from the first to the last, and the inner loop should iterate over the elements of each row in turn. To process the matrix column by column, the outer loop should iterate over the columns, and the inner loop over the rows. Figure 6.1 shows the flowchart for processing a matrix by rows and Figure 6.2 by columns. Here  $i$  is the row number,  $j$  is the column number,  $N$  is the number of rows, and  $M$  is the number of columns on matrix  $A$ .

Consider the main operations performed on matrices when solving problems.

## 6.1 Element Input / Output

Matrix input/output, like arrays, must be done element by element. First, enter the dimensions of the matrix, and then enter the elements using a double loop. The flowchart for entering matrix elements into the matrix is shown in Figure 6.3.

Output can be done by row or by column, but it is better if elements are arranged row by row, as shown below, for example,

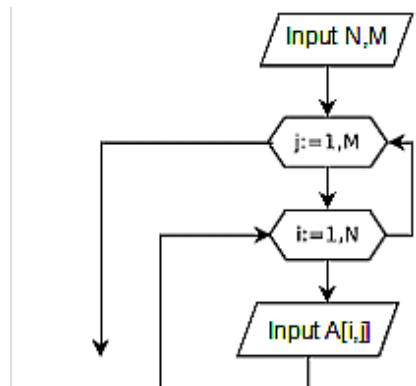


Figure 6.3: Matrix element input

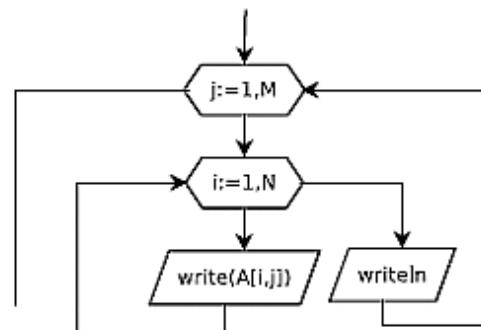


Figure 6.4: Matrix output by rows

2	3	13	35
5	26	76	37
52	61	79	17

The algorithm for the row-by-row output of matrix elements is shown in Figure 6.4.

The declaration of matrices in Pascal was described in Section 5.2 of Chapter 5. Matrix element  $A_{ij}$  can be referred to by using the construct  $A[i,j]$  or  $A[i][j]$ .

Consider the implementation of matrix Input / Output in console applications. Use the **read** statement for row-by-row matrix input in a double loop, through rows and columns.

```

for i := 1 to N do
  for j := 1 to m do
    read(A[i,j]);
  
```

In this case, separate the elements in each row with a space or tab, for example, and press Enter only at the end of the row.

EXAMPLE 6.1. Write a console application for the matrix input of real numbers and displaying the matrix on screen.

The matrix input/output console application is shown below.

```

var
  a: array [1..20, 1..20] of real;
  i, j, n, m: integer;
begin
  
```

```

{ Enter matrix dimensions }
writeln('Enter the number of rows and columns in matrix A:');
readln(N, M);
{ Enter matrix elements. }
writeln('Enter the matrix:');
for i := 1 to N do
  for j := 1 to m do
    read(a[i,j]);
{ Output matrix elements. }
writeln;
writeln('Matrix A:');
for i := 1 to n do
begin
  for j := 1 to m do
    write(a[i,j]:8:3, ' ');      { Print a row. }
  writeln;                      { Move to the next row. }
end;
end.

```

Figure 6.5 shows the results of the program.

```

Enter the number of rows and columns in matrix A:
3 4
Enter the matrix:
1 2 3 4
5 6 7 8
10 12 17 16

Matrix A:
1.000 2.000 3.000 4.000
5.000 6.000 7.000 8.000
10.000 12.000 17.000 16.000

```

Figure 6.5: Results from the program for Example 6.1

Matrix input can also be done using the following loop.

```

for i := 1 to N do
  for j := 1 to m do
begin
  write('a(',i,',',j,') = ');
  readln(a[i, j]);
end;

```

The authors invite the reader to independently discover what will be different in this case.

For matrix input/output, a TStringGrid component (introduced in Chapter 5) can be used.

Consider the following problem, for example.

EXAMPLE 6.2. Write a program to transpose<sup>2</sup> matrix A.

The flowchart for matrix transposition is shown in Figure 6.6. Transposing matrix A(N,M) will result in matrix B(M,N).

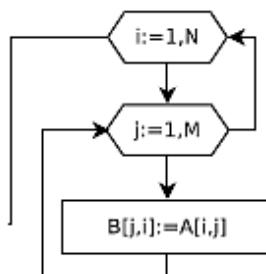


Figure 6.6: Transposing a matrix

Consider the transposition of a matrix of a fixed size A(4,3).

On a form, place labels Label1 and Label2 with Captions *Source Matrix A* and *Transposed Matrix B*, two TStringGrid components with the properties as shown in Table 6.1, and a button with caption *Transpose Matrix*.

The application form is shown in Figure 6.7.

Below is the commented code for a subroutine that will execute if the user clicks on the *Transpose Matrix* button. Note that *matrix mathematics and the TStringGrid component use different order of indices: matrix[row, col] and StringGrid.cells[col, row]*.

```

procedure TForm1.Button1Click(Sender: TObject);
const
  n = 4;
  m = 3;
var
  i, j: byte;
  A: array [1..n, 1..m] of integer;      { Source matrix }
  B: array [1..m, 1..n] of integer;      { Transposed matrix }
begin
  // Fill matrix A with data from StringGrid1 cells.

```

Table 6.1: StringGrid1, StringGrid2 properties

Property	StringGrid1	StringGrid2	Property Description
Top	10	10	Distance from the top grid border to top of form
Left	10	200	Distance from left grid border to the left edge of form
Height	120	120	Grid height
Width	180	180	Grid Width
ColCount	4	5	Number of columns
RowCount	5	4	Number of rows
DefaultColWidth	30	30	Column width
DefaultRowHeight	20	20	Row height
Options.goEditing	true	false	Allow grid cell editing

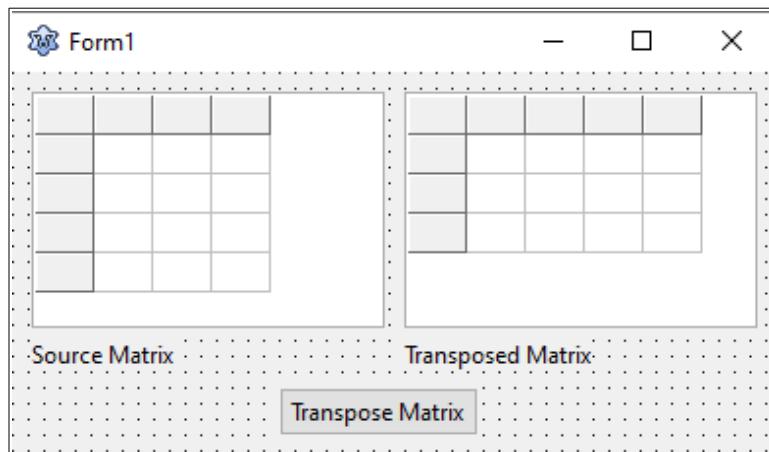


Figure 6.7: Matrix transposition form

```

for i := 1 to n do                      { Loop thru rows. }
  for j := 1 to m do                  { Loop thru columns. }
    A[i,j] := StrToInt(StringGrid1.Cells[j,i]);
// Fill transposed matrix B.
for i := 1 to n do
  for j := 1 to m do
    B[j,i] := A[i,j];
// Display matrix B.
for i := 1 to n do
  for j := 1 to m do
    StringGrid2.Cells[i,j] := IntToStr(B[j,i]);
end;

```

The results of the program are shown in Figure 6.8.

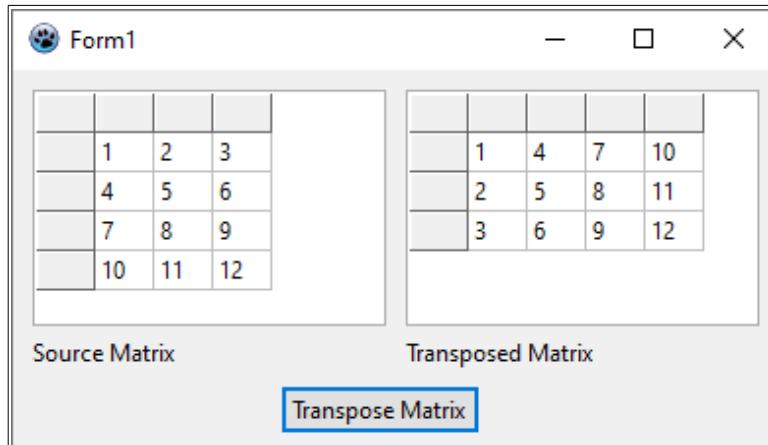


Figure 6.8: The results of transposing matrix A(3,4)

Fixed size matrices A(4,3) and B(3,4) were used to demonstrate matrix input/output using the TStringGrid component. Let us now consider the general case of solving a matrix transposition problem.

Place the following components on a form:

- Label1 with the Caption 'Enter matrix dimensions:';
- Label2 with the Caption 'Rows:';
- Label3 with the Caption 'Columns:';
- Label4 with the Caption 'Source Matrix A';
- Label5 with the Caption 'Transposed Matrix B';
- Edit1 edit box for entering N;
- Edit2 edit box for entering M;
- StringGrid1 for input of the source matrix A;
- StringGrid2 for storing and displaying transposed matrix B;
- Button1 with Caption '*Input*' to enter matrix A dimensions;
- Button2 with Caption '*Clear*' to clear the grids;
- Button3 with Caption '*Transpose*', to transpose matrix A;
- Button4 with Caption '*Exit*' to exit the program.

Components should be placed on the form as shown in Figure 6.9.

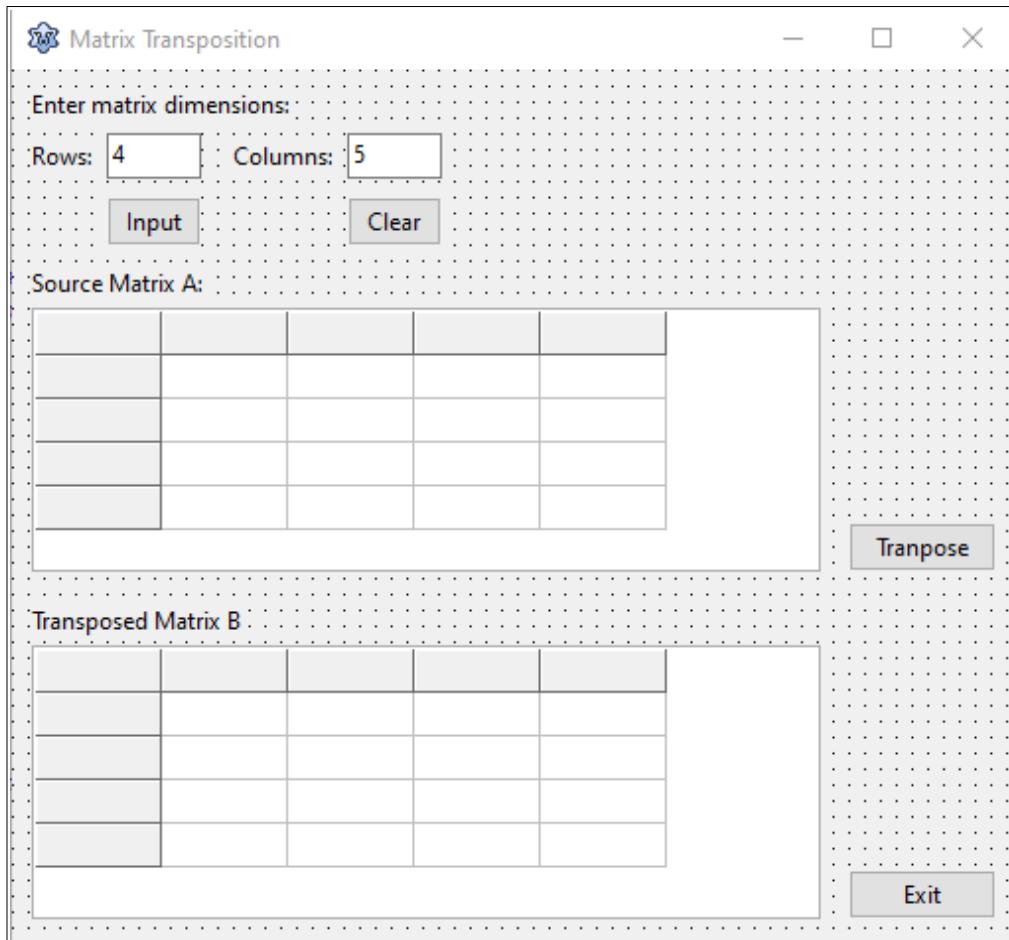


Figure 6.9: Form for matrix transposition

Set the visibility property (Visible) to *False* for Label4, Label5, StringGrid1, StringGrid2, Button2 and Button3. After that, when the program starts, the only components visible will be the components responsible for entering the matrix dimensions, and the Exit button the program (see Figure 6.10)<sup>3</sup>. Matrices A and B, their sizes N, M will be declared globally.

```
type
  { TForm1}
  {Form declaration}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
```

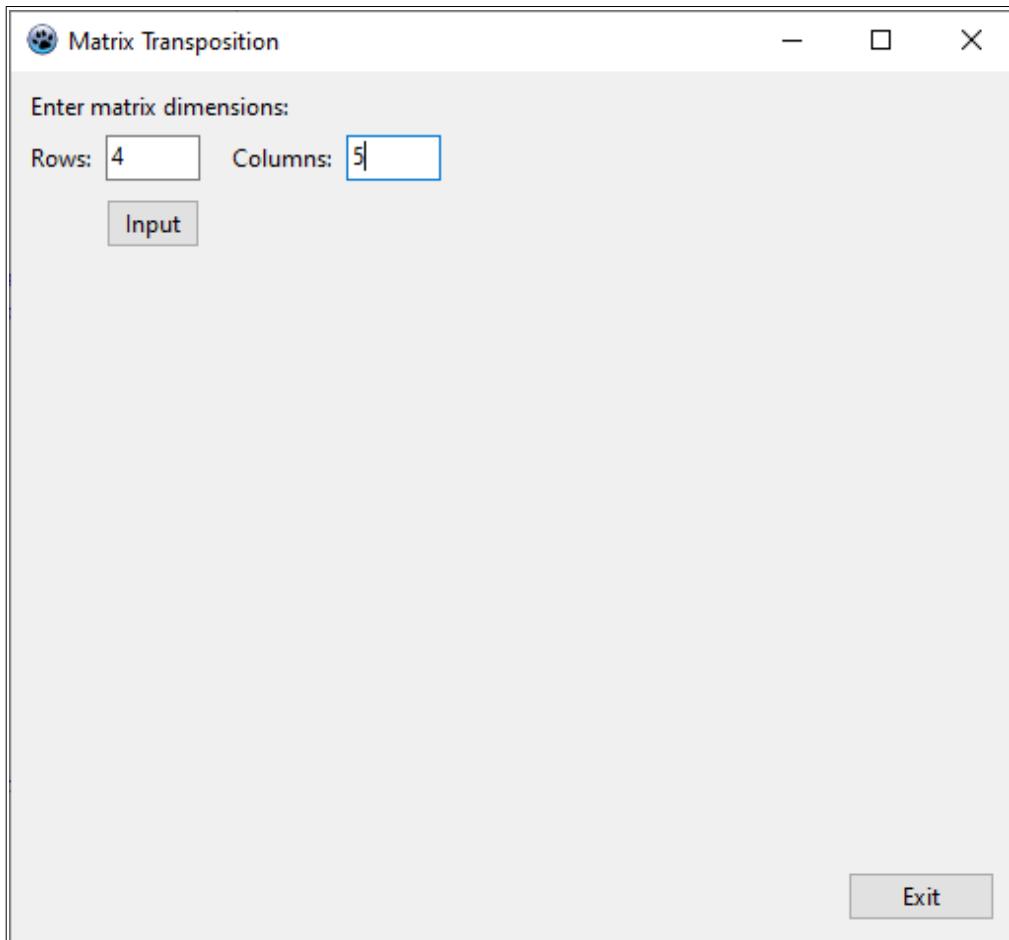


Figure 6.10: Startup window for the matrix transposition program

```
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
StringGrid1: TStringGrid;
StringGrid2: TStringGrid;
private
  {private declarations}
public
  {public declarations}
end;

var
  A, B: array [1..25, 1..25] of integer;      { Matrices A, B }
  N, M: integer;                            { and their sizes }
  Form1: TForm1;
```

The *Exit* button handler is standard and is presented below.

```
procedure TForm1.Button4Click (Sender: TObject);
begin
  Close;
end;
```

Now let us write the handler for the Input button, which must input and check the correctness of the matrix dimensions, set the properties of StringGrid1 and StringGrid2 (number of rows and columns), make StringGrid1 and the *Transpose* button visible, and make invisible the components responsible for entering the dimensions of the matrix (Label1, Label2, Label3, edit boxes fields Edit1 and Edit2, and the Input button).

```
procedure TForm1.Button1Click (Sender: TObject);
var
  i: byte;
  kn, km: integer;
begin
  { Enter the matrix dimensions. Character information is
    converted to numeric and written to }
  val(Edit1.Text, N, km);
  val(Edit2.Text, M, kn);
  { If the conversion was successful and the entered dimensions
    are consistent with the declaration of matrices A and B, }
  if (kn = 0) and (km = 0) and (N > 0) and (N < 26) and (M > 0) and
  (M < 26) then
  begin
    StringGrid1.Visible := true;
    Label4.Visible := true;      { "Source Matrix A" }
    Button2.Visible := true;    { Clear button }
    Button3.Visible := true;    { Transpose button }
    with StringGrid1 do
    begin
      ColCount := M + 1;
      RowCount := N + 1;
      // Number the matrix rows and columns.
      for i := 1 to RowCount - 1 do
        Cells[0,i] := IntToStr(i);
      for i := 1 to ColCount - 1 do
        Cells[i,0] := IntToStr(i);
      // Make cells in StringGrid1 editable
      Options := Options + [goEditing];
    end;
    StringGrid2.ColCount := N + 1;
    StringGrid2.RowCount := M + 1;
  end
  else
  begin
    // Display message if the input is incorrect.
  end
end;
```

---

```

    MessageDlg('The matrix dimensions were entered incorrectly!', 
mtInformation,[mbOk], 0);
    Edit1.Text := '4';           { Reset to default value. }
    Edit2.Text := '3';           { Reset to default value. }
  end;
end;

```

Now let us write a handler for the *Transpose* button. When you click on this the button StringGrid2, which stores transposed matrix B, and its label (Label5) become visible, and matrix B is calculated. Matrix B will be displayed by StringGrid2. The Input button becomes invisible. The handler code is shown below.

```

procedure TForm1.Button3Click (Sender: TObject);
var
  i, j: integer;
begin
  // Make StringGrid2 and its label visible
  StringGrid2.Visible := true;
  Label5.Visible := true;
  // Populate matrix A with data from StringGrid1
  for i := 1 to N do          { Loop thru rows. }
    for j := 1 to M do        { Loop thru columns. }
      A[i,j] := StrToInt(StringGrid1.Cells[j,i]);
  // Number the fixed rows and columns in StringGrid2
  with StringGrid2 do
  begin
    for i := 1 to RowCount - 1 do
      Cells[0,i] := IntToStr(i);
    for i := 1 to ColCount - 1 do
      Cells[i,0] := IntToStr(i);
  end;
  // Populate transposed matrix B.
  for i := 1 to N do
    for j := 1 to M do
      B[j,i] := A[i,j];
  // Display matrix B in StringGrid2.
  for i := 1 to n do
    for j := 1 to m do
      StringGrid2.Cells[i,j] := IntToStr(B[j,i]);
  Button1.Visible := false;
end;

```

Only the writing of an event handler for the *Clear* button remains. When you click on this button, the following should occur:

- clear the contents of StringGrid1 and StringGrid2;
- StringGrid1, StringGrid2, Label4, Label5, the *Transpose* and *Clear* buttons

- become invisible;
- the Input button becomes visible;
  - Reset matrix dimensions to default values ( $N = 4, M = 3$ ).

The commented code for the Clear button handler is shown below:

```

procedure TForm1.Button2Click(Sender: TObject);
var
  i, j: integer;
begin
  // Clear StringGrid1.
  with StringGrid1 do
    for i := 1 to RowCount - 1 do
      for j := 1 to ColCount - 1 do
        Cells[j,i] := '';
  // Clear StringGrid2.
  with StringGrid2 do
    for i := 1 to RowCount - 1 do
      for j := 1 to ColCount - 1 do
        Cells[j,i] := '';
  // Make StringGrid1, StringGrid2, Label4 and Label5 invisible.
  StringGrid1.Visible := false;
  StringGrid2.Visible := false;
  Label4.Visible := false;
  Label5.Visible := false;
  // Make the Transpose and Clear buttons invisible.
  Button2.Visible := false;
  Button3.Visible := false;
  // Make the Input button visible.
  Button1.Visible := true;
  // Reset matrix dimensions to defaults (N = 4, M = 3).
  Edit1.Text := '4';
  Edit2.Text := '3';
end;

```

We now have a working program for matrix transposition. Figure 6.11 shows the results of the transposition of matrix A(2,4).

Note the use of the **with** statement:

**with** component\_name **do** statement;

which makes it easier to access the properties of a component. Inside the **with** statement, the component name can be omitted when accessing its properties.

For example, to clear matrix A, instead of statements:

```

for i := 1 to StringGrid1.RowCount - 1 do
  for j := 1 to StringGrid1.ColCount - 1 do

```

```
StringGrid1.Cells[j,i] := '';
```

the **with** statement was used:

```
with StringGrid1 do
  for i := 1 to RowCount - 1 do
    for j := 1 to ColCount - 1 do
      Cells[j,i] := '';
```

Let us consider a few matrix processing tasks. To address them, the reader should recall several matrix properties (Figure 6.12):

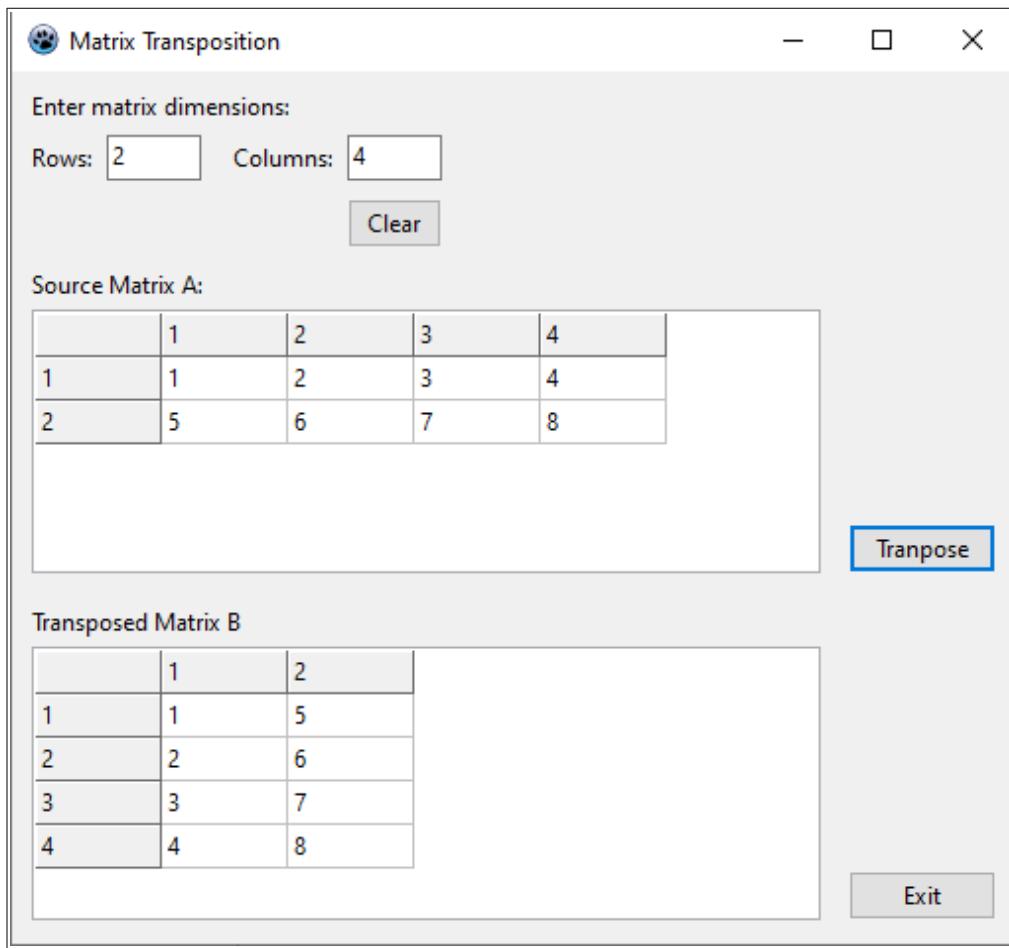


Figure 6.11: Transposition of matrix A(2,4)

- If an element's row index equals its column index ( $i = j$ ), then the element lies on the main diagonal of the matrix;
- If the row index is greater than the column index ( $i > j$ ), then the element is below the main diagonal;

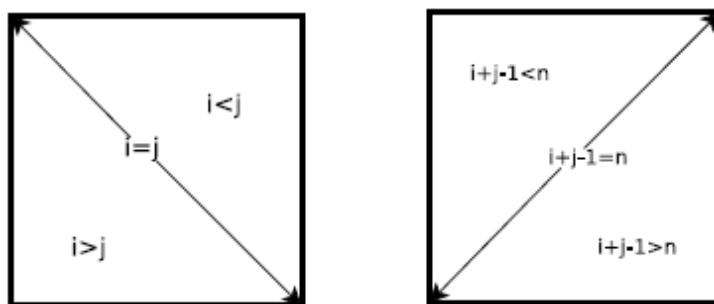


Figure 6.12: Matrix element properties

- If the column index is greater than the row index ( $i < j$ ), then the element is above the main diagonal;
- An element lies on the antidiagonal if its indices satisfy equality  $i + j - 1 = n$ ;
- The inequality  $i + j - 1 < n$  is characteristic of the element located above the antidiagonal;
- Similarly, the inequality  $i + j - 1 > n$  is characteristic of elements below the antidiagonal.

## 6.2 Matrix Algorithms

Consider several examples of solving matrix processing problems.

EXAMPLE 6.3. Find the sum of the elements lying above the main diagonal of a matrix (see Figure 6.13).

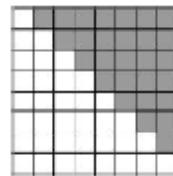


Figure 6.13: Drawing for Example 6.3

Consider two algorithms for solving this problem.

The *first algorithm* for solving this problem (see Figure 6.14) is structured as follows. First, the variable  $S$  for accumulating the sum is set to zero ( $S := 0$ ). Then, using two loops (the first for rows, the second for columns), all the elements of the matrix are accessed, but the summation occurs only if this element is above the main diagonal (if property  $i < j$ ). \

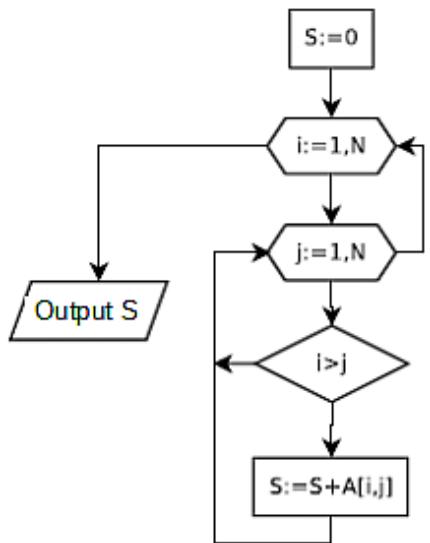


Figure 6.14: Example 6.3 flowchart  
(algorithm 1)

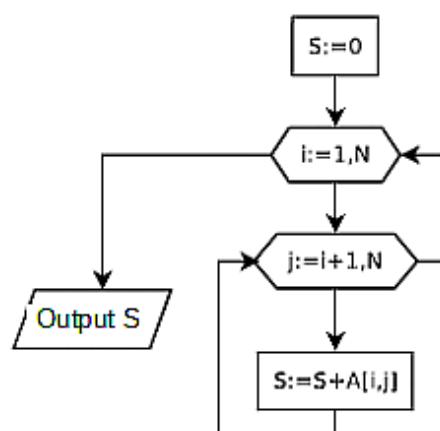


Figure 6.15: Example 6.3 flowchart  
(algorithm 2)

The commented code for a console application with is shown below.

```

var
  a: array [1..15, 1..10] of real;
  i, j, n, m: integer;
  s: real;
begin
  writeln('Enter the matrix dimensions.');
  writeln('n - the number of rows, m - the number of columns');
  readln(n, m);
  writeln('Enter matrix A: ');
  for i := 1 to n do
    for j := 1 to m do
      read(a [i, j]);
  s := 0;
  for i := 1 to n do
    for j := 1 to m do
      if j > i then          { If element above main diagonal }
        s := s + a[i,j];    { add it to sum. }
  writeln('matrix A');
  for i := 1 to n do
  begin
    for j := 1 to m do
      write(a[i,j]:8:3, ' '); { Field width is important here }
      writeln;
  end;
end;

```

```
writeln('Sum of matrix elements: ', s: 8: 3);
end.
```

The results of the program are shown in Figure 6.16.

The second algorithm for this problem is shown in Figure 6.15.

It does not check the condition  $i < j$ , but, nevertheless, it also sums only the elements of the matrix that are above the main diagonal. To understand the functioning of the algorithm, refer to Figure 6.13. In the first row of the given matrix, all the elements, starting with the second, must be summed. In the second row, everything starting from the third, in the  $i$ -th row, the summation will begin with the  $i$ -th + 1 element and so on. So, the first loop goes from 1 to  $N$ , and the second from  $i + 1$  to  $M$ .

We suggest that the reader independently write a program that implements the algorithm described above.

```
Enter the dimensions of the matrix
n - number of rows, m - number of columns
3 4
Enter matrix A
1 2 3 4
5 6 7 8
9 11 12 13
matrix A
1.000    2.000    3.000    4.000
5.000    6.000    7.000    8.000
9.000    11.000   12.000   13.000
Sum of matrix elements 37.000
```

Figure 6.16: Results from the program for Example 6.3

EXAMPLE 6.4. Calculate the number of positive elements in square matrix A, located along its perimeter and on the diagonals.

In a square matrix, the number of rows is equal to the number of columns. Before trying to solve the problem, consider Figure 6.17, which shows the diagonals of square matrices of different dimensions.

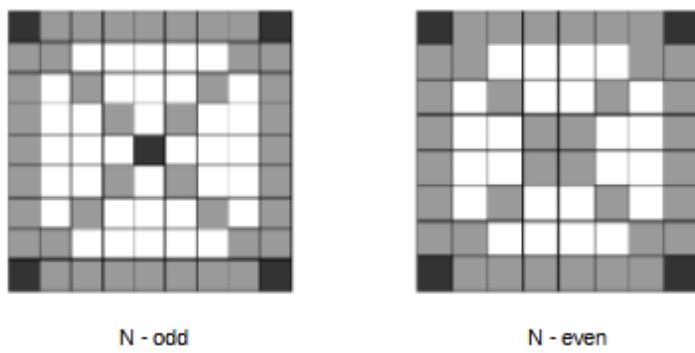


Figure 6.17: Possible matrix diagonals in Example 6.4

The figure shows that there is no need to consider all the elements of the matrix. It is enough to consider the elements in the first and last rows, in the first and last columns and on the diagonals of the square matrix. These elements are all marked in Figure 6.17, in which the elements that on the rows, columns and diagonals are highlighted in black.

For example, element  $A_{1,1}$  is on the first row, the first column, and the main diagonal of the matrix, and element  $A_{N,N}$  is on the last row, last column and on the main diagonal. Also, if  $N$  is an odd number (the matrix on the left in Figure 6.17), then there exists an element with index  $(N \text{ div } 2 + 1, N \text{ div } 2 + 1)$ , which is at the intersection of the main diagonal and the antidiagonal. For an even value of  $N$  (the matrix on the right in Figure 6.17) the diagonals do not intersect.

Consider the algorithm for solving the problem. To refer to the elements on the main diagonal, remember that the row indices of these elements are always equal to the column indices. Thus, if parameter  $i$  changes from 1 to  $N$ , then  $A_{i,i}$  is an element on the main diagonal.

Taking advantage of a property characteristic of elements on the antidiagonal, we obtain:  $i+j-1 = N$ , from which  $j = N-i+1$ . Thus, for rows  $i = 1, 2, \dots, N$ , element  $A_{i,N-i+1}$  is on the antidiagonal. Elements along the perimeter of the matrix are written as follows:  $A_{1,i}$  are elements in the first row ( $i=1, 2, \dots, N$ ),  $A_{N,i}$  are elements in the last row ( $i=1, 2, \dots, N$ ),  $A_{i,1}$  are elements in the first column ( $i=1, 2, \dots, N$ ), and  $A_{i,N}$  are elements in the last column ( $i=1, 2, \dots, N$ ).

We will construct the processing algorithm as follows. First, process the elements on the diagonals of the square matrix. This requires in each row ( $i=1, 2, \dots, N$ ), that the sign of the elements  $A_{i,i}$  and  $A_{i,N-i+1}$  be checked.

```

for i := 1 to N do
begin
  if (a[i,i] > 0) then k := k + 1;
  if a [i,N-i+1] > 0 then k := k + 1;

```

---

```
end;
```

Since the corner elements of the matrix have already been processed while checking the diagonal elements, they can be ignored when processing elements located along the perimeter of the matrix. Thus, we need to iterate over elements from the second to the second to last in the first and last row, and in the first and last column.

```
for i := 2 to N - 1 do
begin
  if (a[1,i] > 0) then k := k + 1; { If item is on first row }
  if (a[N,i] > 0) then k := k + 1; { If item is on last row }
  if (a[i,1] > 0) then k := k + 1; { If item is on first column }
  if (a[i,N] > 0) then k := k + 1; { If item is on last column }
end;
```

Check if the element at the intersection of the diagonals was counted twice. This could only happen if  $N$  is odd and the element at the intersection of the diagonals<sup>4</sup> is positive.

```
if (N mod 2 <> 0) and (a[n div 2 + 1, n div 2 + 1] > 0) then k := k - 1;
```

The full code, with comments, for a console application to solve Problem 6.4, is given below.

```
var
  a: array [1..10, 1..10] of integer;
  i, j, N, k: integer;
begin
  write('Enter array size: ');
  readln(N);
  // Enter the original matrix.
  writeln('Enter matrix A:');
  for i := 1 to N do
    for j := 1 to N do
      read(a[i,j]);
  // Display the original matrix.
  writeln;
  writeln('Matrix A, as entered:');
  for i := 1 to N do
  begin
    for j := 1 to N do
      write(a[i,j], ' ');
    writeln;
  end;
  k := 0;
  // Process elements on the diagonals of matrix.
  for i := 1 to N do
```

```

begin
  if (a[i,i] > 0) then k := k + 1;
  if a[i,N-i+1] > 0 then k := k + 1;
end;
// Process elements along the perimeter of matrix.
for i := 2 to N - 1 do
begin
  if (a[1,i]> 0) then k := k + 1;
  if (a[N,i]> 0) then k := k + 1;
  if (a[i,1]> 0) then k := k + 1;
  if (a[i,N]> 0) then k := k + 1;
end;
{If the element at the diagonal intersection counted twice,}
if (n mod 2 <> 0) and (a[N div 2 + 1, N div 2 + 1] > 0) then
  k := k - 1;           {decrease k by one.}
writeln;
writeln('No of positive elements: ', k);
end.

```

Figure 6.18 shows the results of the program for Problem 6.4.

```

Enter array size: 5
Enter matrix A:
11 22 33 44 -5
-6 77 33 -8 99
22 33 44 55 66
17 -8 -9 99 37
64 76 -7 17 41

Matrix A, as entered:
11 22 33 44 -5
-6 77 33 -8 99
22 33 44 55 66
17 -8 -9 99 37
64 76 -7 17 41

No of positive elements: 16

```

Figure 6.18: Results for Example 6.5

**EXAMPLE 6.5.** Check if a given square matrix is an identity matrix.

An identity matrix is one in which the elements on the main diagonal are ones, and all the others are zeros. For example,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This problem will be solved thus. Assume the matrix is an identity matrix (`ok := true`) and try to prove the opposite. Using a double loop, loop through the rows ( $i := 1, 2, \dots, N$ ) and columns ( $j := 1, 2, \dots, N$ ) to iterate over all the elements of the matrix. If a diagonal element ( $i = j$ ) is not equal to one or an element off the main diagonal ( $i = j$ ), is not equal to zero<sup>5</sup>, then change the Boolean variable `ok` to `false` and stop checking (exit the loop early). After the loop ends, check the value of `ok`. If `ok` is still true, then the matrix is an identity matrix; otherwise, it is not. The algorithm flowchart is shown in Figure 6.19.

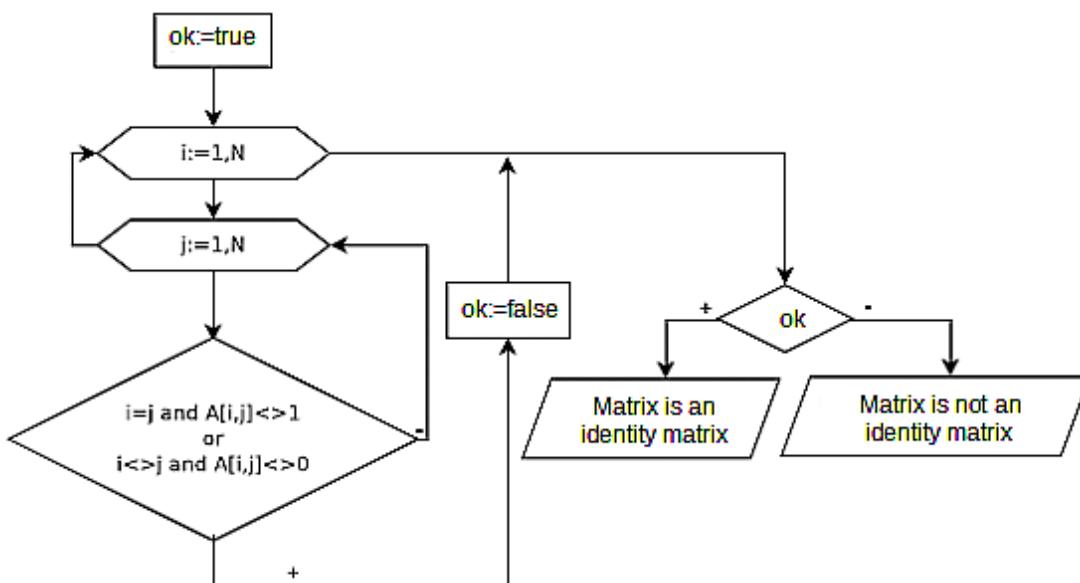


Figure 6.19: Algorithm for Example 6.5

```

program pr_6_5;
var
  a: array [1..10, 1..10] of real;
  i, j, n: integer;
  ok: boolean;
begin
  write('Enter the matrix size: ');
  readln(n);
  writeln('Enter the matrix:');
  for i := 1 to n do
    for j := 1 to n do
      a[i,j] := ...
  end;
  for i := 1 to n do
    for j := 1 to n do
      write(a[i,j], ' ');
    writeln;
end.
  
