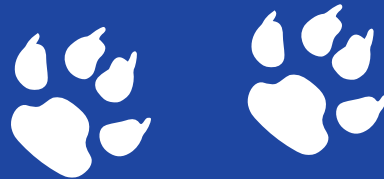
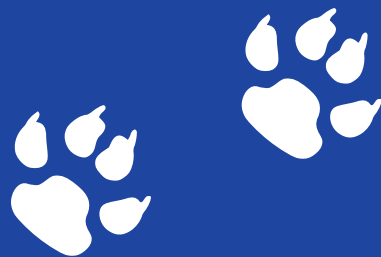


Howard Page-Clark



**Learn to Program
using Lazarus**



BLAISE PASCAL MAGAZINE



PUBLISHER

BLAISE PASCAL MAGAZINE

DELPHI, PRISM, LAZARUS AND
PASCAL RELATED LANGUAGES



WWW.BLAISEPASCAL.EU





Pascal

Learn to program using Lazarus

by Howard Page Clark



Developmental Editor: Detlef Overbeek
Production Editor: Detlef Overbeek
Proofreaders: Peter Bijlsma,

Correctors: Detlef Overbeek

Cover Designer: Detlef Overbeek

Copyright © 2013 by Blaise Pascal Magazine

Copyright © 2012.

All rights reserved by **Blaise Pascal Magazine**

Email: Office@blaisepascal.eu <http://www.blaisepascalmagazine.eu>

Blaise Pascal Magazine grants readers limited permission to reuse the code found in this publication so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product.

Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

This edition is registered by the Dutch Royal Library
Nederlandse Koninklijke Bibliotheek

ISBN: 978-94-90968-04-5

Blaise Pascal Magazine and the Blaise Pascal Magazine logo are either registered trademarks or trademarks of the Pro Pascal Foundation in the Netherlands and/or other countries.

TRADEMARKS: Blaise Pascal Magazine has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The authors and publisher have used their best efforts in producing this book, whose content is based on the latest software releases wherever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s).

The authors and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Pascal is enjoying something of a revival, and this is due in no small part to the quality and success of the Free Pascal Compiler, and the Lazarus project, which depends on it. That success, in turn, owes much to the skill and dedication of the core team members of the two projects.

This book is a tribute to the guys (so far there are no significant gals) starting with Florian Klaempfl who have carried the development of these projects to the significant phase each has reached.

One measure of the lasting value of open source projects is the quality of the programming communities that are drawn together through participation in such projects, and both Free Pascal and Lazarus shine in this respect.

I share the passion of a few in the programming community for good documentation, and particularly for material which makes programming more accessible to newcomers or those who (like me) don't have a formal education in information technology or computer science.

The Pascal language came to birth in the imagination of a Swiss software architect, a rare academic who values clarity and simplicity enough to make both aspects a hallmark of his designs. Pascal later became a widely available compiler (for CP/M, DOS and then Windows) largely through a Danish software architect. Borland incorporated that Danish compiler into an IDE, where Anders Hejlsberg and Chuck Jazdzewski among others oversaw adaptation of the language and support libraries for GUI programming on Windows.

Part of Free Pascal's achievement has been to liberate Pascal from its attachment principally to the Windows/.NET platform (which characterised the major years of US-based Object Pascal Delphi development, since the Kylix fork was not maintained). It was FPC (for a time) that enabled the Delphi IDE to generate code to run on iOS. A maturing Lazarus has also helped slowly to convince developers for MacOS, Linux and more recently Android and other platforms that Pascal can truly serve their needs.

I am grateful to Mattias Gaertner, who encouraged me to write this book during our first conversation, to Michaël van Canneyt, who saw an early draft of the manuscript and whose incisive comments helped enormously in shaping the organisation of the material, and to Detlef Overbeek, whose generous hospitality enabled me to see the obvious enthusiasm for Pascal development among programmers gathered in Utrecht. He is a friend who is ready to take risks in publishing.

Mistakes you find in the following pages are, of course, mine alone. I would be grateful for notification of any needed corrections, so that they can be included in any future printing. Please direct any comments to me via the publisher's website: www.blaisepascal.eu

Howard Page-Clark
Christmas 2012

CHAPTER 1	STARTING TO PROGRAM	
1.a	What to expect in this book	1
1.b	What is programming?	1
1.c	The worlds inside and outside the CPU	2
1.d	Not only digital data, but also digital code	3
1.e	Different computer languages at different levels	3
1.f	Pascal: a universal computer language	4
1.g	Lazarus: an IDE for Pascal	5
1.h	An open source approach to software	6
1.i	Getting help	7
1.j	Review Questions	8
CHAPTER 2	LAZARUS AND PASCAL	
2.a	The layout of the Lazarus IDE	9
2.b	Two different sorts of program	10
2.c	Writing, compiling and running firstproject	12
2.d	The structure of a Pascal program	14
2.e	Comments in Pascal code	16
2.f	Use of names in Pascal	16
2.g	Compiler directives	17
2.h	Review Questions	18
CHAPTER 3	TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS	
3.a	Pascal types	19
3.b	Ordinal types	20
3.c	The boolean type	21
3.d	Enumerated types	22
3.e	Type conversion	24
3.f	Typecasts	25
3.g	Variables	26
3.h	Initialised variables	26
3.i	Assignment: placing a value in a variable	27
3.j	Extended numerical assignment operators	28
3.k	Constants and literal values	28
3.l	A program example: simple_types	29
3.m	Typed constants	31
3.n	Pointers	33
3.o	Review Questions	35
CHAPTER 4	STRUCTURED TYPES	
4.a	Static arrays	36
4.b	Unnamed (anonymous) types	37
4.c	Pascal shortstrings	38
4.d	Dynamic arrays	39
4.e	Ansistrings	39
4.f	Records	41
4.g	The with . . . do statement	42
4.h	Set types	44
4.i	Binary files	45
4.j	Text files	47
4.k	Review Questions	48

CHAPTER 5	EXPRESSIONS AND OPERATORS	
5.a	Operators: forming Pascal expressions	49
5.b	Mathematical operators	49
5.c	Boolean operators: not, and, or, xor	50
5.d	Comparison (relational) operators	50
5.e	Bitwise (logical) operators	51
5.f	A program example: simple_expressions	51
5.g	Review Questions	52
CHAPTER 6	PASCAL STATEMENTS	
6.a	Conditional statement: if	54
6.b	Conditional statement: case of end	55
6.c	Looping statement: for to do	56
6.d	Looping statement: for downto do ; Break and Continue	57
6.e	Looping statement: for in do	57
6.f	Looping statement: while do	58
6.g	Looping statement: repeat until	58
6.h	Exception statements: raise, on, try	58
6.i	Review Exercises	61
CHAPTER 7	ROUTINES: FUNCTIONS AND PROCEDURES	
7.a	Routines and methods	62
7.b	Calling a routine	62
7.c	Passing data to a routine: parameters	63
7.d	Picking up the value returned from a function	63
7.e	Parameter classification: var, const, out	64
7.f	Default parameters	65
7.g	Declaring procedures and functions	65
7.h	A program example: function_procedure	66
7.i	The Exit() procedure	66
7.j	Review Questions	68
CHAPTER 8	CLASS: AN ELABORATE TYPE	
8.a	Generations of classes	69
8.b	Class data fields	70
8.c	Class memory management	71
8.d	Exercising simple class methods	73
8.e	Properties: special access to class data and events	75
8.f	Private, protected, public and published	80
8.g	Events	81
8.h	Event properties	83
8.i	Object oriented design	85
8.j	Review Exercises	85
CHAPTER 9	POLYMORPHISM	
9.a	Cross-platform polymorphism	86
9.b	Polymorphic methods in classes	87
9.c	Polymorphic graphic classes	89
9.d	Overloading	94
9.e	Default parameters	95
9.f	Review Questions	95

CHAPTER 10 UNITS, GUI PROGRAMS AND THE IDE

10.a	Unit structure and scope	96
10.b	The GUI program skeleton	99
10.c	Packages	101
10.d	Changing the program icon	101
10.e	The main form file	102
10.f	Editor Auto-completion	103
10.g	Using the Designer	106
10.h	The Object Inspector	107
10.i	OI Favorites and shortcuts	109
10.j	The OI Restricted page	110
10.k	The Component Palette	111
10.l	Finding a Palette component	111
10.m	Regular, DB and RTTI component types	112
10.n	Non-visual LCL and FCL support classes	113
10.o	Review Exercises	113

CHAPTER 11 DISPLAY CONTROLS

11.a	Display controls: TLabel	114
11.b	Display controls: exploring TLabel properties	115
11.c	Display controls: TStaticText	119
11.d	Display controls: TBevel and TDividerBevel	120
11.e	Display controls: TListBox	121
11.f	Display controls: TStatusBar	122
11.g	Display controls: further options	123
11.h	Review Questions	123

CHAPTER 12 GUI EDIT CONTROLS

12.a	Editing short phrases: TEdit and TLabelEdit	124
12.b	Editing or choosing short phrases: TComboBox	126
12.c	Editing integers and floating point numbers	126
12.d	Multiple-line editors	129
12.e	A component browser	130
12.f	Getting RTTI information for a component	135
12.g	Adding external tools to the IDE	138
12.h	Review Questions	140

CHAPTER 13 LAZARUS GUI PROJECTS

13.a	Planning a project	141
13.b	Creating a project task list	141
13.c	ToDo functionality	142
13.d	Version control	142
13.e	Test-driven software development	142
13.f	Naming	143
13.g	Project directory structure	143
13.h	A template project: SetDemo	144
13.i	Encapsulating set interaction within a new class	146
13.j	The setdemo UI	149
13.k	Review Questions	156

CHAPTER 14 COMPONENT CONTAINERS

14.a	Non-visual RTL classes	157
14.b	Creating new forms	158
14.c	Ownership and Parentage	160
14.d	Programmatic form creation	160
14.e	TGroupBox, TPanel	163
14.f	Resizable children	164
14.g	TFrame	166
14.h	TDatamodule	167
14.i	Review Exercises	167

CHAPTER 15 NON-VISUAL GUI SUPPORT CLASSES

15.a	TPersistent descendants	168
15.b	A chemical TCollection	169
15.c	The TStringList class	172
15.d	Sorting lines in a text file	173
15.e	Streams	176
15.f	TFileStream	177
15.g	TMemoryStream, TStringStream and Blowfish	180
15.h	Visualising a stream	182
15.i	Review Exercises	187

CHAPTER 16 FILES AND ERRORS

16.a	File access in Pascal	188
16.b	Run-time errors and exceptions	188
16.c	An example of string error-handling	189
16.d	File name encoding issues	192
16.e	User-directed file searching and naming – the Dialogs Palette page	193
16.f	Discriminating between text and binary files	196

CHAPTER 17 WORKING WITHIN KNOWN LIMITS

17.a	Using recursion to evaluate factorials	200
17.b	Catching a specific exception	203
17.c	Permutations	204
17.d	Time-consuming routines	205
17.e	Generating anagrams	206
17.f	Review Questions	210

CHAPTER 18 ALGORITHMS AND UNIT TESTS

18.a	Collaboration	211
18.b	The algorithm – a specific plan	211
18.c	A parsing algorithm	212
18.d	Testing the ParseToWords function: the FPCUnit Test	215
18.e	Example tests	218
18.f	Test-driven development	220
18.g	Optimising debugged routines	220
18.h	Profiling and compiler optimisation	220
18.i	Review Questions	221

CHAPTER 19 DEBUGGING TECHNIQUES

19.a	Preventing bugs	222
19.b	Unit tests	223
19.c	Paying attention to compiler messages	223
19.d	Using Assertions	225
19.e	Modularising functionality	227
19.f	Code Observer	228
19.g	Refactoring	232
19.h	Watching variable values	234
19.i	The <code>{\$DEFINE DEBUG}</code> compiler directive	234
19.j	Console debug functions	237
19.k	Program interruption	238
19.l	Logging debug output to a file	239
19.m	The debugserver tool	240
19.n	Getting the compiler to catch bugs	241
19.o	The heaptrc unit	242
19.p	The gdb debugger	245

CHAPTER 20 FURTHER RESOURCES

20.a	Books about Pascal and Lazarus	248
------	--------------------------------	-----

Chapter 1 STARTING TO PROGRAM

1.a What to expect in this book

This is a tutorial guide rather than a reference book. When you've worked your way through the following chapters you should be able to understand how to use Lazarus to tackle increasingly complex programming challenges, and have learned not only how to use the Lazarus IDE and the Pascal language, but how to go about finding out what you still need to learn.

Learning requires engagement with new ideas, often with a new culture, and in the case of learning to program you may encounter several new concepts simultaneously. People vary in their ability to accomplish several tasks at once. Some say that females are better than males in this regard. Whatever the truth of that, readers of this book will be learning on several levels at once:

- learning the fundamental principles of good programming
- learning a new language (*Object Pascal*)
- learning to use complex development software (*Lazarus*)
- learning technical English vocabulary (*if you are not a native English speaker*)

Working through this book will also exercise skills such as typing on a computer keyboard; searching online and offline help resources; thinking through how you would approach finding solutions to the problems posed; remembering the names of programming routines you will often need to use; and so on. Perhaps you are not a complete beginner, and consequently have a head start on some of this.