```

```

read(a[i,j]);
ok := true;           { Assume the matrix is an identity matrix. }
for i := 1 to n do
  for j := 1 to n do
    if ((i = j) and (a[i,j] <> 1)) or ((i <> j) and (a[i,j] <> 0))
then
{ If element is on main diagonal and not equal to one, or the
element is off the main diagonal and not equal to zero, then }
  begin
    ok := false;        { Matrix is not an identity matrix. }
    break;              { Exit loop early. }
  end;
  if ok then          { If matrix is an identity matrix }
    writeln('The matrix is an identity matrix.')
  else
    writeln('The matrix is not an identity matrix.');
end.

```

EXAMPLE 6.6. Modify a matrix so that the last the element in each column was replaced by the difference between the minimum and maximum elements in the same column.

To solve this problem, find in each column the maximum and minimum elements in each column, then change the last element of the column to their difference. The flowchart of the solution algorithm is shown in Figure 6.20.

Below is the code of the console application with comments.

```

program pr_6_6;
var
  a: array [1..25, 1..25] of real;
  i, j, n, m: integer;
  max, min: real;
begin
  write('Enter the 2 dimensions of the matrix: ');
  readln(n, m);
  writeln('Enter the matrix:');
  for i := 1 to n do
    for j := 1 to m do
      read(a[i,j]);
{ Sequentially iterate over all the columns of the matrix.}
  for j := 1 to m do
  begin
{ Assume first element of current column is both the maximum and
minimum. }
    max := a[1,j];

```

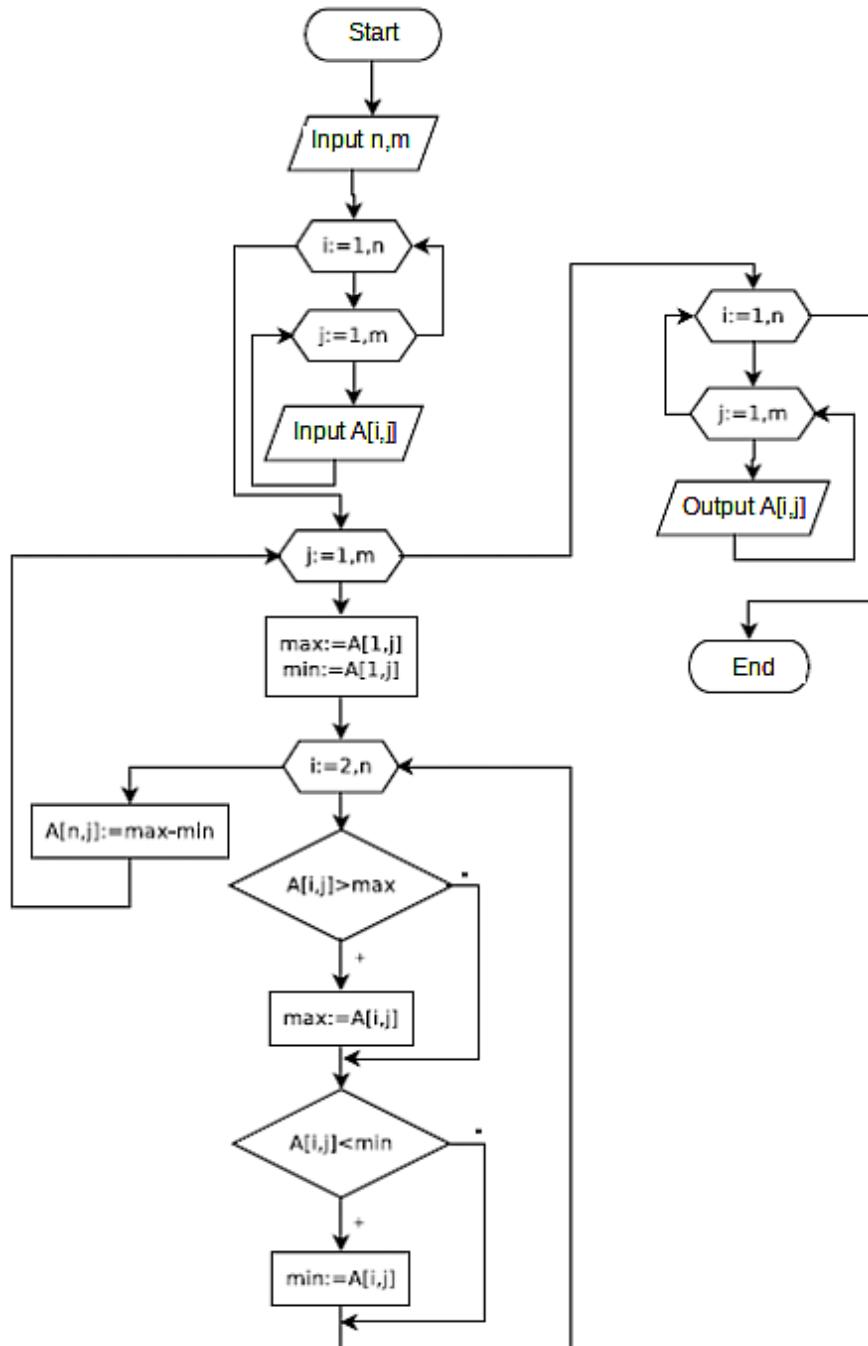


Figure 6.20: Algorithm for Example 6.6

```

min := a[1,j];
for i := 2 to n do { Iterate over the current column. }
begin
  { If current element > max then set max = current element. }
  if a[i,j] > max then max := a[i,j];

```

```

{ If current element < min then set min = current element. }
if a[i,j] < min then min := a[i,j];
end;
a[n,j] := max - min;      { Set last element = max - min. }
end;
{ Output the modified matrix. }
writeln;
writeln('Modified Matrix:');
for i := 1 to n do
begin
  for j := 1 to m do
    write(a[i,j]:7:3, ' ');
  writeln;
end;
end.

```

Let us now create a graphical application to implement this algorithm. As the basis, take the form in Figure 6.9 and the project to transpose matrix A(N,M) that was developed in Problem 6.2. The form window will be changed a bit (Figure 6.9). Rename the *Transpose* button to *Modify* and change the Caption property of Label5 to *Modified Matrix A*. In addition, change the Caption property of the form (see Figure 6.21). The modified form window is shown in Figure 6.21.

The handlers for the *Input*, *Clear* and *Exit* buttons will change a little. Let us consider the algorithm for the *Modify Matrix* button handler. This handler will read the matrix from StringGrid1, modify matrix A according to the algorithm presented in Figure 6.20, and display the transformed matrix in StringGrid2.

The commented code for the graphical application for solving Problem 6.6 is shown below.

```

unit Unit1;
{$mode objfpc} {$H+}
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls, Grids;
// Form declaration
type
{TForm1}
TForm1 = class (TForm)
Button1: TButton;
Button2: TButton;
Button3: TButton;
Button4: TButton;
Edit1: TEdit;
Edit2: TEdit;
Label1: TLabel;

```

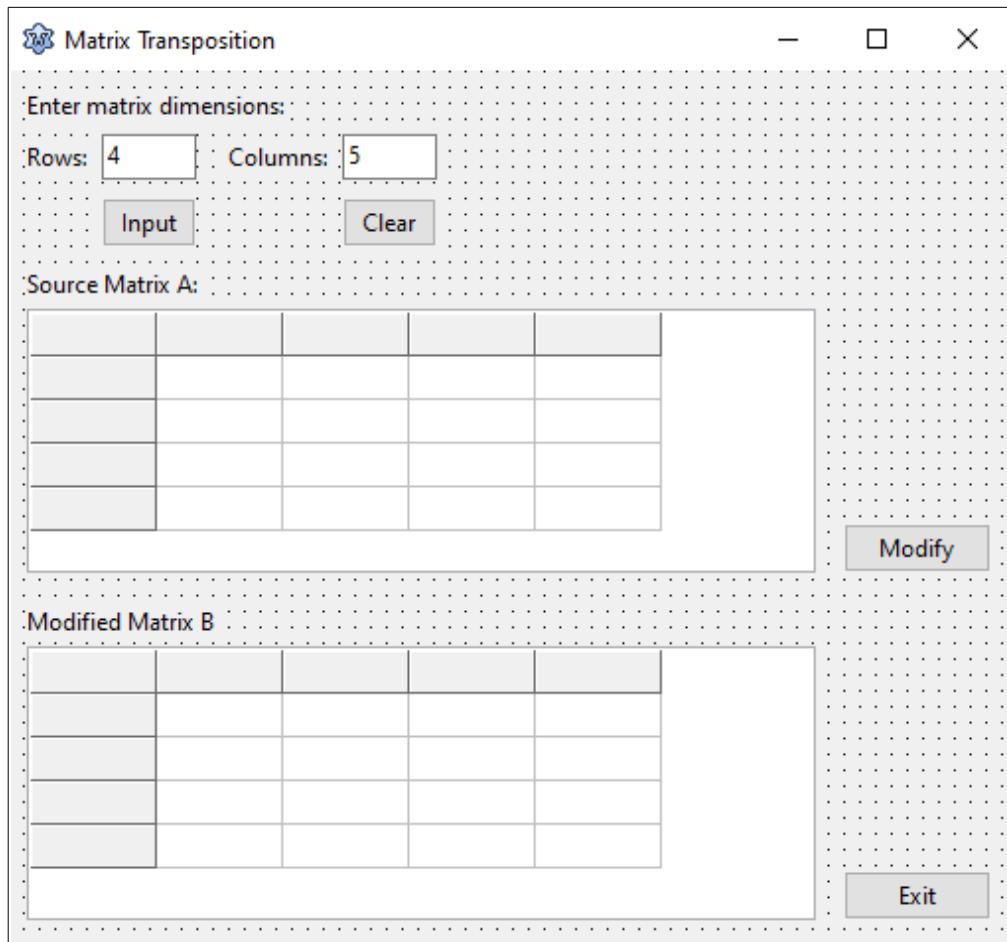


Figure 6.21: Form for Example 6.6

```
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
StringGrid1: TStringGrid;
StringGrid2: TStringGrid;
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button4Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;

var
  A: array [1..25, 1..25] of real;
```

```
N, M: integer;
Form1: TForm1;
implementation
{ TForm1}

procedure TForm1.Button1Click(Sender: TObject);
// Input button handler
var
  i: byte;
  kn, km: integer;
begin
  // Convert matrix dimensions to numbers.
  Val(Edit1.Text, N, kod_m);
  Val(Edit2.Text, M, kod_n);
  // If conversion successful and dimensions ok
  if (kod_n = 0) and (kod_m = 0) and (N > 0) and (N < 26) and (M >
  0) and (M < 26) then
    begin
      // Show StringGrid1, its label and remaining buttons
      StringGrid1.Visible := true;
      Label4.Visible := true;
      Button2.Visible := true;           {Clear button}
      Button3.Visible := true;           {Modify button}
      with StringGrid1 do
        begin
          ColCount := M + 1;
          RowCount := N + 1;
          // Number StringGrid1 fixed rows and columns
          for i := 1 to RowCount - 1 do
            Cells[0,i] := IntToStr(i);
          for i := 1 to ColCount - 1 do
            Cells[i,0] := IntToStr(i);
          // Make cells in StringGrid1 editable
          Options := Options + [goEditing];
        end;
      StringGrid2.ColCount := M + 1;
      StringGrid2.RowCount := N + 1;
    end
  else
    begin
      // If the input is incorrect, display a message.
      MessageDlg ('Matrix dimensions are entered
incorrectly!', mtInformation, [mbOk], 0);
      // Reset the starting parameters in the input fields.
      Edit1.Text := '4';
      Edit2.Text := '3';
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
// Clear button handler
var
  i,j: integer;
begin
  // Clear StringGrid1.
  with StringGrid1 do
    for i := 1 to RowCount - 1 do
      for j := 1 to ColCount - 1 do
        Cells [j, i] := '';
  // Clean uStringGrid2.
  with StringGrid2 do
    for i := 1 to RowCount - 1 do
      for j := 1 to ColCount - 1 do
        Cells[j,i] := '';
  // Make StringGrid1, StringGrid2, Label4, Label5 invisible.
  StringGrid1.Visible := false;
  StringGrid2.Visible := false;
  Label4.Visible := false;
  Label5.Visible := false;
  // Make the Clear and Modify buttons invisible.
  Button2.Visible := false;
  Button3.Visible := false;
  // Make the Input button visible.
  Button1.Visible := true;
  // Reset matrix default size (N = 4, M = 3).
  Edit1.Text := '4';
  Edit2.Text := '3';
end;

procedure TForm1.Button3Click(Sender: TObject);
// Modify button handler
var
  i, j: integer;
  max, min: real;
begin
  // Make StringGrid2 and its label visible
  StringGrid2.Visible := true;
  Label5.Visible := true;
  // Populate matrix A with data from StringGrid1
  for i := 1 to N do                      { Loop thru rows. }
    for j := 1 to M do                      { Loop thru columns. }
      A[i,j] := StrToFloat(StringGrid1.Cells[j,i]);
  // Number fixed rows and columns in StringGrid2
  with StringGrid2 do
  begin
    for i := 1 to RowCount - 1 do
      Cells[0,i] := IntToStr(i);
    for i := 1 to ColCount - 1 do
      Cells[i,0] := IntToStr(i);
  end;
  // Solution to Problem 6.6.
```

```

for j := 1 to m do
begin
  max := a[1,j];           { Assume first element in current }
  min := a[1,j];           { column is both maximum and minimum. }
  for i := 2 to n do {Iterate over current column.}
  begin
    { If current element > max, then max = current element. }
    if a[i,j] > max then max := a [i,j];
    { If current element < min, then min = current element. }
    if a[i,j] < min then min := a[i,j];
  end;
{ Save max - min in last column element. }
  a[n,j] := max - min;
end;
// Display modified matrix A in StringGrid2
for i := 1 to N do           { Loop thru rows. }
  for j := 1 to M do       { Loop thru columns. }
    StringGrid2.Cells[j,i] := FloatToStr(A [i,j]);
// Make the first button invisible.
  Button1.Visible := false;
end;

procedure TForm1.Button4Click(Sender: TObject);
// Exit button handler
begin
  Close;
end;

end.

```

Figure 6.22 shows results of solving the problem.

**EXAMPLE 6.7.** Swap the n-th and r-th columns in the matrix A(K,M).

The problem reduces to exchanging the n-th and r-th elements in all rows of the matrix. The flowchart is shown in Figure 6.23.

Below is a commented code listing of the console application.

```

type
  matrix = array [1..15, 1..15] of real;
var
  a: matrix;
  i,j,k,m,n,r: byte;
  b: real;
begin

```

```
write('Enter Number of Rows: ');
readln(k);
write('Enter Number of Columns: ');
readln(m);
```

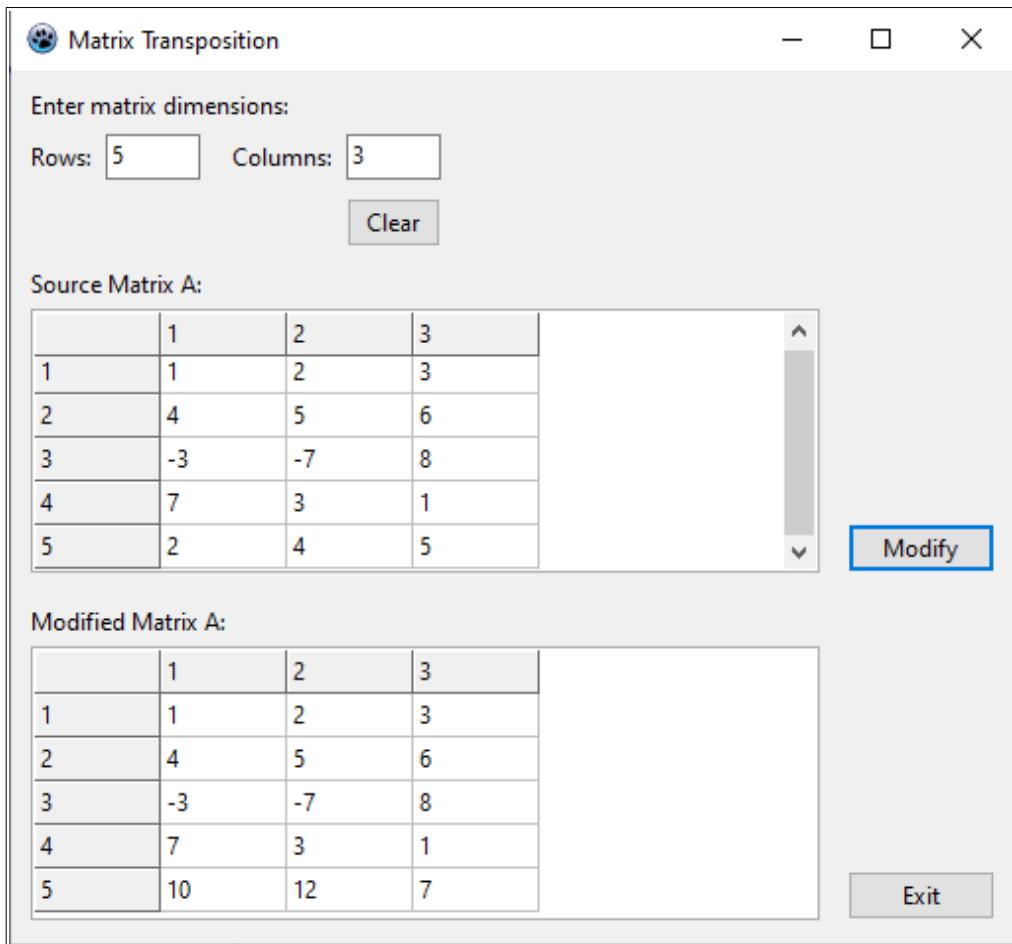


Figure 6.22: Results from Example 6.6

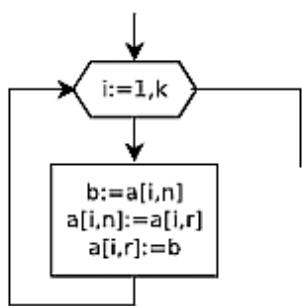


Figure 6.23: Flowchart for Example 6.7

```
// Enter matrix A.
writeln('Matrix A:');
```

```
for i := 1 to k do
    for j := 1 to m do
        read (a[i,j]);
// Enter the numbers of the columns to be exchanged.
repeat
    write('Enter first column to swap: ');
    readln(n);
    write('Enter next column to swap: ');
    readln(r);
    { Entry ok if n and r <= m and n not equal r. }
until (n <= m) and (r <= m) and (n > r);
{ Replace elements in column r with elements in column n. }
for i := 1 to k do
begin
    b := a[i,n];
    a[i,n] := a[i,r];
    a[i,r] := b
end;
writeln;
writeln('Modified matrix A:');
for i := 1 to k do
begin
    for j := 1 to m do
        write(a[i,j]:7:3, ' ');
    writeln;
end;
end.
```

```
Enter number of rows: 4
Enter number of columns: 5
Matrix A:
1 2 3 4 5
6 7 8 9 3
7 5 3 1 2
5 4 3 2 1
Enter first column to swap: 2
Enter next column to swap: 5

Modified matrix A
1.000 5.000 3.000 4.000 2.000
6.000 3.000 8.000 9.000 7.000
7.000 2.000 3.000 1.000 5.000
5.000 1.000 3.000 2.000 4.000
```

Figure 6.24: Results from Example 6.7

The results of the program are shown in Figure 6.24.

**EXAMPLE 6.8.** Modify matrix A ( $m, n$ ) so that rows with odd indices were rearranged in descending order, and the rows with even indices in ascending order.

Each row of the matrix is a one-dimensional array. Thus, you can use the common methods for sorting arrays to rearrange a row or column. To solve this problem, loop from row to row and sort the rows with odd indices in descending order and rows with even indices in ascending order, using the bubble sort method. The flowchart for this algorithm is shown in Figure 6.25.

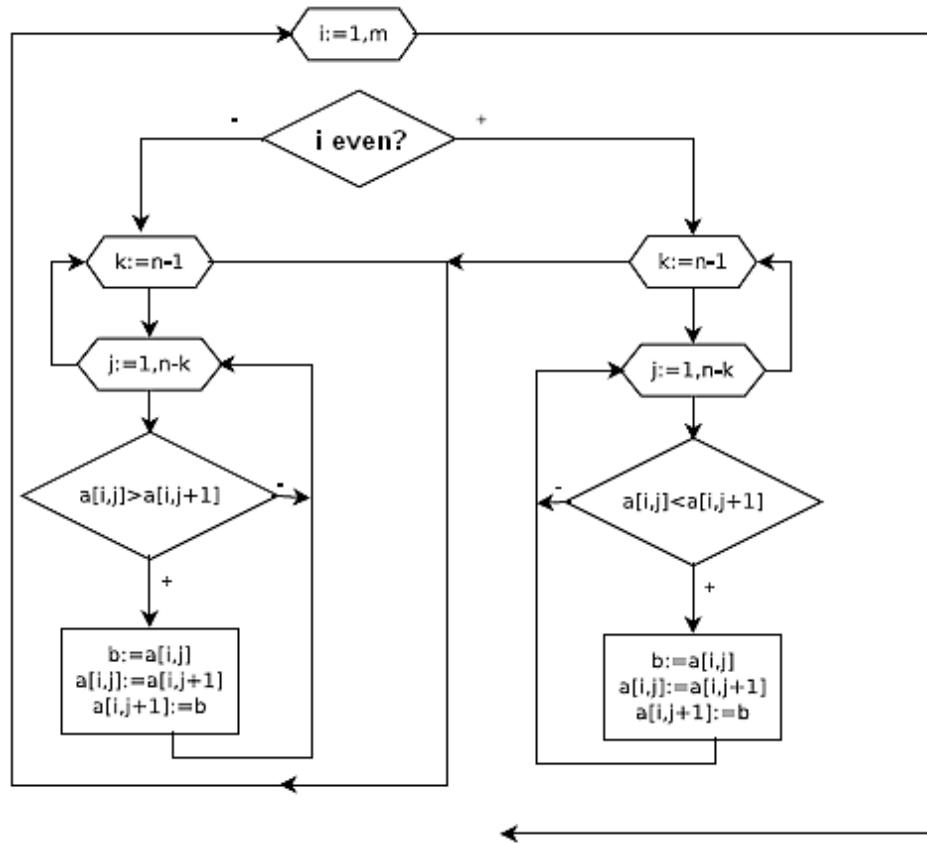


Figure 6.25: Flowchart for Example 6.8

Below is the commented code for a console solution.

```

var
  a: array [1..15, 1..15] of real;
  j, i, k, m, n: byte;
  
```

```
b: real;
begin
  // Enter the size of the matrix.
  writeln('Enter matrix dimensions m and n:');
  readln(m, n);
  // Enter the matrix.
  writeln('Matrix A:');
  for i := 1 to m do
    for j := 1 to n do
      read(a[i,j]);
  // Modify the matrix.
  for i := 1 to m do
    if (i mod 2) = 0 then          { If row is even }
      begin                      { sort elements in ascending order, }
        for k := 1 to n - 1 do    { using the bubble sort method. }
          for j := 1 to n - k do
            if a[i,j] > a[i,j + 1] then
              begin
                b := a[i,j];
                a[i,j] := a[i,j + 1];
                a[i,j + 1] := b;
              end;
        end
      else                      { if row is odd }
        for k := 1 to n - 1 do
          for j := 1 to n - k do
            if a[i,j] < a[i,j + 1] then
              begin
                b := a[i,j];
                a[i,j] := a[i,j + 1];
                a[i,j + 1] := b;
              end;
    end;
  // Display the modified matrix.
  writeln('Modified matrix A:');
  for i := 1 to m do
    begin
      for j := 1 to n do
        write(a[i,j]:7:3, ' ');
      writeln;
    end;
end.
```

Figure 6.26 shows results from the program.

```

Enter matrix dimensions m and n:
5 6
Matrix A:
1 2 3 6 5 4
2 3 4 7 6 5
3 4 5 8 7 6
4 5 6 1 2 3
9 8 7 4 5 6
Modified matrix A:
6.000 5.000 4.000 3.000 2.000 1.000
2.000 3.000 4.000 5.000 6.000 7.000
8.000 7.000 6.000 5.000 4.000 3.000
1.000 2.000 3.000 4.000 5.000 6.000
9.000 8.000 7.000 6.000 5.000 4.000

```

Figure 6.26: Results of solving Example 6.8

EXAMPLE 6.9. Given: a matrix of positive integers  $A(n,m)$ . Create a vector  $P(n)$ , consisting of the sum of the prime numbers in each row of the matrix, in the base-4 number system. If there is no prime number in a row, place a 0 in the corresponding element of the array.

To solve this problem, we need functions to identify a prime number, and to convert an integer to the base-4 number system. The algorithm for identifying a prime number has already been considered several times in this book. This function was discussed in detail in Chapter 5, in Problem 5.7. Therefore, the necessary code will simply be presented here.

```

function prime (N: integer): boolean;
var
  i: integer;
  ok: boolean;
begin
  if N < 1 then ok := false           { Number is negative or 0 }
  else
    begin
      ok := true;
      for i := 2 to N div 2 do
        if (N mod i = 0) then          { Number is not prime. }
          begin
            ok := false;
            break;
          end;
    end;
end;

```

```

prime := ok;
end;
```

Also in Chapter 5, we considered an algorithm (Figure 5.42) and conversion function (function convert ( $N$ : **real**;  $P$ : **word**;  $kvo$ : **word**): **real**) to convert a real number to the base- $p$  number system (Problem 5.10). The function needed to convert a whole number to the base-4 number system is a special case of the convert function. Below is the code for the convert4 function, which converts a positive integer to the base-4 number system.

```

// Function to convert an integer N to the base-4 number system.
function convert4( $N$ : word): word;
var
   $S$ ,  $i$ ,  $q$ ,  $rem$ : word;
begin
   $S$  := 0;           { base-4 number }
   $q$  := 1;           { q = 10^x. Set x = 0 initially. }
  while ( $N <> 0$ ) do
    begin
       $rem$  :=  $N \bmod 4$ ;   { rem = next digit in converted number}
       $S$  :=  $S + rem*q$ ;       { Put together base-4 number. }
       $N$  :=  $N \div 4$ ;
       $q$  :=  $q*10$ ;          { Multiply by 10 for position value. }
    end;
    // Return the base-4 number.
    convert :=  $S$ ;
  end;
```

You need to find the sum of the prime numbers in each row, and then convert the resulting sum to a base-4 number. Therefore, in each row ( $i := 1, 2, \dots, n$ ), the following actions are necessary: Set sum  $S$  to zero ( $S := 0$ ), loop through the elements of the row ( $j := 1, 2, \dots, m$ ), while checking if the current element  $A_{ij}$  is a prime number. If it is, add it to the sum  $S$ . After exiting the loop with loop variable  $j$ , check if the row with index  $i$  had prime numbers ( $S > 0$ ). If it did, convert  $S$  to the base-4 number system and create the corresponding element of array  $P$  ( $P[i] := convert4(S)$ ).

The flowchart for the algorithm is shown in Figure 6.27.

The full code of the console application is shown below.

```

program pr_6_9;

function prime( $N$ : integer): boolean;
var
   $i$ : integer;
```

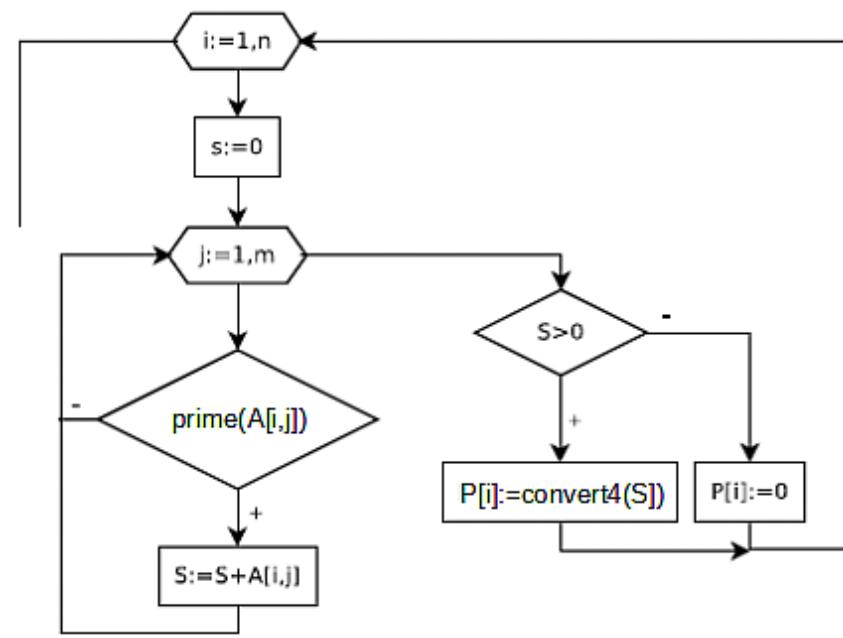


Figure 6.27: The algorithm for Example 6.9

```

ok: boolean;
begin
  if N < 1 then
    ok := false
  else
    begin
      ok := true;
      for i := 2 to N div 2 do
        if (N mod i = 0) then
          begin
            ok := false;
            break;
          end;
    end;
    prime := ok;
  end;

function convert4(N: word): word;
var
  S, q, rem: word;
begin
  S := 0;           { Sum in base-4 number format }
  q := 1;           { q = 10^x. Set x = 0 initially. }
  while (N <> 0) do
    begin
      rem := N mod 4;   { rem = next digit in converted number }
      ...
    end;
  ...
end;
  
```

```

S := S + rem*q;           { Put together base-4 number. }
N := N div 4;
q := q*10;                { Multiply by 10 for position value. }
end;
// Return the base-4 number.
convert4 := S;
end;

//Main program
var
  S, i, j, n, m: word;
  a: array [1..25, 1..25] of word;
  p: array [1..25] of word;
begin
  // Enter the size of the matrix.
  writeln('Enter (two) matrix dimensions:');
  readln(n, m);
  // Enter the matrix.
  writeln('Enter matrix A:');
  for i := 1 to n do
    for j := 1 to m do
      read(A[i,j]);
  // Loop over each row to find sum of prime numbers in row.
  for i := 1 to n do
    begin
      // Initially, set the sum to zero.
      S := 0;
      // Loop over elements in the i-th row.
      for j := 1 to m do
        // If next element is a prime number, add it to sum.
        if prime(A[i,j]) then S := S + A[i,j];
      // If S > 0, convert to base-4 and write in p[i].
      if S > 0 then p[i] := convert4(S);
      // If there were no prime numbers in row, then p[i] := 0.
      else
        p[i] := 0;
    end;
  // Output the generated array P.
  writeln;
  writeln('Array P');
  for i := 1 to n do write(P[i], ' ');
  writeln;
end.

```

The results of the program are shown in Figure 6.28.

```
Enter (two) matrix dimensions:
```

```
4 5
```

```
Enter matrix A:
```

```
2 4 6 8 13
```

```
6 8 10 12 14
```

```
13 11 6 10 11
```

```
34 56 76 80 5|
```

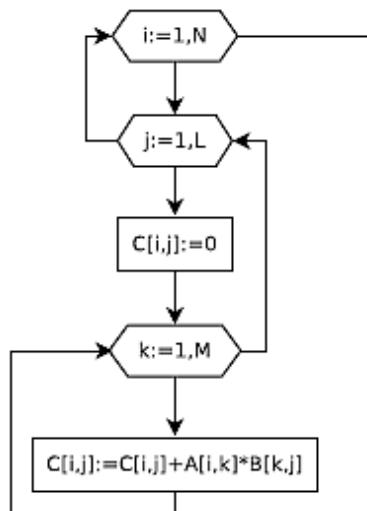
```
Array P:
```

```
33 0 203 11
```

*Figure 6.28: The results from Example 6.9*

EXAMPLE 6.10. Write a program to multiply two matrices A(N,M) and B(M,L).

Let us recall some information from math. You can only multiply two matrices for which the number of columns in the first matrix equals the number of rows in the second matrix. The product matrix has as many rows



*Figure 6.29: Multiplying two matrices*

as were in the first matrix and as many columns as were in the second. Thus, multiplying matrix A(N,M) by matrix B(M,L) gives matrix C(N,L). Each element of matrix  $C[i,j]$  is a scalar product of the  $i$ -th row of matrix A and the  $j$ -th column of matrix B. The formula to find element  $c_{ij}$  of the matrix has the general form:

$$c_{i,j} = \sum_{k=1}^M a_{ik} b_{kj} \quad (6.1)$$

where  $i = 1, \dots, N$  and  $j = 1, \dots, L$ .

Let us consider in more detail the formation of the matrix  $C(3,2)$  as the product of matrices  $A(3,3)$  and  $B(3,2)$ .

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \end{pmatrix}$$

Remember that  $A \cdot B \neq B \cdot A$ .

A flowchart that implements the multiplication of each element of matrix  $C$  per formula (6.1) is shown in Figure 6.29.

Below is the commented code for multiplying two matrices.

```

type
  matrix = array [1..15, 1..15] of real;
var
  A, B, C: matrix;
  i, j, M, N, L, k: byte;
begin
  // Enter the matrix sizes.
  writeln('Enter N, M and L');
  readln(N, M, L);
  // Enter matrix A.
  writeln('Matrix A:');
  for i := 1 to N do
    for j := 1 to M do
      read(A[i,j]);
  // Enter matrix B.
  writeln('Matrix B:');
  for i := 1 to M do
    for j := 1 to L do
      read(B[i,j]);
  // Populate matrix C.
  for i := 1 to N do
    for j := 1 to L do
      begin
        {C[i,j] will store the scalar result of multiplying
         the i-th row by the j-th column.}
        C[i,j] := 0;
        for k := 1 to M do
          C[i,j] := C[i,j] + A[i,k]*B[k,j];
      end;
  
```

```

    end;
    // Output of the matrix C = A*B.
    writeln;
    writeln('Matrix C = A*B');
    for i := 1 to N do
    begin
        for j := 1 to L do
            write(C[i,j]:7:3, ' ');
        writeln;
    end;
end.

```

Figure 6.30 shows results from the program.

```

Enter N, M and L:
3 4 5
Matrix A:
1 2 3 4
3 4 5 6
7 8 0 9
Matrix B:
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45

Matrix C = A * B
310.000 320.000 330.000 340.000 350.000
518.000 536.000 554.000 572.000 590.000
614.000 638.000 662.000 686.000 710.000

```

*Figure 6.30: Results from program to multiply two matrices*

EXAMPLE 6.11. Find the rows in a matrix of natural numbers  $A(N,M)$  which contain the largest prime number. Rearrange the elements in these rows in ascending order. If there are no prime numbers in the matrix, then leave it unchanged.

Before solving the problem, note some of its features<sup>6</sup>. The matrix may have no prime numbers, or several maximum values, or several maximum values in the same row.

The following subroutines are needed to solve this problem:

- 1) The *prime* function, which checks if the number P (word type) is a prime. It returns true if P is a prime, and false otherwise. The function header looks like this:

```
function prime (P: word): boolean;
```

- 2) The *remove* procedure, which removes values from array X that occur more than once. The procedure has two reference parameters: array X and its size N. The procedure header looks like this:

```
procedure remove (var X: arr; var N: word);
```

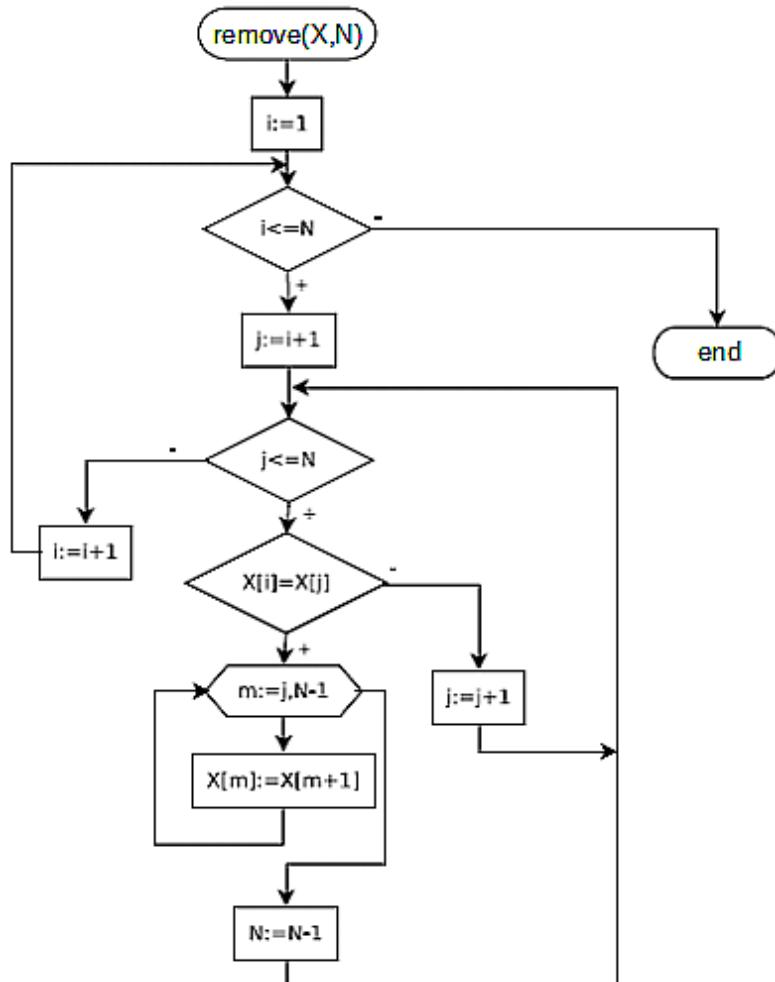


Figure 6.31: Remove procedure

Before declaring the procedure, declare the arr data type (for example, arr = array [1..200] of word). The rem procedure flowchart is shown in Figure 6.31.

Remove duplicate elements as follows. Checking all elements, starting with the first, compare the i-th element with all subsequent ones. If a duplicate

element is found, remove the element with index j from the array. The deletion algorithm was discussed in detail in Chapter 5.

- 3) The *present* function returns true if a specified number is present in the array b, and false otherwise. The procedure header looks like this:

```
function present(a: word; b: arr; N: word);
```

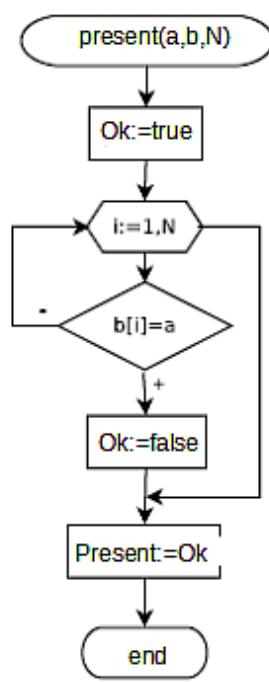


Figure 6.32: The present function

The function flowchart is shown in Figure 6.32.

- 4) The *ascend* procedure sorts array x in ascending order. Sorting algorithms were discussed in Chapter 5. The bubble sort method was chosen in this case. The *ascend* procedure has two parameters: array x (passed by reference) and its size N (passed by value). The procedure header is:

```
procedure ascend(var x: arr; N: word);
```

Consider in more detail the algorithm for solving Problem 6.11, which is given in Figure 6.33 and Figure 6.34.

After the input of the matrix (Blocks 1-4), assume that there is no prime. Set the Boolean variable ok to **false** but change it to **true** if a prime number is found. Set the number of maximum values among the prime numbers to 0 (k := 0) (Block 5).

To check if an element is a prime number, call the prime function. If the number is

a prime number (Block 8), check if it is the first prime number in the matrix (Block 9). If this is the first prime number, then save it in the variable max, set the variable k equal 1 (the number of maxima) and save the index of the row in which the maximum is located, in array arr at index k<sup>7</sup>. Set ok to **true** because the matrix contains prime numbers (Block 10). If this is not the first prime number, compare A [i,j] with max. If A [i,j] > max (Block 11), then save A[i,j] in max, set k equal 1 (there is one maximum), save i, the index of the row containing the maximum, in mas[k] (Block 12).

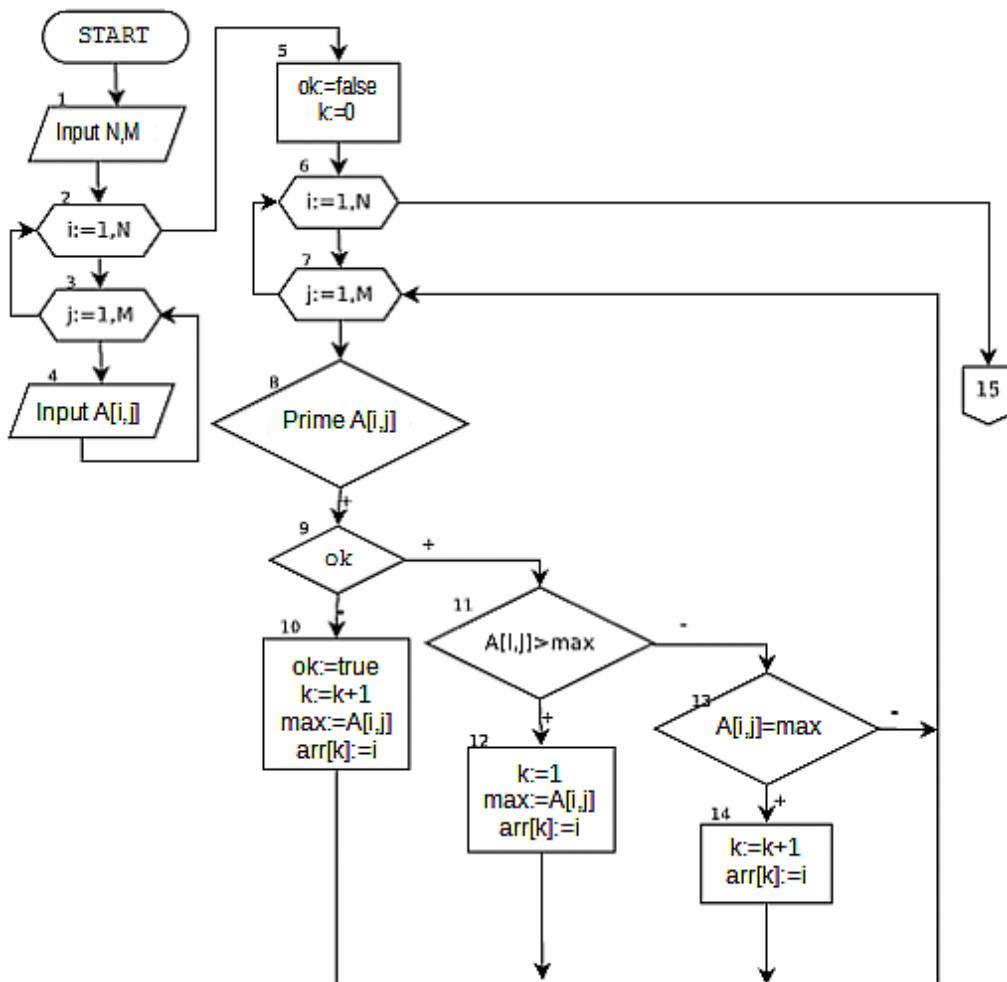


Figure 6.33: Example 6.11 flowchart (first part)

If A[i,j] = max (Block 13), then the element is equal to max. In this case, increase k by 1 and save the row index containing the element that equals the maximum in mas[k]. After processing all the matrix elements using a double loop (Blocks 6 to 14), max will contain the maximum prime number, k will contain the number of maxima and array mas will contain k elements, storing the row indices where the

maximum prime numbers were found. The variable *ok* stores true if the matrix contains prime numbers, or false if it does not.

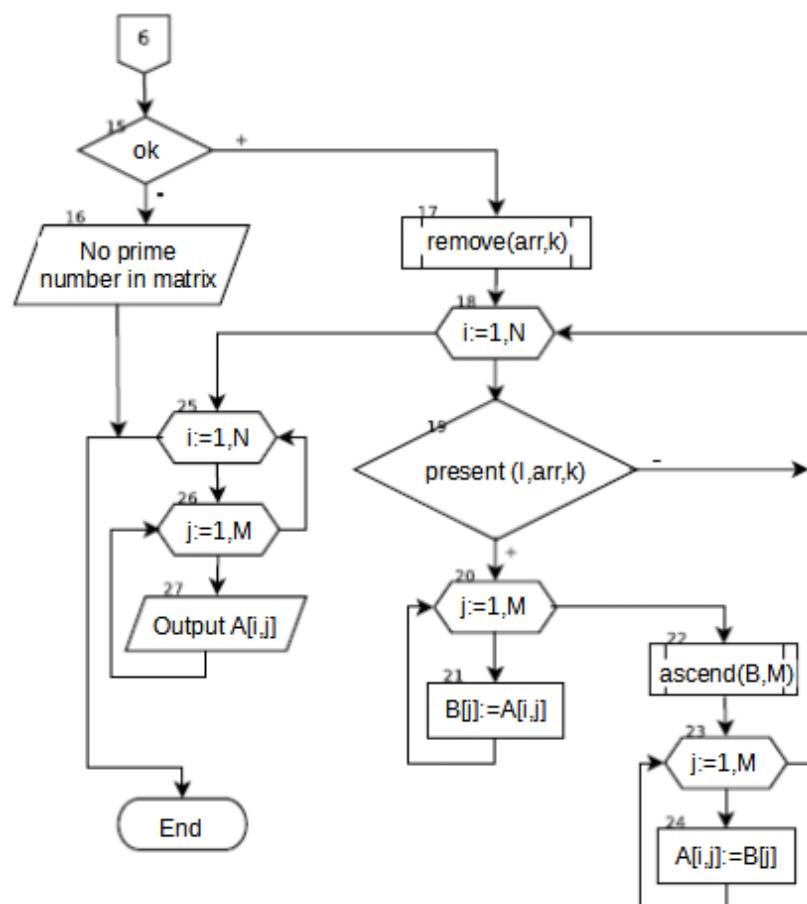


Figure 6.34: Example 6.11 flowchart (second part)

If there are no prime numbers in the matrix (Block 15), display the corresponding message (Block 16). Otherwise, delete *arr* elements from the array, including maxima occurring more than once<sup>8</sup>, using the *remove* procedure (Block 17). Then loop through all the rows of the matrix (the loop begins at Block 18). If the row index occurs in the *arr* array (Block 19), then copy the current row into array *b* (Blocks 20 to 21) and call *ascend*, the procedure for sorting an array increasing order (Block 22). Copy the sorted array *b* to the *i*-th row matrix *A* (Blocks 23 to 24). As the last step, display matrix *A* after modification (Blocks 25 to 27).

The entire code listing with detailed comments is shown below.

```

type
  arrtype = array [1..200] of word;
  
```

---

```

function prime (N: word): boolean;
{ Checks if N is prime (true) or not (false). }
var
  ok: boolean;
  i: word;
begin
  if N > 0 then
    begin
      ok := true;           { Assume N is prime. }
      for i := 2 to N div 2 do
        if N mod i = 0 then
          begin
            ok := false;       { N is not prime and }
            break;              { exit the loop early. }
          end;
    end
    else
      ok := false;
    prime := ok;
  end;

procedure remove(var X: arrtype; var N: word);
{ The remove procedure removes elements that occur more than once
from array X. X and N are reference parameters, since these values
are returned to the calling program from the remove procedure. }
var
  i, j, m: word;
begin
  i := 1;
  while (i <= N) do
    begin
      j := i + 1;
      while (j <= N) do
        if X[i] = X[j] then           { There is a duplicate element. }
          begin
            { Remove it (X[j]) from array. }
            for m := j to N - 1 do
              X[m] := X[m + 1];
            N := N - 1;
          end
        else
          j := j + 1;
      i := i + 1;
    end;
  end;

function present(a: word; B: arrtype; N: word): boolean;
{ The present function returns true if the number a occurs in array
B or false - otherwise. }
var
  ok: boolean;

```

```
i: word;
begin
{ Assume array B does not contain a and set ok = false. }
ok := false;
for i := 1 to N do
{ If next element in array B := a, then ok := true. }
if B[i] = a then
begin
ok := true;
break;
end;
present := ok;
end;

procedure ascend(var X: arrtype; N: word);
{ The ascend procedure sorts array X in ascending order. X is a
reference parameter. It is an array that is returned to the program
calling the ascend procedure. }
var
i, j, b: word;
begin
for i := 1 to N - 1 do
for j := 1 to N - i do
if X[j] > X[j + 1] then
begin
b := X[j];
X[j] := X[j+1];
X[j+1] := b;
end;
end;

// Main program
var
N, M, i, j, k, max: word;
A: array [1..20, 1..20] of word;
ok, L: boolean;
arr, B: arrtype;
begin
{ Enter elements of matrix A. }
write('Enter no of rows in matrix A: ');
readln(N);
write('Enter no of columns in matrix A: ');
readln(M);
writeln('Enter Matrix A:');
for i := 1 to N do
for j := 1 to M do
read(A[i,j]);
{ Assume no primes in matrix. }
ok := false;
{ Number of maxima is 0. }
k := 0;
```

```
for i := 1 to N do
  for j := 1 to M do
begin
  { Call prime function to check if A[i,j] is a prime. }
  L := prime(A[i,j]);
  { If the number is prime and }
  if L then
    { if this is the first prime number, }
    if not ok then
begin
  ok := true;
  { increase the number of maxima by 1. }
  k := k + 1;
  { Save this number in max. Assume that the first prime
  number is the maximum. }
  max := A[i,j];
  { Save row index, where A[i,j] occurs, in arr[k]. }
  arr[k] := i;
end
else
  { If A[i,j] is not first prime number, compare to max. }
  if A[i,j] > max then
begin
  { Reset the number of maxima to 1. }
  k := 1;
  { Save A[i,j] in max, }
  max := A[i,j];
  { Save the row index where A[i,j] occurs, in arr[k]. }
  arr[k] := i;
end
else
  { If A[i,j] = max (element = max), then }
  if A[i,j] = max then
begin
  { increase the number of maxima by 1, }
  k := k + 1;
  { Save row index where A[i,j] occurs in arr[k]. }
  arr[k] := I;
end;
end;
{If ok = false, display message "No prime number in matrix", }
if not ok then
  writeln('No prime number in matrix.')
else
begin
  { otherwise, remove duplicate elements from the arr array. }
  remove(arr,k);
  for i := 1 to N do
begin
  L := present(i, arr, k);
  { If row index is present in array arr, }
```

```
if L then
begin
{ then save the row in array B, }
for j := 1 to M do
  B[j] := A[i,j];
{ sort array B in ascending order. }
ascend(b, M);
{ Replace the i-th row of matrix A with the sorted array. }
for j := 1 to M do
  A[i,j] := B[j];
end;
end;
writeln;
writeln('Modified matrix A');
for i := 1 to N do
begin
  for j := 1 to M do
    write(A[i,j], ' ');
  writeln;
end;
end;
end.
```

The results from the program are shown in Figure 6.35.

```
Enter no of rows in matrix A: 5
Enter no of columns in matrix A: 6
Enter Matrix A:
13 2 7 6 1 9
3 4 6 2 1 5
13 3 13 6 7 1
4 6 8 2 3 9
4 9 7 13 1 2

Modified matrix A
1 2 6 7 9 13
3 4 6 2 1 5
1 3 6 7 13 13
4 6 8 2 3 9
1 2 4 7 9 13
```

Figure 6.35: Results from Example 6.11.

The authors recommend that the reader develop graphical applications based on the algorithms and console applications in Examples 6.7 to 6.11, like those developed for Examples 6.2 and 6.6.

We conclude this chapter with a look at dynamic matrices.

## 6.3 Dynamic Matrices

The concept of a dynamic array (see Section 5.10) can be extended to matrices. A dynamic matrix may be declared as follows.

```
arr: array of array of real;
```

Consider the use of a dynamic matrix in the following example.

EXAMPLE 6.12. For each row in a matrix of real numbers  $B(N, M)$  sort the elements between the maximum and minimum values in ascending order.

Chapter 5 discussed sorting algorithms and the current chapter discussed the basic principles of working with matrices. Thus, the code comments will focus on the specifics of working with dynamic matrices.

```
var
  Nmax, Nmin, i, j, n, m, k: word;
  B: array of array of real;      { Declare dynamic matrix B. }
  a, max, min: real;
begin
  { Enter number of rows n and number of columns m. }
  write('Enter no of rows: ');
  readln(n);
  write('Enter no of columns: ');
  readln(m);
  { Allocate memory for a n x m matrix of real numbers. }
  SetLength(B,n,m);
  writeln('Enter Matrix B:');
  for i := 0 to n-1 do
    for j := 0 to m-1 do
      read(B[i][j]);
  { In each row find the maximum and minimum elements and their
  indices and sort the elements between them using bubble sort. }
  for i := 0 to n-1 do
    begin
      { Find the minimum and maximum elements in the i-th row of the
      matrix and their indices. }
      max := B[i][0];
      Nmax := 0;
      min := B[i][0];
      Nmin := 0;
      for j := 1 to m-1 do
        begin
          if B[i][j] > max then
            begin
```

```
max := B[i][j];
Nmax := j;
end;
if B[i][j] < min then
begin
  min := b [i][j];
  Nmin := j;
end;
end;
{ If min element comes after max, swap Nmin and Nmax. }
if Nmax < Nmin then
begin
  j := Nmax;
  Nmax := Nmin;
  Nmin := j;
end;
{ In the i-th row sort the elements located between Nmin and
Nmax, using bubble sort. }
j := 1;
while Nmax - 1 - j >= Nmin + 1 do
begin
  for k := Nmin + 1 to Nmax - 1 - j do
    if B[i][k] > B[i][k+1] then
      begin
        a := B[i][k];
        B[i][k] := B[i][k+1];
        B[i][k+1] := a;
      end;
    j := j + 1;
  end;
end;
{ Output the modified matrix. }
writeln;
writeln('Sorted matrix B');
for i := 0 to n-1 do
begin
  for j := 0 to m-1 do
    write(B[i][j]:6:2, ' ');
  writeln;
end;
end.
```

The results of the program are shown in Figure 6.36.

A dynamic matrix can be quite large, and its size is in fact limited only by the amount of available memory.

Practice problems to be solved independently are presented at the end of the chapter.

```
Enter no of rows: 3
Enter no of columns: 9
Matrix B:
1 8 7 6 5 4 3 2 9
11 18 17 16 15 2 3 4 5
9 8 7 6 5 4 3 2 1

Ordered matrix B
1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00
11.00  18.00  15.00  16.00  17.00  2.00   3.00   4.00   5.00
9.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   1.00
```

Figure 6.36: Results from Example 6.12

## 6.4 Exercises

- 1) Find the row and column indices of the maximum prime number of a rectangular matrix  $A(n,m)$ . Count the number of zero elements and print their indices.
- 2) Find the geometric mean of the elements on the perimeter and diagonals of a square matrix  $X(n,n)$ , if possible. If the geometric mean cannot be found, swap the positions of the maximum and minimum elements.
- 3) Create vector  $D$ , each element of which is the arithmetic mean of the elements in the rows of matrix  $C(k,m)$ , and vector  $G$ , each element of which is the product of elements in the corresponding column of matrix  $C$ .
- 4) Replace the maximum element in each column of matrix  $A(n,m)$  with the product of negative elements in the same column.
- 5) Find the maximum element above the main diagonal, and the minimum element below the antidiagonal of matrix  $A(n,n)$ . Then sort each column in ascending order.
- 6) Exchange the row of matrix  $P(n,m)$  with the smallest sum of the elements in the row, and the row with the maximum element in the matrix.
- 7) Move the maximum element of matrix  $F(k,p)$  to the upper right corner, and the minimum element to the bottom left.
- 8) Verify if matrix  $A(n,n)$  is a diagonal matrix (all elements are 0, except for the main diagonal), identity matrix (all elements are 0, with only ones on the main diagonal) or zero matrix (all elements are 0).
- 9) Create from matrix  $A(n,n)$  an upper triangular matrix  $B(n,n)$  (all elements

below the main diagonal are 0), a lower triangular matrix  $C(n,n)$  (all elements above the main diagonal are 0) and the diagonal matrix  $D(n,n)$  (all elements are 0, except for the main diagonals).

- 10) Given matrices  $A(m,n)$  and  $B(n,m)$ , find matrix  $C = (AB)^4$ .
- 11) Determine if the matrix  $B(n,n)$  is inverse of  $A(n,n)$ . The product of A and B in this case is the identity matrix.
- 12) Find the number of prime numbers that are not on the diagonals of matrix  $B(n,n)$ .
- 13) Verify if the maximum negative element in matrix  $A(n,n)$  lies on the main diagonal.
- 14) Copy the prime numbers from matrix A to array B. Sort the array in descending order.
- 15) Copy the positive numbers from integer matrix A to array B. Remove numbers from array B that have more ones than zeros in their binary form.
- 16) Four square integer matrices are given:  $A(n,n)$ ,  $B(n,n)$ ,  $C(n,n)$  and  $D(n,n)$ . Find the matrix containing the largest prime number.
- 17) Four square integer matrices are given;  $A(n,n)$ ,  $B(n,n)$ ,  $C(n,n)$  and  $D(n,n)$ . Find the matrices with prime numbers on their diagonals.
- 18) Three rectangular matrices are given:  $A(n,m)$ ,  $B(r,p)$  and  $C(k,q)$ . Find the matrices with only negative numbers on their perimeters.
- 19) Verify if the minimum positive element in matrix  $A(n,n)$  lies on the antidiagonal.
- 20) Matrices  $D(n,n)$ ,  $A(m,n)$  and  $B(n,m)$  are given. Find matrix  $C = BA$ . Determine if matrix  $C(n,n)$  is the inverse of  $D(n,n)$ . The product of matrices C and D in this case is the identity matrix.
- 21) Four square matrices  $A(n,n)$ ,  $B(n,n)$ ,  $C(n,n)$ ,  $D(n,n)$ , storing integers are given. Determine which of these matrices have numbers consisting of eights on the antidiagonal.
- 22) Replace the column in matrix  $P(n,m)$  with the largest sum of its elements with the column containing the maximum number of ones.
- 23) Four square integer matrices are given:  $A(n,n)$ ,  $B(n,n)$ ,  $C(n,n)$  and  $D(n,n)$ . Determine if any of these matrices contains only ones and twos on its antidiagonal.
- 24) Copy the prime numbers from the integer matrix A to array B. Delete the elements between the maximum and minimum elements in array B.
- 25) In matrix  $A(n,n)$  storing integers, sort the rows that do not contain a seven on the main diagonal.

**Endnotes:**

- 
- 1 Or  $h[2][4]$ .
  - 2 A transposed matrix is a matrix obtained by exchanging the rows and columns of the original matrix  $A(N,M)$ .
  - 3 Form Caption property is set to *Matrix Transposition*.
  - 4 The element at the intersection of the diagonals has indices  $(N \text{ div } 2 + 1, N \text{ div } 2 + 1)$ .
  - 5 Using the Boolean operations **and** and **or**, this complex condition can be written thus: *if (( $i = j$ ) and ( $a[i,j] <> 1$ )) or (( $i <> j$ ) and ( $a[i,j] <> 0$ )) then ...*
  - 6 The authors recommend that readers carefully study this example. It brings together almost all the main points covered up to this point.
  - 7 The mas array will store the row index containing the maximum.
  - 8 If some of the maximum elements are in the same row, then the mas array will have duplicate elements.

This page deliberately left blank.

# Chapter 7. Files in Free Pascal

This chapter will examine the Free Pascal's capabilities for working with files. While presenting the material, the reader will also become acquainted with Lazarus components for file selection. First, let us get acquainted with file types.

## 7.1 File Types in Free Pascal

Entering data from the keyboard is convenient when processing small amounts of data. When processing arrays containing hundreds of elements, however, it is inconvenient to enter data from the keyboard. In such cases, the data is conveniently stored in files. The program should then read the files, process the data and output the results to a screen or file. Let us see how this can be done. From a programmer's point of view, files can be divided into three classes:

- typed;
- untyped;
- text.

Files containing data of the same type (integer, real, arrays, etc.), the quantity of which is not necessarily known in advance, are called *typed files*. They end with a special "end of file" character and are saved in binary format which cannot be viewed with a regular text editor, but only with a specially written program.

In *untyped files*, data are read and written in blocks of a defined size. Such files store data of any kind and structure.

*Text files* contain characters. When saving data in a text file, all data are converted to a character type, in which they are saved. The data in such a file can be viewed using a text editor. Data in a text file are saved line by line. A special "end of line" marker is saved at the end of each line. The end of the file itself is denoted by the end of file marker.

To work with files, a file variable should be declared in the program. To work with a text file, the file variable (for example `f`) is declared using the `TextFile` (or `text`) keyword.

**var f: TextFile;**

To declare *typed*<sup>1</sup> files, you can declare a file variable as follows:

**var f: file of type;**

An *untyped file* is declared using the `file` keyword. Here are some examples of declaring file variables.

```

type
  arr = array [1..25] of real;
  ff = file of real;
var
  a: TextFile;   { File variable for a text file. }
  b: ff;          { File variable for a file of real numbers. }
  c: file of integer; { File variable for file of integers. }
  d: file of arr; { File variable d for a typed file whose
                     elements are arrays of 25 real numbers. }

```

Let us consider working with files of each type.

## 7.2 Working with Typed Files

Let us start our acquaintance with the methods of handling typed files with the procedures that are common to all file types.

### 7.2.1 The AssignFile Procedure

To start working with a file, a file variable in the program must be linked with a file on disk. For this, the **AssignFile(f,s)** procedure is used, where f the file variable name and s is the full file name on disk (the file must be in the current directory if the directory path is not provided).

Consider an example of using **AssignFile** for various operating systems.

```

var
  f: file of real;
begin
  // Example of AssignFile procedure for Windows.
  AssignFile(f, 'd:\pascal\abc.dat');
  // Example AssignFile procedure for Linux.
  AssignFile(f, '/home/norm/pascal/abc.dat');

```

### 7.2.2 The Reset and Rewrite Procedures

After linking the file variable to the file name on disk, open the file using the **reset** or **rewrite** procedures. The **reset(f)** procedure, where f is the file variable name, opens the file associated with the file variable f, after which the first item saved in the file becomes accessible. Data can now be read from, or written to, the file.

The **rewrite(f)** procedure creates an empty file (the file location on the disk is determined by the **AssignFile** procedure) for subsequently writing to it.



#### Attention

If a file associated with the file variable f already exists on the disk, then all the data in it will be destroyed after invoking the rewrite procedure.

### 7.2.3 The CloseFile Procedure

The **CloseFile(f)** procedure, where f is a file variable name, closes the file that was previously opened by the **rewrite** or **reset** procedure. **CloseFile(f)** must be used when closing the file to which the data was written.

Procedures for writing to a file do not write directly to the disk. They write data to a special memory area called the *file buffer*. After the buffer is full, all the data in it is transferred to file. The **CloseFile** procedure finishes transferring data from the file buffer to disk first, before closing the file. If the file is not closed manually, then it will be closed automatically when the program ends. However, when the file is closed automatically, any data remaining in the file buffer is not transferred to the disk and will be lost.



#### Attention

After reading or writing information to a file, be sure to close it using the **CloseFile** procedure. Closing a file after reading only is not necessary, but is strongly recommended, because the number of file handles available is limited.

### 7.2.4 The Rename Procedure

Renaming the file associated with the file variable f *must be done while it is closed*, using the **rename(f,s)** procedure, where f is the file variable and s is the new filename (a string).

### 7.2.5 The Erase Procedure

Deleting the file associated with the variable f is done using **erase(f)**, where f is the name of a file variable. *The file must be closed for this operation to be performed correctly.*

### 7.2.6 The eof Function

The **eof(f)** function, where f is the file variable name, returns **true** if the end of the file has been reached, or **false** otherwise. This function helps to determine if the end of the file has been reached and if the next piece of data is ready for reading.

### 7.2.7 Reading and Writing Data to File

Use the **write** procedure to write data to a file:

```
write(f, x1, x2, ..., xn);
write(f, x);
```

where:

- f is the file variable name,
- x, x1, x2, ..., xn are names of variables containing data to be written to the file.

The file type must be the same as the variable type. When the write procedure executes, the values x1, x2, ..., xn are written sequentially to the file (starting from the current position) associated with the file variable f.

To read data from a file associated with a file variable f, use the **read** procedure:

```
read(f, x1, x2, x3, ..., xn);
read(f, x);
```

where:

- f is the file variable name,
- x, x1, x2, ..., xn are the names of the variables into which data from the file will be placed.

The read procedure sequentially reads data from the file associated with the file variable f into variables x1, x2, ..., xn. After reading the current item, the next one becomes available. The read procedure does not check for the end of the file. Check for this using the eof function.

To write data to a file, do the following:

- 1) Declare a file variable.
- 2) Link it with a physical file (AssignFile procedure).
- 3) Open the file for writing (rewrite procedure).
- 4) Write data to the file (write procedure).
- 5) Close the file (CloseFile procedure).

Let us consider the creation of a typed file by solving the following simple task as an example.

EXAMPLE 7.1. Create a typed file and write n real natural numbers to it.

The algorithm for solving the problem has the following steps:

- 1) Open a file for writing using the rewrite statement.
- 2) Enter the value n.
- 3) Loop from 1 to n, enter the next real number a, and write it at once to the file using the write procedure.
- 4) Close the file using the closefile procedure.

The console application code for this problem is given below.

```
program pr1;
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
{ You can add units after this };
var
  f: file of real;
  i, n: integer;
  a: real;
begin
  // Link the file variable to the file on disk.
  // AssignFile(f, '/ home/norm/pascal/abc.dat'), in Linux;
  AssignFile(f, 'abc.dat');
  // Open an empty file (destructively) for writing.
  Rewrite(f);
  // Enter the number of items in the file.
  write('Enter no of items: ');
  readln(n);
  // In loop, enter the next item and write it to file.
  for i := 1 to n do
    begin
      write('a = ');
      readln(a);
      write(f,a);
    end;
  // Close the file after writing. This must be done!!!
  CloseFile(f);
end.
```

EXAMPLE 7.2. The previous problem created a file of real numbers with a dat extension. Open this file and remove the numbers that are less than the arithmetic mean, that are located between the largest and smallest numbers.

The problem requires that the user select a file for processing. For this, you will need a special Lazarus component to select the file. To solve Example 7.2 let us begin with an introduction to the OpenDialog component. This component creates a standard file selection dialog and is the first component shown on the Dialogs tab (Figure 7.1).

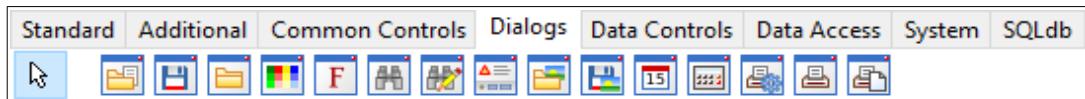


Figure 7.1: The components on the Dialogs Tab

Among the main properties of this component are:

- *FileName*: *String* is the full name of the selected file;
- *Filter*: *String* is the string that contains the filters for selecting files. This property can be set using the filter editor or special strings.

To call the filter editor (Figure 7.2), click on the ellipsis (...) on the Filter line of the Properties tab in the Object Inspector for this dialog.

The filter editor window consists of two columns: *Filter name* (for example, "Pascal programs") and *Filter*, which contains filter masks (for example, \*.pas, which selects all files with a pas extension).

A special string holds the filter names and their corresponding masks, separated by "|". The special string for the filters in Figure 7.2, has the form:

```
OpenDialog1.Filter :=  
'C_programs|*.c|Pascal_programs|*.pas| All_files|*.*';
```

The first filter in the list is the default filter. In the example shown in Figure 7.2 the default filter is "C programs". *InitialDir* is the name of the default directory for file selection. *DefaultExt* is the extension added to the filename by default if none was provided.

The main method for the OpenDialog component is the Boolean function Execute, which opens the dialog box, based on its properties. The Execute function returns **true** if the user selected a file by some method. If the user clicks Cancel, then the Execute method will return **false**. The name of the selected file is saved in the *FileName* property.

Thus, the call to the dialog box can be written like this:

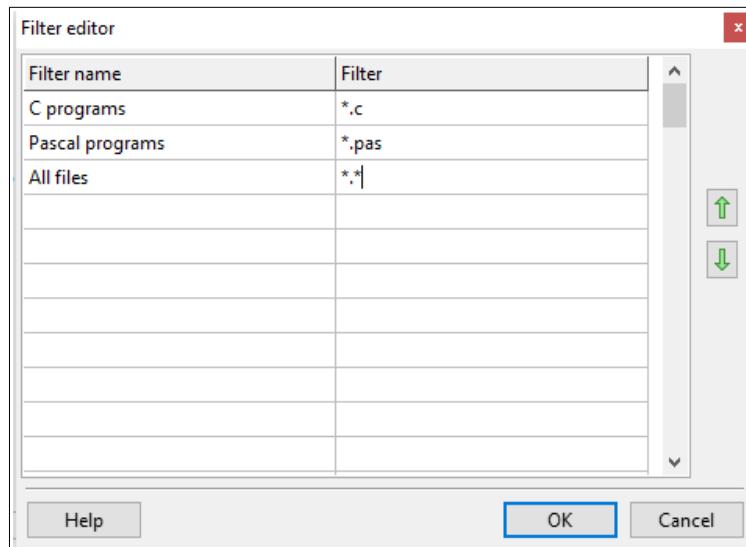


Figure 7.2: Filter editor

```

var
  s: String;
begin
  if OpenDialog1.Execute then
    s := OpenDialog1.FileName;
  {The name of the selected file is stored in s.}
end;
  
```

Having dealt with the file selection component, let us return to Example 7.2. The following steps to solve this example can be identified.

- 1) Select the file.
- 2) Read data from the file into an array of real numbers.
- 3) Remove from the array of real numbers all numbers less than the arithmetic mean, located between the largest and smallest numbers.
- 4) Write the modified array to the file.

Let us start developing a graphical application to solve this example (Project|Create Project|Application).

Place the following components on a form:

- 1) Two labels for the 2 memos below.
- 2) Memo1 stores and displays the source file.
- 3) Memo2 stores and displays the modified file.
- 4) OpenDialog1 for choosing the file to be processed.

- 5) Button1 starts the file processing.
- 6) Button2 ends the program.

Set the following form and component properties (Table 7.1-7.8).

*Table 7.1: Form properties*

<b>Property</b>	Caption	Name
<b>Value</b>	File Processing	frmFile

*Table 7.2: Label1 properties*

<b>Property</b>	Caption	Visible
<b>Value</b>	Original File:	False

*Table 7.3: Label2 properties*

<b>Property</b>	Caption	Visible
<b>Value</b>	Modified File	False

*Table 7.4: Memo1 properties*

<b>Property</b>	Lines	Visible
<b>Value</b>	"	False

*Table 7.5: Memo2 properties*

<b>Property</b>	Lines	Visible
<b>Value</b>	"	False

*Table 7.6: OpenDialog1 properties*

<b>Property</b>	InitDir	DefaultExt
<b>Value</b>	/home/pascal/6/pr2/	dat

*Table 7.7: Button1 properties*

<b>Property</b>	Caption	Name	Visible
<b>Value</b>	Process file	btnFile	True

Table 7.8: Button2 properties

Property	Caption	Name	Visible
Value	Exit	btnExit	False

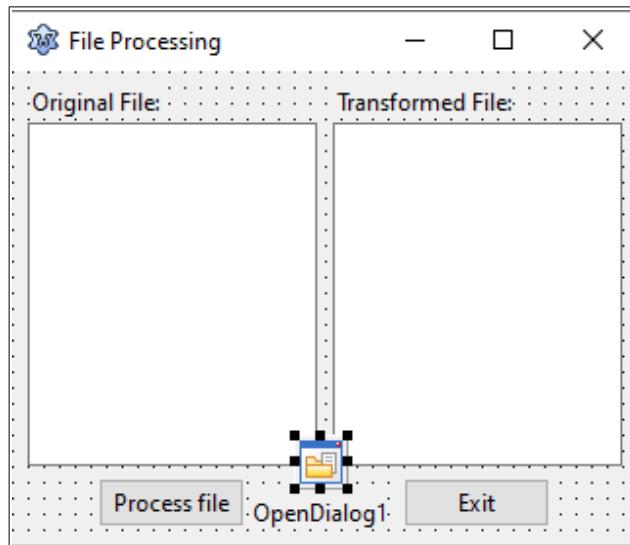


Figure 7.3: Form with components

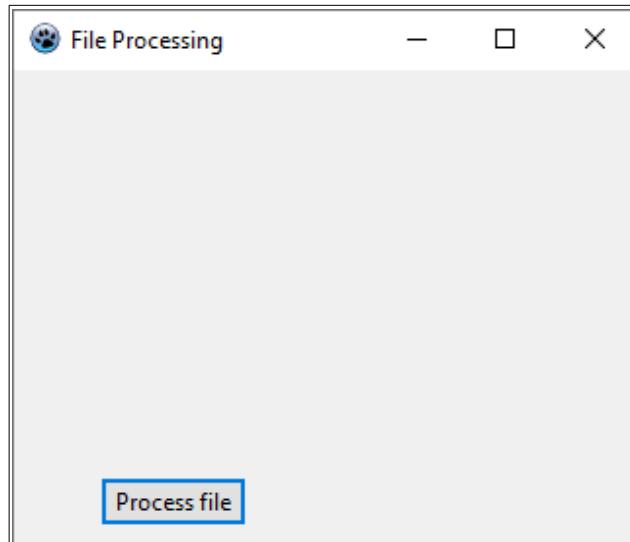


Figure 7.4: Application at startup

Arrange the components on the form as shown in Figure 7.3.

When the program starts, all the components except the *Process File* button will be invisible, with their *Visible* property set to **false**. Figure 7.4 shows the application

window at startup.

The design of the application interface is now complete. The code for the event handlers now needs to be written.

Set the Filter property of the OpenDialog1 component just before opening the Open File dialog box.

When you click the *Process File* button, the following actions should occur:

- 1) Set the filter property of the File Open dialog.
- 2) Select a file using the File Open dialog box.
- 3) Read data from the selected file into an array of real numbers.
- 4) Set the Visible property to **true** for Label1 and Memo1.
- 5) Output the contents of the array to Memo1.
- 6) Process the array.
- 7) Write the modified array to a file.
- 8) Set the Visible property to **true** for Label2 and Memo2.
- 9) Output of the modified array to Memo2.
- 10) Set the Visible property to **true** for btnExit.

Below is the commented code for the btnFileClick event handler.

```
procedure TfrmFile.btnFileClick(Sender: TObject);
var
  f: file of real;
  s: string;
  a: array [1..100] of real;
  nmax, nmin, i, j, n: integer;
  sum, max, min: real;
begin
  // Set the filter property of the File Open dialog.
  OpenDialog1.Filter := 'Real_files|*.dat|All_files|*.*';
  // Open a dialog box to select a file.
  if OpenDialog1.Execute then
    begin
      // Save name of the selected file in s.
      s := OpenDialog1.FileName;
      // Link the file variable to the file on disk.
      AssignFile(f, s);
      // Open the file ((non-destructively) for reading.
      reset(f);
      // Zero the array element counter.
      n := 0;
      // Make Label1 and Memo1 visible.
      Label1.Visible := true;
      Memo1.Visible := true;
```

```
// Save the sum of the array items in the variable sum.  
sum := 0;  
// Read next item Until the end of the file is reached.  
while not eof(f) do  
begin  
    // Increase the array index by 1.  
    n := n + 1;  
    // Read the next item from the file into array.  
    read(f, a [n]);  
    // Accumulate the sum of the array elements.  
    sum := sum + a[n];  
    // Display the current item in Memo1.  
    Memo1.Lines.Add(FloatToStr(a[n]));  
end;  
// Close the file.  
CloseFile(f);  
// Find the arithmetic mean of array items.  
sum := sum / n;  
{ Search for the maximum, minimum items  
  in the array, and their indices. }  
max := a[1];  
min := max;  
nmin := 1;  
nmax := 1;  
for i := 2 to n do  
begin  
    if a[i] > max then  
    begin  
        max := a[i];  
        nmax := i;  
    end;  
    if a[i] < min then  
    begin  
        min := a[i];  
        nmin := i;  
    end;  
end;  
{ If the maximum element comes before the minimum, swap nmin  
  and nmax. }  
if nmax < nmin then  
begin  
    i := nmax;  
    nmax := nmin;  
    nmin := i;  
end;  
i := nmin + 1;  
// Loop to remove items between maximum and minimum.  
while (i < nmax) do  
begin  
    // If item is less than arithmetic average, then  
    if a[i] < sum then
```

```

begin
  // remove it.
  for j := i to n - 1 do
    a[j] := a[j + 1];
  { After deleting, decrease number of items and index of
    the number maximum by 1. }
  n := n - 1;
  nmax := nmax - 1;
end
// If item is greater than arithmetic average, then
else
  // move on to the next one.
  i := i + 1;
end;
// Make Label2 and Memo2 visible.
Label2.Visible := true;
Memo2.Visible := true;
// Open the file for writing.
rewrite(f);
// Save modified file to a file and output to Memo2.
for i := 1 to n do
begin
  write(f, a[i]);
  Memo2.Lines.Add(FloatToStr(a[i]));
end;
// Close the file, and make the Button2 visible.
CloseFile(f);
btnExit.Visible := True;
end;
end;

```

When you click on the Close button, the program should exit. Therefore, the btnExitClick handler text will be very simple.

```

procedure TfrmFile.btnExitClick (Sender: TObject);
begin
  Close;
end;

```

Let us discuss how the application just created works. After launching the application, a window like that shown in Figure 7.4 appears. Clicking the *Process file* button opens a dialog box for selecting a file (Figure 7.5).

After clicking on the OK button, the application window becomes like that shown in Figure 7.6. Clicking the Close button exits the application. Example 7.2 is now solved and a functioning application has been developed.