The approach here is fairly fast-paced, presenting new ideas and topics in every chapter, together with hands-on exercises accompanying the tutorial text. Chapter 2 is an introduction to the Pascal language and the Lazarus IDE. From Chapter 3 onwards new aspects of the Pascal language are introduced one by one. As the topics progress, so various Lazarus features are also introduced. There is a shift from a focus on learning Pascal to learning about Lazarus and its support libraries from Chapter 10 onwards.

In addition to the **Table of Contents** at the beginning of the book the PDF version offers an **Index** to help you quickly locate key topics. However, this is not designed as a reference book – *Lazarus (and particularly Free Pascal)* already has a great deal of good documentation, and a growing library of help files.

The help is far from perfect, but perhaps you will soon be able to contribute to its improvement. Most chapters conclude with a few **Review Questions** or **Review Exercises** designed to help you reflect on what you have been reading, and to apply the principles outlined in that chapter. The **Questions** or **Exercises** section can, of course, be ignored by readers who would rather press on to the next chapter. However pondering applications of the principles given in the previous pages is a good way to help them stay with you after the book is closed.

As a beginning programmer you can look forward to satisfaction and opportunities to be really creative, as well as experiencing frustration and bewilderment or confusion. You will need to have determination, because you will write code that seems correct but does not have the desired effect, and you'll puzzle over what could possibly be wrong with it. You will probably find there are things you cannot figure out, or which seem supremely difficult to understand, particularly at the first reading.

1.b What is programming?

Computer programming is the art of making a computer do exactly what you want without errors. Computers have as their brain one or more central processing units (*CPUs*) containing numerous miniature electrical circuits in which electrons may or may not be moving, and these circuits can indicate to the world beyond the CPU whether those particular electrons are moving or not.

Chapter 1 STARTING TO PROGRAM

This simple difference between a circuit that is switched on (*where electrons move*), and the same circuit when it is switched off (*when no electrons move*) is the basis of the incredible complexity we know as today's software. The computer's circuitry, CPUs, disks, memory, screen and so on are known as **hardware**.

The programs that run on this hardware are termed **software**. You can touch hardware, just as you can touch the human body with its hands and face. Software is more like the plans, thoughts and words that a human can communicate – the ideas, commands and requests that cannot be physically touched.

At the lowest level then, a computer can speak a language with two words: On and Off (*or Yes and No, or if we use numbers rather than words to represent the electrical state, 1 and 0*). Even the most advanced multi-processor supercomputers that execute billions of instructions each second have this fundamental **binary** design. It is extraordinary in fact how much complexity has evolved from such a simple building block as a circuit that can communicate just one of two possible states: On or Off. (*Engineers may object that the detected difference is between a High signal and a Low signal rather than strictly a distinction between On and Off – however oversimplifications of this sort are par for a beginners' course*).

Software complexity is possible because CPU hardware designers and manufacturers have found ways to pack literally millions of these circuits into very small 'chips', and have found ways to prevent the heat caused by millions of electron movements from melting the chip. The complexity is also possible because software engineers have devised ways to make streams of on/off signals (*which is all a CPU can process*) assume meaningful patterns that correspond to the world outside a CPU chip. Understanding the world inside the CPU is termed working at a low level. Understanding the world outside the CPU is termed working at a higher level.

1.c The worlds inside and outside the CPU

Computer processors are designed to handle streams of on/off signals (*or 1/0 signals*).

Information theory calls a state which can be either On or Off (*either 1 or 0*) a binary state, and the knowledge of whether that state is 1 or 0 is a piece of information called a **bit**. Information processed as bits is known as binary information, or **digital** information since the smallest element of such information, the bit, can be represented by the digits 1 or 0.

There is an unambiguous separation between the two states 1 and 0 (*On and Off*). Each state is unique and discrete. The CPU does not ever mistake one for the other (*unless it is faulty*). There is no half-On or half-Off.

Information in the wider world is sometimes clearly digital, particularly numerical information. Numbers other than 0 and 1 can be represented as **combinations** of 0s and 1s. This is familiar to people who have studied binary arithmetic. It is possible to count using only the digits 0 and 1 – rather tedious, but possible. In this severely digit-limited binary system it is obvious which digits represent zero and one. But by combining 0 and 1 in logical patterns whose meaning everyone can agree on, we can also represent the numbers beyond 0 and 1 (*even though we deliberately discard the digits 2, 3, 4 etc. which we use in decimal arithmetic, and stick to just 0 and 1*). The first few numbers in the binary system are written like this

Decimal	0	1	2	3	4	5	6	7	8
Binary	0	1	10	11	100	101	110	111	1000

Patterns of 0s and 1s such as this, then, are the basic language of the computer, which is why computers are often termed digital devices. We can assign any meaning we want to these patterns of 0s and 1s. If we regard a pattern of binary digits as **integers**, then the pattern 1000 means the number 8. If, on the other hand we regard the pattern of binary digits as letter **characters**, then one standard meaning of the pattern 1001011 is the capital letter 'K'.

Chapter 1 STARTING TO PROGRAM

These are examples of treating a binary digit pattern as **data**, that is, as information that represents an agreed arithmetic or alphabetic entity. It is straightforward to agree on a mapping that relates such patterns of bits to numbers, or to characters, each of which is unique and discrete.

You may know the very old quip: “There are just 10 sorts of people in the world – those who understand binary and those who don’t”. The state of being either-0-or-1 is the minimum amount of information we can know. Streams of 0s and 1s are termed **bit patterns**, and when we combine eight bits of information together that larger amount of information is termed a **byte**. You are probably familiar with kilobytes (*kB*) and megabytes (*MB*) and gigabytes (*GB*) used as measurement units for hard disk and USB memory stick capacity, or memory (*RAM*) capacity installed in a computer or a mobile. These quantify the amounts of information that can be stored on that device, and let you know how big, say, a particular downloaded file is.

With some effort, it has proved possible to represent even analogue data too (*which on first appearance is far from being digital*) by unique bit patterns. Consider the colours of the rainbow, for instance, that in differing intensities and brush strokes make up the work of an Impressionist painter. Works of art such as Monet's Water Lilies can be represented digitally as a collection of millions of points of colour, each colour point itself being represented as one of thousands of possible subtle shades of colour.

The result of such **digitisation** is that even complex colour images in 2-D or 3-D can be translated into a form that computers can process. A corresponding process allows music to be translated into mp3 and other formats that computers can store and process. In this way information that seemed rather unpromising as grist for the computer mill has proved to be just that. Provided it can be digitised, the world outside the computer can be represented fairly faithfully in the internal world of a computer's CPU.

1.d Not only digital data, but also digital code

It is also possible, and immensely useful for controlling computers, to assign **instructional** meanings to binary patterns. One pattern might mean “add the next two numbers together”, another pattern might mean “send the result of the addition to the output”. This means we have a digital way not only of feeding data to a CPU, but of getting it to **process** the data in some way.

Computers are devices designed to process information. They do this by receiving everything they need to know about as streams of bit patterns which represent either the (*digitised*) **data to be processed**, or **instructions about how to process** the data, or both. This is most unfortunate for humans, who find patterns of bits not only tedious and error-prone to write, but meaningless, because we cannot distinguish at a glance the meaning of 1000110101011 from 1000110101001. Indeed, we might not even notice there was a difference between these two patterns (*whereas most people immediately notice the difference between the two character patterns ‘dame’ and ‘damn’*). CPUs are designed specifically to excel at processing such bit patterns, whether these bit patterns are machine code (*instructions*) or data to be processed (*information*).

Computer programming means clearly distinguishing what is **data** and what is **code** (*instructions*) and using the computer's short term **memory** and long term storage to keep track of the massive amounts of data flowing in and out of the CPU, while sending the correct instructions (*code*) so that processed data then flows to the appropriate place (*perhaps to the display to be viewed, or to an earpiece to be heard, or to a plotter to be printed*).

1.e Different computer languages at different levels

The bit patterns (*machine code*) that instruct the CPU what to do with the data it receives are not understood by humans. There is a computer language called **assembler** that maps human-readable symbolic instructions to each machine code instruction.

Chapter 1 STARTING TO PROGRAM

Learning and using assembler means working at a very low level, dealing with the minutiae of small on-chip memory areas called registers, and paying careful attention to the size of all data and instructions at the level of individual bits and bytes. This is error-prone, and difficult to master.

The following is a tiny example of the sort of instructions assembly language uses:

```
movl (%esp), %eax
addl $5, %eax
```

Assembler programming has the further disadvantage that it is processor-specific. Instructions that work on one manufacturer's processor may not work or even have an equivalent on another make of processor. Just as a car mechanic knows that a VW alternator will not fit as a replacement for a faulty alternator on an Audi. Little about hardware is standardised to the extent that different manufacturers' parts are interchangeable or software-compatible. The situation is complicated further by increasingly rapid technical advances which render the entire spectrum of IT devices a constantly changing marketplace.

Hardware that was standard yesterday may be almost completely obsolete in just a short time from now. From the point of view of software developers what is needed is a single language that is applicable to various different platforms, whatever the underlying CPU, peripherals and operating system.

What most programmers need is a **higher level language** (*higher than assembler*) that uses meaningful instructional words, whose syntax is closer to the pattern of familiar instructions such as "draw the phrase Free Pascal Compiler on the screen". A language which would shield the programmer from having to deal with low-level, processor-specific interaction with the CPU's registers and internal clock cycles.

High level languages are designed to be close to natural languages.

They combine mathematical symbols with natural language specifically chosen for communicating with a computer's CPU and operating system. High level languages provide a convenient, comprehensible interface between the programmer and the computer, but they need to interface to the CPU as well. This is achieved by a process called **compiling**, in which the meaning of the high level language is translated into the low level instructions that are the language the CPU can process.

1.f Pascal: a universal computer language

Lazarus is a development tool that uses the **Pascal** programming language. Niklaus Wirth designed this computer language to encourage good programming practice, publishing his work in 1970. Pascal uses a small vocabulary of English words which are all self-descriptive. Pascal was extended in the 1980s to be object-oriented, and so the Pascal dialect used in Lazarus is termed **Object Pascal**.

Object-oriented means that Lazarus uses code constructs called objects – Lazarus actually calls them **classes** – which model real-world situations more satisfactorily than the original plain procedural Pascal, which was designed to teach programming well. (*Pascal was not designed originally for commercial-quality software production, for which it has come to be used since – but that is the power and potential of a well-designed computer language.*)

The Oxford English Dictionary lists over 500,000 English word forms, and there are about the same number of additional unlisted English technical and scientific terms. By contrast, Object Pascal (*as used in Lazarus and Free Pascal*) has just over 70 **reserved words** (*or keywords*), and you will use less than half of those regularly. In terms of vocabulary, then, learning to use Pascal is far easier than learning to speak English! Here is a selection of the most commonly used words in the Pascal language (*notice that one or two like const and var are abbreviations of longer English words*):

Chapter 1 STARTING TO PROGRAM

and	file	or	then
array	for	procedure	to
begin	function	program	type
class	if	property	unit
const	implementation	record	uses
do	interface	repeat	var
else	not	set	while
end	of	string	

Rather than diving into details about the Pascal language at this point, we shall look now more closely at Lazarus, the main tool you will be using to write Pascal code. Details of Pascal syntax and usage follow in later chapters.

1.g Lazarus: an IDE for Pascal

High level computer programming languages, including Pascal, allow the programmer to use a specialised language (*nearly always based on English*) to give instructions to the computer's CPU. The CPU does not understand Pascal, so the Pascal code has to be changed into digital electronic signals that the CPU can accept. This translation process (*producing machine code that a CPU can accept*) is called **compilation** and **assembly**. Sections of compiled, assembled code then require **linking** (*that is, joining together appropriately*) to produce an executable program file that the operating system can run. Lazarus uses the Free Pascal Compiler (*FPC*), which on some platforms has its own internal linker, to transform Pascal code into an executable program. Lazarus and similar programming tools such as Eclipse and Visual Studio are termed **Integrated Development Environments** or **IDEs** because they integrate in one toolbox the many functions that are needed for you to produce an executable program starting with only an idea, and no written code. The Lazarus IDE provides:

- an exceptionally versatile **text editor** with numerous features designed for rapidly writing, modifying and navigating your program source code
- a **visual editor** for designing your program's user interface (*UI*)
- three major **libraries** of ready-to-run code routines and components: the RTL, FCL and LCL
- tools to compile, assemble and link your program's modules into a finished executable
- tools to analyse, test, document and debug the code you write

The Lazarus IDE includes many specialised ancillary components including tools which aid in:

- setting initial values for individual program components such as typeface, width, colour
- debugging programs
- writing specialised forms such as dialog windows
- preparing programs for internationalisation and translation of included texts
- converting Delphi projects into Lazarus/Free Pascal projects
- customising the formatting of source code
- creating data dictionaries and writing boilerplate code for object-database mapping
- code analysis and review
- testing code modules
- documentation of routines, libraries and programs

The following illustration depicts the process which starts with writing high level Pascal code in the Lazarus IDE right through to creating and running the final executable application.