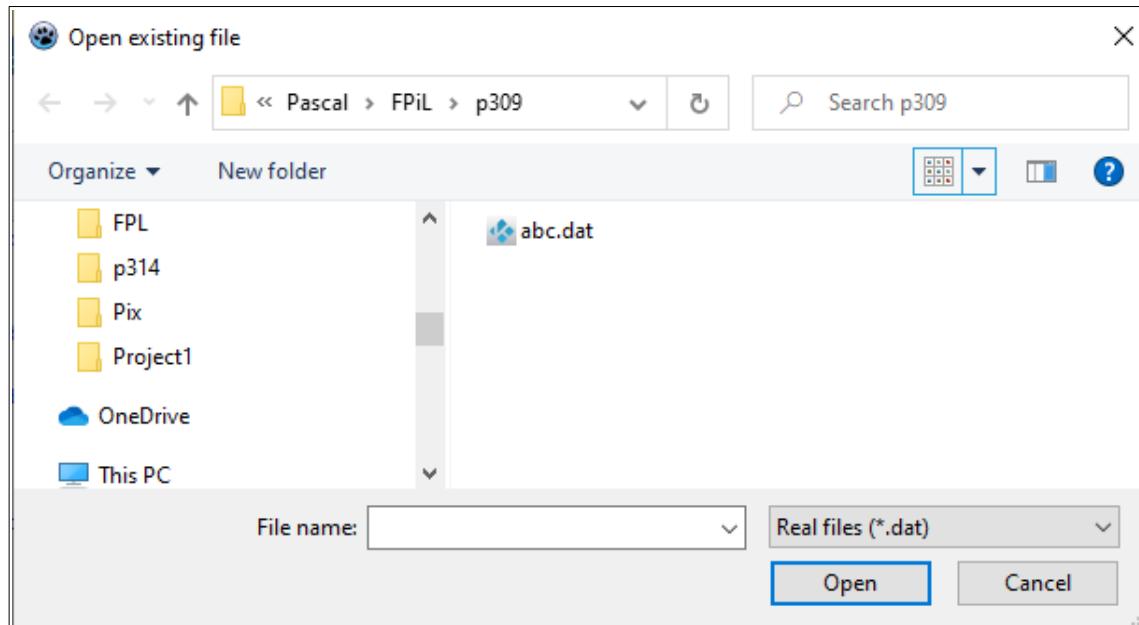


Figure 7.5: File selection dialog

In the solution to the problem, the entire contents of the file had to be read into an array, because the file is a sequential data structure. The data had to be processed in an array before writing them to a file. This method is not always convenient.

In addition, when reading data from a file into an array, we are limited by the declaration of the static array. If the file contains more elements than were declared in the static array (in our case > 100), the program will not work correctly. Using a dynamic array could solve this problem, but not conveniently if the number of elements in the array is unknown.

To solve this problem, Free Pascal allows direct access to files using the file routines discussed below.

### 7.2.8 The Filesize Function

One of the problems when copying data from a typed file to an array is the inability to correctly allocate memory for the array. When trying to allocate memory in this case, the number of elements in the file is not known in advance. The **filesize(f)** function, where f is a file variable, solves this problem. It returns a **longint** value containing the number of real components in the open file associated with the file variable f. For an empty file the function returns 0.

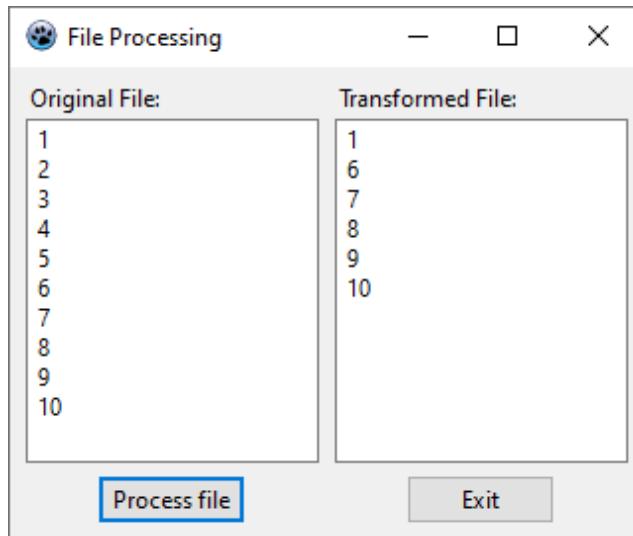


Figure 7.6: Example 7.2 program results

Let us consider the use of the filesize function for copying a set of real numbers from a file of real numbers to an array.

EXAMPLE 7.3. Copy the contents of a file of real numbers to an array.

```

program Project1;
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
  { You can add units after this };
var
  f: file of real;
  a: array of real;
  n, i: word;
begin
  // Link the file variable to the real file on disk.
  // assignfile(f, '/home/norm/pascal/abc.dat');
  assignfile(f, 'abc.dat');
  // Open the file for reading.
  reset(f);
  // n is the number of elements in the file f.
  n := filesize(f);
  writeln('The file has ', n, ' numbers.');
  // Allocate memory for dynamic array a, for n real numbers.
  Setlength(a, n);
  for i := 0 to n-1 do
    begin

```

```
// Read element from file to array.  
read(f, a[i]);  
// Display element on screen.  
write(a[i]:1:3, ' ')  
end;  
// Close the file.  
CloseFile(f);  
readln;  
end.
```

### 7.2.9 The filepos Function

The **filepos(f)** function returns a longint containing the current position in an open file associated with the file variable f. Immediately after opening a file the value of **filepos(f)** is 0. After reading the last item from file, **filepos(f)** is the same as **filesize(f)**. Reaching the end of a file can be checked as follows:

```
if filepos(f) = filesize(f) then  
  writeln('End of file reached.');
```

### 7.2.10 The seek Procedure

The **seek(f,n)** procedure sets a pointer in an open file associated with file variable f, to item n (item indexing begins from 0). The item can then be read.

### 7.2.11 The truncate Procedure

The **truncate(f)** procedure, where f is the name of the file variable, deletes the rest of the open file, starting with the current item, and adds the end of file marker in its place. Examples of the use of the **seek** and **truncate** procedures will be shown in the next two file handling problems.

EXAMPLE 7.4. Swap the largest and smallest numbers in the file abc.dat, containing real numbers.

Let us consider two options for solving this example.

In the first, a console version of the program, after reading the file into an array, search for the largest and smallest numbers and their indices. Then the largest and smallest numbers are swapped in the file.

```
program Project1;
{$mode objfpc }{$H+}
uses
  Classes, SysUtils
  {You can add units after this};
var
  f: file of real;
  i, nmax, nmin: integer;
  n: longint;
  a: array of real;
  max, min: real;
begin
  // AssignFile(f, '/home/norm/pascal/abc.dat');
  AssignFile(f, 'abc.dat');
  reset(f);
  n = filesize(f);
  setlength(a, n);
  // Read the file components into an array a.
  for i := 0 to n - 1 do
  begin
    begin
      read(f, a[i]);
      write(a[i]:1:2, ' ');
    end;
    max := a[0]; nmax := 0;
    min := a[0]; nmin := 0;
    { Main loop to find the largest and smallest numbers
      and their indices.}
    for i := 1 to n - 1 do
    begin
      if a[i] > max then
      begin
        max := a[i];
        nmax := i;
      end;
      if a[i] < min then
      begin
        min := a[i];
        nmin := i;
      end;
    end;
  end;
  // Overwrite the maximum and minimum values to the file.
  // Move the file pointer to the maximum element.
  seek(f, nmax);
  // Write the minimum in place of the maximum element.
  write(f, min);
  // Move the file pointer to the minimum element.
  seek(f, nmin);
  // Write the minimum in place of the maximum element.
  write(f, max);
  // Be sure to close the file.
  closefile(f);
```

```
readln;
end.
```

The second version of the program is a full-fledged graphical application. An array will not be used in this version. There will be a single loop, in which each item will be read into the variable `a` and the search for minimum and maximum elements in the file, and their indices, will be carried out. After this, the minimum and maximum values will be swapped in the file.

Start developing the program with the template for a graphical application. (Project|Create Project|Application). Place the following components on the form:

- 1) Two labels, Label1 and Label2.
- 2) Text box Edit1, to store the content of the source file.
- 3) Text box Edit2, to store the modified file.
- 4) OpenDialog1, to choose the file to be processed.
- 5) Button1, to start the program.
- 6) Button2, to end the program.

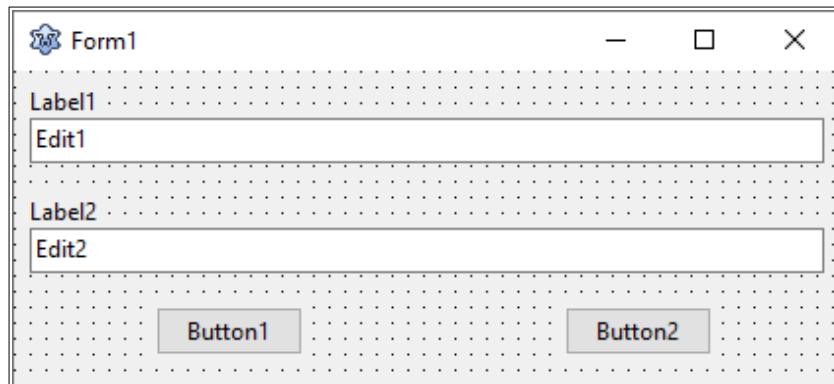


Figure 7.7: Initial form and components

Place the components on the form as shown in Figure 7.7.

The main properties of the components will be set programmatically when creating form. The FormCreate routine will be:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Caption := 'Swap Maximum & Minimum in File';
  Label1.Caption := 'Source file:';
  Label1.Visible := false;
  Label2.Caption := 'Modified file:';
```

```

Label2.Visible := false;
Edit1.Text := '';
Edit1.Visible := false;
Edit2.Text := '';
Edit2.Visible := false;
OpenDialog1.Filter := 'Real number files (*.dat)|All files (*.*)';
// OpenDialog1.InitialDir := '/home/norm/pascal/pr1';
Button1.Caption := 'Modify File';
Button2.Caption := 'Exit';
Button2.Visible := false;
end;

```

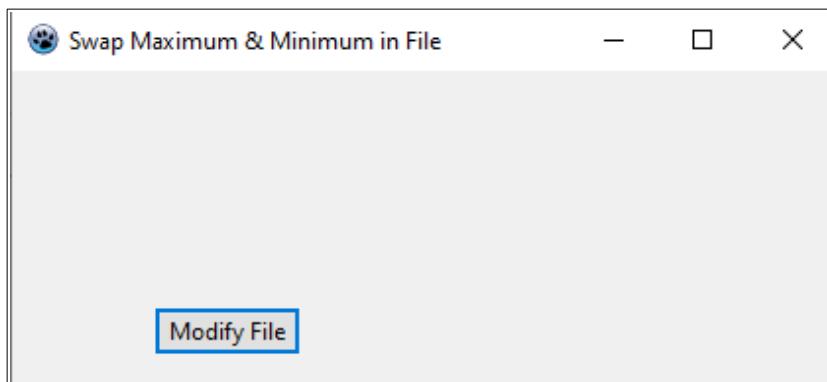


Figure 7.8: Program for Example 7.4 at startup

The authors hope that for readers who have read the book up to this point, the code for TForm1.FormCreate procedure needs no explanation.

When the program starts, it should look like the form shown in Figure 7.8.

The program will execute its main actions when the Modify File button is clicked. The commented code of the corresponding subroutine is shown below.

```

procedure TForm1.Button1Click(Sender: TObject);
var
  f: file of real;
  i, nmax, nmin: integer;
  a, max, min: real;
  s1, s: string;
begin
  // Open a dialog box to select a file.
  if OpenDialog1.Execute then
  begin
    // Save the full file name in the variable s.
    s := OpenDialog1.FileName;
    // Link the file variable to the file on disk.
    assignfile(f,s);
    // Open the file in read mode.

```

```
reset(f);
// Make Label1 and Edit1 visible.
Label1.Visible := true;
Edit1.Visible := true;
// The string s1 will store the file contents.
S1 := '';
// Loop and read all file elements sequentially.
for i := 0 to FileSize(f) - 1 do
begin
  // Read file element into variable a.
  read(f, a);
  if i = 0 then
    { Initial assignment of maximum and minimum values
      and their indices.}
  begin
    max := a; nmax := i;
    min := a; nmin := i;
  end
  else
  begin
    // Compare current item with the maximum (minimum).
    if max < a then
    begin
      max := a;
      nmax := i;
    end;
    if min > a then
    begin
      min := a;
      nmin := i;
    end;
  end;
  // Add another item to the output string s1.
  s1 := s1 + FloatToStr(a) + ' ';
end;
// Display file contents in Edit1.
Edit1.Text := s1;
// Disallow editing the text in the text box.
Edit1.ReadOnly := true;
// Overwrite the maximum and minimum values in the file.
// Move the file pointer to the maximum element.
seek(f, nmax);
// Write the minimum in place of the maximum element.
write(f, min);
// Move the file pointer to the minimum element.
seek(f, nmin);
// Write the maximum in place of the minimum element.
write(f, max);
// Be sure to close the file.
CloseFile(f);
reset(f);
```

```
// Make Label2 and Edit2 visible.  
Label2.Visible := true;  
Edit2.Visible := true;  
// Read data from the modified file  
s1 := '';  
for i := 0 to FileSize(f) - 1 do  
begin  
    read (f, a);  
    // Add item from modified file to output string s1.  
    s1 := s1 + FloatToStr(a) + ' ';  
end;  
// Display the modified file contents in Edit2.  
Edit2.Text := s1;  
// Disallow editing text in the text box.  
Edit2.ReadOnly := true;  
// Make Button2 visible.  
Button2.Visible := true;  
CloseFile(f);  
end;  
end;
```

When you click the Exit button, the program should exit. Thus, the Button2Click handler text will be very simple.

```
procedure TForm1.Button2Click (Sender: TObject);  
begin  
    Close;  
end;
```

When you click the Modify File button, a file selection window appears, like that shown in Figure 7.5. After selecting a file, the maximum and minimum elements in it are swapped and the original and modified file contents are displayed (see Figure 7.9). When the Exit button is clicked, the program ends.

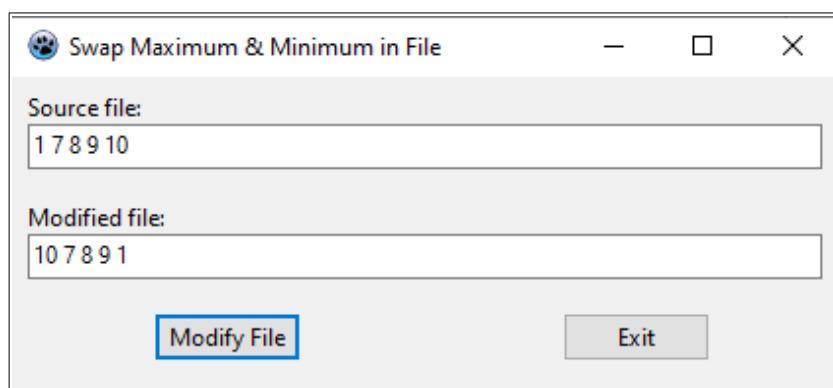


Figure 7.9: Example 7.4 output after modifying file

When solving Example 7.4, the seek procedure was used, which made it possible to change the data in the file directly on the disk, without reading them into an array.

EXAMPLE 7.5. Delete the largest and smallest numbers from a file of real numbers abc.dat.

Let us consider two programs to solve this problem. Both will be console applications, that may be developed in Geany or Lazarus. The first algorithm is as follows. Read the file elements into an array and process the array to find the largest and the smallest numbers and their indices. Open the file for writing and add to it all the numbers, except the largest and smallest.

```
program Project1;
  {$mode objfpc} {$H+}
uses
  Classes, SysUtils
  {you can add units after this};
var
  f: file of real;
  max, min: real;
  j, i, nmax, nmin: integer;
  a: array [1..300] of real;
begin
  // AssignFile(f, '/home/norm/pascal/abc.dat');
  AssignFile(f, 'abc.dat');
  reset(f);
  // Save the number of file elements in the j variable.
  j := filesize(f);
  // Read the file elements into an array a.
  for i := 1 to j do
    read(f, a[i]);
  for i := 1 to j do
    write(a[i]:1:2, ' ');
  writeln;
  closefile(f);
  // Open the file (destructively) for writing.
  rewrite(f);
  { Initial assignment of the maximum and
    minimum array elements and their indices.}
  max := a [1]; min := a [1];
  nmax := 1; nmin := 1;
  { The main loop for finding the maximum and
    minimum array elements and their indices. }
  for i := 2 to j do
  begin
```

---

```

if a[i] > max then
begin
  max := a[i];
  nmax := i;
end;
if a[i] < min then
begin
  min := a[i];
  nmin := i;
end;
end;
{ Copy array elements to file, except
  elements with indices nmax and nmin. }
writeln('Modified file:');
for i := 1 to j do
  if (i <> nmax) and (i <> nmin) then
begin
  write(f, a[i]);
  // Output file element to screen.
  write(a[i]:1:2, ' ');
end;
closefile(f);
readln;
end.

```

The second program works as follows. Find the maximum and the minimum file elements and their indices. If  $n_{\min} > n_{\max}$ , then swap them. Elements lying between the minimum and maximum (between elements with indices  $n_{\min}$  and  $n_{\max}$ ), should then be shifted one place to the left. This will delete element  $n_{\min}$ . After that, all elements after the element with index  $n_{\max}$  should be shifted two places to the left. This will delete the maximum element. Finally, the last two file elements should be removed.

```

program Project1;
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
  { you can add units after this };
var
  f: file of real;
  a: real;
  max, min: real;
  i, nmax, nmin: integer;
begin
  // assignfile(f, '/home/norm/pascal/abc.dat');
  assignfile(f, 'abc.dat');
  reset(f);
  // Find maximum and minimum file elements and their indices.
  writeln('Source file:');

```

```
for i := 0 to filesize(f) - 1 do
begin
  read(f, a);
  write(a:1:2, ' ');
  if i = 0 then
begin
  max := a;
  nmax := i;
  min := a;
  nmin := i;
end
else
begin
  if a > max then
begin
  max := a;
  nmax := i;
end;
  if a < min then
begin
  min := a;
  nmin := i;
end;
end;
end;
writeln;
// If nmax < and nmin, swap them.
if nmax < nmin then
begin
  i := nmax;
  nmax := nmin;
  nmin := i;
end;
// Shift elements between nmin and nmax one place left.
for i := nmin to nmax - 2 do
begin
  seek(f, i + 1);
  read(f, a);
  seek(f, i);
  write(f, a);
end;
// Move the items after nmax two places left.
for i := nmax to filesize(f) - 3 do
begin
  seek(f, i + 1);
  read(f, a);
  write(a:1:2);
  seek(f, i - 1);
  write(f, a);
  write(a:1:2);
end;
```

---

```
// Truncate last two elements.
truncate(f);
closefile(f);
reset(f);
// Output the modified file.
writeln('Modified file:');
for i := 1 to filesize(f) do
begin
  read(f, a);
  // Output file element to screen.
  write(a:1:2, ' ');
end;
closefile(f);
readln;
end.
```

After running the program, the largest and smallest numbers in the real number file were deleted.

In addition to typed files, untyped binary files are widely used to process numeric data.

## 7.3 Untyped Binary Files in Free Pascal

It is convenient to use untyped binary files to store data of various types. When opening untyped binary files, use the extended syntax for the reset and rewrite procedures.

```
reset(var f: file; BufSize: word);
rewrite(var f: file; BufSize: word);
```

The optional BufSize parameter defines the size of the data transfer block, which is the number of bytes read or written to the data file per call. If this parameter is absent, then the default value (128) is used. Maximum flexibility can be achieved with a block size of 1 byte. To write data to an untyped file, use the BlockWrite procedure:

```
BlockWrite(var f: file; var X; Count: word; var WriteCount: word);
```

where f is file variable name, X is the data to be written to the file, Count is the number of blocks of size BufSize, written to file and the optional WriteCount parameter is the number of blocks that were actually written to the file.

The BlockWrite routine writes Count blocks to the file associated with the file variable f<sup>2</sup>, from variable X. If the data were written to the file correctly, WriteCount will be the same as Count.

Use the BlockRead procedure to read data from an untyped file:

**BlockRead(var f: file; var Y; Count: word; var ReadCount: word);**

where f is the file variable name, Y is the variable that stores the data read from the file, Count is the number of BufSize blocks to read from file and the optional ReadCount parameter is the number of BufSize blocks actually read from the file.

BlockRead reads Count blocks from the file associated with file variable f into variable Y. When the data is read correctly from the file, ReadCount will be the same as Count.

Working with untyped binary files will be explored by solving several example problems.

EXAMPLE 7.6. An untyped binary file stores an array of real numbers and the number of array elements in it. Remove the largest real number from the file.

Write a console application to create the file.

```
program Project1;
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
  {you can add units after this};
type
  arr = array [1..100000] of real;
var
  i, N: word;
  f: file;
  x: ^arr;
begin
  // Assignfile(f, '/home/norm/pascal/new_file.dat') in Linux;
  Assignfile(f, 'new_file.dat');
  // Open file for writing. Data transfer block size = 1 byte.
  rewrite(f, 1);
  // Enter N - the number of positive integers in the file
  write('N = ');
  readln(N);
  { Writing N to file f takes 2 blocks at 1 byte each,
    since (sizeof(word) = 2). }
  BlockWrite (f, N, sizeof(word));
  // Allocate memory for N real elements.
  getmem(x, N * sizeof(real));
  // Input array.
  writeln('Enter an array:');
  for i := 1 to N do
  begin
    write('x(', i, ') = ');
```

```

readln(x^[i]);
end;
for i := 1 to N do
{ Writing real number x^[i] to file f takes 8 blocks at
  1 byte each, since (sizeof(real) = 8).}
  BlockWrite(f, x^[i], sizeof(real));
{ Writing an array to a file can be done without a loop,
  by calling the BlockWrite function as follows:
  BlockWrite (f, x^, N * sizeof (real));
  Writing an array to a untyped file without a loop, by using
  a single BlockWrite(f, X^, N * sizeof (type)) statement
  is considered the preferred option by the authors.}
// Close the file.
CloseFile(f);
// Free up memory.
freemem(x, N * sizeof(real));
readln;
end.

```

Start developing the program for solving Example 7.6 by using the application template (Project|Create| Project Application) in Lazarus. Place the following components on the form:

- 1) Two labels, Label1 and Label2.
- 2) A text box Edit1, to store and display the source file.
- 3) A text Edit2, to store and display the modified file.
- 4) A file selection dialog OpenDialog1, for selecting a file to process.
- 5) A button Button1, to modify the file.

Place the components on the form, as shown in Figure 7.10.

The main properties of the components will be set in the program when creating the form. The program for solving Example 7.6 will run when the button is clicked.

The full code for the unit, with the necessary comments, is given below:

```

unit Unit1;
{$mode objfpc} {$H+}

interface

uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls;
type
  { TForm1}
TForm1 = class(TForm)
  // The form contains:
  // A button.

```

```
Button1: TButton;
```

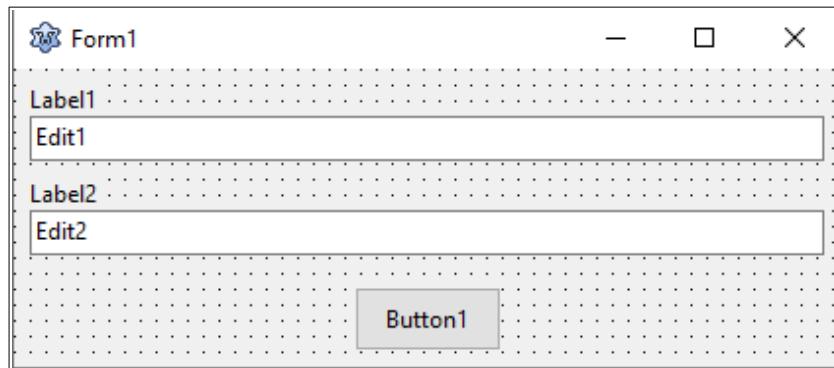


Figure 7.10: Form with components

```
// A text box, to display the original file.
Edit1: TEdit;
// A text box, to display the modified file.
Edit2: TEdit;
// Labels for labeling text boxes.
Label1: TLabel;
Label2: TLabel;
// Component for selecting a file.
OpenDialog1: TOpenDialog;
procedure Button1Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
arr = array [1..50000] of real;
var
  Form1: TForm1;
implementation
{$R *.lfm}

{TForm1}

// Event handler for clicking on the button.
procedure TForm1.Button1Click(Sender: TObject);
var
  X: ^arr;
  f: file;
  i, N, Nf, nmax: word;
  max: real;
  s: string;
begin
```

```
// File => array X => Edit1
if OpenDialog1.Execute then
begin
  // Select file.
  s := OpenDialog1.FileName;
  // Link the file variable to the file in variable s.
  AssignFile(f, s);
  // Open file for reading, data transfer block = 1 byte.
  reset(f, 1);
  { Read a whole number (word) N from file f, or 2 blocks
    at 1 byte each, since (sizeof(word) = 2).}
  BlockRead(f, N, sizeof(word));
  // Allocate memory for N real number elements.
  getmem(X, N * sizeof (real));
  // Nf ensures memory will be freed correctly.
  Nf := N;
  { Read N real numbers into variable x from file f,
    or read N * sizeof(real) blocks of 1 byte.
    Populate array X with values from file.}
  BlockRead(f, X^, N * sizeof(real));
  // Make Label1 and Edit1 visible.
  Label1.Visible := true;
  Edit1.Visible := true;
  for i := 1 to N do
    { Add element from array X to Edit1. Thus, Edit1 will
      display the original array.}
    Edit1.Text := Edit1.Text + FloatToStrF(X^[i], ffFixed, 5, 2) +
  ';
  // Find the maximum element and its index.
  max := X^[1];
  nmax := 1;
  for i := 2 to N do
    if X^[i] > max then
      begin
        max := X^[i];
        nmax := i;
      end;

  // Remove the maximum element from the array.
  for i := nmax to N - 1 do
    X^[i] := X^[i + 1];
  // Decrease the number of elements in the array.
  N := N - 1;
  // Close the file.
  CloseFile(f);

  // Reopen original file for writing.
  AssignFile(f, s);
  rewrite(f, 1);
  { Write an integer N, or 2 blocks to the file f,
    at 1 byte per block, since sizeof(word) = 2.}
```

```
BlockWrite(f, N, sizeof(word));
// Write array to file using BlockWrite.
BlockWrite(f, X^, N * sizeof(real));
// Close the file.
CloseFile(f);
// Free memory.
freemem(X, Nf * sizeof(word));

// Read data from the modified file.
AssignFile(f, s);
reset(f, 1);
{ Read from file f an integer (word) N, or 2 blocks
  at 1 byte per block, since sizeof(word) = 2.}
BlockRead(f, N, sizeof(word));
// Allocate memory for N elements of array x real.
getmem(X, N * sizeof(real));
Nf := N;
{ Read from file f N real numbers into variable X,
  or read N * sizeof(real) blocks of 1 byte.
  Fill the X array with the values stored in the file.}
BlockRead(f, X^, N * sizeof(real));
// Make Label2 and Edit2 visible.
Label2.Visible := true;
Edit2.Visible := true;
// Edit2 displays the contents of array X.
for i := 1 to N do
  Edit2.Text:=Edit2.Text+FloatToStrF(x^[i], ffFixed,5,2)+' ';
// Close the file and free the memory.
CloseFile(f);
freemem(X, Nf * sizeof(real));
end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
{ Set properties of Edit1, Edit2, Label1, Label2 and Button1.
  Make all components invisible, except button. }
Form1.Caption := 'Untyped Binary File';
Label1.Caption := 'Content of source file:';
Edit1.Clear;
Label2.Caption := 'Content of modified file:';
Edit2.Clear;
Button1.Caption := 'Modify File';
Label1.Visible := false;
Edit1.Visible := false;
Label2.Visible := false;
Edit2.Visible := false;
end;

end.
```

When the program starts, the application window will look like that shown in Figure 7.11. After clicking the Convert File button, a file selection window will appear(see Figure 7.12).

Figure 7.13 shows the program after modifying the source file.

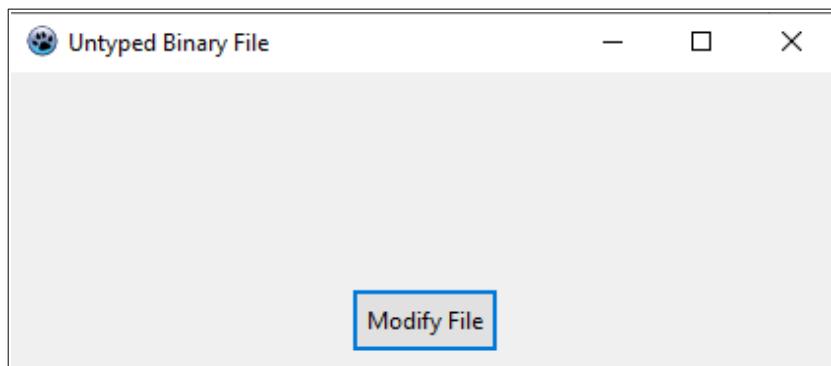


Figure 7.11: Program for Example 7.6 at startup

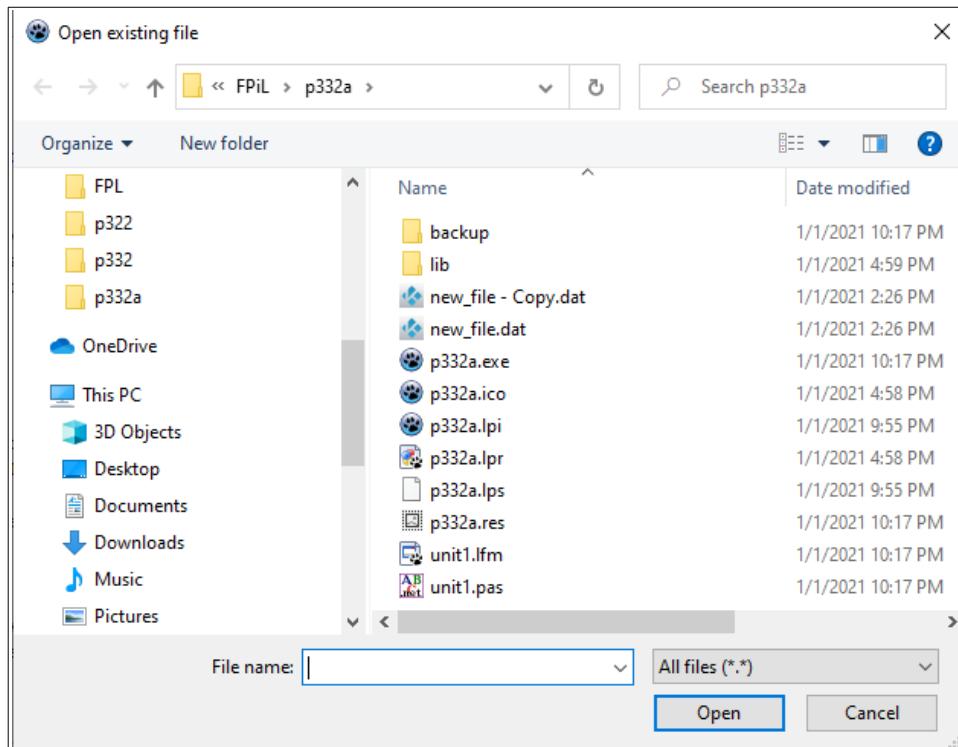


Figure 7.12: File selection window

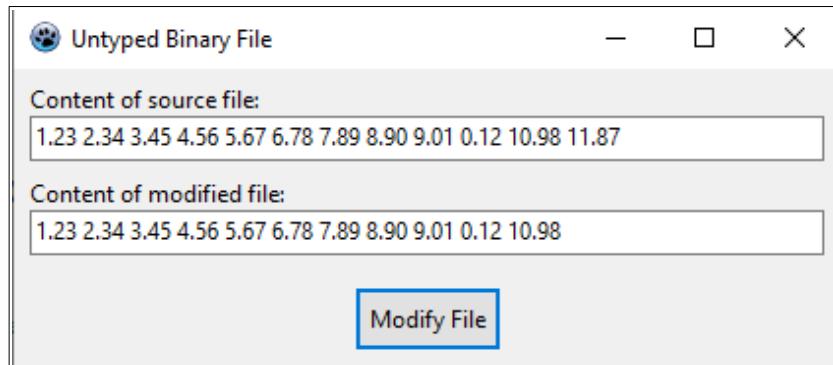


Figure 7.13: Running program

EXAMPLE 7.7. An untyped file contains matrices  $A(N,M)$  and  $B(P,L)$  of real numbers, and their sizes. Multiply, if possible,  $A$  by  $B$ , and add the resulting matrix  $C = A \cdot B$  to the file.

Start solving the problem by writing a console file creation program. To do this, Let define the file structure. Let it store two values  $N$  and  $M$  of word type, then the matrix of real numbers  $A(N,M)$ , then the values  $P$  and  $L$  of word type and matrix  $B(P,L)$ .

Below is the listing for the program for creating the untyped binary file, with comments.

```
program Project1;
{$mode objfpc}{$H+}

uses
  Classes, SysUtils
  {you can add units after this};
var
  f: file;
  i, j, n, m, l, p: word;
  a, b: array [1..20, 1..20] of real;
begin
  AssignFile(f, 'prim.dat');
  // Open file for writing, with data transfer block = 1 byte.
  rewrite(f, 1);
  write('N = ');
  readln(N);
  write('M = ');
  readln(M);
  { Write an integer N, or 2 blocks to file f, since
    sizeof(word) = 2, one byte each from variable N.}
```

```

BlockWrite(f, N, sizeof(word));
{ Write an integer N, or 2 blocks to file f, since
  sizeof(word) = 2, one byte each from variable M.}
BlockWrite (f, M, sizeof(word));
writeln('Matrix A:');
for i := 1 to N do
  for j := 1 to M do
    begin
      // Enter the next element of the matrix.
      read(A[i,j]);
      { Write real number A[i,j] to file f, in sizeof(real)
        blocks of one byte.}
      BlockWrite(f, A[i,j], sizeof(real));
    end;
  write('L = ');
  readln(L);
  write('P = ');
  readln(P);
  // Write to file 2 blocks, one byte each for variable L.
  BlockWrite(f, L, sizeof(word));
  // Write to file 2 blocks, one byte each from variable P.
  BlockWrite(f, P, sizeof(word));
writeln('Matrix B:');
for i := 1 to L do
  for j := 1 to P do
    begin
      read(B[i,j]);
      // Write to file sizeof(real) blocks one byte from B[i,j].
      BlockWrite(f, B[i,j], sizeof(real));
    end;
  // Close the file.
  CloseFile(f);
end.

```

Now, write a Lazarus console application, that will read matrices A, B and their sizes from the untyped file, after which we will calculate matrix C, the product of A and B.

```

program Project1;
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
{you can add units after this};
var
  f: file;
  k, i, j, n, m, l, p: word;
  A, B, C: array [1..20, 1..20] of real;
begin
  // Link the file variable to the file on disk.
  AssignFile(f, 'prim.dat');
  // Open the file and set the block size to 1 byte.

```

```
reset(f, 1);
{ Read 2 bytes into N from file f, since (sizeof(word)) = 2
  blocks, at one byte at a time.}
BlockRead(f, N, sizeof(word));
{ Read 2 bytes into M from file f, since (sizeof(word)) = 2
  blocks one byte at a time.}
BlockRead(f, M, sizeof(word));
for i := 1 to N do
  for j := 1 to M do
    { Read into A[i,j] from file f, sizeof(real) bytes,
      (sizeof(real)) blocks one byte at a time.}
    BlockRead(f, A [i, j], sizeof(real));
// Read 2 bytes into variable L from file f.
BlockRead(f, L, sizeof(word));
// Read 2 bytes into variable P from file f.
BlockRead(f, P, sizeof(word));
for i := 1 to L do
  for j := 1 to P do
    // Read into B[i,j] from file f sizeof (real) bytes.
    BlockRead (f, B[i,j], sizeof(real));
// Display the matrix A on the screen.
writeln;
writeln('Matrix A:');
for i := 1 to N do
begin
  for j := 1 to M do
    write(A[i,j]:1:2, ' ');
  writeln;
end;
// Display matrix B on the screen.
writeln;
writeln('Matrix B:');
for i := 1 to l do
begin
  for j := 1 to P do
    write(B[i,j]:1:2, ' ');
  writeln;
end;
// Check if matrix multiplication is possible, if yes,
if m = 1 then
begin
  // then multiply matrix A by B
  for i := 1 to n do
    for j := 1 to p do
      begin
        C[i,j] := 0;
        for k := 1 to m do
          C[i,j] := C[i,j] + A[i,k] * B[k,j]
      end;
  // Output matrix C.
  writeln;
```

```

writeln('Matrix C = A * B:');
for i := 1 to n do
begin
  for j := 1 to p do
    write(C[i,j]:1:2, ' ');
  writeln;
end;
// Add integer n to file f, or 2 blocks of one byte each
BlockWrite(f, n, sizeof(word));
// Add integer p to file f, or 2 blocks of one byte each
BlockWrite(f, p, sizeof(word));
for i := 1 to n do
  for j := 1 to p do
    { Write a real number C[i,j] to file f, or
      sizeof(real) blocks of one byte.}
    BlockWrite(f, C[i,j], sizeof(real));
end
else
  writeln('Matrices cannot be multiplied.');
// Close the file.
CloseFile(f);
end.

```

## 7.4 Text Files in Free Pascal

The following should be considered when working with text files:

- 1) The behavior of the **reset**, **rewrite**, **close**, **rename**, **erase** and **eof** procedures is the same as with *typed files*.
- 2) The **seek**, **truncate** and **filepos** procedures do not work with *text files*.
- 3) The **append(f)** procedure can be used for opening a text file, where f file variable name. This procedure is used to open file in append mode to the end of the file. It only applies to existing files and opens and prepares them for adding information at the end of the file.
- 4) Reading and writing to a text file is carried out using the **write**, **writeln**, **read** and **readln** procedures, with the following structure:

```

read(f, x1, x2, x3,..., xn);
read(f, x);
readln(f, x1, x2, x3,..., xn);
readln(f, x);
write(f, x1, x2, x3,..., xn);
write(f, x);
writeln(f, x1, x2, x3,..., xn);
writeln(f, x);

```

In these statements f is the file variable. In the read statements (**read**, **readln**), x,

x1, x2, x3, ..., xn are the variables that hold data read from a file. In the write statements (**write**, **writeln**), x, x1, x2, x3, ..., xn are the variables or constants, that are written to the file.

There are several peculiarities when working with the write, writeln, read, readln statements with text files. Variable names can be integer, real, character and string type. Before writing data to a text file, the write procedure converts them to a string type. The writeln statement differs by adding an End of Line marker after the specified variables and constants have been written to the file.

When reading data from a text file using the read or readln procedures, string type data are converted to the desired data type. If the conversion is not possible, then an error code is generated, the value of which can be found by referring to the IOResult function. The Free Pascal compiler allows you to generate program code in two modes: with Input / Output validation and without it.

A compilation mode key may be included in the program. Besides, a transfer of Input / Output error control from one state to another is provided, thus:

- {\$I+} if the Input / Output error checking mode is enabled;
- {\$I-} if the Input / Output error checking mode is disabled.

The default is usually {\$I+}. The modes can be repeatedly enabled or disabled, creating regions with or without input control. All compilation keys are described in the appendix.

When the check mode is enabled, any Input / Output error will be fatal, and the program halts with an error number.

If you remove the check mode, then when an Input / Output error occurs the program will not stop but will continue to work from the next operation. The result of the Input / Output operation will be undefined.

To poll the error code, it is better to use the special IOResult function, but remember that you can only poll it once after each input or output operation. It resets its value after each call. IOResult returns an integer corresponding to the code of the last Input / Output error. If IOResult = 0, then there was no Input / Output error. Otherwise IOResult returns an error code, some of which are listed in Table 7.9.

- 1) On opening, check if the specified file exists and data can be read from it.