Chapter 1 STARTING TO PROGRAM

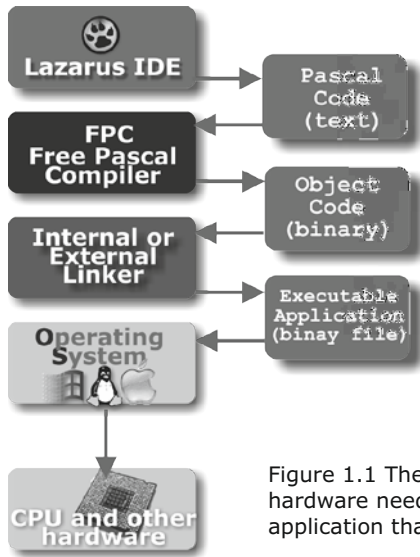


Figure 1.1 The software and hardware needed to develop an application that runs on a computer

1.h An open source approach to software

Since 1998 the phrase 'open source' has been officially sanctioned to designate that category of software which publishes its source in the public domain. This radical approach to software development has flourished alongside the long-established closed source, commercially licensed software typical of the products of many of the IT companies that mushroomed in the latter part of twentieth century.

Most open source software is free (*no payment is charged*), and all of the source code is publicly available. This means that it can be continuously corrected and improved by a large community of interested people, none of whom may be employees whose wages depend on their work for the project. Knowledgeable users who find **bugs** can submit solutions (*known as patches*) which, if accepted, improve the software for all subsequent users, often within a matter of days following the discovery of the bug. Patches might also be improvements or enhancements adding new functionality, not related to bugs at all.

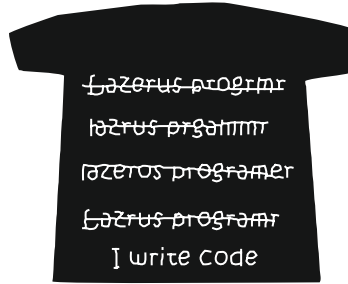
Open source software usually has many, many versions. Often a new 'bleeding edge' version is available daily. This is true of both the Free Pascal Compiler (FPC) and the Lazarus IDE that are principal topics in this book. However, periodically a 'release' version is tagged and numbered. This is a version that has received extensive testing in the field (*including use in commercial programs developed using Lazarus/FPC*), and for which bug fixes have been found and applied for known bugs identified up to the point of the release.

Screenshots in this book are based on **Lazarus 1.0** which was released on 29th August, 2012 and a bug-fix release Lazarus 1.0.4 dated 3rd December 2012. It uses the **Free Pascal Compiler version 2.6.0** which was released on 1st January 2012. Version 2.6.2 was about to be released as this book went to press. If you are using a later Lazarus (*or Free Pascal*) release the code examples should work identically, but you may find some screenshots do not match your version exactly. Windows is the most popular Lazarus platform (*if SourceForge download statistics are a reliable guide*), and this book's screenshots are based on a 32-bit Windows 7 Lazarus installation. If you are working on Ubuntu or other Linux, or Mac OS X, or on one of the various other Lazarus-supported platforms you will of course find this book's screenshots somewhat different from the appearance of Lazarus running on your operating system with your current theme. However Lazarus is truly cross-platform. Both the IDE itself and the code examples that come with it compile on all supported platforms, both 32-bit and 64-bit.

Chapter 1 STARTING TO PROGRAM

1.1 Getting help

Experienced software developers often encounter coding problems. How much more likely is this for beginners! But your choice of **Lazarus** to learn programming is a good one for several reasons.



Lazarus uses Pascal, one of the best languages for a beginning programmer to learn. Moreover the better open source projects attract and consolidate a community around them. Not only are there the core developers (*in the case of Lazarus this is currently a team of about twenty regular contributors*) but scores of other experienced people who are involved in the project as users, testers, bug reporters, patch submitters, translators, documenters, contributors to the forums and mailing lists.

To get help in the Source Editor of the Lazarus IDE you place the cursor somewhere in the word you want help on and press [F1]. If that is unhelpful you can also consult the wiki:

<http://wiki.lazarus.freepascal.org>

You can browse the Lazarus and Free Pascal forums and mailing lists where you can pose questions and ask for (or offer) assistance on specific problems. This is the main Lazarus forum link:

<http://www.lazarus.freepascal.org/index.php?action=forum>

Of course you might encounter a lunatic on one of the forums (as you might anywhere on the web), but generally the Lazarus and Free Pascal online community is a source of help, support and encouragement. You can also download and consult the documentation. Indeed you would be daft to overlook such a vast and generally well-organised resource. Most of the information in this book is based on it.

<http://sourceforge.net/projects/lazarus/files/Lazarus%20Documentation>

Or you can search one of the mailing list archives such as:

<http://lists.lazarus.freepascal.org/pipermail/lazarus>

There are several free online tutorials available which offer alternatives to the material presented here. One of them can be found here: **http://wiki.lazarus.freepascal.org/Lazarus_Tutorial**

An online book about Pascal and Lazarus by Motaz Azeem can be found here:

<http://code.sd/startprog/index.html>

Chapter 20 entitled Further Resources at the end of this book lists a number of published Pascal/Lazarus resources, and authors to look out for. If you have trouble installing Lazarus look in the Installation section of the main forum here:

<http://www.lazarus.freepascal.org/index.php?action=forum>

Chapter 1 STARTING TO PROGRAM

Often googling for a related Delphi topic will throw up useful leads, since Lazarus is largely Delphi-compatible. However you will find that Delphi-based information on the internet is often rather Windows-centric, because (*except for the short-lived Kylix venture*) until recently Delphi was itself Windows-only.

There are a large number of example projects included with Lazarus in the examples folder, located immediately below the main Lazarus folder in your installation. The IDE provides a tool to explore this local collection of example programs and code routines. You will find it via **Tools | Example Projects...**

There is an excellent and growing repository of example code you can both browse and use in your own projects (*and perhaps contribute to yourself*) located at:
<http://sourceforge.net/projects/lazarus-ccr/>

Last, but not least, you have **immediate access to every line of source code**. This may not be supremely helpful to a complete beginner, but you will be surprised how quickly you will come to value the immense benefit of being able to look up exactly how some routine is coded in the source in order to better understand its operation.

Another advantage of open source projects is that new (*free*) resources are contributed all the time, so when you read this you will almost certainly find that further helpful material has been added that was unavailable when the above list and Chapter 20 were written. For example the Brook framework released by Silvio Clécio

<http://github.com/silvioprogram>

just as this book went to press opens up web programming in a wonderful new way for Pascal programmers.

1.j Review Questions

1. What would you say are the main differences between low level and high level languages? Can you give an example of each?
2. What is the main 'brain' of a computer called?
3. Who invented the Pascal language, and in which year did he publish his work?
4. What units are used to measure the size of items of information?
5. How would you write the (decimal) number 11 in binary arithmetic?
6. What distinguishes software from hardware?
7. Why do you need an IDE to develop software?



Chapter 2 LAZARUS AND PASCAL

2.a The layout of the Lazarus IDE

This book assumes you have downloaded and installed Lazarus and its help system and companion debugger, gdb. If you're having problems getting Lazarus up and running on your system, the following wiki link may be helpful:

http://wiki.lazarus.freepascal.org/Installing_Lazarus

The book also assumes you have a basic familiarity with your operating system, and can save and manage files and folders, navigate standard menus, choose from drop-down lists and use other common operating system widgets to save preferred settings and so on.

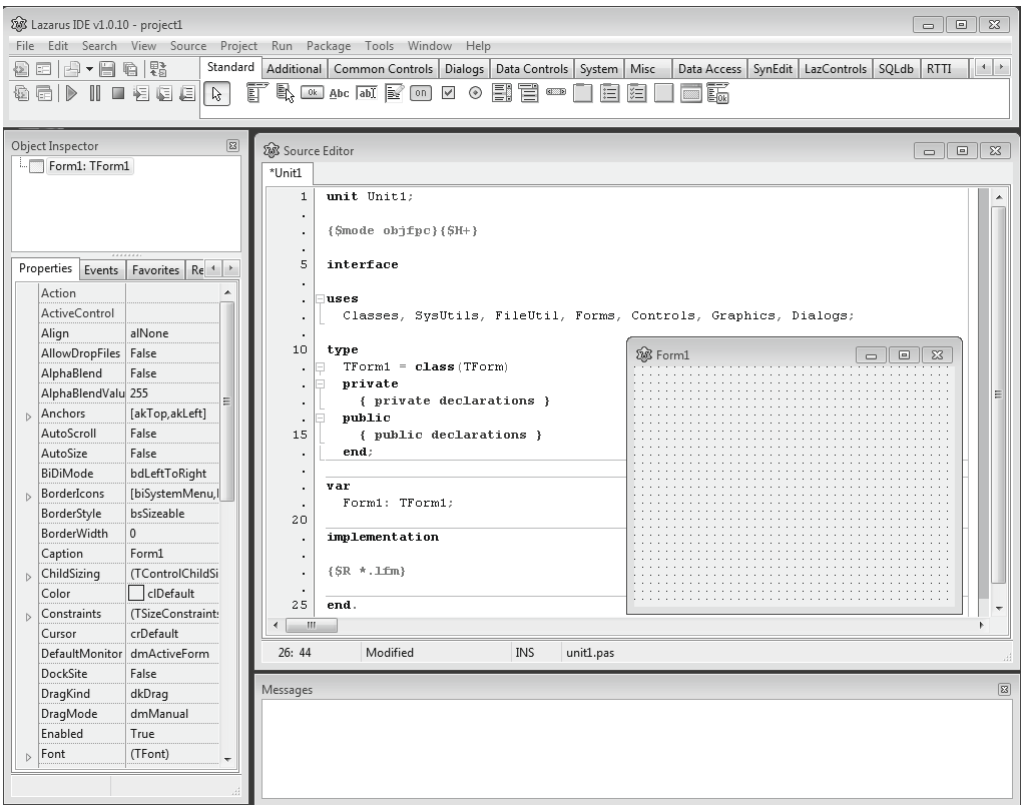


Figure 2.1 The Lazarus IDE when first installed, with various sections labelled

Lazarus always loads an empty project when started (*though you can configure it to load the last edited project if you prefer*). A Lazarus **project** is a collection of files and settings all related to a particular application. Projects contain the information needed to build that application (*program*). This book deals both with **console** and with **GUI** applications – the latter being windowed apps that users interact with via mouse and keyboard, that utilise the components (*edits, labels, file dialogs etc.*) provided by the operating system. A project is stored in its own folder (*directory*).

Note: Lazarus will not prevent you from storing more than one project in a single folder, but to do so is asking for trouble.

Chapter 2 LAZARUS AND PASCAL

The default project name Lazarus assigns to its opening project is `project1`. This is unsatisfactory as a long term name, but will do for now. The project's name is helpfully displayed in the **title bar** at the top of the main Lazarus window, and if you save the project (*until you first save a project it only exists in memory*) the main program file will be saved using the project name with an `.lpr` extension. Actions in Lazarus can often be accomplished in one of four alternative ways. For instance, to save a project you can:

1. Navigate the **main menu** to choose your desired action – in this case **Project | Save Project** (or the menu item *Project | Save Project as...*).
2. Click a **toolbutton** in the toolbar area, in this case the floppy-disk-icon toolbutton either the 4th or 5th from the left.
3. Press the **shortcut key** combination `[Ctrl][S]` or `[Shift][Ctrl][S]`.
4. **Right-click** to open a context menu with selectable options (*not applicable in this case of saving a project*). In addition to the title bar, main menu and toolbar area already mentioned, the IDE includes at its first default showing (see Figure 2.1) the following:

- **Component Palette**
- **Source Editor**
- **Form Designer** (or *Form Editor*)
- **Object Inspector**
- **Messages Window**

The Form Designer may not be visible. It can be hidden by the Source Editor. The convenient toggle key `[F12]` brings the Designer forward above the Source Editor, or places it behind the Editor on the next `[F12]` key press. This is a quicker alternative to using the main menu option **View | Toggle Form/Unit View**.

Note: The shortcut key combinations mentioned in this book are the defaults installed with Lazarus, which you may want to customise (see **Tools | Options... | Editor, Key Mappings**).

To save undue verbosity this book will refer to the Component **Palette** as the Palette, the Source **Editor** as the Editor, the Form Designer as the **Designer**, the Object Inspector as the **OI**, and the Messages Window as **Messages**. There are a multitude of further dialogs and windows in the IDE which you will use in due course, but the above-mentioned items are (*by default*) always on display, and usually needed in developing any project. Any window can be closed by clicking on the **[X]** Close icon in its title bar; or by using the `[Ctrl][F4]` shortcut; or by right-clicking on its title bar and choosing **Close**. Windows (*whether hidden or not*) are listed in the **Window** menu, and if you cannot find a particular window you can locate it in the **Window** menu list and click on its name to focus it in the IDE. You can use the **View** menu to open many IDE windows if the one you want is not yet listed under the **Window** menu. **Note:** Versions of Lazarus after 1.0 may have introduced bug-free support for window docking. If so, you may prefer to move to a docked window layout.

2.b Two different sorts of program

Lazarus enables you to produce two rather different styles of program, either **console** or **GUI** programs. Console programs run in the Linux Terminal (*or equivalent*) or in a console window under Windows. Console programs have a text-only interface, and typically you pass data to a console program when it runs initially via parameters given on a command-line such as

```
dir /w/s/p      (Windows)
man -t ascii   (Linux)
```

Chapter 2 LAZARUS AND PASCAL

Console programs tend to be designed to do one particular task well, and are often lightning fast in execution because of their text-based, lightweight user interface. They were the only kinds of programs possible when Wirth first designed Pascal.