```
assign(f, 'abc.dat');
{$I-}
reset(f);
{$I+}
if IOResult <> 0 then
```

```

writeln('File not found or not readable')
else
begin
  read(f, ...);
  close(f);
end;

```

Table 7.9: Input / Output error codes

Error Code	Description
2	File not found
3	Path not found
4	Too many open files
5	Access denied
12	Invalid access mode
15	Wrong disk number
16	Current directory cannot be deleted
100	Error while reading from disk
101	Error writing to disk
102	AssignFile procedure not applied
103	File not open
104	File not open for input
105	File not open for output
106	Wrong number
150	Disk is write protected

Let us look at some practical examples of handling Input / Output errors:

- 2) Check if the number entered from the keyboard is an integer.

```

var i: integer;
begin
{$I-}
repeat
  write('Enter integer i: ');
  readln(i);
until(IOResult = 0);
{$I+}
{ This loop repeats until the user enters an integer. }
end.

```

When working with a text file, there are special rules for reading and writing the values of variables:

- When entering numeric values, two numbers may be separated by at least one space between them, or by a tab, or by an End of Line character.
- When entering strings, the current string starts right after the previously entered string. The number of characters that can be entered equals the declared string length. If the End of Line character is encountered while reading, then the current string ends. The End of Line character itself is a separator and is never added to a string.
- After the **readIn** procedure reads the current line in a file, the cursor moves to the next line and continues reading.

As an example of working with text files, consider the following problem.

EXAMPLE 7.8. The text file abc.txt contains matrices A(N,M) and B(N,M) and their sizes. Compute matrix C = A + B and add it to abc.txt.

First, create a text file abc.txt with the following structure. The first row contains the matrix sizes N and M, separated by a space. The following rows contain matrices A and B.

```
4 5
1.1 2.1 3.2 4.5 5.6
3.5 6.4 7.8 8.8 0.23
4.5 6.7 8.1 9.4 4.4
7.1 4.2 3.9 6.5 3.7
11.1 12.1 -3.12 -4.25 8.6
3.5 6.4 7.8 8.8 0.23
4.5 6.7 8.1 9.4 4.4
7.1 4.2 3.9 6.5 3.7
```

Figure 7.14: abc.txt file

Figure 7.14 shows an example of file abc.txt, containing matrices A(4,5) and B(4,5). Create it using the console application developed for this purpose to solve Example 7.7.

The code for a console application for solving Problem 7.8 is given below, with comments.

```
program project1;
```

```
{$mode objfpc} {$H+}
uses
  Classes, SysUtils
  {you can add units after this};
var
  f: text;
  i, j, n, m: word;
  A, B, C: array [1..1000, 1..1000] of real;
begin
  // Link the file variable f to a file on disk.
  AssignFile(f, 'abc.txt');
  // Open the file in read mode.
  reset(f);
  // Read from n and m from the first row of abc.txt.
  read(f, n, m);
  // Read the elements of matrix A from the file row by row.
  for i := 1 to n do
    for j := 1 to m do
      read(f, A[i,j]);
  // Read the elements of matrix B from the file row by row.
  for i := 1 to n do
    for j := 1 to m do
      read(f, B[i,j]);
  // Create matrix C = A + B.
  for i := 1 to n do
    for j := 1 to m do
      C[i,j] := A[i,j] + B[i][j];
  // Close file f.
  CloseFile(f);
  // Open the file in write mode.
  Append(f);
  // Add matrix C to file.
  for i := 1 to n do
  begin
    for j := 1 to m do
      // Add next element and a space.
      write(f, C[i,j]:1:2, ' ');
      // After writing matrix row, move to next row in text file.
      writeln(f);
  end;
  // Close the file.
  CloseFile(f);
end.
```

After running the program, the abc.txt file will look something like that shown in Figure 7.15.

```
4 5
1.1 2.1 3.2 4.5 5.6
3.5 6.4 7.8 8.8 0.23
4.5 6.7 8.1 9.4 4.4
7.1 4.2 3.9 6.5 3.7
11.1 12.1 -3.12 -4.25 8.6
3.5 6.4 7.8 8.8 0.23
4.5 6.7 8.1 9.4 4.4
7.1 4.2 3.9 6.5 3.7

12.20 14.20 0.08 0.25 14.20
7.00 12.80 15.60 17.60 0.46
9.00 13.40 16.20 18.80 8.80
14.20 8.40 7.80 13.00 7.40
```

Figure 7.15: abc.txt file after adding matrix C

## 7.5 Exercises

Write two programs for all the problems. The first should create a typed file and the second should read data from this file, carry out required calculations and write results to a text file.

- 1) Create a typed file with n integers. Create arrays of even and odd numbers from this source file. Find the largest negative and smallest positive elements in the file.
- 2) Create a typed file with n integers. From this source file, create an array of tripled even file elements. Sort it in descending order.
- 3) Create a typed file with n integers. Create an array of positive numbers divisible by seven without remainder, using the elements of the source file. Sort the array in ascending order.
- 4) Create a typed file with n real numbers. Create arrays of numbers greater than ten and less than two from the source file. Calculate the number of zero elements in the file.
- 5) Create a typed file with n integers. Create an array of prime numbers from the elements of the source file that are located after the maximum element.
- 6) Create a typed file with n integers. Create an array of even numbers from the elements located before the minimum element in the file.
- 7) Create a typed file with n real numbers. Create an array of elements that are greater than the average of the positive elements in the source file.
- 8) Create a typed file with n integers. Create an array containing the numbers

- located in the file before the maximum element and after the minimum.
- 9) Create a typed file with  $n$  integers. Create an array of prime and perfect numbers located between the minimum and maximum elements in the file.
  - 10) Create a typed file with  $n$  integers. From this file, create an array containing the even numbers first, and then the odd numbers. Find the indices of the largest odd and smallest even components.
  - 11) Create a typed file with  $n$  integers. Swap the smallest positive element and the third prime element in the file.
  - 12) Create a typed file with  $n$  integers. Copy all prime numbers located after the maximum element in this file to new file.
  - 13) Create a typed file with  $n$  integers. Find the arithmetic mean of the positive numbers located before the second prime number.
  - 14) Create a typed file with  $n$  integers. Swap the last perfect and third negative numbers in the file.
  - 15) Create a typed file with  $n$  integers. Copy all the perfect and prime numbers from the file to an array and sort the array in ascending order.
  - 16) Create a typed file with  $n$  integers. Copy the last group of consecutive positive numbers from the source file to text file.
  - 17) Create a typed file with  $n$  integers. Find in it the longest group of consecutive prime numbers.
  - 18) Create a typed file with  $n$  integers. Create arrays of prime and negative numbers from the file. Find the smallest prime and largest perfect numbers in the file.
  - 19) Create a typed file with  $n$  integers. From this file, create an array of elements that are not prime numbers and are located before the maximum element in the file.
  - 20) Create a typed file with  $n$  integers. Create an array of elements from the file that are multiples of 5 and 7, located after the maximum file element.
  - 21) Create a typed file with  $n$  real numbers. From this file, create an array of elements that exceed the average of the positive values in the file.
  - 22) Create a typed file with  $n$  real numbers. Swap the last negative number in the file and the fourth number.
  - 23) Create a typed file with  $n$  real numbers. Find the sum of the third group of consecutive negative elements.
  - 24) Create a typed file with  $n$  integers. Delete from it the fourth group of consecutive prime numbers.
  - 25) Create a typed file with  $n$  integers. Find the difference between the sum of

the prime numbers and the largest negative number in the file.

**Endnotes :**

---

- 1 Typed files are sometimes called components.
- 2 The block size is determined when executing the reset or rewrite procedures.

This page deliberately left blank.

# Chapter 8. Strings and Records

In this chapter, we will get acquainted with the two data types: strings and records. The operation of basic string processing functions will be shown through examples, after which the reader can start exploring records. A *record* is a complex data structure consisting of a fixed number of objects, called record *fields*. Unlike arrays, record fields can be different data types.

## 8.1 Working with Text

Text often must be processed in programming tasks. Processing text in Free Pascal includes character processing and string processing. The concept of strings was already introduced in Chapter 2, along with the main functions for working with strings. Let us recall the fundamental points.

A *character* is a letter, number or sign. The character code table has 256 positions, i.e. each character has its own unique code from 0 to 255. Since the largest character code is 255, then obviously one character will take 1 byte of computer memory. The **char** data type (1 byte) exists for working with characters.

A *string* is an array of characters. A text string has a defined length. The length of a string is the number of characters it contains. Thus, *if one character takes 1 byte*, then a string of N characters would occupy N bytes of memory. The **string** data type exists for working with strings.

Note that in Lazarus strings are encoded in UTF8, in which characters can consist of up to four bytes, as in the German 'ü'.

Use single quotes to assign a value to a character or string variable in a program, such as, for example:

```
s := 'Q';
s1 := 'Hello';
```

The length of the text saved in a string variable can be limited. To do this, the string variable should be declared as follows (the maximum string length is shown in brackets):

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

Let us look at basic string operations.

One of the main operations with strings is concatenation. For this they can be added like numbers. For example:

```
var s: string;
begin
  s := 'text-' + '-' + '*1';
```

This will result in the string: text--\*1.

A string is a character array. So, any character can be obtained from a string by specifying its index in square brackets after the string name. *Note that the index of a string is 1-based*, i.e. the index of the first character is 1. For example:

```
var s: string; c: char;
begin
  s := 'Howdy';
  c := s[4];
```

The result will be the character d.

Now let us look at the main functions and procedures in Table 2.7 for processing strings, using examples. Below is the listing of a program with comments. Results are shown in Figure 8.1.

```
program project1;
var
  str1, str2, str3, str4: string;
  word: string;
  k, L: integer;
begin
  str1 := 'Sergei';
  str2 := 'Ivanov';
  writeln('String str1: ', str1);
  writeln('String str2: ', str2);
  // Concatenate 1st and 2nd strings
  str3 := str1 + ' ' + str2;
  // Find the length of the string
  L := Length(str3);
  writeln('String str1 + str2: ', str3);
  writeln('String length: ', L);
  str4 := 'v';
  // Find an occurrence of the letter v in str3
  k := pos(str4, str3);
  writeln('Letter v first occurs at position: ', k);
  { Copy 6 characters from str3 to str4,
    starting from the eighth character.}
  str4 := copy(str3, 8, 6);
  writeln('Str4: ', str4);
  { Search for the first word. Copy characters to word
    from str3, starting from first character to a space.}
  word := copy(str3, 1, pos(' ', str3) - 1);
```

```
writeln('First word: ', word);
// Remove 2 characters from str3, starting from 12th position
delete(str3,12, 2);
writeln('Str3 after removing characters: ', str3);
readln;
end.
```

```
String str1: Sergei
String str2: Ivanov
String str1 + str2: Sergei Ivanov
String length: 13
Letter v first occurs at position: 9
Str4: Ivanov
First word: Sergei
Str3 after removing characters: Sergei Ivan
```

Figure 8.1: Results from the string program

In the following example, we will read an array of numbers from a text box on a form and find their sum. To do this, create a new project, and on the form, place a text box (Edit1), a ListBox component (ListBox1) for displaying the results and a button (Button1), as shown in Figure 8.2.

Assign the following string to the Text property of Edit1: 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0. By doing so, the array will be already entered when the program starts. When entering the array any number of spaces could be placed between the elements. The program will reduce the number of spaces between elements to one.

Below is a listing of the program with comments. Results from the program is shown in Figure 8.3.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  TForm1
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    ListBox1: TListBox;
    procedure Button1Click (Sender: TObject);
```

```
private
  {private declarations}
public
  {public declarations}
end;
```

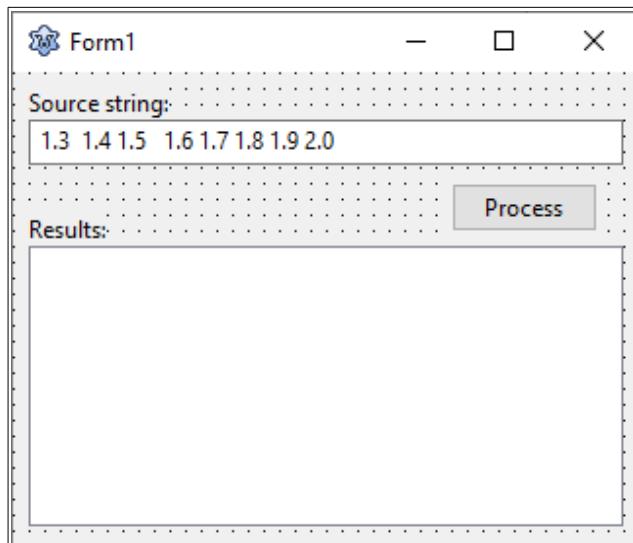


Figure 8.2: Form for reading array

```
var
  Form1: TForm1;
implementation
  {TForm1}

// Button 1 click event handler
procedure TForm1.Button1Click (Sender: TObject);
var
  str1, str2, str3: string;
  i: integer;
  word: string;
  X: array [0..100] of real;
  Sum: real;
begin
  // Read string from Edit1
  str1 := Edit1.Text;

  // Output the original string to listbox
  str2 := 'Source string:';
  ListBox1.Items.Add(str2);
  ListBox1.Items.Add(str1);

  // If the first character is a space, then remove it.
  if str1[1] = ' ' then delete(str1, 1, 1);
```

```
// Loop over string to replace double spaces with single space.  
while pos(' ', str1) > 0 do // 2 spaces!  
  delete(str1, pos(' ', Str1), 1); // 2 spaces!  
  
// Space needed at end of string for correct text processing.  
if str1[Length(str1)] <> ' ' then  
  str1 := str1 + ' ';  
  
// Output string after removing extra spaces.  
str2 := 'String after removing spaces';  
ListBox1.Items.Add (str2);  
ListBox1.Items.Add(str1);  
  
i := 0;  
Sum := 0;  
  
// Loop to find substrings and convert to numbers.  
repeat  
  // Select substring up to a space  
  word := copy(str1, 1, pos(' ', str1) - 1);  
  // Convert substring to number and save it to array.  
  X[i] := StrToFloat(Word);  
  Sum := Sum + X[i];  
  inc(i);  
  // Remove substring from str1  
  delete(str1, 1, Length(word) + 1);  
  // Output substring to ListBox1  
  ListBox1.Items.Add (word);  
until Length(str1) = 0;  
  
// Output sum and number of array elements  
str2 := 'Sum: ' + FloatToStr(Sum);  
str3 := 'Number of elements: ' + IntToStr(i);  
ListBox1.Items.Add(str2);  
ListBox1.Items.Add(str3);  
end;  
  
end.
```

## 8.2 Working with Records

In most cases, simple types are used when writing programs with simple data types (numbers, strings). It often becomes necessary, however, to combine several different data types into a single data type. Free Pascal uses a structured data type known as a *record*, for this. A *record* consists of a fixed number of elements called *record fields*. The general syntax to declare a record looks like:

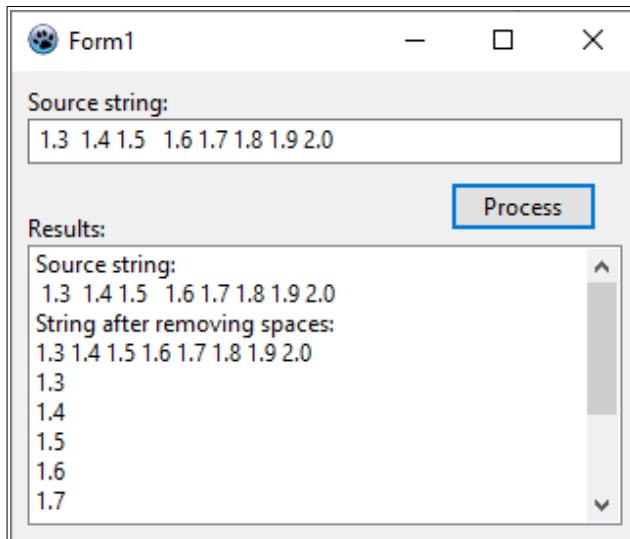


Figure 8.3: Results after processing source string

```
type
  record_name = record
    field1: type;
    field2: type;
    ...
    fieldN: type;
  end;
```

In a program, a record type variable is declared as follows:

```
var variable_name: record_name;
```

Each field of a record can be accessed using its composite name. This consists of the variable name, then a period and then the name of the field.

Consider the following example. Create a triangle record with the three fields, representing the sides of the triangle. The program calculates the area of triangle using Heron's formula.

```
program Project1;
type
  Triangle = record
    a, b, c: real;                                // Sides of the triangle
  end;
var
  x: Triangle;                                    // Declare a Triangle type
  p, s: real;
begin
  write('Side a = ');
  readln(x.a);                                  // Read in field a
  write('Side b = '');
```

```

readln(x.b);                      // Read in field b
write('Side c = ');
readln(x.c);                      // Read in field c
p := (x.a + x.b + x.c)/2; // Compute semi-perimeter
{ Test for the existence of a triangle. Radical expression
  for Heron's formula must be positive.}
if (p-x.a) * (p-x.b) * (p-x.c) > 0 then
begin
  // Compute and display area of triangle
  s := sqrt(p * (p-x.a) * (p-x.b) * (p-x.c));
  writeln('Area of triangle = ', s:7:2);
end
else
  writeln('Not a triangle.');
readln;
end.

```

The elements of a record may be either simple or structured types. There is no limit on the level of nesting of structures. As an example, create a student record, including the following fields: lastname, firstname, group, grades in five courses and address. In turn, the address field is also a record, containing fields: city, street, house number and apartment.

```

type
  residence = record
    city, street: string;
    house, apartment: integer;
  end;
  student = record
    lastname, firstname: string;
    class: string;
    grade: array [1..5] of integer;
    address: residence;
  end;

```

After declaring such a record, access the fields as follows:

```

var
  Johnson: student;
  X: array [1..100] of student; // Student list
begin
  Johnson.class := 'Math010';
  Johnson.address.city := 'York';
  X[1].grade[5] := 3;          // First student on list

```

Using the **with** keyword as shown below, the fields of a record can be accessed without specifying the field name as a prefix:

**with variable do statement**

For example:

```
with Johnson do
begin
  with address do
  begin
    city := 'York';
    street := 'Main';
    house := 145;
    apartment := 31;
  end;
  lastname := 'Johnson';
  firstname := 'Andrew';
  class := 'Math010';
  grade[1] := 3; grade[2] := 5; grade[3] := 4;
  grade[4] := 3; grade[5] := 5;
end;
```

**EXAMPLE 8.1.** Create a database containing student information. The program calculates the student's average score, sorts alphabetically and outputs results to screen and a text file.

Create a new project. On the form (Figure 8.4) place the required number of TEdit objects for inputting data and a StringGrid object to display the results. For the StringGrid1 object, set the properties ColCount = 8 and RowCount = 1. Let us also add three buttons:

- *Save data.* After reading and saving data in the text boxes, the text boxes are cleared for new input;
- *Sort Data.* After clicking this button, the records are sorted alphabetically by surname;
- *Display Data.* The results are displayed in the grid and saved in a text file.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls, Grids;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
```

```
Button2: TButton;
```

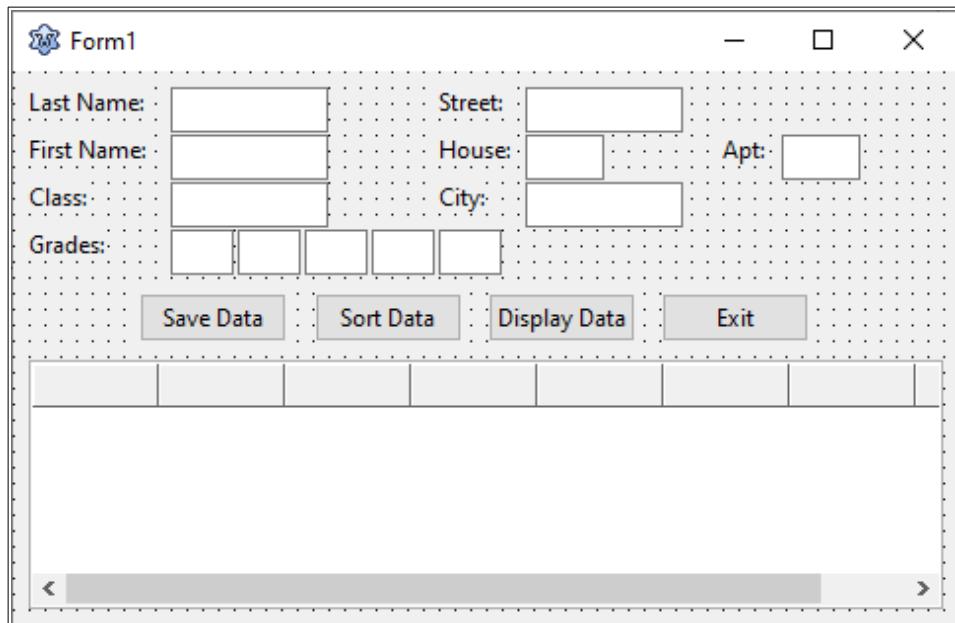


Figure 8.4: Form for Example 8.1

```
Button3: TButton;
Edit1: TEdit;
Edit10: TEdit;
Edit11: TEdit;
Edit12: TEdit;
Edit2: TEdit;
Edit3: TEdit;
Edit4: TEdit;
Edit5: TEdit;
Edit6: TEdit;
Edit7: TEdit;
Edit8: TEdit;
Edit9: TEdit;
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Label6: TLabel;
Label7: TLabel;
Label8: TLabel;
Label9: TLabel;
StringGrid1: TStringGrid;
// Event handler for clicking the Save Data button
procedure Button1Click (Sender: TObject);
// Event handler for clicking the Display Data button
procedure Button2Click (Sender: TObject);
```

```
// Event handler for clicking Sort Data button
procedure Button3Click (Sender: TObject);
// Form initialization procedure
procedure FormCreate (Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
// Declare the student residence record
residence = record
  city, street: string;
  house: integer;
  apt: string;
end;
// Declare the student record
student = record
  firstname, lastname: string;
  group: string;
  grades: array [1..5] of integer;
  address: residence;
  GPA: real;           // grade point average
end;
var
  Form1: TForm1;
  // Array of student variable
  X: array [0..100] of student;
  // Student counter
  i: integer;
implementation
{TForm1}

// Form initialization
procedure TForm1.FormCreate (Sender: TObject);
begin
  i := 0;           // Set number of students = 0
end;

// Event handler for clicking the Save Data button
procedure TForm1.Button1Click (Sender: TObject);
var
  sum, j: integer;
begin
  // Read data from the form
  X[i].lastname := Edit1.Text;
  X[i].firstname := Edit2.Text;
  X[i].group := Edit3.Text;
  X[i].address.city := Edit5.Text;
  X[i].address.street := Edit6.Text;
  X[i].address.house := StrToInt(Edit7.Text);
  X[i].address.apt := Edit8.Text;
```

```
X[i].grades[1] := strToInt(Edit4.Text);
X[i].grades[2] := strToInt(Edit9.Text);
X[i].grades[3] := strToInt(Edit10.Text);
X[i].grades[4] := strToInt(Edit11.Text);
X[i].grades[5] := strToInt(Edit12.Text);
// Compute the student's average score
sum := 0;
for j := 1 to 5 do
  sum := sum + X[i].grades[j];
X[i].GPA := sum / 5;
inc(i);
// Clear input fields for new data
Edit1.Text := ''; Edit2.Text := '';
Edit3.Text := ''; Edit4.Text := '';
Edit5.Text := ''; Edit6.Text := '';
Edit7.Text := ''; Edit8.Text := '';
Edit9.Text := ''; Edit10.Text := '';
Edit11.Text := ''; Edit12.Text := '';
end;

// Event handler for clicking the Display Data button
procedure TForm1.Button2Click (Sender: TObject);
var
  f: textfile;
  j: integer;
  s: string;
begin
  { Number of StringGrid rows will be 1 more than
    number of students (plus 1 row for the header) }
  StringGrid1.RowCount := i + 1;
  // Display the table header
  StringGrid1.Cells[1,0] := 'Last Name';
  StringGrid1.Cells[2,0] := 'Name';
  StringGrid1.Cells[3,0] := 'Group';
  StringGrid1.Cells[4,0] := 'City';
  StringGrid1.Cells[5,0] := 'Street';
  StringGrid1.Cells[6,0] := 'House / Apt';
  StringGrid1.Cells[7,0] := 'GPA';
  // Output student info in the j-th line
  for j := 1 to i do
  begin
    StringGrid1.Cells[1,j] := X[j-1].lastname;
    StringGrid1.Cells[2,j] := X[j-1].firstname;
    StringGrid1.Cells [3,j] := X[j-1].group;
    StringGrid1.Cells [4,j] := X[j-1].address.city;
    StringGrid1.Cells [5,j] := X[j-1].address.street;
    s := IntToStr(X[j-1].address.house) + '/' +
      X[j-1].address.apt;
    StringGrid1.Cells[6, j] := s;
    StringGrid1.Cells [7,j] := FloatToStr(X[j-1].GPA);
  end;
```

```
// Output the results to a text file
assignfile(f, 'student.txt');
rewrite(f);
for j := 1 to i do
begin
  writeln(f, X[j-1].lastname:20, X[j-1].firstname:15,
    X[j-1].address.city:15, ', ', X[j-1].address.street:15,
    X[j-1].address.house:4, '/', X[j-1].address. apt, 'GPA = ',
    X[j-1].GPA:4:1);
end;
closefile(f);
end;

// Event handler for clicking Sort Data button
procedure TForm1.Button3Click (Sender: TObject);
var
  j, k: integer;
  temp: student;
begin
  for j := 0 to i - 1 do
    for k := j + 1 to i - 1 do
      if X[j].lastname > X[k].lastname then
        begin
          temp := X[j];
          X[j] := X[k];
          X[k] := temp;
        end;
end;
end.
```

When you start the program and enter information, the form window looks like Figure 8.5.

When the entry is entered, click on the button Save data, when This clears the input fields for the next entry. After clicking the button Display data, the table is filled with the entered information (Figure 8.6).

Form1

Last Name:	Johnson	Street:	Main St																								
First Name:	Andrei	House:	100																								
Class:	Math010	City:	York																								
Grades:	3	3	3																								
	4	4																									
<input type="button" value="Save Data"/> <input type="button" value="Sort Data"/> <input type="button" value="Display Data"/> <input type="button" value="Exit"/>																											
<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td colspan="8"></td></tr><tr><td>&lt;</td><td colspan="6"></td><td>&gt;</td></tr></table>																				<							>
<							>																				

Figure 8.5: Form for entering student information

Form1

Last Name:	Johnson	Street:	Main St																																			
First Name:	Andrei	House:	100																																			
Class:	Math010	City:	York																																			
Grades:	3	3	3																																			
	4	4																																				
<input type="button" value="Save Data"/> <input type="button" value="Sort Data"/> <input type="button" value="Display Data"/> <input type="button" value="Exit"/>																																						
<table border="1"><thead><tr><th></th><th>Last Name</th><th>Name</th><th>Group</th><th>City</th><th>Street</th><th>House / Apt</th></tr></thead><tbody><tr><td></td><td>Johnson</td><td>Andrei</td><td>Math010</td><td>York</td><td>Main St</td><td>100/5D</td></tr><tr><td></td><td>Wilkins</td><td>Thomas</td><td>Phy100</td><td>Yants</td><td>Main St</td><td>80/</td></tr><tr><td></td><td>Christian</td><td>Willaby</td><td>Math010</td><td>York</td><td>Bent St</td><td>1200/40</td></tr><tr><td></td><td>Ali</td><td>Aftab</td><td>Phy100</td><td>Yants</td><td>Drury Ln</td><td>234/7C</td></tr></tbody></table>					Last Name	Name	Group	City	Street	House / Apt		Johnson	Andrei	Math010	York	Main St	100/5D		Wilkins	Thomas	Phy100	Yants	Main St	80/		Christian	Willaby	Math010	York	Bent St	1200/40		Ali	Aftab	Phy100	Yants	Drury Ln	234/7C
	Last Name	Name	Group	City	Street	House / Apt																																
	Johnson	Andrei	Math010	York	Main St	100/5D																																
	Wilkins	Thomas	Phy100	Yants	Main St	80/																																
	Christian	Willaby	Math010	York	Bent St	1200/40																																
	Ali	Aftab	Phy100	Yants	Drury Ln	234/7C																																

Figure 8.6: Form for displaying student information

	Last Name	Name	Group	City	Street	House / Apt
	Ali	Aftab	Phy100	Yants	Drury Ln	234/7C
	Christian	Willaby	Math010	York	Bent St	1200/40
	Johnson	Andrei	Math010	York	Main St	100/5D
	Wilkins	Thomas	Phy100	Yants	Main St	80/

Figure 8.7: Form after sorting alphabetically by last name

After clicking the Sort Data button, you need to click again on the Display Data button to see the sorted list (Figure 8.7).

## 8.3 Exercises (Strings)

A string consists of words separated from each other by a space. Perform the actions specified below on the string.

- 1) Find the number of commas in the string.
- 2) Replace all digits in the string with a space. Print the number of replacements.
- 3) Find the number of digits in the string.
- 4) Remove all commas from the string.
- 5) Find the number of words in a line.
- 6) Remove from the string all words starting with an “o”.
- 7) After each space, insert a “\*”.
- 8) Find the longest word in the string.
- 9) Insert a space and a “+” before each space.
- 10) Calculate the sum of all numbers in the string.
- 11) Count the words in the string starting with “Ab”.
- 12) Replace double spaces in the string with single spaces. Print the number of

replacements.

- 13) Insert a comma after each word.
- 14) Insert a ";" character after each word.
- 15) Find the number of ":" and ";" characters in the string.
- 16) Remove all digits from the string.
- 17) Find the number of words ending in "on" in the string.
- 18) Find the shortest word in the string.
- 19) Insert the word "eve" after each word ending with an "e".
- 20) Remove all five-letter words from the string.
- 21) Find the number of words starting with an "a" and ending with a "t" in the string.
- 22) Delete the second, third and fifth words from the string.
- 23) Insert the symbol "No." in front of each digit.
- 24) Count the number of vowels in the string.
- 25) Remove words starting and ending with an "e" from the string.

## 8.4 Exercises (Records)

All tables must contain at least ten records.

- 1) Create a data structure per Table 8.1.

*Table 8.1: Population growth in cities*

<b>City</b>	<b>Population growth, thousand people</b>				
	1999	2000	2001	2002	2003
Izhevsk	2.5	1.3	-0.2	-0.1	0.6
...					

Add and calculate an "Average Growth" field in the structure. Find number of cities with negative growth in 2003. Arrange records in ascending order by average growth.

- 2) Create a data structure per Table 8.1. Add and calculate a "Minimum Growth" field in the structure. Determine the number of cities with a growth of more than 2 thousand in 2003. Sort records alphabetically by city name.
- 3) Create a data structure per Table 8.2:

*Table 8.2: Product details*

<b>Name</b>	<b>Factory</b>	<b>Price</b>	<b>Produced</b>	<b>Quantity</b>
Locomotive	Toy	125.00	02/01/2007	
...				

Add and calculate a “Discount Price” field in the structure by assuming a percentage discount from prices shown in the form. Find the total number of toys from the Toy factory. Sort records in descending order by Price.

- 4) Create a data structure per Table 8.2. Add and calculate a “Sales” field in the structure. Find the number of names of toys with a price that is less than the average price of all the toys. Sort records by toy name.
- 5) Create a data structure per Table 8.3:

*Table 8.3: Information about schoolchildren*

<b>Surname</b>	<b>Name</b>	<b>Date of Birth</b>	<b>School</b>	<b>Class</b>
Johnson	Douglas	05/05/1994	112	9-A
...				

Add and calculate an Age field in the structure, based on the current date. Determine the number of students with the name Douglas. Sort records by school number.

- 6) Create a data structure per Table 8.3. Add and calculate a “Grade” in the structure, by removing the letter from the class name. Find the number of ninth grade students. Sort records alphabetically by surnames.
- 7) Create a data structure per Table 8.4:

*Table 8.4: Sales Details*

<b>Printer</b>	<b>Quantity, pcs</b>			<b>Price</b>
	Jan	Feb	May	
Samsung CLP-310	25	20	26	1200
...				

Add and calculate a “Revenue” field in the structure. Find the average printer price. Sort records in ascending order by Price.

- 8) Create a data structure per Table 8.4. Add and calculate an “Average Quantity” field in the structure. Find the average number of printers per month. Sort records by printer name.
- 9) Create a data structure per Table 8.4. Add and calculate a “Total” field in the

structure. Find the total number of printers sold per month. Sort records in ascending order by Total.

- 10) Create a data structure per Table 8.4. Add and calculate a "Discount Price" field in the structure, by assuming the discount percentage from prices shown in the form. Find the number of printers with a price greater than 1500 dollars. Sort records in ascending order by price.
- 11) Create a data structure per Table 8.5:

*Table 8.5: Employee information*

Full name	Date of Birth	Position	Years	Salary
Johnson D.I.	03/12/1966	Manager	2	1250
...				

Add and calculate a "Bonus" field in the structure, calculated as follows: 20% of salary for employees with 10 or more years of service, or 10% otherwise. Find the number of employees with over 10 years of experience. Sort records by position.

- 12) Create a data structure per Table 8.5. Add and calculate an "Age" field in the structure, using the current date. Find medium the salary of all employees. Sort records by name.
- 13) Create a data structure per Table 8.5. Add and calculate an "Age" field in the structure, using the current date. Define the number of young professionals (under 25). Sort records in ascending order by salary.
- 14) Create a data structure per Table 8.6:

*Table 8.6: Tour Sales Information*

Destination	Quantity, trips			Price
	Jul	Aug	Sep	
Mazatlan	255	203	198	1400
...				

Add and calculate an "Average" field in the structure. Find the total number of trips per month. Sort records by destination.

- 15) Create a data structure per Table 8.6. Add and calculate an "Income" field in the structure. Find the average tour price. Sort records in ascending order by price.
- 16) Create a data structure per Table 8.7. Add and calculate a "Salary" field in the structure, calculated as follows: add 15% of the bonus from the salary to the

salary. Sort records by name.

- 17) Create a data structure per Table 8.7. Add and calculate an "Age" field in the structure, using the current date. Find the number of men and women. Sort records by title.

*Table 8.7: Employee details*

Full name	Date of Birth	Position	Sex	Salary
Johnson D.I.	03/12/1966	Manager	Male	4000
...				

- 18) Create a data structure per Table 8.8:

*Table 8.8: Academic employee details*

Surname	Initials	Academic Class	Birth Year	Articles
Johnson	D.I.	Assistant professor	1971	7
...				

Add and process an "Activity" field in the structure, based on the following. If the number of articles is more than 5, then place a space in the field. Otherwise, place "Work harder" in the field. Sort records by surname.

- 19) Create a data structure per Table 8.8. Remove an employee by surname from the form. Find the number of associate professors. Sort records by title.
- 20) Create a data structure per Table 8.9:

*Table 8.9: Book publication information*

Title	Author	Typography	Year published	Price	Quantity
Light appearance	I. D. Johnson	KP	2003	20	15000
...					

Add and calculate a "Publishing Cost" field in the structure. Find the quantity of books published in 2005. Sort posts by author.

- 21) Create a data structure per Table 8.9. Delete all books published in 2000. Find the average price of books printed with KP typography. Sort records by year of publication.
- 22) Create a data structure per Table 8.10. Add and calculate a "Cost of Call" field

in the structure. Find the average cost of calls to a city, on request. Sort entries by subscriber.

*Table 8.10: Information about phone calls*

<b>Subscriber</b>	<b>Number called</b>	<b>Date</b>	<b>City</b>	<b>Cost / min</b>	<b>Minutes</b>
Mole R.Y.	956-250-781	05/12/2003	New York	3.65	2
...					

- 23) Create a data structure per Table 8.10. Delete all records of calls to numbers starting with "3". Sort entries by city.  
 24) Create a data structure per Table 8.11:

*Table 8.11: Instrument Information*

<b>Instrument Type</b>	<b>Model</b>	<b>Release Date</b>	<b>Quantity</b>	<b>Warranty, months</b>
Microscope	M12-08	06/12/2006	200	24
...				

Add and calculate a "Warranty Service" field in the structure, as follows. Place "1 year" in the field if the warranty period is more than 3 years, or "No service" otherwise. Find the average number of instruments. Sort records by instrument type.

- 25) Create a data structure per Table 8.11. Delete all records with a warranty of less than 6 months. Sort records by release date.

This page deliberately left blank.

# Chapter 9. Object Oriented Programming

This chapter is dedicated to learning object-oriented programming (OOP). OOP is a program development technique based on the concept of an object, as a specific structure that emulates an object in the real world, its behavior and interaction with other objects.

## 9.1 Basic Concepts

The basis of object-oriented programming is an object. An object consists of three main parts:

- 1) *Name* (e.g. car);
- 2) *State, or state variables* (for example, car make, color, weight, number of seats, etc.);
- 3) *Methods*, or functions that perform specific actions on objects and determine how the object interacts with the surrounding environment.

To work with objects, the concept of a class was added to Free Pascal. A *class* is a complex structure, that includes the definition of data, procedures and functions that can be performed by the object.

A *class* is a composite data type, with members (elements) that are functions and variables (fields). The concept of a class is based on the fact that "*various operations can be performed by objects*". Object properties are defined using class variables (fields) and actions by objects are defined using subroutines called class methods. Objects are called *class instances*.

Object Oriented Programming (OOP) is technology for developing programs using objects. In object-oriented languages there are three basic concepts: encapsulation, inheritance and polymorphism. *Encapsulation* is a union in a class of data and the subroutines to process them. *Inheritance* is when any class can be generated from another class. The generated class (child) automatically inherits all fields, methods, properties and events from the parent class. *Polymorphism* allows the use of the same names for methods in different classes.

To declare a class, use the following construction:

```
type
  <class name> = class (<parent class name>)
    <class fields and methods>
private
  <fields and methods accessible only within the unit>
```

```
protected
  <fields and methods accessible only in descendant classes>
public
  <fields and methods accessible from other units>
published
  <fields and methods visible in the object inspector>
end;
```

Any valid Free Pascal identifier can be used as a class name. The *parent class name* is the name of the class from which the given class descends. This is an optional parameter. If it is not specified, then the given class will descend from the predefined **TObject** class.

Records differ from classes in that the fields of a record are always accessible. When using classes, there may be members accessible everywhere (*public* members), and *private* members, which can be accessed only through public class methods. This also applies to class methods.

Fields, properties and methods in the *public* section have no restrictions on their visibility. They are accessible from other functions and methods of objects both in this unit and in all others referring to it. When accessing public fields outside a class, the *.* (dot) operator is used.

Fields, properties and methods located in the *private* section are accessible only in the class methods and in functions contained in the same unit that contains the class definition. This allows you to completely hide the details of the internal implementation of the class. Private methods should be called from public ones.

The *published* section contains properties that the user can set up at the design stage and which can be accessed for editing in the object inspector.

The *protected* section contains fields and methods that are accessible inside the class and in any of its descendant classes, including those in other units.

Fields and methods declared before the section names, are public.

When programming using classes, the programmer must decide which members and methods should be declared public, and which private. The general principle is: The lesser the public data in the program. the better. Reducing the number of public members and methods will minimize the number of errors.

Fields can be of any type, including classes. Declaring fields is carried out in the same way as declaring ordinary variables:

```
field1: datatype;
field2: datatype;
```

Methods in a class are declared just like regular subroutines:

```
function method1(parameter list): result type;
procedure method2(parameter list);
```

Procedures and functions implementing methods are declared after the word **implementation** in the unit where the class is declared:

```
function class_name.method1(parameter list): result type;
begin
  function body;
end;
procedure class_name.method2(parameter list);
begin
  procedure body;
end;
```

Declaring a class type is called *creation (initialization)* of an object (class instance). A class instance is declared in the same block used for declaring variables:

```
var variable_name: class_name;
```

After declaring the variable, its field and methods can be called the same way as calling record fields, using the “.” operator.

For example:

```
variable_name.field1 := expression;
variable_name.method1 (parameter list);
...
```

The **with** expression may also be used:

```
with variable_name do
begin
  field1 := expression;
  method1(parameter list);
...
end;
```

Free Pascal has many classes for declaring an application form and its components (buttons, text boxes, check boxes, etc). In the process of constructing the form, software objects automatically to the code. For example, when adding a component to a form, the component class declaration is created, and when creating an event handler, a method declaration is added to the class declaration.

Let us consider this on an example of a project with a form containing a button (Button1).

```
unit Unit1;
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
{ TForm1 }
  // Declaration of the form class TForm1
  TForm1 = class(TForm)
    // Declaration of the button component Button1
    Button1: TButton;
    // Declaration of the event handler for clicking on Button1
    procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
var
  // Declaration of the class variable for the form (TForm1)
  Form1: TForm1;
implementation
{ TForm1 }
  // Event handler for clicking on Button1
  procedure TForm1.Button1Click(Sender: TObject);
begin
  // The text of the event handling procedure
end;

end.
```

In Free Pascal, a class (object) is a dynamic structure. Unlike a static object, it contains not the data, but links to them. Therefore, the programmer must take care of memory allocation for the data.

A *constructor* is a special method that creates and initializes an object. The constructor declaration looks like this:

**constructor** Create;

Declare a constructor just like any other method, after the implementation keyword in the unit in which the class is declared.

```
constructor class_name.Create;
begin
  field1 := expression1;
  field2 := expression2;
  ...
inherited Create;
end;
```

Because of the constructor, all the class fields are initialized. Ordinal types are set to 0 as initial values, and strings are set to an empty value.

The *destructor* is a special method that destroys an object and releases the memory it occupies. The destructor is declared as:

**destructor** Destroy;

If an object is no longer needed, then the expression

classname.Free;

using the free method, calls the destructor and frees the memory occupied by fields of the classname object.

Consider an example of “complex number”<sup>1</sup> class. Call this class “TComplex”. Its members are: x, the real part of the complex number and y, the imaginary part. Class methods will include:

- a Create constructor, which will set the real and imaginary parts to 0;
- the modulus() function for computing the modulus;
- the argument() function for computing the argument;
- the ComplexToStr() function to represent a complex number as a string, for output.

Create a new project. On a form, place a button, two text boxes to input the real and imaginary parts, and a memo to show the results. Clicking on the button will generate an instance of the Complex number class, then its modulus and argument will be computed. Display the results in Memo1: the number in an algebraic form, its modulus and argument. Below is the commented code for the unit, which demonstrates working with this class.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  TForm1
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
```

```
Memo1: TMemo;
procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
type
  // Complex number class declaration
  TComplex = class
private
  x: real;      // real part
  y: real;      // imaginary part
public
  constructor Create;
  function modulus(): real;    // Compute the unit
  function argument(): real;   // Compute the argument
  // Write a complex number as a string
  function ComplexToStr(): String;
end;
var
Form1: TForm1;
// Declare a variable of complex number class type
number: TComplex;
implementation
// Constructor
constructor TComplex.Create;
begin
  x := 0; y := 0;
  // Call the parent constructor
  inherited Create;
end;

// Compute modulus.
function TComplex.modulus(): real;
begin
  modulus := sqrt(x*x + y*y);
end;

// Compute argument.
function TComplex.argument(): real;
begin
  argument := arctan(y/x) * 180/pi;
end;

// Convert complex number to string.
function TComplex.ComplexToStr(): string;
begin
  if y >= 0 then
    ComplexToStr := FloatToStrF(x, ffFixed, 5, 2) + '+' +
      FloatToStrF(y, ffFixed, 5, 2) + 'i'
```

```

else
  ComplexToStr := FloatToStrF(x, ffFixed, 5,2) +
    FloatToStrF(y, ffFixed, 5,2) + 'i'
end;

{ Button handler. Instantiate the "Complex number class",
compute its modulus and argument, output the number
in algebraic form, its modulus and argument. }
procedure TForm1.Button1Click(Sender: TObject);
var
  Str1: string;
begin
  // Create an instance of the complex number class
  number := TComplex.Create;
  // Read in the real and imaginary parts
  number.x := StrToFloat(Edit1.Text);
  number.y := StrToFloat(Edit2.Text);
  Str1 := 'Complex number: ' + number.ComplexToStr();
  // Output complex number to Memo1 field line-by-line,
  Memo1.Lines.Add(Str1);
  // its modulus
  Str1 := 'Modulus: ' +
    FloatToStrF(number.Modulus(), ffFixed, 5, 2);
  Memo1.Lines.Add (Str1);
  // and argument
  Str1 := 'Argument: ' +
    FloatToStrF(number.argument(), ffFixed, 5, 2);
  Memo1.Lines.Add(Str1);
  // Destroy the object
  number.Free;
end;

end.

```

The result of the program is shown in Figure 9.1.

In this example, a constructor was written for the complex number class with no parameters. In Free Pascal, you can write a constructor with parameters that takes input values and initializes the class fields with those values. Rewrite the previous example as follows. Read the real and imaginary parts from the form and pass them to the constructor to initialize a complex number object. The program listing is shown below.

```

unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;

```

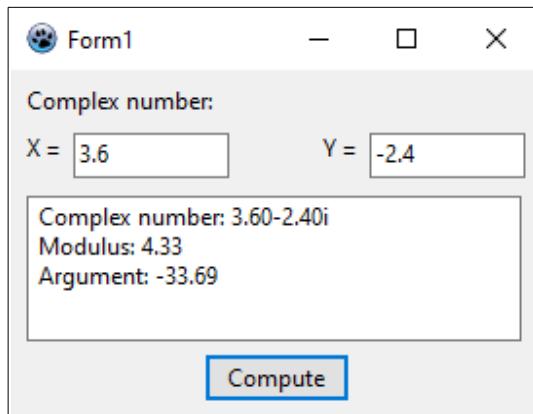


Figure 9.1: The complex number class program

```
type
{ TForm1}
TForm1 = class(TForm)
  Button1: TButton;
  Edit1: TEdit;
  Edit2: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Memo1: TMemo;
  procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
type
  TComplex = class
private
  x, y: real;
public
  // constructor declaration
  constructor Create (a, b: real);
  function modulus(): real;
  function argument(): real;
  function ComplexToStr(): string;
end;
var
  Form1: TForm1;
implementation
{ Constructor takes as input two real numbers and writes them
  to the real and imaginary parts of the complex number. }
constructor TComplex.Create (a, b: real);
```

```
begin
  x := a; y := b;
  inherited Create;
end;
function TComplex.modulus(): real;
begin
  modulus := sqrt(x*x + y*y);
end;
function TComplex.argument(): real;
begin
  argument := arctan(y/x) * 180/pi;
end;
function TComplex.ComplexToStr(): string;
begin
  if y >= 0 then
    ComplexToStr := FloatToStrF(x, fffFixed, 5, 2) + '+'
      + FloatToStrF(y, fffFixed, 5, 2) + 'i'
  else
    ComplexToStr := FloatToStrF(x, fffFixed, 5, 2) +
      FloatToStrF(y, fffFixed, 5, 2) + 'i';
end;

procedure TForm1.Button1Click (Sender: TObject);
var
  Str1: string;
  x1, x2: real;
  number: TComplex;
begin
  x1 := StrToFloat(Edit1.Text);
  x2 := StrToFloat(Edit2.Text);
  number := TComplex.Create(x1, x2);
  Str1 := 'Complex number: ' + number.ComplexToStr();
  Memo1.Lines.Add(Str1);
  Str1 := 'Modulus: ' + FloatToStrF(number.modulus(), fffFixed, 5,
2);
  Memo1.Lines.Add(Str1);
  Str1 := 'Argument: ' + FloatToStrF(number.argument(),
  fffFixed, 5, 2);
  Memo1.Lines.Add(Str1);
  number.free;
end;

end.
```

## 9.2 Encapsulation

Encapsulation one of the most important mechanisms of object-oriented programming (along with inheritance and polymorphism). A class is a unity of three entities: fields, properties and methods, which is encapsulation. Encapsulation

allows you to create a class as something integrated, which has a certain functionality. For example, the TForm class has (encapsulates) everything needed to create a dialog box.

The main idea of encapsulation is to protect fields from unauthorized access. Therefore, it is advisable to declare fields in the private section. Direct access to object fields, reading and updating their content should be done through appropriate methods. Class properties serve such a purpose in Free Pascal.

*Properties* are a special class mechanism that controls access to fields. Properties are declared using reserved words **property**, **read** and **write**. Usually a property is associated with a specific field and specifies those class methods to be used for writing to, or reading from, this field. The syntax for declaring properties is as follows:

```
property property_name: type read read_name write write_name
```

The **read** reserved word defines the method for reading from, and **write** the method for writing to, the properties of an object. **Read\_Name** and **Write\_Name** are the names of methods for reading or writing properties, respectively.

If the property should be read-only or write-only, the **write** or **read** part should be omitted, respectively.

Consider the following example. Create a polygon class with the name TPolygon. The fields of the class will be:

- K, the number of sides of the polygon, and
- p, an array to store the lengths of the sides.

The class methods will be:

- the Create constructor, zeroing the elements of the p array;
- the Perimeter() function for calculating the perimeter;
- the Show()function to display information about the figure (number of sides and perimeter);
- the Set\_Input() function to check the initial data.

Place a button and a label on the form. When the button is clicked, a window for entering the number of sides of the polygon should appear. If the number of sides is entered correctly, the Polygon object is initialized with the number of sides entered. Otherwise the default (50) is used. After this, the perimeter is calculated, and the results are shown using the label.

Below is the program listing with comments. Results from the program can be seen in Figure 9.2.

```
unit Unit1;
```

```
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
private
  { private declarations }
public
  { public declarations }
end;
type
  // TPolygon class declaration
  TPolygon = class
  // Private fields
  private
    K: integer;                      // Number of sides
    p: array of real;                // Lengths of sides
  // Public methods
  public
    constructor Create;
    function Perimeter(): real; // Calculate perimeter
    function Show(): string;        // Show results
  // Protected methods
  protected
    procedure Set_Input(m: integer); // Data validation
  published
  { Declare property n, which operates on field K.
    In the property declaration there is a field name (K)
    after read. This means that there is no specified
    read function and the user can read this field value
    directly. Reference to the Set_Input function after
    write means that this function will have to be used to
    modify K. }
    property n: integer read K write Set_Input;
  end;

  var
    Form1: TForm1;
    // Declare TPolygon variable
    Figure: TPolygon;
implementation
// Constructor.
constructor TPolygon.Create;
var
  i: integer;
```

```
begin
  K := 50;           // Assign initial values to fields.
  SetLength(p, K);    // Allocate memory for array P
  for i := 0 to K - 1 do
    p[i] := 0;
  inherited Create;
end;
// Function for calculating the perimeter.
function TPolygon.Perimeter(): real;
var
  Sum: real;
  i: integer;
begin
  Sum := 0;
  for i := 0 to K - 1 do
    Sum := Sum + p[i];
  perimeter := Sum;
end;
// Display information about polygon.
function TPolygon.Show(): String;
begin
  Show := 'Polygon with number of sides ' + IntToStr(K) + chr(13)
  + 'Perimeter = ' + FloatToStr(Perimeter());
end;
// Method for writing data to field K.
procedure TPolygon.Set_Input (m: integer);
begin
  { If the value entered value > 1 then set K = entered value.
  Otherwise, use the default value. }
  if m > 1 then K := m else K := 50;
end;

{ TForm1 }
// Button click event handler
procedure TForm1.Button1Click(Sender: TObject);
var
  i, m: integer;
  s: string;
begin
  // Enter the number of sides of the polygon
  s := InputBox('Input', 'Enter the number of sides for the
polygon','6');
  val(s, m);
  Figure := TPolygon.Create; // Initialize object
  with Figure do
  begin
    // method for checking the initial data
    Set_Input(m);
    // Generate an array of random numbers
    for i := 0 to K - 1 do
      p[i] := random(50);
```

```
// call to the method for calculating the perimeter  
s := Show();  
end;  
Label1.Caption := s;           // Display results  
end;  
  
end.
```

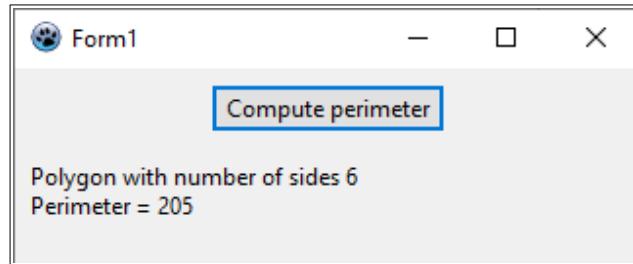


Figure 9.2: Program with polygon class

## 9.3 Inheritance and Polymorphism

The second foundational component of object-oriented programming is inheritance. The concept of inheritance is this: if a new class is needed that differs only slightly from an existing class, then there is no need to rewrite the existing fields and methods. A new class can be declared, which inherits the existing class, and new fields, methods and properties can be added to it. The new class inherits the members of the previously defined base class (parent). The descendant class is derived from the base class, and it too can become the base class for the newly derived classes.

In Object Pascal, all classes are descendants of the `TObject` class. Therefore, if you create a child class directly from the `TObject` class, then `TObject` may be omitted from the definition.

A derived class inherits the fields and methods of its parent class. If method names match, they are said to overlap. Depending on what actions occur when called, methods can be divided into the following groups:

- static methods;
- virtual methods;
- dynamic methods.

All methods are *static* by default. Base class methods are completely hidden in descendant classes if they are redefined. In such a case, the method may be completely rewritten. If the method is called through the base class then the method in the base class will be used. If called through the derived class then the

derived class method will be used.

*Virtual* and *dynamic* methods have the same names and types in the base and derived classes. These methods are overloaded in inherited classes. The method called depends on which class is called.

The main difference between virtual and dynamic methods is in the way they are called. Information about virtual methods is stored in the virtual method table (VMT). The virtual methods of a given class and its ancestors are stored in the VMT. When creating a class descendant, the entire VMT of the ancestor is transferred to the descendant and new methods are added to it. Finding a required method takes little time, since the class has complete information about its virtual methods. Dynamic methods are not duplicated in the dynamic method table (DMT) of the descendant. A class's DMT contains only methods declared in this class. When calling a dynamic method, the class's DMT is searched first, and if the method is not found then in the ancestor's DMT is searched, etc. Thus, using virtual methods requires more memory to store massive VMTs, but they are called faster.

By changing the algorithm of a method in derived classes, the programmer can give descendant classes specific properties that are not present in the parent class. To change a method, its descendant should be overloaded, i.e. a method with the same name should be declared in the derived class and the necessary functionality added to it. As a result, the parent and child classes will have two methods with the same name but with different functionality. This is called object *polymorphism*.

Virtual and dynamic methods are declared the same way as static methods, except that the **virtual** or **dynamic** keyword is added the end of the declaration:

```
type
  parent_class_name = class
  ...
  method; virtual;
  ...
end;
```

To overload a virtual method in a derived class, add the **override** keyword after it in the class declaration:

```
type
  successor_class_name = class (parent_class_name)
  ...
  method; override;
  ...
end;
```

Let us consider inheritance in classes in the following example. Create a base class

TTriangle with fields for the coordinates of the vertices of the triangle. The class will have the following methods:

- Check(), to verify the existence of a triangle (that the 3 points do not lie on a straight line);
- Perimeter(), to calculate the perimeter of the triangle;
- Area(), to calculate the area;
- a(), b() and (c), to calculate the lengths of the sides;
- Set\_Tr(), to obtain the coordinates;
- Show(), to display information about a triangle.

Based on this class, we will create a derived class TEqTriangle (equilateral triangle), which inherits all fields and methods of the base class, but the Check and Show methods will be modified.

On a form, place a label, a button and six TEdit components to input the vertices for the two triangles. After clicking the button, two objects, a triangle and an equilateral triangle, are created the perimeter and area of each triangle calculated and the results are displayed below in Label15 and Label16. The following is a listing of the program, with comments.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  {TForm1}
TForm1 = class(TForm)
  Button1: TButton;           // "Calculate" button
  // Arrays x1, y1 - vertex coordinates
  // The first triangle is a general case
  // The second triangle is equilateral.
  Edit1: TEdit;               // Coordinate x1[1]
  Edit10: TEdit;              // y2[2] coordinate
  Edit11: TEdit;              // Coordinate x2[3]
  Edit12: TEdit;              // y2[3] coordinate
  Edit2: TEdit;               // y1[1] coordinate
  Edit3: TEdit;               // Coordinate x2[2]
  Edit4: TEdit;               // y1[2] coordinate
  Edit5: TEdit;               // Coordinate x1[3]
  Edit6: TEdit;               // y1[3] coordinate
  Edit7: TEdit;               // Coordinate x2[1]
  Edit8: TEdit;               // y2[1] coordinate
  Edit9: TEdit;               // Coordinate x2[2]
  Label1: TLabel;
```

```
Label10: TLabel;
Label11: TLabel;
Label12: TLabel;
Label13: TLabel;
Label14: TLabel;
Label15: TLabel;
Label16: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Label6: TLabel;
Label7: TLabel;
Label8: TLabel;
Label9: TLabel;
procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;

// Base class - triangle
type
  TTriangle = class
    private
      // Arrays to store vertices
      x, y: array [0..2] of real;
    public
      constructor Create;
      // Method for obtaining input data
      procedure Set_Tri(a, b: array of real);
      // Methods for calculating the sides of the triangle.
      function a(): real;
      function b(): real;
      function c(): real;
      { Virtual method to check existence triangle,
        to be overloaded in the derived class. }
      function Check(): boolean; virtual;
      function Perimeter(): real; // Calculate perimeter
      function Area(): real; // Calculate area
      // Virtual method to display results
      function Show(): string; virtual;
    end;

// Derived class - equilateral triangle
type
  TEqTriangle = class(TTriangle)
    public
      { Overloaded methods to check if a triangle is equilateral,
        and to display results. }
```

```
function Check(): boolean; override;
function Show(): string; override;
end;

var
  Form1: TForm1;
// Declare of a variable of type Triangle
  Figure1: TTriangle;
// Declare a variable of type Equilateral Triangle
  Figure2: TEqTriangle;
implementation

// Constructor to set coordinates to zero.
constructor TTriangle.Create;
var
  i: integer;
begin
  for i := 0 to 2 do
  begin
    x[i] := 0; y[i] := 0;
  end;
end;

// Obtain coordinates of vertices.
procedure TTriangle.Set_Tr (a, b: array of real);
var
  i: integer;
begin
  for i := 0 to 2 do
  begin
    x[i] := a[i]; y[i] := b[i];
  end;
end;

// Calculate the sides of triangle a, b, or c
function TTriangle.a(): real;
begin
  a := sqrt(sqr(x[1] - x[0]) + sqr(y[1] - y[0]));
end;
function TTriangle.b(): real;
begin
  b := sqrt(sqr(x[2] - x[1]) + sqr(y[2] - y[1]));
end;
function TTriangle.c(): real;
begin
  c := sqrt(sqr(x[0] - x[2]) + sqr(y[0] - y[2]));
end;

// Calculate perimeter of triangle.
function TTriangle.Perimeter(): real;
begin
```

```
    Perimeter := a() + b() + c();
end;
// Calculate area of triangle.
function TTriangle.Area(): real;
var
  p: real;
begin
  p := Perimeter()/2;           // Semi-perimeter
  Area := sqrt((p - a()) * (p - b()) * (p - c()));
end;

{ Method for checking the existence of a triangle:
  If one of the three points lies on the equation of a straight
  line formed by the other two points, then the three lines are
  on the same straight line and cannot form a triangle. }
function TTriangle.Check(): boolean;
begin
  if (x[0]-x[1])/(x[0]-x[2]) = (y[0]-y[1])/(y[0]-y[2]) then
    Check := false
  else
    Check := true;
end;

// Display results
function TTriangle.Show(): string;
begin
// If triangle exists, then display results
if Check() then
  Show := 'Tr' + chr(13) + 'a = ' + FloatToStrF(a(),ffFixed,5,2)
+chr(13) + 'b = ' + FloatToStrF(b(),ffFixed,5,2) + chr(13) +'c = '
+FloatToStrF(c(),ffFixed,5,2) + chr(13) + 'P = ' +FloatToStrF
(Perimeter(),ffFixed,5,2) + chr(13) + 'S = ' +FloatToStrF(Area(),
ffFixed, 5, 2)
else
  Show := 'Not Triangle';
end;
// Check for equilateral triangle.
function TEqTriangle.Check(): boolean;
begin
  if (a() = b()) and (b() = c()) then
    Check := true
  else
    Check := false
end;

// Display results for equilateral triangle.
function TEqTriangle.Show(): string;
begin
// If triangle is equilateral, then create info string
  if Check() = true then
    Show := 'Tr' + chr(13) + 'a = ' + FloatToStrF (a(), ffFixed,
```

```
5,2) +chr(13) + 'P = ' + FloatToStrF (Perimeter(), ffFixed, 5, 2) +
chr(13) + 'S = ' + FloatToStrF (Area(), ffFixed, 5, 2)
else
  Show := 'Not equilateral triangle';
end;

{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
// Arrays x1, y1 - coordinates of triangle
// Arrays x2, y2 - coordinates of equilateral triangle
var
  x1, y1, x2, y2: array [1..3] of real;
  s: string;
begin
  // Read the coordinates from the text boxes
  x1[1] := StrToFloat(Edit1.Text);
  y1[1] := StrToFloat(Edit2.Text);
  x1[2] := StrToFloat(Edit3.Text);
  y1[2] := StrToFloat(Edit4.Text);
  x1[3] := StrToFloat(Edit5.Text);
  y1[3] := StrToFloat(Edit6.Text);
  x2[1] := StrToFloat(Edit7.Text);
  y2[1] := StrToFloat(Edit8.Text);
  x2[2] := StrToFloat(Edit9.Text);
  y2[2] := StrToFloat(Edit10.Text);
  x2[3] := StrToFloat(Edit11.Text);
  y2[3] := StrToFloat(Edit12.Text);
  // Initialize triangle object
  Figure1 := TTriangle.Create;
  // Initialize equilateral triangle object
  Figure2 := TEqTriangle.Create;
  Figure1.Set_Tr (x1, y1);
  Figure2.Set_Tr (x2, y2);
  // Create output strings
  s := Figure1.Show();
  Label15.Caption := s;
  s := Figure2.Show();
  Label16.Caption := s;
  // Destroy objects
  Figure1.Free;
  Figure2.Free;
end;

end.
```

The results of the program are shown in Figure 9.3.

An *abstract* method is a virtual or dynamic method that is not defined in the class where it is declared. It is assumed that this method will be overloaded in a derived class. Call the method only in those classes where it is overloaded. An abstract

method is declared by placing the **abstract** keyword after the words virtual or dynamic. For example:

```
method1; virtual; abstract;
```

Triangle:		Equilateral Triangle:	
X1:	2	X1:	1
Y1:	4	Y1:	1
X2:	3	X2:	12
Y2:	12	Y2:	1
X3:	5	X3:	6
Y3:	6	Y3:	19

**Calculate**

Tr  
 $a = 8.06$   
 $b = 6.32$   
 $c = 3.61$   
 $P = 17.99$   
 $S = 3.67$

Not equilateral triangle

Figure 9.3: Calculating the triangle parameters

Consider the following example. Create a base class TFigure. Based on this class, derive classes for real figures (circle, quadrilateral, etc.). In our example, consider two derived classes TCircle (circle) and TRectangle (rectangle).

Place a button on a form. When the button is clicked, two objects, a circle and a rectangle, will be initialized and their perimeter (or circumference) and area calculated. Results will be shown in Label1 and Label2. Below is the program listing.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
{ TForm1 }
TForm1 = class(TForm)
  Button1: TButton;
  Label1: TLabel;
  Label2: TLabel;
  procedure Button1Click(Sender: TObject);
end;
```

```
private
  {private declarations}
public
  {public declarations}
end;

// Declare base class
type
  TFigure = class
private
  n: integer;           // Number of sides
  p: array of real;    // Array of lengths of sides
public
  // Abstract constructor. Overload in derived class
  constructor Create; virtual; abstract;
  function Perimeter(): real; // Calculate perimeter
  // Abstract method to compute area. Must be overloaded.
  function Area(): real; virtual; abstract;
  // Abstract method to show results. Must be overloaded.
  function Show(): string; virtual; abstract;
end;

// Declare derived class - circle
type
  TCircle = class(TFigure)
public
  constructor Create; override;
  function Perimeter(): real;
  function Area(): real; override;
  function Show(): string; override;
end;

// Declare derived class - rectangle
type
  TRectangle = class(TFigure)
public
  constructor Create; override;
  function Area(): real; override;
  function Show(): string; override;
end;

var
  Form1: TForm1;
  Figure1: TCircle;
  Figure2: TRectangle;
implementation
  // Define base class perimeter calculation method
  function TFigure.Perimeter(): real;
  var
    i: integer;
    s: real;
```

```
begin
  s := 0;
  for i := 0 to n - 1 do
    s := s + p[i];
  Perimeter := s;
end;

// Define circle constructor. Reload abstract parent constructor
constructor TCircle.Create;
begin
  // Set number of sides for circle to 1
  n := 1;
  // Allocate memory for 1 array element
  SetLength (p, n);
  p[0] := 5;      // Side - radius of circle
end;

// Overload method to calculate perimeter.
function TCircle.Perimeter(): real;
begin
  Perimeter := 2*Pi*p[0];           // Compute circumference
end;

// Overload the area calculation method.
function TCircle.Area(): real;
begin
  Area := Pi*sqr(p[0]);
end;

// Define Show() for circle.Overload abstract parent method.
function TCircle.Show(): string;
begin
  Show := 'Circle' + chr(13) + 'r =' +
    FloatToStr(p[0]) + chr(13) + 'P =' +
    FloatToStr(Perimeter()) + chr(13) +
    'S =' + FloatToStr(Area());
end;

{ Define constructor for rectangle class. Overload abstract
parent constructor.}
constructor TRectangle.Create;
begin
  n := 2; // number of sides - two
  SetLength (p, n); // memory allocation for two elements
  p[0] := 4; p[1] := 2; // side lengths
end;

// Overload abstract parent Area method
function TRectangle.Area(): real;
begin
  Area := p[0] * p[1];
```

```

end;

// Define rectangle Show. Overload parent abstract method.
function TRectangle.Show(): string;
begin
  Show := 'Rectangle:' + chr(13) + 'a = ' + FloatToStr(p[0]) +
  chr(13) + 'b = ' + FloatToStr(p[1]) + chr(13) + 'P = ' +
  FloatToStr(Perimeter()) + chr(13) + 'S = ' + FloatToStr(Area());
end;

{TForm1}
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  // Initialize a circle object
  Figure1 := TCircle.Create;
  // Initialize a rectangle object
  Figure2 := TRectangle.Create;
  // Generate info to display results
  s := Figure1.Show();
  Label1.Caption := s;
  s := Figure2.Show();
  Label2.Caption := s;
  Figure1.Free;
  Figure2.Free;
end;

end.

```

## 9.4 Operator Overloading

In Free Pascal, you can overload not only functions, but also operators. For example, the `*` operator could be overloaded to perform matrix multiplication when applied to matrices and complex number multiplication when applied to complex numbers.

To do this, a special method must be written. The overload method is declared after the class declaration and looks like this:

**operator** symbol (parameters: type) output\_variable\_name: type;

where:

- **operator** is a keyword;
- symbol is the overloaded operator;
- parameters are the variables involved.

Define the overloaded method in the same manner as other methods, after the implementation keyword in the unit in which the class is declared. Let us look at a

few examples.

EXAMPLE 9.1. Create a class for working with complex numbers, in which overloads the addition and subtraction operators.

On the form, place four TEdit components to enter the real and imaginary parts of two complex numbers. Also, add a TMemo to display the results, and a button. Clicking on the button creates four complex numbers. The first and second numbers are initialized using data entered. The third results from adding the first two, and the fourth is their difference. The commented code is shown below, and Figure 9.4 shows the program in action.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    // To enter the real part of the first number:
    Edit1: TEdit;
    // To enter the imaginary part of the first number:
    Edit2: TEdit;
    // To enter the real part of the second number:
    Edit3: TEdit;
    // To enter the imaginary part of the second number:
    Edit4: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Memo1: TMemo;      // Field for displaying results
procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;

// Declare a complex number class
type
```

```
TComplex = class
private
  x: real;           // Real part
  y: real;           // Imaginary part
public
  constructor Create;
  function Modulus(): real;
  function Argument(): real;
  function ComplexToStr(): string;
end;

// Declare overloaded + operator for complex numbers
operator + (const a, b: TComplex) r: TComplex;
// Declare overloaded - operator for complex numbers
operator - (const a, b: TComplex) r: TComplex;

var
  Form1: TForm1;
  // Declare complex numbers
  complex1: Tcomplex;
  complex2: TComplex;
  complex3: TComplex;
  complex4: TComplex;
implementation
  // Define complex number constructor.
  constructor TComplex.Create;
  begin
    x := 0; y := 0;
    inherited Create;
  end;

  function TComplex.Modulus(): real;
  begin
    modulus := sqrt(x*x + y*y);
  end;

  function TComplex.Argument(): real;
  begin
    argument := arctan(y / x) * 180 / pi;
  end;

  // Overloading + operator. a and b are complex numbers to add.
  // r is the resulting complex number.
  operator + (const a, b: TComplex) r: TComplex;
  begin
    r := TComplex.Create;
    { The real part of the new complex number is the result of
      adding the real parts of the first and second operands. The
      imaginary part of the new complex number is the result of
      adding the imaginary parts of the first and second operands.
      Operands are complex numbers passed to the method. }
  end;
```

```
r.x := a.x + b.X;
r.y := a.y + b.Y;
end;

// Overloading the - operator
operator - (const a, b: TComplex) r: TComplex;
begin
  r := TComplex.Create;
  r.x := a.x - b.x;
  r.y := a.y - b.y;
end;

function TComplex.ComplexToStr(): string;
{ The real part of the new complex number is the difference
  between the real parts of the first and second operands.
  The imaginary part of the new complex number is the difference
  between the imaginary parts of the first and second operands.
  Operands are complex numbers passed to the method. }
begin
  if y >= 0 then
    ComplexToStr := FloatToStrF(x, ffFixed,5,2) + '+' +
      FloatToStrF(y, ffFixed,5,2) + 'i'
  else
    ComplexToStr := FloatToStrF(x, ffFixed,5,2) +
      FloatToStrF(y, ffFixed,5,2) + 'i';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Str1: string;
begin
  // Initialize objects
  complex1 := TComplex.Create;
  complex2 := TComplex.Create;
  complex3 := TComplex.Create;
  complex4 := TComplex.Create;
  // Read real and imaginary parts of original complex numbers
  complex1.x := StrToFloat(Edit1.Text);
  complex1.y := StrToFloat(Edit2.Text);
  complex2.x := StrToFloat(Edit3.Text);
  complex2.y := StrToFloat(Edit4.Text);
  // Use overloaded + operator to add two complex numbers.
  complex3 := complex1 + complex2;
  // Use overloaded - operator to subtract two complex numbers.
  complex4 := complex1 - complex2;
  Str1 := 'complex1 + complex2: ' + complex3.ComplexToStr();
  Memo1.Lines.Add(Str1);
  Str1 := 'Complex3 modulus: ' + FloatToStrF (complex3.Modulus(),
    ffFixed,5,2);
  Memo1.Lines.Add (Str1);
  Str1 := 'Complex3 argument: ' +
```

```

      FloatToStrF(complex3.Argument(), fffFixed,5,2);
      Memo1.Lines.Add(Str1);
      Str1 := 'complex1 - complex2: ' + complex4.ComplexToStr();
      Memo1.Lines.Add(Str1);
      Str1 := 'Complex modulus: ' + FloatToStrF(complex4.Modulus(),
      fffFixed,5,2);
      Memo1.Lines.Add (Str1);
      Str1 := 'Complex4 argument: ' +FloatToStrF(complex4.Argument(),
      fffFixed, 5, 2);
      Memo1.Lines.Add(Str1);
end;

end.

```

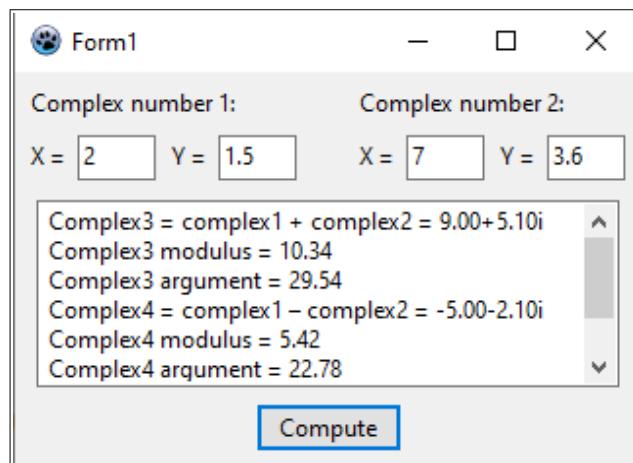


Figure 9.4: Program for Example 9.1

EXAMPLE 9.2. Create a class for working with square matrices, in which the addition and multiplication operators are overloaded. Create a method to check if the matrix is an identity matrix.

Two methods for overloading the addition and multiplication operators will be created in the class. In the first case, two matrices are added and multiplied. In the second case, a real number is added to the matrix, or the matrix is multiplied by real number.

Place a button and eight TLabel components on the form for displaying results. When the button is clicked, six matrix objects will be created and filled with random numbers.

The third matrix will then be populated with the sum of the first two. The fourth matrix will be populated with the product of multiplication of the first matrix to the

second. The fifth matrix is obtained from the first by adding the number 10 to it, and the sixth by multiplying the first matrix by the number 5.

The first and the second matrices are also checked to see if they are identity matrices.

The commented code is shown below, and Figure 9.5 shows the program results.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    {private declarations}
  public
    {public declarations}
  end;

  // Declare matrix class
type
  TMatrix = class
  private
    x: array [0..5, 0..5] of real;
  public
    constructor Create;
    // Create output string to display matrix elements
    function MatrixToStr(): string;
    // Checking for identity matrix
    function Identity(): boolean;
  end;

  // Overload the + operator for adding two matrices.
operator + (const a, b: TMatrix) r: TMatrix;
  // Overload the + operator for adding a number to a matrix.
operator + (const a: TMatrix; b: real) r: TMatrix;
  // Overload the * operator for multiplying two matrices.
operator * (const a, b: TMatrix) r: TMatrix;
```

```
// Overload * operator for multiplying a matrix by a real number.
operator * (const a: TMatrix; b: real) r: TMatrix;
var
  Form1: TForm1;
  mat1: TMatrix;
  mat2: TMatrix;
  mat3: TMatrix;
  mat4: TMatrix;
  mat5: TMatrix;
  mat6: TMatrix;
implementation

// Matrix constructor fills the matrix with random numbers
constructor TMatrix.Create;
var
  i, j: integer;
begin
  for i := 0 to 4 do
    for j := 0 to 4 do
      x[i,j] := random(10);
  inherited Create;
end;

// Check for identity matrix.
function TMatrix.Identity(): boolean;
var
  i, j: integer;
begin
  // Assume that matrix is an identity matrix
  Result := true;
  for i := 0 to 4 do
    for j := 0 to 4 do
      // If element on main diagonal is not 1
      // or element off main diagonal is not 0,
      if ((i = j) and (x[i,j] < 1)) or ((i >> j) and (x[i,j] <>
        0)) then
        begin
          // Matrix is not an identity matrix.
          Result := false;
          break;           // Exit loop early
        end;
  end;

  // Overload + operator to add two matrices
operator + (const a, b: TMatrix) r: TMatrix;
var
  i, j: integer;
begin
  r := TMatrix.Create;
  for i := 0 to 4 do
    for j := 0 to 4 do
```

```
r.x[i,j] := a.x[i,i] + b.x[i,j];
end;

// Overload * operator to multiply two matrices
operator * (const a, b: TMatrix) r: TMatrix;
var
  i, j, k: integer;
begin
  r := TMatrix.Create;
  for i := 0 to 4 do
    for j := 0 to 4 do
      begin
        r.x[i,j] := 0;
        for k := 0 to 4 do
          r.x[i,j] := r.x[i,j] + a.x[i,k] * b.x[k,j];
      end;
  end;

// Overload + operator to add a matrix and a real number
operator + (const a: TMatrix; b: real) r: TMatrix;
var
  i, j: integer;
begin
  r := TMatrix.Create;
  for i := 0 to 4 do
    for j := 0 to 4 do
      r.x[i,j] := a.x[i,j] + b;
  end;

// Overload * operator to multiply matrix by a real number.
operator * (const a: TMatrix; b: real) r: TMatrix;
var
  i, j: integer;
begin
  r := TMatrix.Create;
  for i := 0 to 4 do
    for j := 0 to 4 do
      r.x[i,j] := a.x[i,j] * b;
  end;

// Create string with matrix elements.
function TMatrix.MatrixToStr(): string;
var
  s: string;
  i, j: integer;
begin
  s := '';                      // First line is empty
  for i := 0 to 4 do
    begin
      for j := 0 to 4 do
        // Add element and space to s
        s := s + FloatToStrF(x [i,j],fffFixed,5,2) + ' ';
```

```
// Move to next row
s := s + chr(13);
end;
MatrixToStr := s;
end;

// Handle the button click:
// Create instances of the Matrix class.
// Calculate the resulting matrices by
// adding and multiplying the original matrices.
// Multiply matrix by number.
procedure TForm1.Button1Click(Sender: TObject);
var
  Str1: string;
begin
  // Initialize objects:
  // mat1, mat2 - initial matrices, filled with random numbers;
  // mat3 is obtained by adding the two original matrices;
  // mat4 is obtained by multiplying the two initial matrices;
  // matr5 is obtained by adding 10 to the first matrix;
  // matr6 is obtained by multiplying the first matrix by 5.
  mat1 := TMatrix.Create;
  mat2 := TMatrix.Create;
  mat3 := TMatrix.Create;
  mat4 := TMatrix.Create;
  mat5 := TMatrix.Create;
  mat6 := TMatrix.Create;
  // Calculate mat3, mat4, mat5 and mat6,
  // using overloaded operators.
  mat3 := mat1 + mat2;
  mat4 := mat1 * mat2;
  mat5 := mat1 + 10;
  mat6 := mat1 * 5;

  // Create output string for mat1 and display in Label1.
  Str1 := 'Matrix1 ' + chr(13) + mat1.MatrixToStr();
  Label1.Caption := Str1;

  // Check if mat1 is an identity matrix
  // and display the corresponding message.
  if mat1.Identity() = true then
    Str1 := 'Matrix1 is an identity matrix'
  else
    Str1 := 'Matrix1 is not an identity matrix';
  Label5.Caption := Str1;

  // Check if mat2 is an identity matrix
  // and display the corresponding message.
  if mat2.Identity() = true then
    Str1 := 'Matrix2 is an identity matrix'
  else
```

```
Str1 := 'Matrix2 is not an identity matrix';
Label6.Caption := Str1;

Str1 := 'Matrix2 ' + chr(13) + mat2.MatrixToStr();
Label2.Caption := Str1;
// Output matrix elements mat3 = mat1 + mat2
Str1 := 'Matrix1 + Matrix2 ' + chr(13) + mat3.MatrixToStr();
Label3.Caption := Str1;
// Output matrix elements mat4 = mat1 * mat2
Str1 := 'Matrix1 * Matrix2 ' + chr(13) + mat4.MatrixToStr();
Label4.Caption := Str1;
// Output matrix elements mat5 = mat1 + 10
Str1 := 'Matrix1 + 10 ' + chr(13) + mat5.MatrixToStr();
Label7.Caption := Str1;
// Output matrix elements mat5 = mat1 * 5
Str1 := 'Matrix1 * 5 ' + chr(13) + mat6.MatrixToStr();
Label8.Caption := Str1;
end;

end.
```

EXAMPLE 9.3. Create a class for the common fraction, for which the < and > operators are overloaded.

Place four TEdit components on a form for entering the numerators and denominators of two common fractions and a Memo1 component for displaying results. Also place a button on the form that, when clicked, will create two common fraction objects. They will then be compared and the result displayed in Memo1. The program listing with comments is shown below and the results of the program are shown in Figure 9.6.

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
```

Matrix1	Matrix2
5.00 5.00 7.00 8.00 6.00	3.00 9.00 8.00 0.00 3.00
8.00 5.00 8.00 4.00 6.00	0.00 6.00 0.00 3.00 8.00
6.00 3.00 4.00 2.00 8.00	9.00 7.00 1.00 8.00 8.00
0.00 9.00 2.00 3.00 4.00	9.00 4.00 7.00 8.00 4.00
7.00 8.00 5.00 4.00 5.00	5.00 7.00 6.00 1.00 7.00

Matrix1 is not an identity matrix      Matrix2 is not an identity matrix

Matrix1 + Matrix2	Matrix1 + 10
8.00 14.00 15.00 8.00 9.00	15.00 15.00 17.00 18.00 16.00
8.00 11.00 8.00 7.00 14.00	18.00 15.00 18.00 14.00 16.00
15.00 10.00 5.00 10.00 16.00	16.00 13.00 14.00 12.00 18.00
9.00 13.00 9.00 11.00 8.00	10.00 19.00 12.00 13.00 14.00
12.00 15.00 11.00 5.00 12.00	17.00 18.00 15.00 14.00 15.00

Matrix1 * Matrix2	Matrix1 * 5
180.00 198.00 139.00 141.00 185.00	25.00 25.00 35.00 40.00 30.00
162.00 216.00 136.00 117.00 186.00	40.00 25.00 40.00 20.00 30.00
112.00 164.00 114.00 65.00 138.00	30.00 15.00 20.00 10.00 40.00
65.00 108.00 47.00 71.00 128.00	0.00 45.00 10.00 15.00 20.00
127.00 197.00 119.00 101.00 176.00	35.00 40.00 25.00 20.00 25.00

**Compute**

Figure 9.5: Results from Example 9.2

```

type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;      // Numerator for first fraction
    // Denominator of the first fraction
    Edit2: TEdit;
    // Numerator of the second fraction
    Edit3: TEdit;
    // Denominator of the second fraction
    Edit4: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Memo1: TMemo;      // Display results
    procedure Button1Click(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;
// Declare the common fraction class

```

```
type
TComfrac = class
private
  Nu: integer;           // Numerator
  De: integer;           // Denominator
public
  constructor Create(a, b: integer);
  // Create string to display fraction
  function FracToStr(): string;
end;

// Overload the < operator
operator < (const a, b: TComfrac) r: boolean;
// Overload the > operator
operator > (const a, b: TComfrac) r: boolean;

var
  Form1: TForm1;
  // Declare common fractions
  d1, d2: TComfrac;
implementation

constructor TComfrac.Create (a, b: integer);
begin
  Nu := a; De := b;
  inherited Create;
end;

// Overload the < operator to compare common fractions
operator < (const a, b: TComfrac) r: boolean;
begin
  if a.Nu * b.De < b.Nu * a.De then
    r := true
  else
    r := false;
end;

// Overload the > operator to compare common fractions
operator > (const a, b: TComfrac) r: boolean;
begin
  if a.Nu * b.De > b.Nu * a.De then
    r := true
  else
    r := false;
end;

// Create a string formation to display the fraction
function TComfrac.FracToStr(): string;
begin
  if De <> 0 then
    if Nu * De > 0 then
```

```
        FracToStr := IntToStr(Nu) + '/' + IntToStr(De)
    else
        FracToStr := '-' + IntToStr(abs(Nu)) + '/' + IntToStr(abs(De))
    else
        FracToStr := 'Division by zero'
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Str1: string;
    a, b: integer;
begin
    // Read data from form
    a := StrToInt(Edit1.Text);           // Numerator
    b := StrToInt(Edit2.Text);           // Denominator
    // Initialize the first fraction
    d1 := TComfrac.Create(a, b);
    // Read data from form
    a := StrToInt(Edit3.Text);
    b := StrToInt(Edit4.Text);
    // Initialize the second fraction
    d2 := TComfrac.Create(a, b);
    // Create output string and add it to Memo1.
    // Output original fraction d1.
    Str1 := 'Fraction 1 = ' + d1.FracToStr();
    Memo1.Lines.Add(Str1);
    // Output original fraction d2.
    Str1 := 'Fraction 2 = ' + d2.FracToStr();
    Memo1.Lines.Add(Str1);
    // Compare fractions using overloaded
    // < and > operators and display the message.
    if d1 < d2 then
        Str1 := 'Fraction 1 < Fraction 2'
    else
        if d1 > d2 then
            Str1 := 'Fraction 1 > Fraction 2'
        else
            Str1 := 'Fraction 1 = Fraction 2';
    Memo1.Lines.Add(Str1);
end;

end.
```

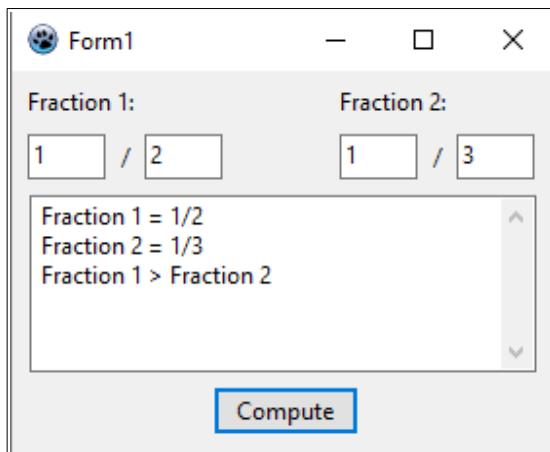


Figure 9.6: Results from Example 9.3

## 9.5 Exercises

- 1) Create a class for a complex number in algebraic form  $z = x + yi$ , with fields for the real ( $x$ ) and imaginary ( $y$ ) parts of the number. Include methods that calculate the root of the complex number and display the complex number. Overload the addition, subtraction, division and multiplication operators for complex numbers.
- 2) Create a *square matrix* class, with fields for dimension and matrix elements. Include a method to display the matrix. Overload the addition, subtraction and multiplication operators, and check if one matrix is inverse of another ( $A \cdot A^{-1} = I$ , where  $I$  is the identity matrix).
- 3) Create a *vector on a plane* class, with fields for the vector coordinates. Include methods to calculate directional cosines and output all vector characteristics. Overload the addition operator, and the multiplication operator for the scalar and cross product of vectors.
- 4) Create a common fraction class, with fields for the numerator and denominator. Include methods for fraction reduction and output. Overload the addition, subtraction, division and multiplication operators.
- 5) Create a *square* class, with a field for the length of each side. Provide methods to calculate and display the perimeter, area and diagonal of the square. Create a derived class *cube* and add a method to compute the volume of the cube. Overload the area calculation method and the method display to display information for the object.
- 6) Create *square matrix* class, with fields for the dimension and matrix elements. Include methods to calculate the sum of all elements and output

- the matrix. Overload the addition, subtraction and multiplication operators to multiply matrices and a matrix by a number.
- 7) Create a *line* class, with fields for its end coordinates ( $x_1, y_1$ ) and ( $x_2, y_2$ ). Include methods to derive the equation of the straight line in the form  $y = ax + b$ . Overload operators to determine if line is parallel with another, and the angle between the line and another.
  - 8) Create a *complex number* class in trigonometric form  $a = \rho(\cos\varphi + i\sin\varphi)$ , with fields for the modulus ( $\rho$ ) and argument ( $\varphi$ ). Include methods to raise the complex number to a power, and to display a complex number in algebraic and trigonometric forms. Overload the addition, subtraction, division and multiplication operators for complex numbers.
  - 9) Create a *vector on a plane* class, with fields for vector coordinates. Include methods to calculate of vector length and output vector characteristics. Overload the addition operator and the multiplication operator to compute the scalar and the vector product of vectors.
  - 10) Create a *common fraction* class, with fields for the numerator and denominator. Include methods for inverse fraction detection and fraction output. Overload the addition, subtraction, division and multiplication operators for common fractions.
  - 11) Create a *square matrix* class, with fields for the dimension and matrix elements. Include methods to check if the matrix is an upper or lower triangular matrix, and for matrix output. Overload the addition, subtraction multiplication operators, for matrix multiplication, and matrix multiplication by a number.
  - 12) Create a *triangle* class, with fields for the length of three sides. Include methods to check the existence of a triangle, and to calculate and output the length of the sides, angles, perimeter and area. Derive an isosceles triangle class, provide a method to check if the triangle is an isosceles triangle.
  - 13) Create a complex number class in exponential form  $a = \rho e^{i\varphi}$ , with fields for the modulus ( $\rho$ ) and argument ( $\varphi$ ). Include methods to output the complex number in algebraic, trigonometric and exponential forms. Overload the addition, subtraction, division and multiplication operators for complex numbers.
  - 14) Create a *line* class, with fields for the coefficients of the equation  $y = ax + b$ . Include methods to derive the equation of a line and find the points of intersection with the axes. Overload operators to check if the line is perpendicular to another, and to find the angle between the line and another.

- 15) Create a *square matrix* class, with fields for the dimension and matrix elements. Include methods for checking if the matrix is a diagonal or null matrix, and for matrix output. Overload the addition, subtraction and multiplication operators, for matrix multiplication and adding a number to a matrix.
- 16) Create a *triangle* class, with fields for the coordinates of the three vertices. Provide methods to check for the existence of a triangle, and to calculate and display the length of sides, angles, perimeter and area. Derive class *right triangle* class and provide a method to check if the triangle is a right triangle.
- 17) Create a *complex number* class in trigonometric form  $a = p(\cos \varphi + i \sin \varphi)$ , with fields for the modulus ( $p$ ) and argument ( $\varphi$ ). Include methods to extract a root from the number and display the complex number in algebraic and trigonometric forms. Overload the addition, subtraction, division and multiplication operators for complex numbers.
- 18) Create a *common fraction* class, with fields for the numerator and denominator. Include methods to raise the fraction to a power and output the fraction. Overload the addition, subtraction, division and multiplication operators for fractions.
- 19) Create a *triangle* class, with fields for the length of three sides. Include methods to check for the existence of a triangle, and to calculate and display the length of the sides, angles, perimeter and area. Derive an *equilateral triangle* class and overload the checking method to check if the triangle is equilateral.
- 20) Create a *complex number* class in algebraic form  $z = x + yi$ , with fields for the real ( $x$ ) and imaginary ( $y$ ) parts of the number. Include methods to calculate the modulus and argument of the complex number, and to display the complex number. Overloading the addition and subtraction operators for complex numbers and for checking the conjugacy of two complex numbers.
- 21) Create a *circle* class, with a field for the radius  $R$ . Provide methods to calculate and display the area and circumference. Derive a class for *cylinder with height h* and add a method to determine its volume. Overload the methods for calculating the area and displaying information about the object.
- 22) Create a *vector on a plane* class, with fields for vector coordinates. Include methods to calculate the vector length and output vector characteristics. Provide methods for overloading operators for addition, the scalar and cross product of vectors, and for calculating the angle between vectors.
- 23) Create a *square matrix* class, with fields for the dimension and matrix

elements. Include methods to check if the matrix is symmetric ( $A = A^T$ ), and to output the matrix. Overload the addition, subtraction and multiplication operators, for matrix multiplication, and the addition of a number to a matrix.

- 24) Create a *common fraction* class, with fields for the numerator and denominator. Include a method to output the fraction. Overload the addition, subtraction, division, multiplication and comparison operators of fractions.
- 25) Create a *square* class, with a field for the length of a side. Provide methods to calculate and display the diagonal, perimeter and area. Derive a class for a *right square prism with height H* and add a method to determine the volume of the object. Overload the methods to calculate the area and display information about the object.

#### Endnotes:

---

- 1 We have already encountered complex numbers in previous chapters. Read more about them you on the page [http://kvant.mccme.ru./1982/03/kompleksnye\\_chisla.htm](http://kvant.mccme.ru./1982/03/kompleksnye_chisla.htm).

This page deliberately left blank.

## Chapter 10. Graphics in Lazarus

This chapter looks at the graphical capabilities of the language. The main procedures and functions for working with graphics are examined. An example for graphing functions will be considered.

### 10.1 Drawing Tools in Lazarus

When developing a project with drawing capability, you have at your disposal a canvas (Canvas property), pencil (Pen property), and brush (Brush property). The following components have a Canvas property:

- form (class TForm);
- grid (TStringGrid class);
- bitmap image (class TImage);
- printer (TPrinter class).

When drawing on a component that has a Canvas property, the component itself is considered a rectangular grid consisting of individual points, called pixels. The position of a pixel is characterized by its vertical (X) and horizontal (Y) coordinates. The upper left pixel has a coordinate (0,0). Vertical coordinates increase from top to bottom, and horizontal from left to right. The total number of pixels vertically is determined by the Height property, and horizontally by Width. Every pixel can have its own color. To access any point on the canvas, use the Pixels[X, Y]: TColor property. This property defines the color of the pixel with coordinates X (integer), Y (integer).

The color of any pixel on the canvas can be changed, using the following assignment expression:

```
Component.Canvas.Pixels[X,Y] := Color;
```

where Color is a variable or constant of type TColor.

The following color constants are defined (Table 10.1).

The color of any pixel can be obtained using the following expression:

```
Color := Component.Canvas.Pixels[X,Y];
```

where Color is a variable of type Tcolor.

The point color class Tcolor is defined as a longint (integer). Variables of this type occupy four bytes in memory, which contain information about the proportions of blue (B), green (G) and red (R) colors and are arranged as follows: \$00BBGGRR.

For drawing, TCanvas methods can draw figures (line, rectangle, etc.) or display text in graphics mode. Three classes that define the display of figures and text are listed below:

*Table 10.1: Color properties*

Constant	Color	Constant	Color
clBlack	Black	clSilver	Silver
clMaroon	Chestnut	clRed	Red
clGreen	Green	clLime	Light green
clOlive	Olive	clBlue	Blue
clNavy	Dark blue	clFuchsia	Hot pink
clPurple	Pink	clAqua	Turquoise
clTeal	Azure	clWhite	White
clGray	Grey		

- `TFont`;
- `TPen`;
- `TBrush`.

**TFONT class.** The following are properties of the `Canvas.TFont` object:

- Name (string), the name of the font.
- Size (integer), the font size in points, which is a font measuring unit equal to 0.353 mm or 1/72 inch.
- Style, which can be normal, **bold** (`fsBold`), *italic* (`fsItalic`), underline (`fsUnderline`) or strikethrough (`fsStrikeOut`). Several styles can be combined. For example, set the style to ***bold italic*** as follows:

```
Object.Canvas.Font.Style := [fsItalic, fsBold]
```

- Color (Tcolor), the color of characters.
- Charset (type 0..255), the font character set. Every font supports one or more character sets. Table 10.2 shows a few values for Charset.

**TPEN class.** A pen is a tool to draw points, lines, geometric shapes, etc. Basic properties of `Canvas.TPen` include:

- **Color** (Tcolor) defines the line color;
- **Width** (integer) sets the line width in pixels;
- **Style** makes it possible to choose a line type. This property can take the values shown in Table 10.3.

*Table 10.2: Charset values*

<b>Constant</b>	<b>Value</b>	<b>Description</b>
ANSI_CHARSET	0	ANSI characters
DEFAULT_CHARSET	1	Set by default. The font can be selected only by Name and Size only. If the requested font is not available in the system, another font will be substituted.
SYMBOL_CHARSET	2	Standard character set
MAC_CHARSET	77	Macintosh characters
GREEK_CHARSET	161	Greek characters
RUSSIAN_CHARSET	204	Cyrillic characters
EASTEUROPE_CHARSET	238	Includes dialectical signs (signs, added to letters to modify their pronunciation) for East European languages

*Table 10.3: Line types*

<b>Value</b>	<b>Description</b>
psSolid	Solid line
psDash	Dashed line
psDot	Dotted line
psDashDot	Dash-dotted line
psDashDotDot	Line with alternating dash and two dots
psClear	No line

- **Mode** defines how the pen and canvas colors interact. The choice of the value of this property produces various effects. Possible values of Mode are shown in Table 10.4. By default, a line is drawn with the color specified in Pen.Color, but it is possible to select the inverse line color relative to the background color. This will cause the line to be always visible, even if the line and background colors are the same.

**TBRUSH class.** The brush (Canvas.Brush) is used fill inside closed shapes. The brush has two main properties:

- **Color** (Tcolor), the color used to fill the enclosed area;
- **Style** (TBrushStyle), the fill pattern, including solid (bsSolid), transparent

(bsClear), horizontal lines (bsHorizontal), vertical lines (bsVertical), diagonal lines (bsFDiagonal, bsBDiagonal), grid (bsCross), and lattice (bsDiagCrossdiagonal).

*Table 10.4: Values for the Mode property*

Mode	Operation	Pixel color
pmBlack	Black	Always black
pmWhite	White	Always white
pmNop		Unaltered
pmNot	Not Screen	Inverse color with respect to background color
pmCopy	Pen	Color specified in Pen color property (this value is default)
pmNotCopy	Not pen	Pen color inversion
pmMergePenNot	Pen or Not Pen	Disjunction of pen color and inverse background color
pmMaskPenNot	Pen and Not Screen	Conjunction of pen color and inverse background color
pmMergeNotPen	Not pen or screen	Disjunction of background color and inverse pen color
pmMaskNotPen	Not pen and screen	Conjunction of background color and inverse pen color
pmMerge	Pen or Screen	Disjunction of pen color and background color
pmNotMerge	Not (Pen or Screen)	Inverse pmMerge mode
pmMask	Pen and Screen	Conjunction of pen color and background color
pmNotMask	Not (Pen and Screen)	Inverse pmMask mode
pmXor	Pen xor Screen	Xor operation on pen color and background color
pmNotXor	Not (Pen xor Screen)	Inverse pmXor mode

**TCANVAS class.** This class is the main tool for drawing graphics. Let us look at its most used methods.

**procedure MoveTo(X, Y: integer);**

The MoveTo method changes the current pen position to the position at point (X,Y).

The current position is stored as a TPoint type in PenPos. The definition of the TPoint type is:

```
type TPoint = record
  X: longint;
  Y: longint;
end;
```

The current pen position can be read using the PenPos property:

```
X := PenPos.X;
Y := PenPos.Y;
```

**procedure** LineTo(X, Y: **integer**);

The LineTo method connects the current pen position and point (X,Y) with a straight line. The current position of the pen moves to point (X, Y).

An example will illustrate the procedure. Place a button on a form and enter the procedure for handling the TForm1.Button1Click event, which draws straight lines:

```
procedure TForm1.Button1Click(Sender: TObject)
begin
  Form1.Canvas.LineTo(30,50);
end;
```

When the button is clicked, a straight line will be drawn on the form, connecting the point with coordinates (0,0) to the point with coordinates (30,50). When the button is clicked again, the procedure will draw the same line over the old one.

Now, rewrite the event handler as shown below:

```
procedure TForm1.Button1Click(Sender: TObject)
begin
  Form1.Canvas.LineTo(Canvas.PenPos.X+30, Canvas.PenPos.Y+50);
end;
```

The first click will draw a similar line. clicking again, the procedure will draw a line connecting the current point to the point obtained by adding 30 to the X coordinate, and 50 to the Y coordinate. That is, after clicking again, the procedure connects points (30,50) and (60,100) with a straight line. The third click will connect points (60,100) and (90,150), etc.

**procedure** PolyLine(**const** Points **array of** TPoint);

The PolyLine method draws a polyline, with the coordinates of its vertices stored in an array of Points.

An example will illustrate the procedure. Place an Exit button on a form and enter the following event handlers:

```
// Form OnPaint event handler
procedure TForm1.FormPaint(Sender: TObject);
var
  temp: array [1..25] of TPoint;
  i: byte;
  j: integer;
begin
  j := 1;
  for i := 1 to 25 do
  begin
    // Calculate the coordinates of the polyline vertices
    temp[i].x := 25 + (i - 1) * 10;
    temp[i].y := 150 - j * (i - 1) * 5;
    j := -j;
  end;
  Form1.Canvas.Polyline(temp);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Close;
end;
```

After starting the program, the form will look like Figure 10.1.

**procedure Ellipse(X1, Y1, X2, Y2: integer);**

The Ellipse method draws an ellipse or circle on the canvas. X1, Y1, X2, Y2 are coordinates of the rectangle enclosing the ellipse. If the rectangle is a square, a circle is drawn.

**procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: integer);**

The Arc method draws an arc of an ellipse. X1, Y1, X2, Y2 are coordinates of the ellipse of which the arc is a part. X3, Y3 define the starting point of the arc. X4, Y4 define its end point. The arc is drawn counterclockwise.

**procedure Rectangle(X1, Y1, X2, Y2: integer);**

The Rectangle method draws a rectangle. X1, Y1, X2 and Y2 are the coordinates of its top-left and bottom-right corners.

**procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: integer);**

This method draws a rectangle with rounded corners. X1, Y1, X2, and Y2 are coordinates of the upper left and lower right corners of the rectangle. X3, Y3 is the size of the ellipse, one quarter of which is used to draw the rounded corner.

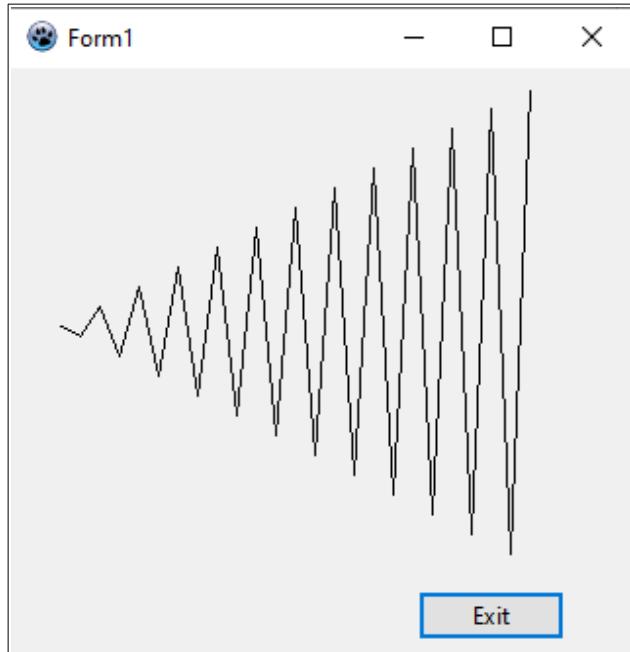


Figure 10.1: Using the PolyLine procedure

**procedure** PolyGon (**const** Points **array of** TPoint);

The PolyGon method draws a closed polygon through the vertices in the Points array. The first and last points are connected by a straight line, unlike the Polyline method, which does not connect the endpoints. Drawing is done using the current Pen, and the inside area of the shape is painted with the current Brush.

**procedure** Pie (X1, Y1, X2, Y2, X3, Y3, X4, Y4: **integer**);

The Pie method draws a closed sector of a circle or ellipse, using current Pen settings. The inside area is filled using the current Brush. Points (X1,Y1) and (X2,Y2) define the rectangle enclosing the ellipse. The arc start point is the intersection of the ellipse with a line passing through its center and point (X3,Y3). The arc end point is the intersection of the ellipse and the line passing through its center and point (X4,Y4). The arc is drawn counterclockwise, from the start to end point. Lines are drawn that bound the segment and pass through the center of the ellipse and points (X3,Y3) and (X4,Y4).

Create a form and set its Height and Width to 500. At the bottom of the form, place a button and set its Caption to Draw. When the program starts and this button is clicked, various figures will be drawn on the form (Figure 10.2). The program listing, demonstrating the methods, is shown below.

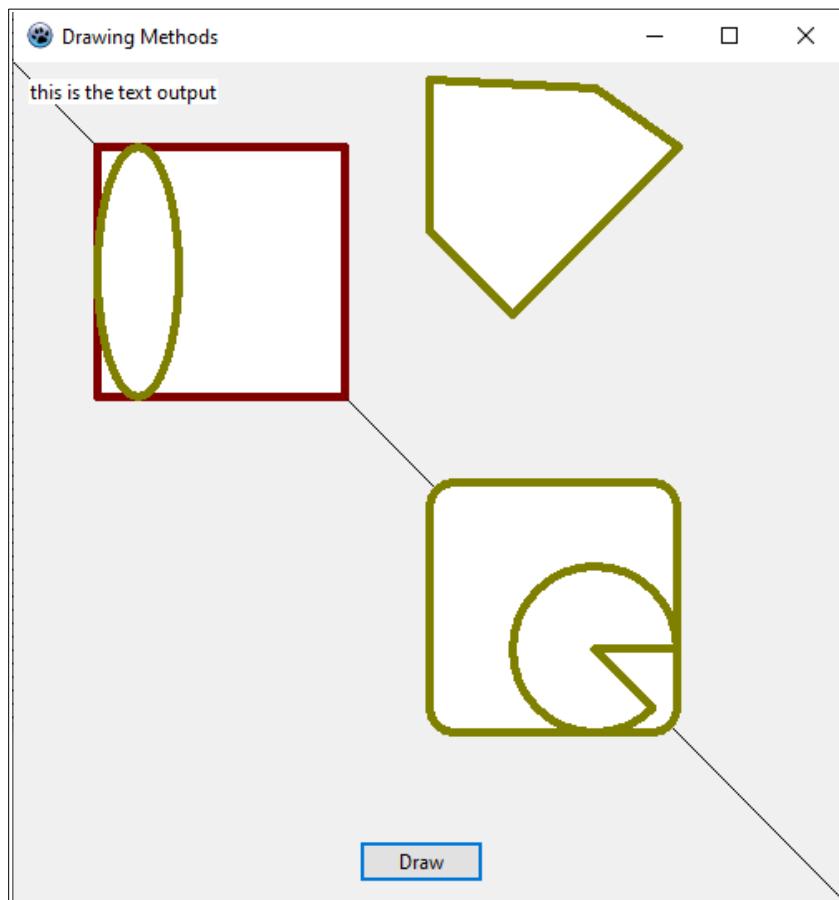


Figure 10.2: Drawing methods example

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormPaint(Sender: TObject);
  private
    {private declarations}
  public
    {public declarations}
  end;
var
```

```

Form1: TForm1;
Clicked: boolean = false;
implementation
{ TForm1}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Clicked := true;
  FormPaint(Sender);
end;
procedure TForm1.FormPaint(Sender: TObject);
var
  t: array [1..5] of TPoint;
begin
  if Clicked then
  begin
    Form1.Canvas.LineTo(500,500);      // Draw a line
    Form1.Canvas.Pen.Color := clMaroon; // Change pen color
    Form1.Canvas.Pen.Width := 5;       // Change line width
    Form1.Canvas.Rectangle(50,50,200,200); // Draw rectangle
    Form1.Canvas.Pen.Color := clOlive;
    Form1.Canvas.Ellipse (50,50,100,200); // Draw ellipse
    // Draw a rounded rectangle
    Form1.Canvas.RoundRect(250,250,400,400,30,30);
    // Draw a sector of a circle
    Form1.Canvas.Pie(300,300,400,400,350,350,500,500);
    // Populate an array with vertices of a pentagon
    t[1].x := 250; t[1].y := 10;
    t[2].x := 350; t[2].y := 15;
    t[3].x := 400; t[3].y := 50;
    t[4].x := 300; t[4].y := 150;
    t[5].x := 250; t[5].y := 100;
    Form1.Canvas.Polygon(t);
    Form1.Canvas.TextOut(10,10,'this is the text output');
  end;
end;
end.
```

**procedure** TextOut(X, Y: **integer**; **const** Text: **string**);

This function writes a line of text, starting at (X,Y). The current PenPos position of the Pen moves to the end of the displayed text. The text is displayed using the current Font settings, while the text background is displayed using the current Brush settings. To align text on the canvas, use methods to determine

the height and length of the text in pixels, including TextExtent, TextHeight, and TextWidth. Let us look at these functions.

**function** TextExtent(**const** Text: **string**): TSize;

This function returns a structure of type TSize containing the width and height in pixels of the Text to be written on the canvas with the current font.

```

type
  TSize = record
    cx: LongInt;           // Width in pixels
    cy: LongInt;           // Height in pixels
  end;

```

**function** TextHeight(**const** Text: **string**): **integer**;

This function returns the height in pixels of the text to be written on the canvas with the current font.

**function** TextWidth(**const** Text: **string**): **integer**;

The function returns the length in pixels of the Text to be written on the canvas in the current font. Before outputting text to the canvas, this helps to determine the text size and how to arrange it and other image elements in the best possible way.

If the brush color differs from the canvas color when text is displayed, the text will be displayed in a colored rectangular frame, with the same dimensions as the text.

We have covered the basic drawing functions and can now start drawing. Before that, however, it should be noted that if a form with an image is closed and then restored, then the image in the form will disappear. Resizing the form can also ruin the image in it. The Object.FormPaint and Object.FormResize event handlers can be used to solve this problem. Object.FormPaint executes when the form appears on screen, and Object.FormResize after resizing the form. Therefore, all drawing expressions should be placed in Object.FormPaint and duplicated in Object.FormResize.

## 10.2 Plotting Graphs

The algorithm for plotting the graph of a continuous function  $y = f(x)$  in segment  $[a; b]$  is as follows. Construct points  $(x_i, f(x_i))$  in Cartesian coordinates and connect them with straight lines. The coordinates of the points may be defined using the following formulas:

$$hx = (b - a)/N,$$

where  $N$  is the number of segments on the segment  $[a; b]$ .

$$x_i = a + (i - 1) hx; y_i = f(x_i),$$

where  $i = 0, \dots, N$ .

The more points drawn, the smoother the plotted graph.

When transferring this algorithm to a form or other component, Lazarus considers

the size and features of the component (the X axis goes from left to right, its coordinates range from 0 to *Width*; the Y axis goes downward, its coordinates range from 0 to *Height*). The X and Y coordinate values must be integers.

All points must be converted from a “paper” coordinate system (X varies from a to b, Y varies from minimum to maximum function values) into “component”<sup>1</sup> (in this coordinate system, the X axis is denoted by the letter U,  $0 \leq U \leq \text{Width}$ , and the Y axis by the letter V,  $0 \leq V \leq \text{Height}$ ).

To convert the X coordinate to the U coordinate, construct a linear function  $cX + d$ , which will translate points in the interval (a;b) to points in the interval  $(X_0, \text{Width} - X_k)$ <sup>2</sup>. Since point *a* in the paper coordinate system becomes point  $X_0$  in “screen coordinates”, and point *b* becomes  $\text{Width} - X_k$ , then the system of linear equations for finding the coefficients c and d has the form:

$$\begin{cases} c \times a + d = X_0 \\ c \times b + d = \text{Width} - X_k \end{cases}$$

After solving it, the coefficients c, d are:

$$\begin{cases} d = X_0 - c \times a \\ c = (\text{Width} - X_k - X_0)/(b - a) \end{cases}$$

To transform the Y coordinate into the V coordinate, construct a linear function  $V = gY + h$ . The *min* point in the paper coordinate system will become the point  $\text{Height} - Y_k$  and point *max* becomes  $Y_0$ . To find the coefficient coefficients g and h, solve the system of linear algebraic equations:

$$\begin{cases} g \times \text{min} + h = \text{Height} - Y_k \\ g \times \text{max} + h = Y_0 \end{cases}$$

Its solution will allow us to find the coefficients g and h.

$$\begin{cases} h = Y_0 - g \times \text{max} \\ g = (\text{Height} - Y_k - Y_0)/(\text{min} - \text{max}) \end{cases}$$

Before describing the plotting algorithm, let us clarify the formulas to calculate coefficients c, d, g and h. *Width* is the component width, including the left and right frames, and *Height* is the full component height, including the frame. If a form is used as a component, then the window title bar is included. To display a graph, however, the vertical and horizontal dimensions without the frames and titles is

needed. These dimensions are stored in the properties ClientWidth (width of the client area excluding the border) and ClientHeight (height of the client area, excluding the frame and title bar). Thus, it is more logical to compute the coefficients c, d, g and h as follows:

$$\begin{cases} d = X_0 - c \times a \\ c = (\text{ClientWidth} - X_k - X_0) / (b - a) \end{cases} \quad (10.1)$$

$$\begin{cases} h = Y_0 - g \times \text{max} \\ g = (\text{ClientHeight} - Y_k - Y_0) / (\text{min} - \text{max}) \end{cases} \quad (10.2)$$

The algorithm for plotting a graph on screen can be divided into the following steps:

- 1) Define number of segments N and incremental step in X.
- 2) Create arrays X, Y, and calculate the maximum (max) and minimum(min) value of Y.
- 3) Find coefficients c, d, g and h by formulas (10.1), (10.2).
- 4) Create arrays  $U_i = cX_i + d$ ,  $V_i = gY_i + h$ .
- 5) Connect adjacent points with straight lines using LineTo.
- 6) Draw coordinate system, grid lines and captions.

When plotting several continuous functions, instead of arrays Y and V, it is rational to use matrices of size k x n, where k is the number functions. The elements of each row of a matrix are coordinates of the corresponding graphs in paper (Y) and in component (V) coordinate systems.

The flowchart for the graph display algorithm is shown in Figure 10.3, and the flowchart for plotting the graphs of k continuous functions is shown in Figure 10.4.

Let us consider each flowchart step by step.

In Figure 10.3, blocks 2 and 3 show the first stage of the algorithm. Blocks 4 to 13 implement the second stage. Block 14 computes the coefficients in Step 3. In blocks 15 to 16, arrays U and V, containing the values of the component coordinate system (Step 4), are created. Blocks 17 to 19 are the display the graphs on screen, and block 20 implements Step 6.

For the second flowchart (Figure10.4) blocks 4 to 15 implement Step 2. Block 16 computes coefficients c, d, g and h. Blocks 17 to 20 create the arrays in Step 4. Blocks 21 to 24 display the graph, and block 25 plots the axes.

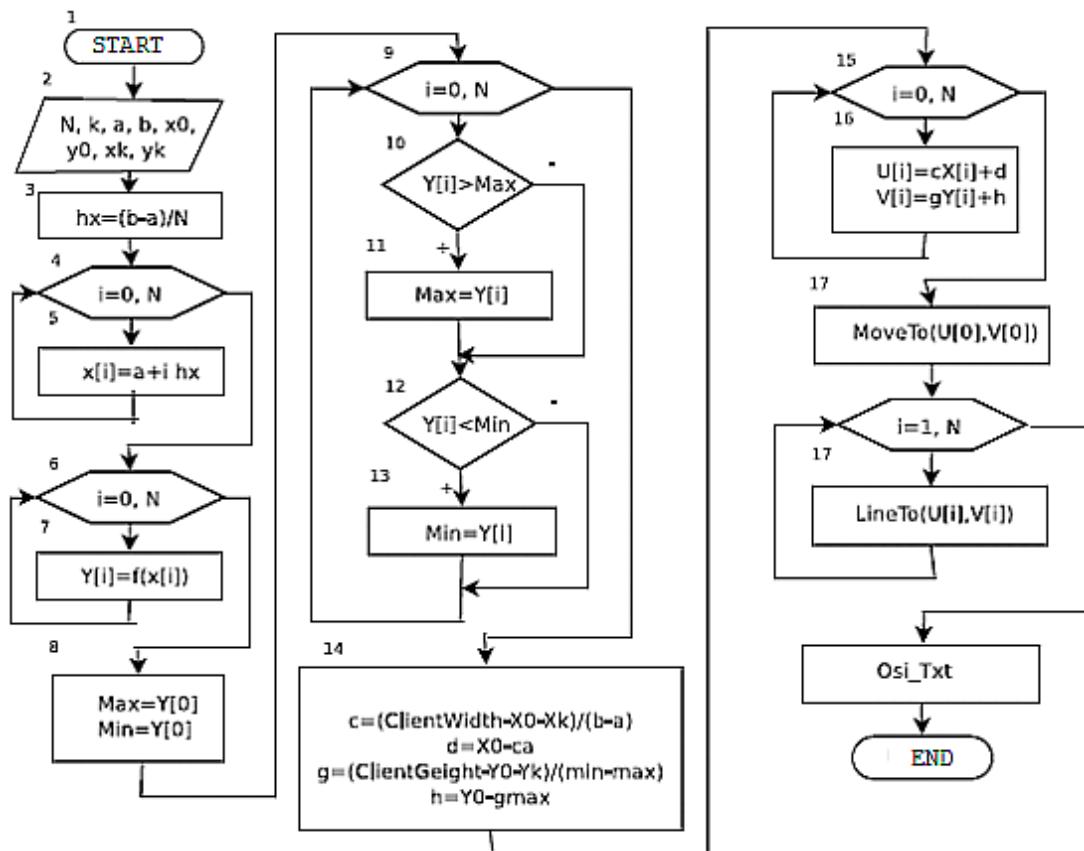


Figure 10.3: Flowchart for the function plotting algorithm

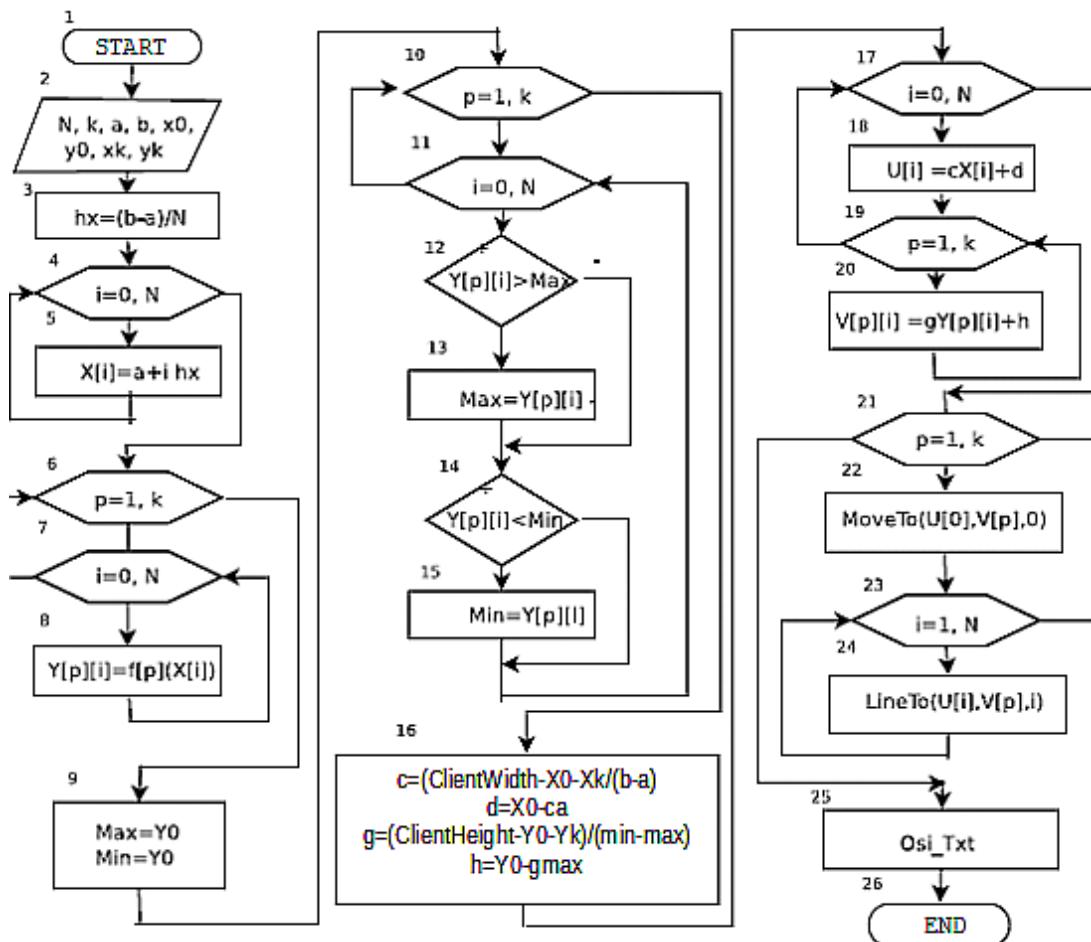
EXAMPLE 10.1. Plot the function  $f(x)$  in the interval  $[a, b]$ . The function is defined as follows:

$$\begin{aligned} f(x) &= \sin(x/2), \text{ if } x \leq 0 \\ f(x) &= \sqrt{(1+x)/3}, \text{ if } x > 0 \end{aligned}$$

Create a new project and make the height and width of the form large enough to display the graph. Set Width = 800 and Height = 700, for example. On the form, place a button and a TImage component.

The TImage1 object is a bitmap that will be used to display the chart after clicking Button1. The size of the raster image will be slightly smaller than the size of the form.

Set the form Caption property to "Graph of function".



*Figure 10.4: Flowchart of a multi-function graphing algorithm*

To avoid of the problem with redrawing the graph when resizing the form, prevent the resizing of the form and remove the Maximize and Minimize buttons. The form `BorderStyle` property defines the external appearance and behavior of the frame around the form window. To prevent the resizing of the form, set the value of its `BorderStyle` property to `bsSingle`, to specify a standard border around the form and prevent the resizing of the form. To remove the buttons to minimize and maximize the form, set the form's `BorderIcons.BiMaximize` and `BorderIcons.BiMinimize` properties to **false**.

Set the button Caption property to “Plot graph”.

After setting all the properties as described, the form should look like that shown in Figure 10.5.

Enter the interval a and b using dialog boxes when the program starts (in the form initialization method FormCreate). Below is the program listing for drawing a

graph, with comments:

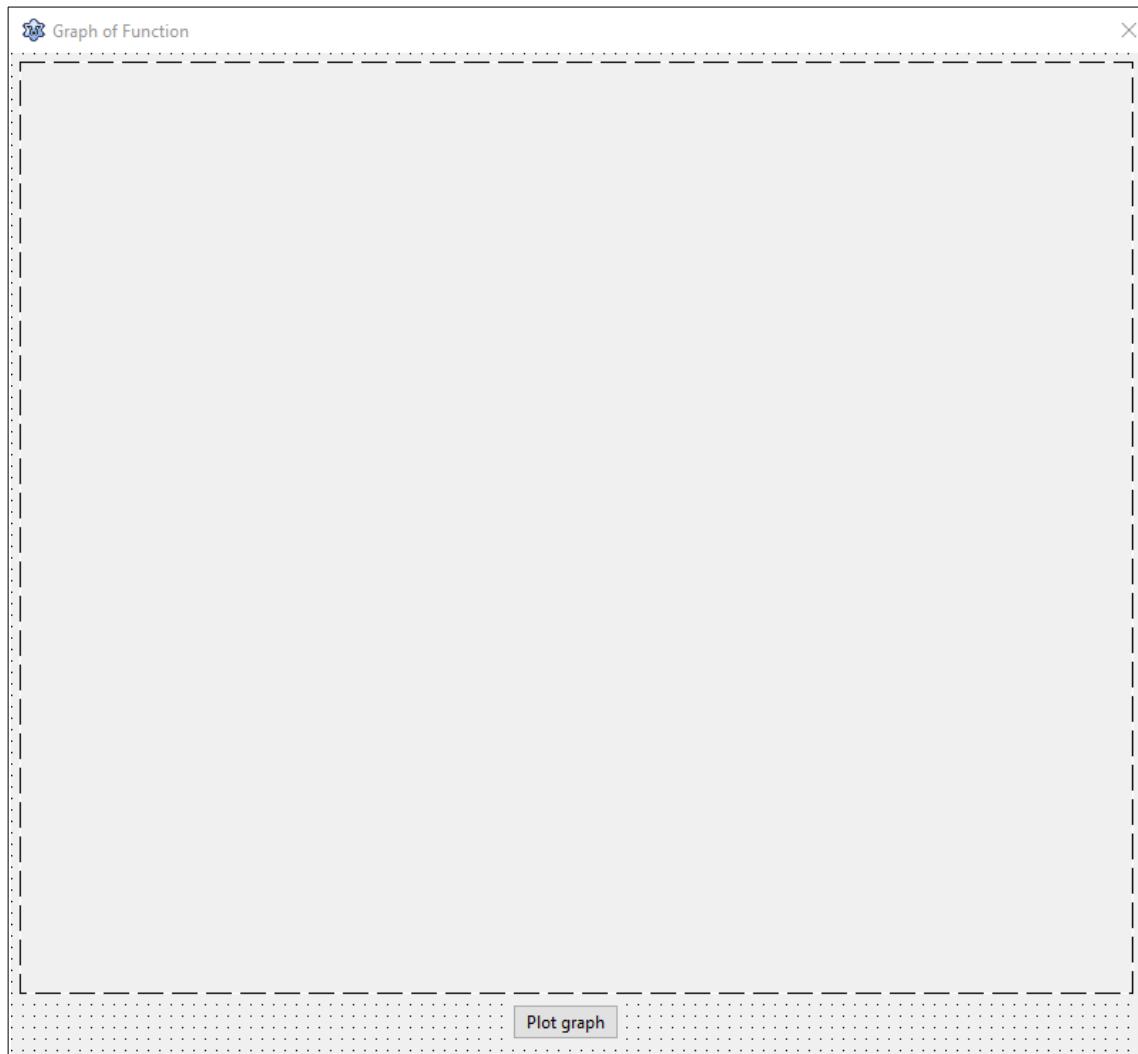


Figure 10.5: Form window after setting properties

```
unit Unit1;
{$mode objfpc} {$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, ExtCtrls, StdCtrls;
  // Forward declaration. Mathematical function to be plotted
  function f(x: real): real;
  // Forward declaration. Draw graph on component.
  procedure Graph(a, b: real);
type
  { TForm1 }
  TForm1 = class(TForm)
```

```
Button1: TButton;
Image1: TImage;
procedure Button1Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
private
  {private declarations}
public
  {public declarations}
end;

var
  Form1: TForm1;
  { x0,xk,y0,yk - left, right, top and bottom frame offsets
  X,Y - arrays for paper coordinates of chart points
  U,V - arrays for component coordinates of chart points
  N - number of points. }
  x0, y0, xk, yk, a, b: real;
  X, Y: array [0..1000] of real;
  U, V: array [0..1000] of integer;
  N: integer;
implementation

// Function to be displayed on the Image1 component
function f(x: real): real;
begin
  if x <= 0 then
    Result := sin(x/2)
  else
    Result := sqrt((1 + x)/3);
end;

// Draw the graph on the Image1 component
procedure Graph(a, b: real);
// Kx + 1 - number of grid lines perpendicular to X axis
// Ky + 1 - number of grid lines perpendicular to Y axis
const
  Kx = 5; Ky = 5;
var
  dx, dy, c, d, g, h, max, min: real;
  i, tempx, tempy: integer;
  s: string;
begin
  // Compute step change on X axis
  h := (b - a)/(N - 1);                                // p. 421, N=100
  // Populate arrays X and Y
  X[0] := a;
  Y[0] := f(x[0]);
  for i := 1 to N do
  begin
    X[i] := X[i-1] + h;
    Y[i] := f(X[i]);
  end;
```

```
// Find maximum and minimum values in the Y array
max := Y[0]; min := Y[0];
for i := 1 to N do
begin
  if Y[i] > max then max := Y[i];
  if Y[i] < min then min := Y[i];
end;
// Conversion factors from paper to component coordinates
c := (Form1.Image1.ClientWidth - x0 - xk)/(b - a);
d := x0 - c * X[0];
g := (Form1.Image1.ClientHeight - y0 - yk)/(min - max);
h := yk - g * max;
// Fill arrays with points in screen coordinate system
for i := 0 to N do
begin
  U[i] := trunc(c * X[i] + d);
  V[i] := trunc(g * Y[i] + h);
end;
Form1.Image1.Canvas.Pen.Color := clGray;
Form1.Image1.Canvas.Pen.Mode := pmNot;
// Draw function graph on the Image1 component
Form1.Image1.Canvas.MoveTo(U[0], V[0]);
Form1.Image1.Canvas.Pen.Width := 2;
Form1.Image1.Canvas.Pen.Color := clGreen;
for i := 1 to N do
  Form1.Image1.Canvas.LineTo(U[i], V[i]);
Form1.Image1.Canvas.Pen.Width := 1;
Form1.Image1.Canvas.Pen.Color := clBlack;
// Draw coordinate axes if they fall in chart area
Form1.Image1.Canvas.MoveTo(trunc(x0), trunc(h));
if (trunc(h) > yk) and (trunc(h) <
  trunc(Form1.Image1.ClientHeight - y0)) then
  Form1.Image1.Canvas.LineTo(trunc(Form1.Image1.ClientWidth -
    xk), trunc(h));
Form1.Image1.Canvas.MoveTo(trunc(d), trunc(yk));
if (trunc(d) > x0) and (trunc(d) <
  trunc(Form1.Image1.ClientWidth - xk)) then
  Form1.Image1.Canvas.LineTo(trunc(d),
    trunc(Form1.Image1.ClientHeight - y0));
{ Draw grid lines. Compute distance between grid lines
  in component system coordinates perpendicular to X axis. }
dx := (Form1.Image1.ClientWidth - x0 - xk) / Kx;
// Select the line type for the grid lines
for i := 0 to Kx do
begin
  // Draw first and last grid lines as normal solid line
  if (i = 0) or (i = Kx) then
    Form1.Image1.Canvas.Pen.Style := psSolid
  // Draw the rest with dotted lines
  else
    Form1.Image1.Canvas.Pen.Style := psDash;
```

```
// Draw a grid line perpendicular to X axis
Form1.Image1.Canvas.MoveTo(trunc(x0 + i*dx), trunc(yk));
Form1.Image1.Canvas.LineTo(trunc(x0 + i * dx),
  trunc(Form1.Image1.ClientHeight - y0));
end;
{ Compute distance between grid lines in component system
  coordinates perpendicular to Y axis. }
dy := (Form1.Image1.ClientHeight - y0 - yk) / Ky;
for i := 0 to Ky do
begin
  // Draw first and last grid lines with regular solid line
  if (i = 0) or (i = Ky) then
    Form1.Image1.Canvas.Pen.Style := psSolid
  // Draw the rest with dotted lines
  else
    Form1.Image1.Canvas.Pen.Style := psDash;
  // Draw a grid line perpendicular to Y axis
  Form1.Image1.Canvas.MoveTo(trunc(x0), trunc(yk + i * dy));
  Form1.Image1.Canvas.LineTo(trunc(Form1.Image1.ClientWidth -
    xk), trunc(yk + i * dy));
end;
Form1.Image1.Canvas.Pen.Style := psSolid;
{ Display labels under the axes. Define dx - the distance
  between the output under the OX axis with values }
dx := (b - a) / Kx;
tempy := trunc(Form1.Image1.ClientHeight - y0 + 10);
for i := 0 to Kx do
begin
  // Convert the output value to a string
  Str(a + i * dx: 5: 2, s);
  // x abscissa of value under X axis in component system
  tempx := trunc(x0 + i * (Form1.Image1.ClientWidth - x0 -
    xk)/Kx) - 10;
  // Display value under X axis
  Form1.Image1.Canvas.TextOut(tempx, tempy, s);
end;
if (trunc(d) > x0) and (trunc(d) < Form1.Image1.ClientWidth -
  xk) then
  Form1.Image1.Canvas.TextOut(trunc(d) - 5, tempy, '0');
// Define dy - distance between the values to left of Y axis.
dy := (max - min) / Ky;
tempx := 5;
for i := 0 to Ky do
begin
  // Convert the output value to a string
  Str(max - i * dy: 5: 2, s);
  // y ordinate of output to left of Y axis in component system
  tempy := trunc(yk - 5 + i * (Form1.Image1.ClientHeight - y0 -
    yk) / Ky);
  // Display value to left of Y axis
  Form1.Image1.Canvas.TextOut(tempx, tempy, s);
```

```
end;
if (trunc(h) > yk) and (trunc(h) < Form1.Image1.ClientHeight -
y0) then
  Form1.Image1.Canvas.TextOut(tempx + 10, trunc(h) -5, '0');
tempx := trunc(x0 + i * (Form1.Image1.ClientWidth - x0-xk)/2);
Form1.Image1.Canvas.TextOut(tempx, 10, 'Function graph');
end;

{ TForm1 }
procedure TForm1.FormCreate(Sender: TObject);
var
  s: string;
  code: integer;
begin
  N := 100;
  x0 := 40; xk := 40;
  y0 := 40; yk := 40;
  repeat
    s := InputBox ('Plot Continuous Function', 'Enter Left
      Limit:', '-10');
    val(s, a, code);
  until code = 0;
  repeat
    s := InputBox ('Plot Continuous Function', 'Enter Right
      Limit', '10');
    val(s, b, code);
  until code = 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Graph(a, b);
end;

end.
```

When starting the project, dialog boxes to enter the left (Figure 10.6) and right (Figure 10.7) limits of the interval will appear.

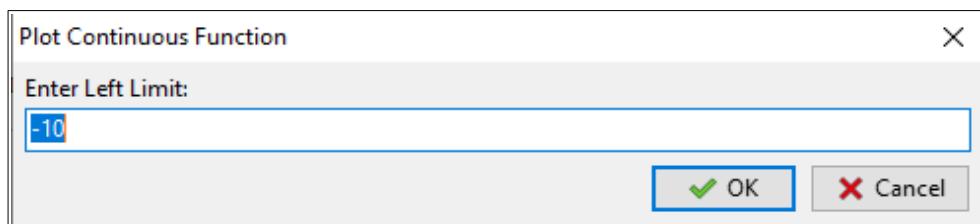


Figure 10.6: Left limit input

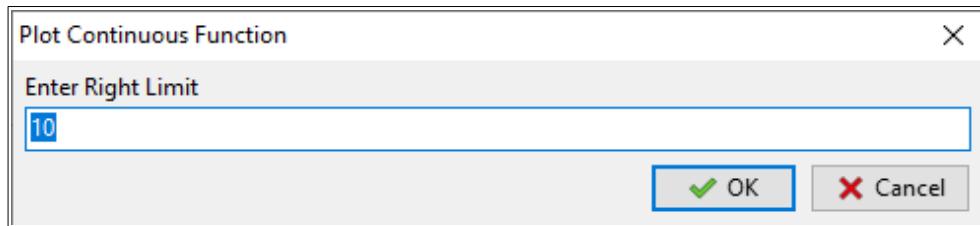


Figure 10.7: Right limit input

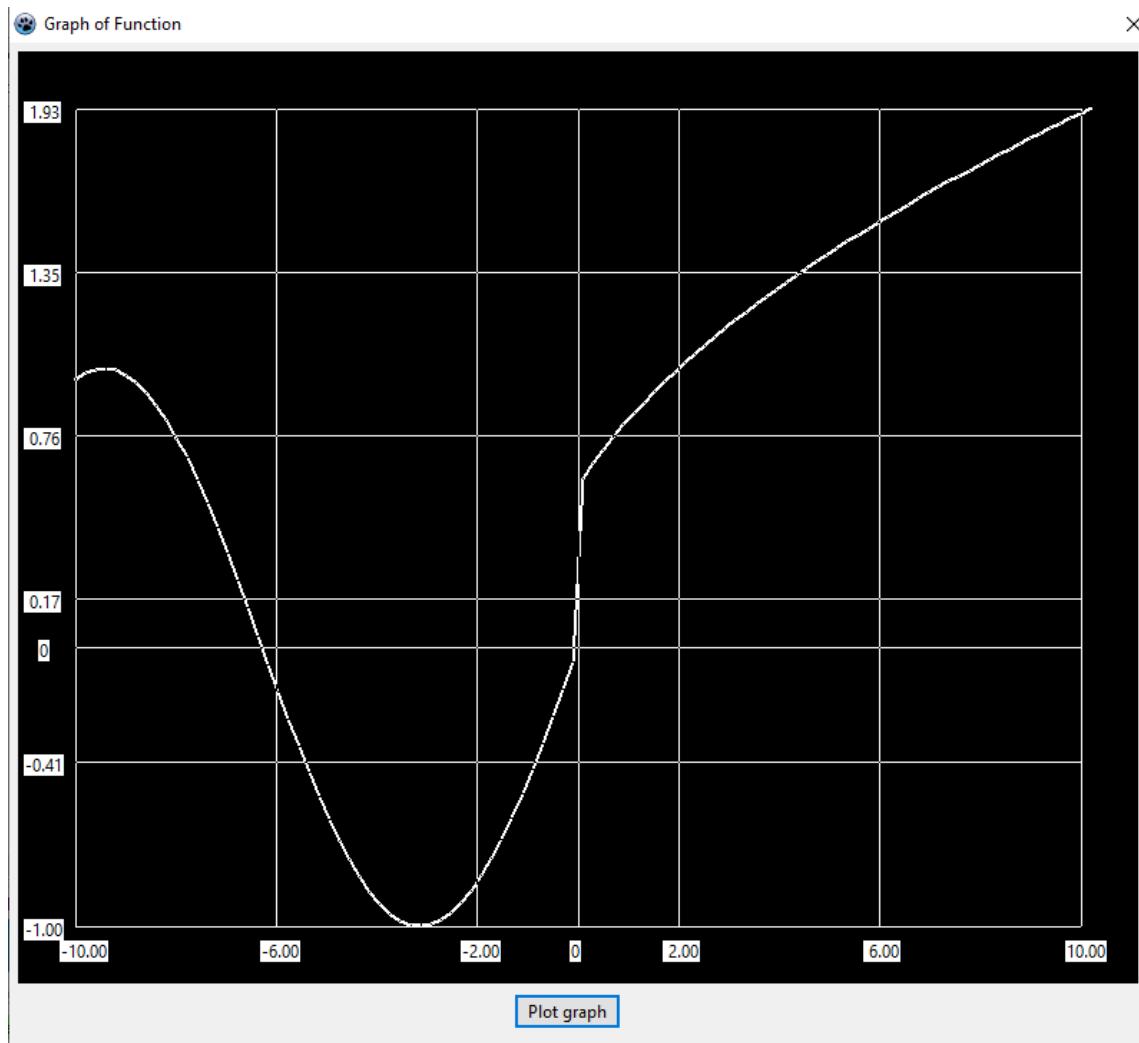


Figure 10.8: Window with a rendered graph

The form with the button will then appear. After clicking the button, the graph of the function will be drawn (Figure 10.8).

## 10.3 Exercises

Plot the function  $f(x)$  in the interval  $[a; b]$ . The function is given by the following equations:

- 1) 
$$\begin{aligned} f(x) &= \sin((x^3 + 2)/3) \text{ if } x \leq 5 \\ f(x) &= \sqrt[5]{(1+x)/3} , \quad \text{if } x > 5 \end{aligned}$$
- 2) 
$$\begin{aligned} f(x) &= (x^2 - x + 2)/x^3 , \quad \text{if } x \geq 1 \\ f(x) &= \sqrt[5]{e^x/2+x^2} , \quad \text{if } x < 1 \end{aligned}$$
- 3) 
$$\begin{aligned} f(x) &= \sin(\pi + x) , \quad \text{if } x \leq -1 \\ f(x) &= \sqrt[3]{e^{x+1}} , \quad \text{if } x > -1 \end{aligned}$$
- 4) 
$$\begin{aligned} f(x) &= \sin((x + 3.6)/x^3) , \quad \text{if } x < 0 \\ f(x) &= \sqrt[3]{1+x} , \quad \text{if } x \geq 0 \end{aligned}$$
- 5) 
$$\begin{aligned} f(x) &= \log((1 + x)/3) , \quad \text{if } x > 2 \\ f(x) &= \sqrt[3]{1+x^3} , \quad \text{if } x \leq 2 \end{aligned}$$
- 6) 
$$\begin{aligned} f(x) &= (x^3 + 3x^2)/3 , \quad \text{if } x \leq -2 \\ f(x) &= \sqrt[3]{\log(1.5+x^2)} , \quad \text{if } x > -2 \end{aligned}$$
- 7) 
$$\begin{aligned} f(x) &= \sqrt{\cos(x+2)^3}/3.5 , \quad \text{if } x > 2 \\ f(x) &= \sin(x+2^2) , \quad \text{if } x \leq 2 \end{aligned}$$
- 8) 
$$\begin{aligned} f(x) &= e^{x/4} , \quad \text{if } x > -2.5 \\ f(x) &= \sqrt[5]{x^2} , \quad \text{if } x \leq -2.5 \end{aligned}$$
- 9) 
$$\begin{aligned} f(x) &= \cos((x-6)/(x-3)) , \quad \text{if } x > 5 \\ f(x) &= \sqrt{1+x^4} , \quad \text{if } x \leq 5 \end{aligned}$$
- 10) 
$$\begin{aligned} f(x) &= \cos((x+2)/x^3) , \quad \text{if } x \geq 4 \\ f(x) &= \sqrt[3]{e^x+x^2} , \quad \text{if } x < 4 \end{aligned}$$
- 11) 
$$\begin{aligned} f(x) &= x^3 + 3x^2 , \quad \text{if } x \leq -1 \\ f(x) &= \sqrt[3]{\ln(10.5+x)} , \quad \text{if } x > -1 \end{aligned}$$
- 12) 
$$\begin{aligned} f(x) &= x^2 - x + 2 , \quad \text{if } x \geq -1 \\ f(x) &= \sqrt[5]{e^x+7} , \quad \text{if } x < -1 \end{aligned}$$
- 13) 
$$\begin{aligned} f(x) &= \cos((x^2 + 2)/x) , \quad \text{if } x > 3 \\ f(x) &= \sqrt{12+x^2} , \quad \text{if } x \leq 3 \end{aligned}$$

14)  $f(x) = \log((1+x^3)/2)$ , if  $x > 2.5$   
 $f(x) = \sqrt[3]{1+2x^2-x^3}$ , if  $x \leq 2.5$

15)  $f(x) = x^3 \sin(x)$ , if  $x > -3.5$   
 $f(x) = \sqrt[5]{|x/2|}$ , if  $x \leq -3.5$

16)  $f(x) = x^3 + 3(x+2)^2$ , if  $x \leq 3$   
 $f(x) = \sqrt[5]{\sin(10.5+x)^2}$ , if  $x > 3$

Construct graphs of functions  $f_1(x)$ ,  $f_2(x)$ ,  $f_3(x)$  in one coordinate system in the interval  $[a;b]$ . The functions are defined by the following dependencies:

17)  $f_1(x) = \cos\left(\frac{x-6}{x^2+3}\right)$   
 $f_2(x) = x^3 \cdot \sin(x)$   
 $f_3(x) = \sqrt{5x^3} \cdot \sin(x^2)$

18)  $f_1(x) = 1 + 2x^2 - x^3$   
 $f_2(x) = e^{x/2} + 7$   
 $f_3(x) = \sin(x/3)$

19)  $f_1(x) = \sqrt[3]{1+x^3}$   
 $f_2(x) = 14 + 2x^2 - 3x^3$   
 $f_3(x) = \cos(\sin(x))$

20)  $f_1(x) = \sin(x/3) + e^x$   
 $f_2(x) = \sqrt{7+2x^4}$   
 $f_3(x) = \sqrt[3]{(3+5x^2-x^3)^2}$

21)  $f_1(x) = \sqrt[5]{7-x^3}$   
 $f_2(x) = \cos(x/2 + \pi)$   
 $f_3(x) = \cos(\sin(x^2))$

22)  $f_1(x) = \sqrt[3]{\ln(12+x^3)}$   
 $f_2(x) = e^{x/5}$   
 $f_3(x) = \cos(x/\pi)$

23)  $f_1(x) = \cos\left(\frac{x+2}{\pi}\right)$   
 $f_2(x) = \sqrt[4]{1+x^6+2x^2}$   
 $f_3(x) = 3x^4 - 5x^2 + 7x - 2$

$$24) \quad f_1(x) = \sqrt[3]{(1+x^2-x^3)^2}$$
$$f_2(x) = 2 \cdot \sin(x/2 + \pi)$$
$$f_3(x) = 5x^2 - 3x^3$$

$$25) \quad f_1(x) = \frac{x}{2} + \sin(2 \cdot x \cdot \pi)$$
$$f_2(x) = \sqrt[3]{((1+x)(x^3-4))^2}$$
$$f_3(x) = e^{x/7} + 4$$

Endnotes:

---

- 1 The coordinate system for a component of class TForm, TImage, TPrinter, etc.
- 2  $X_0, X_k, Y_0, Y_k$  padding from the left, right, bottom and top component borders

This page deliberately left blank.

## In Place of a Conclusion

The last page of this book has been turned. Now what? The authors hope that this acquaintance with the Free Pascal language will be only the first step in learning programming. The reader's desire to improve something in the book: to rewrite the included programs, propose simpler and faster algorithms, write your own programs and units, would be the best way to thank the authors. If you, the reader, now have such a desire, then we would have accomplished our task of teaching you the basics of programming.

The next step in mastering programming would be developing your own algorithms and writing real working programs for various operation systems.

We hope this is only the first book dedicated to programming. We think that our book will be the beginning of a new series of books dedicated to programming in Linux. The next may be a book dedicated to the solution more complex problems in Free Pascal.

This page deliberately left blank.

## About the Authors

**Evgeniy Rostislavovich Alekseev** is an *Associate Professor* in the Department of Computational Mathematics and Programming at Donetsk National Technical University. He has been teaching computer science and programming at universities in Donetsk for over 15 years. He is the author of over 10 books on computer science and programming, published from 2001 to 2008, and more than 60 scientific and methodological papers. His areas of interest include programming, computational mathematics and open source software.

**Oksana Vitalievna Chesnokova** is a *Senior Lecturer* in the Department of Computational Mathematics and Programming at Donetsk National Technical University. She has been teaching computer science and programming at universities for over 10 years. She is the author of over 10 books on computer science and programming, published from 2001 to 2008 and more than 30 scientific and methodological papers. Her areas of interest include programming and computational mathematics.

**Tatyana Viktorovna Kucher** is an *Assistant* in the Department of Computational Mathematics and Programming at Donetsk National Technical University. She has been teaching computer science and programming at universities in Donetsk for over 10 years. Her areas of interest include programming and computational mathematics.

This page deliberately left blank.

## About the Translation

This book was translated from Russian by **Norman Gonsalves**, who was a foreign student at the Leningrad Polytechnic Institute (now [St Petersburg Polytechnic University](#)) from 1977 to 1984. He graduated with a Master of Science in Hydraulic Structures and has been a Registered Civil Engineer in the United States for over 25 years. His areas of interest include open source programming using Pascal and C++, engineering hydrology, hydraulics, structures and numerical analysis. He has a website with civil engineering resources at <https://www.plainwater.com>.

This translation preserves the original structure and content as much as possible, as a tribute to the original authors. Small sections about memory management in Chapter 5 and Graphics in Chapter 10 were updated, to reflect the current state of Free Pascal and Lazarus.

Valuable feedback and advice for improving the translated text came from several members of the Freepascal / Lazarus community. In particular, **Werner Pamler** (who uses the handle "wp" on the Lazarus forum) provided technical feedback, **Trevor Roydhouse** ("trev") spent many hours ferreting out typos, grammatical errors and other such gremlins that torment writers, and **Howard Page-Clark** ("howardpc") provided general tips on book formatting. A most heartfelt gratitude goes out to these gentlemen!

Any remaining typos, errors and "mistranslations" remain the fault of the translator ("norm" on the Lazarus forum), who would welcome any feedback on the matter.

Plans for future improvements will build on Werner's suggestion to add "modern Pascal features, such as streams, string-lists and lists, etc". Suggestions for expanding the content would be welcome. The book will keep its tutorial format and provide working code examples, as opposed to in-depth explanations of program features.

The book was prepared using LibreOffice 7.1, which proved to be very much up to the task of preparing the book, its table of contents and index. It also proved to be quite useful for checking the spelling and grammar, and then creating the PDF and epub versions. Thank you, LibreOffice team!

The current and expanded books will continue to use an open source license. Download the PDF version from <https://www.plainwater.com/pubs/FPLazarus.pdf> and the epub version from <https://www.plainwater.com/pubs/FPLazarus.epub>.

This page deliberately left blank.

## Literature

- 1) *Alekseev E.R., Chesnokova O.V.* Turbo Pascal 7.0. Moscow: NT Press, 2006. 320 p.
- 2) *Alekseev E.R.* Learning to Program in Microsoft Visual C++ and Turbo C++ Explorer. General editor *Chesnokova O. V.* M.: NT Press, 2007. 352 p.
- 3) *Bronshtein I.N., Semendyaev K.A.* Handbook of Mathematics for Engineering and University Students. Moscow: Nauka, 1981. 720 p.
- 4) *Faronov V.V.* Delphi. High Level Language Programming: A Textbook for Universities. SPb.: Piter, 2005. 640 p.
- 5) *Chesnokova OV* Delphi 2007. Algorithms and Programs. Learning to Program in Delphi 2007. General editor *Alekseeva E.R.M.* NT Press, 2008. 368 p.
- 6) Free Pascal - Advanced open source Pascal compiler for Pascal and Object Pascal. URL: <http://www.freepascal.org/> (Accessed 03.08.2009).
- 7) Free Pascal wiki (Accessed 03.11.2009). URL: [http://wiki.freepascal.org/Main\\_Page/ru](http://wiki.freepascal.org/Main_Page/ru).
- 8) Free Pascal.ru Information portal for developers on Free Pas-cal & Lazarus & MSE. URL: <http://www.freepascal.ru> (Access:03.11.2009).
- 9) GNU Pascal. URL: <http://www.gnu-pascal.de/gpc/h-index.html> (Accessed 08/03/2009).
- 10) GNU Pascal Wikipedia (Accessed 03.11.2009). URL: [http://ru.wikipedia.org/wiki/GNU\\_Pascal](http://ru.wikipedia.org/wiki/GNU_Pascal) .
- 11) Lazarus - News. URL: <http://www.lazarus.freepascal.org> (Accessed 03.11.2009).
- 12) Lazarus Wikipedia (Accessed 03.11.2009). URL: <http://ru.wikipedia.org/wiki/Lazarus> .

This page deliberately left blank.

## Alphabetical Index

abstract.....	377	close.....	330
actual parameter.....	142	CloseFile.....	299
address.....	172	code templates.....	26
address operator.....	64	Comments.....	52
algorithm.....	5, 79	compound statement.....	81
append.....	330	concatenation.....	339
Arc.....	404	console application.....	43
argument.....	142	const.....	172, 183
array.....	57, 181	constant.....	53
AssignFile.....	298	constant parameter.....	172
assignment statement.....	81	constructor.....	362
begin.....	81	continue.....	111
bitwise.....	62	control variable.....	105
BlockRead.....	321	copy.....	340
BlockWrite.....	320	date.....	68
Boolean.....	55	DateTime.....	55
break.....	111	delete.....	341
brush.....	399	dereferencing.....	65
bsSingle.....	412	destructor.....	363
canvas.....	399	dispose.....	212
case.....	98	div.....	62
character.....	53	do.....	105, 108
Charset.....	400	downto.....	108
CheckBox.....	157	dynamic array.....	181, 207
Checked.....	151	dynamic method.....	371
chr.....	380	Ellipse.....	404
class.....	359	else.....	82
ClientHeight.....	410	encapsulation.....	359, 367
ClientWidth.....	410	end.....	81

enumeration.....	56	integer.....	53
eof.....	299, 330	IntToStr.....	189
erase.....	299, 330	IOResult.....	331
error code.....	331	Keywords.....	52
exit.....	111	Lazarus.....	14
exp.....	65	Left shift.....	63
expression.....	60	LineTo.....	403
field.....	339	ListBox.....	341
file.....	59, 297	local variable.....	141
filepos.....	311, 330	loop.....	105
filesize.....	309	loop body.....	105
FloatToStr.....	42	mark.....	212
FloatToStrF.....	66	mathematical functions.....	65
flowchart.....	79	memo.....	303, 363
for.....	108	MessageDlg.....	101, 190
formal parameter.....	142	Method.....	359
formatted output.....	47	mod.....	62
frac.....	66, 232, 233	Mode.....	401
freemem.....	213	MoveTo.....	402
function.....	141	new.....	211
Geany.....	11, 13, 73	open array parameter.....	209
getmem.....	213	OpenDialog.....	302
global variable.....	141	operator.....	381
goto.....	110	operator precedence.....	60
halt.....	111	overload.....	372
Height.....	399	overloaded.....	381
high.....	207	override.....	372
identifiers.....	52	Pen.....	399
if.....	82	PenPos.....	403
Inheritance.....	359, 371	Pie.....	405
InputBox.....	127, 189	Pixels.....	399

pointer.....	60, 64, 211	Right Shift.....	63
PolyGon.....	405	RoundRect.....	404
PolyLine.....	403	seek.....	311, 330
Polymorphism.....	359, 372	selection structure.....	82
pos.....	340	selector variable.....	98
private.....	360	set.....	59
private member.....	360	SetLength.....	207
Procedural type.....	172	static array.....	181
procedure.....	143	static method.....	371
procedures.....	141	string.....	57, 67, 339
property.....	368	StrToFloat.....	189
protected.....	360	StrToInt.....	189
public.....	360	structured data type.....	57
public member.....	360	Style.....	400
published.....	360	subrange.....	56
RadioButton.....	151	subroutine.....	141
RadioGroup.....	157	TBrush.....	400
read.....	46, 300, 330	TCanvas.....	400
ReadCount.....	321	TColor.....	399
readIn.....	46, 330	text.....	297
real.....	54	text file.....	297, 330
record.....	58, 339	TextExtent.....	407
Rectangle.....	404	TextFile.....	297
recursion.....	168	TextHeight.....	408
reference parameter.....	142	TextOut.....	407
release.....	212	TextWidth.....	408
rename.....	299, 330	TFont.....	400
repeat.....	107, 201	then.....	82
reset.....	298, 330	time.....	68
Result.....	147	TObject.....	360, 371
rewrite.....	298, 330	TPen.....	400

trunc.....	66, 154, 232, 415	value parameter.....	142
truncate.....	311, 330	variable.....	52
TStringGrid.....	191	virtual method.....	371
typed file.....	297	Warnings.....	12
typed files.....	297	while.....	105
unit.....	175	Width.....	399
until.....	107	with.....	59, 346, 361
untyped binary file.....	320	write.....	46, 300, 330
untyped file.....	297	WriteCount.....	320
untyped files.....	297	writeln.....	46, 330
untyped pointer.....	211	Reference parameter.....	143
Val.....	102	function.....	146

## Math Index

Algorithm.....	5, 138	matrix transposition.....	249
base-p number.....	231	palindrome.....	149
bi-quadratic equation.....	95	parallelepiped.....	75
Cardano's method.....	92	perfect number.....	139, 147
common fraction.....	390	polygon class.....	368
complex number.....	89, 158, 176, 363	prime number.....	118, 147, 221, 276
complex roots.....	89	quadratic equation.....	88, 144
cosine rule.....	69	radians.....	45
cubic equation.....	91	semi-perimeter.....	69
digits.....	121, 149	sequence.....	127, 145
discriminant.....	88	sine rule.....	70
dynamic matrix.....	291	TCircle.....	378
Euclidean algorithm.....	111	TComplex.....	363, 383
factorial.....	169	TEqTriangle.....	373
Fibonacci number.....	171	TFigure.....	378
greatest common divisor.....	111	TMatrix.....	386
Heron's formula.....	344	TPolygon.....	368
Heron's theorem.....	69	TRectangle.....	379
identity matrix.....	263	TTriangle.....	373
matrix input.....	247		

*Educational edition*

Alt Linux Library Series

Alekseev Evgeny Rostislavovich  
Chesnokova Oksana Vitalievna  
Kucher Tatiana Viktorovna

**Free Pascal and Lazarus: A Programming Textbook**

Series Editor: K.A. Maslinsky  
Editor: V.M. Zhukov  
Cover design: V. Melamed  
Layout: A.V. Korotkov, K A. Maslinsky

Approved for printing on 31.03.10. Format 70×100/16.  
Headset Computer Modern. Offset printing. Offset paper.  
Cond. print l. 35.48. Uch.-ed. l. 23.04 Circulation 1000 copies. Order

Alt Linux LLC  
Mailing address: 119334, Moscow, 5th Donskoy Proezd, 15, Bldg. 6, (for Alt  
Linux LLC)  
Phone: (495) 662-38-83. E-mail: [sales@altlinux.ru](mailto:sales@altlinux.ru)  
<http://altlinux.ru>

DMK Press Publishing House  
Mailing address: 123007, Moscow, 1st Silikatny Proezd, 14  
E-mail: [books@dmk-press.ru](mailto:books@dmk-press.ru)  
<http://www.dmk-press.ru>