The later sections of this book will focus mainly on GUI programs which are nearly always much larger in executable size (*from approximately 10MB upwards*). The size increase over console programs arises mainly because of the much more complicated user interface code (graphical rather than character-based) that GUI programs link in, and the event-based paradigm GUI programs require; whereas most console programs are purely procedural.

The following seven chapters teach Pascal using console program examples since that is slightly easier for beginners. There is a focus on GUI programs from Chapter 10 onwards. Lazarus loads an empty GUI project by default when first started. However, it is simple enough to discard this and choose to develop a console project. Select **Project | New Project...** from the main menu, which opens the *Create a new project* dialog (see Figure 2.2).

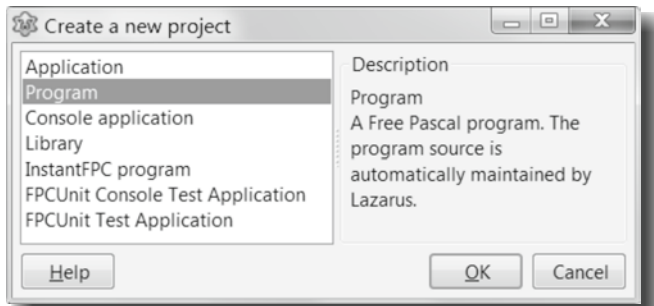


Figure 2.2 Choosing the type of Lazarus project to create

If you click on Program, not on Application or Console application (*which looks promising but is too complex for us at this stage*) and then click on [OK], Lazarus will display a skeleton Pascal console program in the Editor. The code will look like this:

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes

  { you can add units after this };

begin
end.
```

You'll see that Lazarus has named this program `Project1`, and given it the filename `project1.lpr` which is shown both in the tab at the top of the Editor, and in the statusbar at the base of the Editor. However, the program only exists in memory at this point. In order to save the program to disk create a new folder called Chapter 2 in an area where you have file write permission, and choose **Project | Save Project As...** In the save-dialog give the project the name `firstproject.lpi` (*.lpi will be stipulated as the default file extension*).

Note: It is recommended to name **all** programming files in lowercase. This avoids later cross-platform problems arising when you share or publish your code, and will help prevent you loading and working on `firstProject.lpi` later thinking it is the same project (*if you are currently working under Linux or working on a Mac*). On Windows, of course, it would indeed be the very same project. If you use your system's file browser to examine the contents of the Chapter 2 folder you'll see that although you only appeared to save one file (`firstproject.lpi`) Lazarus has in fact saved three files:

Chapter 2 LAZARUS AND PASCAL

```
firstproject.lpi
firstproject.lpr
firstproject.lps
```

The `.lpr` file is the Pascal program file, the text file whose contents are displayed in the Editor. The initial “l” of the three filename extensions stands for Lazarus

- “pr” stands for “**program**” (*it could stand for “project”* - in Lazarus the two terms effectively mean the same for simple projects). If you come from a Delphi background you will find Lazarus projects more limited than Delphi projects in that they cannot contain sub-projects, and there are no project groups.
- “pi” stands for “**program information**”
- “ps” stands for “**project settings**”

The `.lpi` and `.lps` files are quietly created by Lazarus for every new project. They are XML text-format files used internally by Lazarus to store information about this named project, and to store details about your particular setup so that when you next edit this project Lazarus can restore your programming environment as it was, with IDE windows in the positions they were, various files you may have had opened now reopened at the same place etc.

Note: The `firstproject.lpi` file is maintained by Lazarus specifically for this project. If your operating system supports association of file extensions with program executables, you can double-click on an `lpi` file using your system file browser to start Lazarus and load that particular project. The same project can also be loaded by double-clicking on the project's `lpr` file.

2.c Writing, compiling and running firstproject

Whatever sort of project you create in Lazarus (*console or GUI*) the IDE starts you off with some basic Pascal code. The code Lazarus writes for you is a **skeleton** – a template, if you like, provided with the intention that you will flesh it out with further valid Pascal statements so that the program is complete. Then, when it is compiled, it will do something useful. Accordingly, let us add a few lines of Pascal code to the provided skeleton (*which is already a valid Pascal program, save for the fact that it “does nothing”*). In the Editor add lines between **begin** and **end** so that your program looks like this:

```
program firstproject;

{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  {you can add units after this };

begin
  writeln('A Free Pascal program');
  {$IFDEF WINDOWS}
  readln;
  {$ENDIF}
end.
```


Chapter 2 LAZARUS AND PASCAL

Note that the statusbar at the bottom of the Editor now shows the full path to firstproject, and there is an asterisk (*) by the project name in the tab at the top of the Editor, and Modified appears in the Editor statusbar. The asterisk has the same meaning as the Modified notification. It indicates that you have typed text to change the contents of the Editor, and **not yet saved** what you have written. The act of compiling this code will automatically save your changes. Once saved the asterisk and Modified will disappear (*until you edit the code further*). Choose **Run | Run** (or click the green arrow toolbutton 3rd from the left on the second row, or press the shortcut key [F9]) to compile and run firstproject. You will see several messages flash by in the Messages Window, and then a console window (*with the full path to the firstproject executable as its title*) will appear in the foreground. The last message left in the Messages window: Project "firstproject" successfully built confirms what you see in the Terminal or console window (see Figure 2.3).

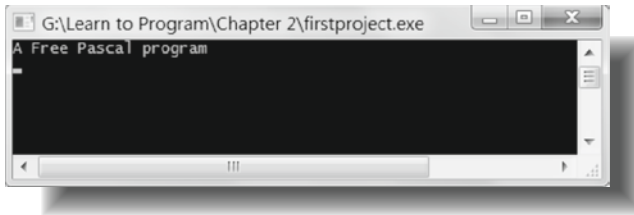
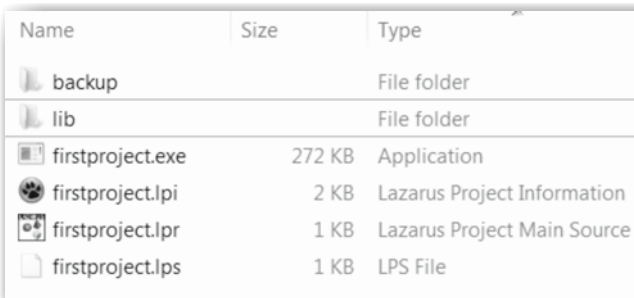


Figure 2.3 The firstproject executable running

Note: If you have ticked the Show compile dialog checkbox on the Environment, Files page of the **Tools | Options...** dialog you will see a Compile Project dialog which tracks compilation progress and gives a statistical summary of number of lines compiled, errors encountered and suchlike.

The Terminal or console window displays the text `A Free Pascal program`, which corresponds to the Pascal statement you inserted: `writeln('A Free Pascal program');` In Windows you can close the program by pressing the [Enter] key, or clicking the [X] close icon to return to the Lazarus IDE.

A screenshot of a Windows file explorer window showing the contents of a directory. The window has a table with three columns: Name, Size, and Type. The table lists several files and folders.

Name	Size	Type
backup		File folder
lib		File folder
firstproject.exe	272 KB	Application
firstproject.lpi	2 KB	Lazarus Project Information
firstproject.lpr	1 KB	Lazarus Project Main Source
firstproject.lps	1 KB	LPS File

Figure 2.4 Firstproject's files and folders

Chapter 2 LAZARUS AND PASCAL

If you check the contents of your project folder you will find a new file named `firstproject` (*Linux*) or `firstproject.exe` (*Windows*) which will be the largest of the four `firstproject.*` files. You will also find two new sub-folders named `lib` and `back-up`.

These are created in every project folder for you by Lazarus (see *Figure 2.4*).

The `back-up` folder, as you would expect, contains backup copies of the three text-format `firstproject` files. They are named by default with an additional `.bak` extension

(e.g. `firstproject.lpr` becomes `firstproject.lpr.bak`). You can customise this naming scheme if the default does not suit you via **Tools | Options... | Environment, Back-up**.

You can turn off the backup feature altogether if you wish.

The `lib` folder contains a sub-folder named according to the platform you are working on, and this sub-folder contains intermediate binary files resulting from the compilation and linking process. You can safely ignore the contents of the `lib` sub-folder. The Pascal language shields us from needing to be concerned with the low-level processes that require the production of these files. If any of them is deleted accidentally, they will be regenerated as necessary, and all this happens largely unseen in the background.

2.d The structure of a Pascal program

A Pascal program has a strict grammar and syntax, and the rules are few and simple.

- A Pascal program begins with a **program heading**. This consists of the reserved word `program` followed by the name you give the program followed by a semicolon separator.
- An (*optional*) **uses clause**. This consists of the reserved word `uses` followed by a comma-separated list of unit names ending with a semicolon separator. A Pascal program is built from source code modules called **units**. Each unit is stored in its own file and compiled separately. The compiled units are linked to create the final executable program.
- A program block containing **statements** which begins with the reserved word `begin` and ends with the reserved word `end`. The program block is closed with a final dot (*period, or full stop*).

To summarise, a Pascal program has this structure:

```
program ProgramName;
uses unitA, unitB, ..., unitN;
begin
    {statements go here}
end.
```

If you look at the `firstproject.lpr` program file in the Editor you can see it takes the form above, but it includes **comments**, which make it look rather more complex than the simple skeleton code outline presented above. Here is that code again:

```
program firstproject;

{$mode objfpc}{$H+}
uses
    {$IFDEF UNIX}{$IFDEF UseCThreads}
    cthreads,
    {$ENDIF}{$ENDIF}
    Classes
    { you can add units after this };

begin
    writeln('A Free Pascal program');
    {$IFDEF WINDOWS}
    readln;
    {$ENDIF}
end.
```

Chapter 2 LAZARUS AND PASCAL

The remaining parts of the code are Pascal statements. Bold indicates a reserved word, or keyword, i.e. a word with a special meaning in Pascal such as **begin**, or text strings. Normal typeface indicates a Pascal name (*the above code contains a program name, two unit names and the names of two Pascal routines declared in the hidden system unit*).

Removing the comments strips the program to its essentials, when it looks like this:

```
program firstproject;  
  
uses  
  cthreads, Classes;  
  
begin  
  WriteLn('A Free Pascal program');  
  ReadLn;  
end.
```

Lazarus has helpfully included two units (cthreads and Classes) assuming we would need them. In fact, for this simple first program neither unit is needed (*unnecessary units are ignored by the compiler, so it does not cause an error to leave them in the source*). The only unit needed is the system unit which is included by default in every Lazarus/FPC program (*and does not need to be – indeed should not be – declared explicitly*). The system unit contains the WriteLn and ReadLn routines we use here. So the bare bones of our working firstproject program is in fact this code:

```
program firstproject;  
  
begin  
  WriteLn('A Free Pascal program');  
  ReadLn;  
end.
```

We have a program heading, no uses clause (*this is not always required*), and a program block with two statements calling predefined Pascal code routines found in the implicit system unit. The first, WriteLn(' . . . '), displays a string of text on the console, and the second, ReadLn, waits for an [Enter] keypress.

The second statement is not required on Unix systems. However it is required on a Windows console, which behaves differently from the Unix Terminal. Windows will scroll screen text away and close the console unless explicitly instructed to wait by using a ReadLn statement. Lazarus is a cross-platform development environment. It minimises or hides the differences between supported platforms as much as possible, but cannot shield us from the more glaring differences. So we sometimes have to wrap Pascal code in a **conditional compiler directive** to take account of platform differences, which is what the `{ $IFDEF }` directives do here. Notice how each statement ends with a semicolon to separate it from the next statement, and how the final end is followed by a dot (*not a semicolon*) to indicate the end of all Pascal code.

Note also how the code is laid out with **indentation** to set off the statements between the **begin** and **end**, and a **blank line** between the program heading and the main code block.

You are free to style your code in a way that is comfortable for you. However, the recommended style is to follow the example of the FPC/Lazarus code itself.

The idea is to make the code as easy to read and understand as possible. Extra white space to set statements off or to emphasise program logic goes a long way to enhance the readability of code, and costs nothing in terms of the final program output (*since the compiler discards all tabs, spaces and newline characters used for formatting*). It is perfectly valid Pascal to write the above program like this:

```
program firstproject;begin writeln('A Free Pascal program');readln;end.
```

Chapter 2 LAZARUS AND PASCAL

The compiler can read and process this just as easily (*perhaps even a microsecond faster*). However, it would be insane to write code like that. Since Pascal allows you to set out code in a way that enhances its readability, take advantage of this facility, and exploit it to the full. Everyone who reads your code (*including yourself returning to your code much later*) will be grateful to you.

2.e Comments in Pascal code

Some parts of code are **comments** or **literal text**. Comments in the above code include `{ $mode objfpc }` and `{ you can add units after this }`. The literal text (*which in Pascal code is always surrounded by single quote marks*) is 'A Free Pascal program'.

As far as the compiler is concerned, comments come in two flavours:

- **compiler directives** starting with `{ $`, which Lazarus colours red.
- **plain comments** starting with `{` or `(*` or `//`, which Lazarus colours blue.

The Editor colours comments appropriately on-the-fly as you type them. (*You can customise these colours in the IDE Options dialog, accessed via **Tools | Options...***).

{Plain comments} have to be surrounded by `{ }` curly brackets. They are not Pascal code, and are completely ignored by the compiler. They are inserted only to be helpful to the programmer, perhaps to document a feature of the code, or (*as here, inserted by Lazarus itself*) to remind you where further units you might need are to be placed: `{ you can add units after this }`. A comment has to be enclosed within curly brackets to signify that it is a comment (*not code*), and comments can span more than one line if they are too long to fit on a single line. There is an alternative C-style one-line comment which opens with a double forward-slash

// this comment cannot be longer than one line

Such comments close with a carriage return or newline character (*the normally hidden character(s) that are inserted when you press [Enter] in the Editor to start a new line of code*).

`{ $ Compiler directives }` are likewise comments enclosed by curly brackets. However they all start with a `$` symbol to indicate their nature as compiler instructions. These are comments that are **not** ignored by the compiler. However, they are not Pascal code. They don't get compiled into code that ends up in your program, but they do influence how the compiler treats your code. So, when required, they are essential.

Note: There is a third type of comment, the `{ToDo comment item}`. The compiler ignores these comments (*as it does all comments that are not compiler directives*). However the IDE knows about `{ToDo comments}` you insert in your code. There is more about this in Chapter 13, Section c.

2.f Use of names in Pascal

A Pascal **program** is built from **units** which are self-contained code modules containing ancillary code used by the main program. Rather than stuffing all needed code into a single huge file, Pascal program code is **modular** in design. This aids both in the reuse and sharing of code among projects and between programmers, and it also speeds compilation (*since pre-compiled units that have not been changed do not need to be compiled again before linking*).

You are free to give your program (*application*) and the new units you create for it whatever names you like, subject to certain **naming restrictions** and **recommendations**:

- Pascal names can contain only alphanumeric characters or the underscore `_` character (no punctuation symbols, or characters such as `%`, `#`, etc. are allowed in names).
- The first character of a name must be an underscore or alphabetical (*it cannot be a digit, so `project1` is OK, whereas `1stProject` is not allowed*).
- Although it is not a Pascal restriction it is recommended to use **only lowercase** names for the names of Pascal files (`.pas` and `.lpr` and `.inc` filenames).

Chapter 2 LAZARUS AND PASCAL

This is because of the risk of ambiguity or confusion on case-sensitive operating systems where `unit1.pas` is a different file from `Unit1.pas`. This is an unavoidable issue when a case-insensitive language (*Pascal*) meets a case-sensitive OS (*all Unixes*).

- The best Pascal names are self-descriptive and self-documenting. The names encountered above (`Classes` and `firstproject`) are good examples of descriptive Pascal names. If you choose the names you introduce carefully, you may not need comments or other documentation for simple programs. You will also be very thankful for the care you took over naming when you come to re-read your code months later – it will then be far easier for you to understand it yourself!
- A common convention is to use camel-casing for names within code (*not for filenames*). For example: `LoopCounter`, `DialogBackgroundColor`. A readable alternative is to use underscores (`Loop_counter`, `Dialog_background_color`) since spaces are not allowed within names.
- Pascal names are not case-sensitive. The identifiers `classes`, `Classes`, `cLaSsEs`, `CLASSES`, `ClAsSeS` and so on are all **identical** in Pascal. If you come from a C-derived language such as C# or Java this will take some getting used to, and possibly be a great relief, saving you from (*some*) hard-to-spot spelling mistake bugs.
- When Lazarus writes code for you it supplies default names such as `project1` and `unit1` and `Form1` because some name has to be given. These names should be changed to something more meaningful as soon as possible. No one names their children: `child1`, `child2`, and `child3`. Likewise you should give your variables and code routines descriptive and meaningful names. This also makes the code more interesting to read, just as a play about `Character1`, `Character2` and `Character3` seems very dull compared to one about `Romeo`, `Juliet` and `Friar Laurence`. A program with variables of type `ex1` named `a`, `b` and `c` gives few clues about its nature compared to a program with variables of type `TCar` named `Volvo`, `Opel` and `Skoda`.
- It does not matter in the slightest whether you call a new variable `aColor` or `aColour`. However because Lazarus is largely Delphi-compatible, and Delphi was developed in American (US) English, many existing names in the LCL and FCL follow Delphi and use American rather than British spelling. Note that though you could name a variable `aGirl`, you could **not** name it `einMädchen` since accented characters are not allowed, but `uneFille` would be OK, just as `unGarçon` would not be.

The program heading (`program firstproject;`) names the program and the name chosen has to be a valid Pascal name.

2.g Compiler directives

Fortunately beginners rarely need to worry about compiler directives (*the comments beginning with `{` that we encountered earlier*). This is because the really essential directives are inserted automatically by Lazarus in the template code written by the IDE. So you can safely learn about Pascal and write many programs without needing to learn any more about compiler directives, and without needing to use one yourself.

Nevertheless, it is worth looking briefly at the directives Lazarus has inserted (*and the one we inserted ourselves*), so you can be familiar with the general principles governing the need for them, and not regard them as mysterious or arcane.

Compiler directives are mostly either:

- general instructions required to cover every part of your project's code.
- specific instructions relating to a particular platform.
- instructions you define and insert to enable you to produce different executables for different circumstances, such as 'debug' and 'release' versions of a program.

If we examine the directives encountered so far in `firstproject`, you will see:

Chapter 2 LAZARUS AND PASCAL

`{$mode objfpc}{$H+}` immediately after the program heading.

`{$mode objfpc}` sets the dialect of Pascal we use (*strict Object Pascal with support for classes, and not the looser Delphi dialect*).

`{$H+}` sets the type of string variable support required (*H+ indicates we will use ansistrings*).

```
{$IFDEF WINDOWS}
  ReadLn;
{$ENDIF}
```

Here are two linked directives wrapping a Pascal routine. A `$IFDEF` must always be followed by a corresponding `$ENDIF`. The `{$IFDEF WINDOWS}` means "If Windows is defined then ..." Lazarus defines `WINDOWS` automatically on Windows machines. So this instruction tells the compiler to include code for `ReadLn` at that point in the program, but only if the program is compiled for Windows. If the program is compiled on Unix, that code is not needed and not wanted, and so is not included in the program.

Such directives enable Lazarus to deal cleanly with all manner of platform differences where different code routines are needed because of local variation between operating systems.

Needing the occasional `$IFDEF` is a consequence of using such a versatile IDE as Lazarus that runs on Windows, Mac, Linux and elsewhere.

The first compiler directive which wraps one of the unit names is more complicated because it is nested (*i.e. one directive is wrapped inside another one*). However the effect is similar to the `$IFDEF WINDOWS` example – it provides specifically for one particular family of platforms (*Unix*) that needs the `cthreads` unit if `UseCThreads` is defined (*which it usually is*). Here it is again:

```
{$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
{$ENDIF}{$ENDIF}
```

2.h Review Questions

1. Why is `Ides-Of-March` not a valid Pascal identifier?
2. What are the two main categories of program Lazarus can produce?
3. Why should you not name a Lazarus project `Begin`?
4. What is wrong with the following code?
`{$IFDEF DARWIN}`
`DoDarwinSpecificRoutine;`
5. Is `1stProgram` a valid Pascal name?



Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.a Pascal types

You will recall that there are two sorts of entities in a computer program: **code** and **data**.

The **data** are the chunks of information that the program deals with (*such as the colour of the typeface used, or the first line of a business address the user enters, or the current date*).

Code, on the other hand, represents instructions to the computer's CPU (*and via the operating system to its display, disk drives and so on*) about how to process that data.

A program is a complex combination of code and data, designed for a particular application purpose. This chapter is largely to do with **data**. Pascal helps you to organise your data by using three concepts, those of

- type (declaration: `type`)
- variable (declaration: `var`)
- constant (declaration: `const`)

Of the three concepts the fundamental, governing idea is that of **type**.

Pascal is known as a **strongly-typed** language. This is a Good Thing, which will help you avoid one source of sloppy programming by forcing you to consider the types you employ at every stage of writing code. Data used in programs is routinely broken down to small elements of known size (*which after breakdown can always be recombined*). This is partly because a computer's memory is a limited resource, and memory usage needs to be planned carefully.

Memory can be squandered but – however much is installed – it is a finite resource that eventually cannot accept any further code or data. If your program attempts to use more memory than is physically available then it suddenly becomes very sluggish. Other running programs are also affected, and (*if it also runs out of virtual memory*) the computer will freeze or crash. Memory should be conserved and used prudently.

In Pascal all data is categorised by **type**. Each data type has a **known size** (*even variable-size data types have a **current** size that is known, otherwise their memory usage could not be managed*).

Data and type sizes are measured in bytes or multiples of bytes: kilobytes(kB), megabytes(MB) and gigabytes(GB).

There are a number of **simple types** which occupy at most a few bytes, and from which larger **structured types** are built. All types must be **declared** before they can be used.

A number of common types are predeclared (*effectively they become part of the Pascal language*) and so have names which you should not reuse for other purposes (*even though in some cases you can do so*). These types are declared in the libraries (*RTL, FCL and LCL*) that Lazarus uses.

It is possible to redefine many of the identifiers from the libraries used by Lazarus programmers. However to do so would be very confusing, not only to you, but to anyone else reading your code. Unless you enjoy managing chaos these names are best left alone.

So although many of the types you will encounter and use are not actually part of the Pascal language definition (*they are simply declared in various units of the RTL/FCL/LCL, just as you will be declaring types of your own before long*), it is safest to think of these predeclared types as if they were somehow set in stone, and immutably part of Pascal.

Floating point types include `single`, `double` and `extended` in Pascal. They are often referred to generically as **real** types, though in the FPC implementation of Pascal, `real` is also a platform-dependent type which equates either to `single` or to `double`. Nevertheless the term **real** is often used as a generic description equivalent to “floating point” in many articles and books about Pascal. A few of the predeclared simple types available to programmers are listed below in Table 3.1.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

Data to be stored	Size in bytes	Predefined Pascal type
ASCII character	1	Char (ansichar)
Boolean value (True or False)	1	boolean
Integer between 0 and 255	1	byte
Integer between -128 and +127	1	shortint
Integer between 0 and 65,535	2	word
Integer between -32,768 and +32,767	2	smallint
Integer between 0 and +4,294,967,295	4	longword (cardinal)
Integer between -2,147,483,648 and +2,147,483,647	4	longint
Real value between 1.5E-45 and 3.4E38	4	single
Integer between -9,223,372,036,854,775,808 and +9,223,372,036,854,775,807	8	int64
Real value between 5.0E-324 and 1.7E308		Double
Dates and times between 01.01.0001 and 31.12.9999	8	TDateTime
Real value between 1.9E-4923 and 1.1E4932	10	extended

Table 3.1 Some predeclared simple types commonly used in Pascal

All types must be **declared** somewhere before use. A **declaration** defines an identifier to be used in Pascal expressions and statements, and in appropriate cases will allocate memory for the identifier. The above predeclared types can be found in the FPC sources (*or internally within the compiler*).

A Pascal type declaration has the format:

```
type TypeName = definition_of_the_type;
```

Here is an example:

```
type integer = longint;
```

This example defines an **alias** type – we can refer to the longint type either as longint or as integer. The two names refer to an identical type on 32-bit computers which can contain values such as -78 or 2,341 or 0. Here is another example:

```
type TDaysInTheMonth = 1..31;
```

This defines a **subrange** type, which limits numeric data of that type to values between 1 and 31 inclusive. You can define a subrange of any **ordinal** type (*see the following section*).

Note: Since this last subrange example is a type we have made up ourselves (*not a predefined Pascal type*) we started the type name with a capital T (*for Type*). This is just a convention, which Lazarus will not force you to follow. However, it is helpful for readers of your code to mark simply, in a visually distinctive way, the difference between the names of types and the names of variables.

The Editor, and this book, uses a bold mono-spaced font to mark Pascal keywords like **end**. Other Pascal identifiers are in a normal mono-spaced font such as `WriteLn`. Keywords should not be used for anything other than the meaning reserved for them in the Pascal language. Although you can reuse identifiers like `WriteLn` for purposes other than the normal RTL usage of the procedure so named, it is very unwise (*and most confusing*) to do so.

3.b Ordinal types

Ordinal types exhibit order. They have a limited set of values that can be ordered exactly. Each value in the value set (*except the first*) has a unique predecessor, and each value in the set (*except the last*) has a unique successor. For integer types the order (*or ordinality*) of the value is the value itself.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

For other ordinal types (`Char`, `boolean`, subrange and enumerated types) the ordinality of the first value is 0, of the second is 1, of the third is 2 and so on. See Section 3.d for more about enumerated types. All the single and dual byte types in the above table (Table 3.1) of simple types are ordinal, as are the four byte `longint` and `longword` types. However floating point types (`single`, `double`, and `TDateTime` which is based on a float type) are not ordinal. It is impossible to say what number comes immediately before or after any floating point value. For instance what value comes after 3.4? Is it 3.41, or 3.401, or ...? Also the eight byte `int64` integer is not a true ordinal type. Pascal provides several predefined functions that operate on all ordinal types. They include the routines listed in Table 3.2

Routine	Parameter	Return value	Comment
<code>Ord()</code>	ordinal expression	ordinality of parameter	Can be used in case selectors
<code>Succ()</code>	ordinal expression	successor of parameter	Can be used only with directly read ordinal class properties
<code>Pred()</code>	ordinal expression	predecessor of parameter	Can be used only with directly read ordinal class properties
<code>Low()</code>	ordinal type or variable	lowest possible value of the given type	Also applicable to arrays and strings
<code>High()</code>	ordinal type or variable	highest possible value of the given type	Also applicable to arrays and strings
<code>Inc()</code>	ordinal variable	incremented variable is passed as a var parameter	Use a 2nd parameter with integers for an increment other than 1. Also applicable to pointer types.
<code>Dec()</code>	ordinal variable	decremented variable is passed as a var parameter	Use a 2nd parameter with integers for a decrement other than 1. Also applicable to pointer types.

Table 3.2 Commonly used routines which take ordinal parameters

3.c The boolean type

The boolean type is a fundamental feature of all programming languages, and provides exactly the right type for uncomplicated Yes/No, Black/White, Present/Absent sort of data.

The `boolean` type is part of the Pascal language, predefined ready for use. It can hold one of two mutually exclusive predefined values: `True` or `False`. A boolean variable is sometimes called a 'flag' in programming jargon, since its value flags up which one of these two mutually exclusive states currently applies to that variable.

The comparison and logical operators always produce boolean expression results (see Chapter 5 for more about Pascal expressions), and conditions in conditional statements (`if`, `while`, `repeat`) must resolve to a boolean expression (see Chapter 6 for more about Pascal statements).

Here is an example of declaration and use of a boolean variable named `finished`:

```
var finished: boolean = False;
begin
  repeat
    {some action that eventually sets finished to True};
  until (finished = True);
end;
```

Note that the final condition could be written more compactly as `until finished`;

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

In addition to the one-byte boolean type which has been part of Pascal since 1970 there are several newer 'boolean' types (`byteBool`, `wordBool`, `longBool` of type `byte`, `word` and `longint`) which are assignment-compatible with `boolean`. They allow straightforward interfacing with routines from C and C++, where any non-zero integer is regarded as `True` when cast as a boolean. Thus `longBool` is useful as a return type for many Windows API functions which return zero for failure and any non-zero integer for success. If `bool` is a boolean variable which is `True`, then `Ord(bool)` returns +1 if `bool` is `boolean`, while `Ord(bool)` returns -1 if `bool` is `byteBool`, `wordBool`, or `longBool`.

3.d Enumerated types

Suppose you are writing a program to record membership in your local singing group. Each person's data includes a boolean value to indicate whether they have paid or not. You call it `HasPaid`, and declare it thus:

```
var HasPaid: boolean = False;
```

Once the member pays their annual membership fee their `HasPaid` value is changed from `False` to `True`. However, it turns out that some members have paid a lifetime subscription, and so don't need to be sent subscription reminders. Do you introduce another boolean variable `LifetimeMember`?

Then a probationary membership is introduced after which new members have an audition and can be accepted long term (if they opt to continue), or they can opt to leave; or perhaps they fail the audition. In that case any fee they have paid is refunded. So the actual possibilities are more complicated than merely a `HasPaid (True)` and `HasPaid (False)` situation. In fact you need a variable that can track the following situations that could describe a member's payment:

- Has not (yet) paid
- Has paid a lifetime subscription and should not ever be sent a subscription reminder
- Has paid the current year's subscription
- Has paid a subscription and had it refunded

Where one option from a range of more than two options is to be stored in a variable you need an **enumerated** type, rather than a boolean type.

Subrange types were mentioned earlier, which are simply a limited range of values taken from an already available base type. For instance:

```
type TLiquidCelsiusRange = 0..100; // a subrange of integer
   TLowercaseChar = 'a'..'z'; // a subrange of Char
```

Enumerated types are similar, in that they are ordinal types in which each value in the type is in a specific order, and has a next value (except the last one) and a previous value (except the first one). Enumerated types differ from subrange types in that they explicitly declare the name of every possible value, as well as its fixed order in the sequence.

The above membership payment situation would be covered by an enumerated type declared as follows:

```
type TPaymentKind = (Unpaid, LifetimePaid, CurrentYearPaid, Refunded);
```

Likewise we could declare a `TCelestialBody` type like this:

```
type TCelestialBody = (planet, moon, asteroid, comet, meteor, star,
   blackHole, darkMatter);
```

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

Notice the use of brackets () to delineate the list of elements in the type, commas to separate the elements in the list and the = sign used in the declaration. We also stick with the convention of starting our own type names with a "T". In fact the boolean type can be thought of as an enumerated type containing only two values defined as follows:

```
type boolean = (False, True);
```

The LCL makes extensive use of enumerated types to give meaningful names to frequently used properties. (See Chapter 8 for more information about classes and properties).

For example, the TForm class has a property called BorderStyle of type TFormBorderStyle which is declared thus:

```
type TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog,  
                        bsToolWindow, bsSizeToolWin);
```

giving you a choice of six border styles for windows in your programs. There are exactly six border styles to choose from – no more, and no less. By contrast some controls (*widgets*) you can drop on to a form have a slightly different BorderStyle property which is of type TBorderStyle (note the slightly different name), which is declared thus:

```
type TBorderStyle = bsNone..bsSingle;
```

Here TBorderStyle is a **subrange** of TFormBorderStyle, in this case having only two elements.

The TPanel component has such a BorderStyle property. It gives panels the option of having no border, or a single line border. It does not offer a sizeable border built in to its functionality as the TForm class has.

FPC lets you assign an ordinal value to some or all of the named enumerations, provided you declare them in ascending order. For instance, the following declaration is legal in Free Pascal:

```
type TLowPrimes = (two=2, three=3, five=5, seven=7, eleven=11,  
                 thirteen=13, seventeen=17);
```

This declares an enumerated type where the Pascal name of each possible value in the type corresponds exactly to its ordinal value. If the TLowPrimes type were declared like this:

```
type TLowPrimes = (two, three, five, seven, eleven, thirteen, seventeen);
```

then, though the names of the type elements are identical with the earlier declaration, the ordinal values would be as for any default enumerated type, starting at zero and incrementing steadily one by one (Ord(two)=0, Ord(three)=1, Ord(five)=2, etc.).

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.e Type conversion

Because Pascal is a strongly typed language, there is often a need to convert one type into another type. It is very convenient to have an integer type such as byte for holding small positive integer values on which you can perform arithmetic:

```
var a: byte = 14; b: byte = 2;
begin
  WriteLn('The value of a x b = ', a * b, '; a minus b = ', a - b);
end;
```

However you cannot directly display a byte value – first it has to be converted to a string to be displayed. The `WriteLn()` procedure does this conversion for us implicitly. The `Write/WriteLn` and `Read/ReadLn` procedures are in fact quite complex routines in order to handle all these implicit type conversions (*in addition to their ability to handle a varying number of arguments*). Most Pascal routines are not so versatile, and are designed to deal with a more limited range of types, often being defined only for one type. This is both an aid in avoiding bugs (*since the compiler will notify you if you try to pass data of the wrong type to a type-specific routine*), and a nuisance when Pascal's type checking seems overly restrictive.

In Chapter 9 the Section **d** dealing with **overloading** discusses ways to circumvent this type-limitation imposed on parameters passed to Pascal routines. There are several ways to work with such a strictly type-checked language as Pascal when you find the compiler's type checking too restrictive. Here are three of them:

- use a type conversion routine (*the subject of this section*).
- use a typecast (*the subject of the next section*).
- use the variant type (*a topic not treated in this book, but particularly useful for COM/OLE programming, for which the variant type was introduced into Pascal; and to a lesser extent useful in database applications, if performance is not critical*).

Pascal provides numerous functions to convert values explicitly from one type to another as `WriteLn` and `ReadLn` do implicitly. The following table (Table 3.3) lists some of the most commonly used routines, though there are a great many more than those listed here.

Name of routine	Type	Result type	Comment
<code>Chr(b: byte): Char</code>	Byte	Char	Equivalent to the typecast <code>Char(b)</code>
<code>Ord(x: ordinal): integer</code>	any ordinal type	positive	Mainly useful for characters and enumerated values
<code>Val(s: string; var V; var code: integer)</code>	string	numeric	The numeric V parameter can be integer, int64 or a real value
<code>Trunc(r: realvalue): int64</code>	floating point	int64	If r is infinite or NaN Trunc gives a runtime error
<code>Round(r: realvalue): int64</code>	floating point	int64	Rounds r to the nearest integer value, using banker's rounding for .5
<code>Int(r: realvalue): realvalue</code>	floating point	floating point	Int returns a floating point value with its fractional part set to zero
<code>IntToStr(value:integer): string</code>	integer, int64, QWord	string	The resulting string has the minimum number of characters
<code>DateToStr(aDate: TDateTime): string</code>	TDateTime	string	The resulting string has the short date format
<code>StrPas(p: PChar): shortstring</code>	PChar	shortstring	The resulting string is truncated at character 255 if longer than this

Table 3.3 Some commonly used Pascal type conversion routines

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.f Typecasts

The FPC expects the left hand side and right hand side of assignment statements to be type compatible, and if they are not will give an error in the Messages window something like “Expecting ansistring and got longint”. If the offending expression is the same size as the incompatible type it is possible to **cast** it (or *typecast* it) using the name of the type and parentheses. For instance:

```
var i: integer;
    b: boolean;
    c: Char;
begin
    i:= integer('A');
    b:= boolean(0);
    c:= Char(43);
end;
```

You can cast between ordinal types freely even when they are of different sizes, and you can freely cast between real types such as from `single` to `double` or between integer types. In fact the compiler does this for you, and often no explicit casting is required. Bear in mind that casting between different sized integer types, even if allowed by the compiler, may lead to loss of data if you cast from a larger to a smaller type. For instance, consider this program snippet:

```
var b:byte;
    i:integer = 2000;
begin
    b:= byte(i);
    WriteLn('The value of b is ',b);
    ReadLn;
end.
```

The display will show that `b` is equal to 208 (*not 2000*), since part of the binary data in the integer variable `i` has been discarded in making the cast to the smaller byte variable. Casting is generally a risky undertaking because you are accessing a value as if it were something other than what it actually is. If you know the underlying storage layout of the types involved you will probably understand what you are doing, and perhaps feel safer. Nevertheless typecasts can lead to hard-to-locate bugs, and it is better to look for a way to code that does not require casting if possible, since casting is a way of circumventing compiler checks that are usually helpful. A compiler range or overflow check is defeated by a typecast like the above

```
b:= byte(i);
```

since the cast specifically tells the compiler that it is OK to treat this integer as a byte (*when it might not be*).

Pascal allows casting between `ansistring` and `PChar` types – see Chapter 4 for more about strings and character arrays.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.g Variables

Types only become useful when we have an actual data item of that type, a **variable**.

A variable is a named memory location which holds a specific type. Its value can change at runtime, and indeed it is this potential for variation that gives variables their name.

Memory is involved because the computer has to put its data somewhere, and be able to refer to it. Computers store all data (*whether or not it will eventually be stored more persistently in a disk file or on a network server in the Cloud*) in short term memory (also called RAM for **R**andom **A**ccess **M**emory).

Naming each variable you use gives you a handle on all uses of that variable.

The compiler associates each program variable with a specific place (*an **address***) in memory where that variable's data is stored. When your program has finished using the variable, the compiler arranges for that particular memory location to be released, and made available for potential reuse by another program. A later section on pointers (3.n) gives details of a Pascal type that can be used to refer to specific memory addresses without knowledge of the name of the variable stored there. Like types, variables have to be declared before they can be used, and all variables must be of some **known type** (*i.e. a type that has been declared somewhere earlier in the program or unit, or in a unit listed in the `uses` clause, or declared in one of the required packages/libraries listed in the Project Inspector as a dependency*).

A variable declaration has the format:

```
var      VariableName: VariableType;
```

Here are some examples:

```
var height, width, length: integer;
    daysInApril: TDaysInTheMonth; // this uses a type declared in Section a
    keyJustPressed: Char;
    CurrentAccountBalance: double;
    OnlyUseCapitalLetters: boolean;
```

You can declare several variables of the same type all together by placing them in a comma-separated list, as above with `height, width, length`.

3.h Initialised variables

A variable (*i.e. a memory location*) contains an unknown (*random, rubbish*) value at the start of a program until the programmer makes some **assignment** that puts a known value into the variable. Some languages initialise variables for you to a known initial value such as zero, but Pascal does not. This is one cause of bugs which arise when a careless programmer omits to give a variable an initial value. However, FPC provides for variables to be initialised at the time they are declared. Global variables are initialised once when they first come into scope at the start of the program to the value zero.

For local variables (*used in routines*) instead of writing:

```
var counter: integer;
begin
    counter := 0;
...
```

you can write the more concise code:

```
var counter: integer = 0;
begin
...
```

This is a very handy FPC feature (*borrowed from C*), enabling you to initialise variables when you first find the need for them, helping you to avoid the uninitialised-variable bugs that happen all too easily since local variables are not initialised automatically for you.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.i Assignment: placing a value in a variable

Once a variable has been declared it is used in a program by means of the **assignment operator** := which is a dual-symbol operator made up of a colon followed immediately by an equals sign. Given the following type and variable declarations, the assignments below are valid Pascal statements:

```
type TDayOfWeek = 1..7;
var  active: boolean;
      dayOfWeek: TDayOfWeek;
      golden_section: single;
begin
  active:= True;
  dayOfWeek:= 5;
  golden_section:= 1.618;
end;
```

Notice how the = equals sign alone is used in a type definition, and the combination := in assignments.

The above three assignments (between **begin** and **end**) are all valid. However the following assignment would produce a Warning:

```
dayOfWeek:= 9;
```

The message would be *Warning: range check error while evaluating constants* since we have defined `dayOfWeek` to be of a type that is limited to values between 1 and 7. So trying to assign the value 9 to `dayOfWeek` is a programming error.

The FPC checks the code you write when you attempt to build or compile a project.

Any warning or error message is shown in the Messages window.

A **warning** will not stop compilation, but you should definitely pay attention to it – ignoring warnings stores up trouble. An **error** means that the code as written is not compilable without correcting the error.

When reading code (*whether aloud or in your head*) the assignment operator can be read as “becomes equal to” so that

```
active:= True;
```

is read as “active becomes equal to True” (or “assign the value True to active”).

In algebra (and C) assignments are made using the equals = sign.

In Pascal this is **not** the case.

You can write

```
active = True
```

in Pascal (*without a closing semicolon*). This is a valid Pascal expression.

However it does **not assign** the value True to active, it **compares** the two values, and the result of the comparison is a boolean value (*True if the values are equal, False if not*).

Chapter 4 has fuller details about Pascal operators and expressions.

Note that the variable has to be on the **left hand side** of this assignment. It is not possible to execute the Pascal statement:

```
True:= active;
```

This will not compile. The Editor cursor will be placed at the beginning of True,

and the message will be: *Error: Variable identifier expected*. This is because True is a predefined Pascal constant, not a variable, so no new value can be assigned to it. By contrast the operand comparison (`True = active`) can be made with the operands in either order.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.j Extended numerical assignment operators

FPC has borrowed the shortcut numerical assignment operators familiar to C users, meaning that a complete list of available assignment operators is as given in Table 3.4 below.

Assignment	Outcome of the assignment	Type of left hand side, A
A := expression	The expression result is stored in A	Any type
A += expression	Adds expression result to A, storing it in A	Numeric, string, set, pointer
A -= expression	Subtracts expression result from A, storing it in A	Numeric, set, pointer types
A *= expression	Multiplies expression result by A, storing it in A	Numeric, set types
A /= expression	Divides A by expression result, storing it in A	Real type

Table 3.4 Free Pascal assignment operators

3.k Constants and literal values

A constant declaration has the format:

```
const ConstantName = value;
```

Constants are entities whose value stays constant, allowing you to give meaningful names to values that do not change as the program executes. They are given a value in your code ('hardwired' at compilation) and keep that value throughout the program. You cannot change them. Unlike variables, constants do not need to be **explicitly** given a type, since the compiler can determine from the value given to the constant what is the most appropriate type for it. All constants do have a type assigned to them by the compiler, but it is implicit.

Any ordinal, real, character or string value can be declared in a program as a constant.

Numeric constants are written simply as the digits of the desired number. For instance:

```
const DaysInTheWeek = 7;  
e = 2.718281;
```

By default it is assumed that integer and float (*real*) constants are **decimal** (*base 10*) values.

You can specify unsigned integer constants as hexadecimal (*base 16*) values by prepending the number with a \$. Likewise you can specify a constant as octal (*base 8*) by prepending it with &, or specify it as binary (*base 2*) by prepending it with %. The following are alternative ways of declaring a constant of decimal value 10:

```
const tenDecimal = 10;  
tenDecimalDefinedInBinary = %1010;  
tenDecimalDefinedInOctal = &12;  
tenDecimalDefinedInHex = $A;
```

Note: This binary and octal notation only operates if `{mode obifpc}` has been specified.

Floating point constants can also be specified using engineering notation, which is essential for very large numbers.

```
const Avogadro = 6.022141E+23;
```

Character and string constants are written as text enclosed in single quotes. The quote marks are not part of the value - they merely signal to the compiler that this is a character or string constant. You cannot use double quotes "" to indicate a string value, only single quotes '' (*apostrophes*). For instance:

```
const Space = ' ';  
IDName = 'Lazarus';  
EmptyChar = '';  
DoubleQuoteChar = ''';
```

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

If the string itself contains a single quote, that quote character has to be doubled when written to inform the compiler that it is an actual quote character, and not yet the end of the text string. So the phrase *Reader's Digest* has to be written in Pascal as:

```
const MagazineName = 'Reader''s Digest';
```

Here are some further examples:

```
const CardsInAPack = 52;
      pi = 3.1415;
      hidden = False;
      QuestionMarkChar = '?';
      OriginalPascalDesigner = 'Niklaus Wirth';
```

Most control characters cannot be typed on a keyboard as printable characters. To specify such characters in code the symbol # is given the special meaning that when followed immediately by a number it specifies that numbered ASCII character. So the following commonly encountered control characters can be inserted in code as follows:

```
#8  backspace
#9  horizontal tab character
#10 newline character (linefeed)
#12 form feed character
#13 carriage return character
#27 escape key
```

In fact any ASCII character can be represented in this way ('A' is #65 for instance). However for readability it is far better to use normal quoted strings for alphanumeric and symbol characters rather than using 'magic numbers' which may merely appear mysterious to future readers of your code.

You can also specify control characters using the older Turbo Pascal notation which combines the ^ character with an alphabetical letter (*so that #10 = ^J, and #13 = ^M*). ANSI characters are not Unicode characters. However, the first 256 Unicode characters correspond exactly to the 256 one-byte ANSI characters.

3.1 A program example: simple_types

This is a good moment to use Lazarus to write a short example program which exercises some of the ideas encountered so far.

Start Lazarus, and use the main menu to choose **Project | New Project...** selecting *Program* from the list at the left of the *Create a new project* dialog to create a console application as you did previously. Save this project (**Project | Save Project As ...**) in a new folder, giving the project the name `simple_types`. As you know, Lazarus will save this project in a main program file called `simple_types.lpr`, with a project file called `simple_types.lpi`. Lazarus will also write the following code for you:

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

```
program simple_types;

{$mode objfpc}{$H+}

uses
{$IFDEF UNIX}{$IFDEF UseCThreads}
cthreads,
{$ENDIF}{$ENDIF}
Classes
{ you can add units after this };

begin
end.
```

Delete the `uses` line and the five lines which follow it because they are not needed for this program (the default Editor shortcut for deleting lines is `[Ctrl]Y`). We will use the versatile `WriteLn()` function to display successive lines on the console showing the values of a constant and two variables of different types.

Copy the following code into the Editor so that your program matches what follows, and then press `[F9]` to save, compile and run it.

```
program simple_types;

{$mode objfpc}{$H+}

var
  smallInteger: SmallInt= -37;
  started: boolean= False;
const
  pName= 'simple_types';
  tab= #9;
begin
  WriteLn('This program is called ',tab,tab,'"',pName,'" ',sLineBreak);

  WriteLn('When initialised smallInteger has the value: ',smallInteger);
  WriteLn('When initialised started has the value: ',started, sLineBreak);

  smallInteger += 100;
  WriteLn('After adding 100 to smallInteger its value is: ',smallInteger);
  started:= not started;
  WriteLn('Negating started gives it the new value: ',started, sLineBreak);

  WriteLn('The highest value a SmallInt can store is: ',High(SmallInt));
  WriteLn('The lowest value a SmallInt can store is: ',Low(SmallInt));
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

We declare two initialised variables, a `SmallInt` and a `boolean`; and a string. The global variable `sLineBreak` is declared in the system unit which is always available even with no `uses` clause. When `WriteLn(sLineBreak)` is executed a blank line is inserted in the display (*whatever platform or OS you use*). The same effect can be achieved by calling `WriteLn` with no parameters (either `WriteLn;` or `WriteLn();`).

The display output of the program should look similar to this:

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

```
This program is called      "simple_types"
When initialised smallInteger has the value: -37
When initialised started has the value: FALSE

After adding 100 to smallInteger its value is: 63
Negating started gives it the new value: TRUE

The highest value a SmallInt can store is: 32767
The lowest value a SmallInt can store is: -32768
```

3.m Typed constants

You can also specify the type of a constant in its declaration. This sort of constant is known as a **typed constant**. So you can define the speed of light either like this:

```
const SpeedOfLight = 299792458; // an ordinary constant
```

or like this:

```
const SpeedOfLight: double = 299792458.0; // a typed constant
```

Typed constants are initialised once dynamically at the start of the program. Their advantage is that they allow you to declare record, array and pointer constants, declarations which are not possible with ordinary untyped constants (though it is may be better to use an initialised variable for this purpose, see above at Section 3.h).

The disadvantage of typed constants is that they are not constant, or rather that their value can be changed. The term `const` is confusing in this case, because you can change the value of typed constants. Whereas ordinary untyped constants cannot have their values altered (which is the intuitively assumed meaning of `const`).

Typed constants are useful for what some languages term **static variables** within subroutines, that is variables that keep the value they last had before the subroutine finished when it was last called. Ordinary local variables are volatile – they are erased at the end of a subroutine, and their value at subroutine exit cannot be recovered if it has not specifically been saved somewhere else in the program.

On the other hand a local typed constant's value (it is effectively a sort of initialised variable) will persist after a routine exits, and retain that value when the routine is next called. This is because local typed constants are not stored on the transient stack as other local variables are which are declared within a routine. By contrast a local initialised variable will always keep its one constant initial value, since it is re-initialised to that value whenever the routine is called.

As an example of how local typed constants and local initialised variables can be used, try the following short program (or you may prefer to wait until you've read Chapter 7 which explains more about functions). Create a new console project in Lazarus, and save it as `typed_const`. Then adapt the skeleton code Lazarus writes to look like the following:

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

```
program typed_const;

{$mode objfpc}{$H+}

Function GetAName: string;
const lastName: string = '';
var  constName: string = 'Lazarus';
begin
  WriteLn(['Last name entered was ', lastName, '']);
  WriteLn(['Value of constName is ',
constName, '']);
  Write('Enter a new name: ');
  ReadLn(Result);
  lastName:= Result;
end;

begin
  WriteLn('First invocation of GetAName');
  WriteLn(GetAName);
  WriteLn;
  WriteLn('Second invocation of GetAName');
  WriteLn(GetAName);
  WriteLn;
  WriteLn(['finished']);
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Although we have not looked in detail at functions, and how they are declared and used (see *Chapter 7 for more on this*), the program is simple enough to follow. The function `GetAName` is called twice as part of two `WriteLn` calls, which display the value of this function display each time. Each invocation of the function asks for user input. A typical program output display is the following:

```
First invocation of GetAName
[Last name entered was ""]
[Value of constName is "Lazarus"]
Enter a new name: Blaise
Blaise

Second invocation of GetAName
[Last name entered was "Blaise"]
[Value of constName is "Lazarus"]
Enter a new name: Pascal
Pascal

[finished]
```

Ironically, in this particular local function, `GetAName`, it is the initialised variable that is initialised to its unvarying value, and the typed constant that varies!

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

3.n Pointers

A pointer is a variable that denotes a memory address. There is a generic `pointer` type that can be used to point to any memory address. A memory address is found by use of the `@` operator. The pointer type is compatible with any other (*generic or typed*) pointer. You can also define **typed pointers** to indicate the kind of data stored at the addresses these pointers point to. These are somewhat safer to use, since the compiler will prevent you from assigning a typed pointer to a typed pointer of a different type.

Pointers allow fairly low level access to the way the compiler uses a computer's memory to store data and sections of code instructions. A pointer is a fixed-size type (*4 bytes on a 32-bit computer*). When it holds the address of another variable we say it **points to** the location of that variable in memory (*i.e. to the data stored there*) – hence the name of this type.

Your business address might be something like
33 Avenue des Champs Élysées, Paris

This enables anyone who knows your address to visit your shop. Likewise a pointer with a valid memory address gives a programmer access to the data or code stored at that specific memory location, even without knowing the name of the variable.

Type checking for pointers in Pascal is looser than for other types, which makes them more dangerous to use, since you are deliberately employing a type for which the compiler cannot put in place some of the checks that apply to other types. So although pointers are particularly useful for certain situations, giving immediate access to memory that might be difficult or impossible to access in any other way, use of pointers means abandoning certain safeguards simply in order to use them at all. In particular they facilitate the allocation of memory **dynamically** in situations where the amount of memory required is not known at the time of compilation.

Typed pointers indicate the type of the data stored at the memory address they point to. You use the caret (*circumflex*) character `^` to declare a pointer type rather than using a keyword to do this.

The `^` operator is used in two ways. Placed immediately **before a typename** the `^` symbol denotes a type that represents pointers to variables of that type. Placed immediately **after a typed pointer variable** the caret **dereferences** the pointer, that is it returns the value the typed pointer is pointing to – the value being the data that is stored at the location the typed pointer points to. The value should be of the type indicated by the name of the typed pointer variable. It is good programming practice to use typed pointers as much as possible (*rather than generic pointers*). This not only helps to prevent you from making errors, but provides built-in documentation to people reading and maintaining your code later (*assuming you have named the typed pointers you use intelligently*).

A pointer is declared explicitly in one of two ways. Either you can write:

```
var p: pointer; // p is a generic pointer variable
```

or you can use the `^` (*circumflex or caret*) pointer operator as follows:

```
var PByte: ^Byte; // a typed pointer which can only point to bytes
```

In addition to `pointer`, FPC defines several useful typed pointer types including:

```
type PInteger = ^integer;  
PChar = ^char;
```

Note the use of a leading 'P' here. This is a helpful convention, because when using pointers it always helps to be able to distinguish them immediately from non-pointer variables.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

Correct use of pointers is one of the more advanced aspects of Pascal. There is lots of code in the libraries that come with Lazarus that uses pointers extensively. For instance the `TList` class declared in the RTL maintains a list of pointers. This versatile class is used extensively elsewhere, and is a template for many other list and collection classes.

Many complex data structures make heavy use of pointers both to build and to navigate the data structure.

Pascal allows you to treat a pointer to any type as an array of that type (*as C does*).

Such an array has an undefined length, and is not reserved simply by declaring the pointer.

The array memory needs to be reserved by explicit declaration of a variable to which the pointer is made to point, perhaps by calling the `GetMem()` or `New()` routines.

The pointer points to the first (*zero-indexed*) element of the array.

(See Chapter 4 for more about array types).

Pascal has built in support for the `PChar` type which is not only a pointer to a `Char`, but a pointer to a null-terminated, zero-indexed character array (*the classic C structure for handling strings*). This facilitates interfacing with libraries written in C or C++ which expect string parameters to be null terminated character arrays. This is true, for instance, for all the DLLs that make up the Windows OS. Use of `PChar` variables and typecasts makes interfacing directly with the Windows API much easier for Pascal users. Likewise `ansistring` variables are really pointers to arrays of characters (*which also hold additional string metadata*). If you come from a C background you will feel at home with the `PChar` type. Otherwise it is probably best to get familiar with the traditional Pascal string types (*shortstrings and ansistrings*) before trying to use `PChar`. Pascal strings are treated in Chapter 4.

Here is an example of a program that makes use of a typed pointer (*a user-defined pointer to a longint*), and manipulates the data the pointer points to both using the pointer, and using the name of the data variable. In the following program the two references `intPtr^` and `anInt` are **aliases** to the same data value. Only one memory location is involved in storing the data.

```
program pointer_project;

{$mode objfpc}{$H+}

type
  Plongint = ^longint;
  // this declares a typed pointer
var
  anInt: longint = 243;
  intPtr: Plongint;
begin
  intPtr:= @anInt; // this points intPtr to a specific integer variable
  WriteLn('The value of intPtr^ is: ',intPtr^); // dereferencing intPtr
  Inc(intPtr^, 4); // this alters the data of the dereferenced value
  WriteLn('The value of anInt after Inc(intPtr^,4) is: ',anInt);
  {$IFDEF WINDOWS}
    ReadLn;
  {$ENDIF}
end.
```

Begin a new console project in Lazarus named `pointer_project` and adapt the skeleton code Lazarus provides to read as above. Compile and run the program, and check if the output is as you expect.

Chapter 3 TYPES, VARIABLES, CONSTANTS AND ASSIGNMENTS

The keyword `nil` is a special constant that can be assigned to any pointer to indicate that the pointer is not pointing to any known memory address, i.e. the pointer is **unassigned**.

```
const nil = pointer(0);
```

The `nil` keyword is a special pointer value that is guaranteed to be distinct from any valid pointer.

Pascal provides a boolean function named `Assigned()` which accepts any pointer variable as a parameter, returning `True` as long as that pointer parameter is not `nil`. `Assigned()` can be used on any pointer-type variable (*pointer, class, dynamic array, string*) to check that it is non-`nil`.

You can perform arithmetic on pointers using the `+` and `-` operators, or (*preferably*) using the `Inc()` and `Dec()` procedures. These two procedures always increment or decrement a typed pointer by the size of the type to which it points, and so are safer than using the `+` and `-` operators with numeric offset values.

It is very easy when programming pointer routines to inadvertently access protected memory and cause a protection fault.

This may just trigger an exception, or (*if you're unlucky*) crash your entire program.

Memory addresses can be invalid. The OS allocates certain memory region(s) to the process running your program.

If you try to access memory outside that permitted memory, a fault or exception is thrown.

This is a severe error in programming terms. It only takes one stray pointer somewhere in your code to precipitate this situation, which is why beginners often avoid pointers altogether.

However, if used sensibly, pointers are a powerful weapon in the programming arsenal.

Pointers let you shoot yourself in the foot in many ways. A good rule of thumb is:

“Only use pointers when you have to”.

Of course, you can shoot yourself in the foot without the help of pointers.

3.0 Review Questions

1. How many bytes does a `longint` occupy?
2. What is the difference between a variable and a type?
3. What types could you use to hold the value 13?
4. How would you declare a constant named `LeapYearDays` so that it had the value 366?
5. What is the value of `Succ('a')`?
6. Consider this code:

```
var smallInteger: SmallInt;  
begin  
  smallInteger := Pred(Low(SmallInt));  
end;
```

Is there a problem?

7. Write and test a program that displays the highest and lowest values an `int64` type can hold.



Chapter 4 STRUCTURED TYPES

The previous chapter which introduced the *type* concept indicated that Pascal provides for structured types in addition to simple types such as `Char`, `integer`, and `single`. Structured types combine simpler types into constructs that come closer to modelling the complexity of much real-world data. This chapter introduces several of Pascal's most useful structured types.

First we consider a type that provides for a number of elements all of the same type to be treated either as a single entity or as an indexed collection of multiple elements: the array. The memory required by an array can be allocated statically (*determined completely when the program is compiled*) or dynamically (*in this case the compiler inserts code to allocate and deallocate memory as needed at runtime as the size of the array varies with use*).

4.a Static arrays

Arrays have a **base type** which is the type of the repeating element. The declaration of a static array states the **dimensions** of the array. The simplest array is a one-dimensional (*linear*) array declared as follows:

```
type arrayTypeName =  
    array[indexRange] of baseType;
```

The `baseType` can be almost any type, though arrays cannot contain files. An example is

```
type T100IntegerArray =  
    array[1..100] of integer;
```

Arrays can have more than one dimension. Each dimension is specified by a different `indexRange` value, separated from the next dimension by a comma. For example:

```
type TCrossword = array[1..5, 1..5] of Char;
```

Individual elements of an array are accessed by index, using values within the declared dimension range(s). Given a variable `crossword` of the above type we could write:

```
var crossword: TCrossword;  
begin  
    FillChar(crossword, SizeOf(crossword), '.');  
    crossword[1, 1] := 'B';  
    crossword[2, 1] := 'o';  
    crossword[3, 1] := 'n';  
    crossword[4, 1] := 'n';  
    crossword[1, 2] := 'e';  
    crossword[1, 3] := 'r';  
    crossword[1, 4] := 'n';  
    crossword[1, 5] := 'e';  
end;
```

If `crossword` represents a 5x5 grid of letters the code above would fill the grid as follows:

```
Bonn.  
e....  
r....  
n....  
e....
```

Notice that we had to **initialise** the variable `crossword` using the `FillChar()` procedure. The first parameter passed to this procedure is the name of the variable to initialise. The second parameter is the size of the variable to be initialised (*for which we used a further function, `SizeOf()` which accurately determines the size of any type in bytes*); and the third parameter is the character used to fill each character in the array (*a `Char` occupies one byte*). The `crossword` variable, when first created, is not empty (*or full of space ' ' characters*).

Chapter 4 STRUCTURED TYPES

It is just a chunk of memory, recently set aside for our program to use. It will contain random bytes, perhaps left over by a previous program, or (*if ours is the first program to use that memory*) just whatever the initial power surge in the computer happened to leave in that memory location. If we want crossword to contain all space characters (*or all dot characters as here*) we have to **deliberately write code to make that happen**, overwriting whatever junk happened to occupy the memory `crossword` occupies before that memory was reserved for our program use.

Writing

```
crossword[1,6] := 'd';
```

gives a Warning on compilation: *range check error while evaluating constants* since the index 6 is outside the declared index range of 1..5. This code will compile, but (*as always*) it is dangerous to ignore Warnings. Memory outside that reserved for crossword will have a 'd' written to it, possibly corrupting data in a running program. The effect is unpredictable. Nothing untoward may happen. Or we may see strange characters appear on screen. Or the program may crash, or some other unforeseen outcome may ensue. It pays to attend carefully to Messages when you build or compile a program, and take corrective action as needed.

Array constants are declared as a comma-separated list of values enclosed in parentheses, with one value for each element in the declared range. Array constants have to be declared as typed constants because it is not possible to declare them as ordinary untyped constants (this facility is available only for simple types). As an alternative you can declare an array as an initialised variable, and if you want it to remain constant simply avoid your code changing that variable in any way. When using arrays you should pay careful attention to the **origin** of the index. Many arrays are declared **zero-based** like this:

```
type   TDaysOfWeek = array[0..6] of string;
const  daysOfWeek: TDaysOfWeek = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri',
                                   'Sat');
```

Notice that this zero-based indexing scheme makes the first array element ('Sun') to be `daysOfWeek[0]`, and not `daysOfWeek[1]` as you might otherwise assume.

4.b Unnamed (anonymous) types

It is possible to construct a new type on-the-fly in a var declaration where new variables are being declared without in the process needing to give such a type a name. Here is an example:

```
var uppercaseCounts: array['A'..'Z'] of integer;
```

Here we are declaring an array variable of what type exactly? This is a valid Pascal construct, but the type used is never given a name. Internally the compiler does name the type, but we do not know what that type name is, so we cannot refer to it. This means that you can never pass the variable `uppercaseCounts` to a procedure or function as a parameter, because all routines which take parameters require the parameters to have named types (*types you can refer to*). This lets the compiler know – by stating what the parameter type is – how much space to reserve for the parameter in the routine's stack memory.

For this reason alone it is unwise to use anonymous types, even though they are allowed (*since you are restricting your future coding options unnecessarily*). In fact there is no compelling reason to use anonymous types at all. Just because “you can” does not mean “you should”. It is better to name every new type you create (even if you don't need to pass variables of that type as parameters to any functions or procedures). A better programming practice is to rewrite the above variable declaration in this fashion:

```
type TUppercaseCounts =
  array['A'..'Z'] of integer;
var  uppercaseCounts: TUppercaseCounts;
```


Chapter 4 STRUCTURED TYPES

4.c Pascal shortstrings

The oldest and simplest **string type** in Pascal is a zero-based array of `Char`, and it is called `shortstring`. Lazarus programs use the `ansistring` type as the default string type by specifying the compiler directive `{$H+}` which is usually exactly what we want.

However, we can use the original `shortstring` type simply by specifying `shortstring` rather than `string` as the type of a variable.

In the `shortstring` scheme the first character element (*the zeroth element*) is used to store a character that represents the actual length of the string that is currently stored in the `shortstring`. So `aShortStringValue[0]` holds the length of the `shortstring`. Shortstrings have a formal maximum length, set when they are first declared, perhaps like this

```
var testString: string[10];
```

The `testString` variable has a maximum length of 10 characters. It can hold words such as `'learning'` and `'Pascal'` but not `'programming'` since it is too long. Test this out by starting a new console project in Lazarus (*Project | New Project... select Program and then click on the [OK] button in the Create a new project dialog*). Save the new project as `shortstrings.lpr`. Adapt the skeleton program, and delete unneeded lines so it looks like the following

```
program shortstrings;

{$mode objfpc}

var testStr: string[10] = 'programming';
begin
  WriteLn('This shows testStr enclosed by square brackets [' , testStr, ']');
  WriteLn('testStr's length is ' , Length(testStr), ' characters');
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Compile and run the program. Notice both the Warning message (*Warning: string "programming" is longer than "10"*), and what gets displayed on the console or Terminal. The word *programming* has a character length of 11. Notice what our program reports. The discrepancy is simply because we have used a `shortstring` variable to hold it that is too short. FPC can only stuff „programming“ into the undersized variable `testStr` by truncating the last character (*which is what the Warning was all about*). Before leaving this program, substitute other words for „programming“, such as

```
testStr := 'Lazarus';
```

Rerun the program. Notice that the `Length` function reports the **current** length of the string, not its **declared** length of 10 (*unless the two are coincidentally identical*). This is a drawback of shortstrings: you have to know in advance the maximum length a string is likely to be in order to declare a `shortstring` length long enough to fit the anticipated string.

Suppose you write a program whose first task is to accept the name of the user, which is stored in a `shortstring` variable. If users were all named something like `'Angela Merkel'` (13 characters including the space in the middle) we could declare a variable to hold the name as

```
var userName: string[15];
```

without having to worry about truncation of surplus characters. However if the president of Iran (*Mahmoud Ahmadinejad*) were to use the program, we would come unstuck. If we then changed the declaration to `string[20]` to accommodate him, we would have to alter the declaration again to accommodate the president of Qatar (*Hamad bin Jassim bin Jaber bin Muhammad Al Thani*).

This is why Lazarus defaults to use `ansistring` rather than `shortstring` (*this is what Lazarus sets up via the {\$H+} compiler directive which it inserts into every project*), and why **dynamic** arrays are important. The two following sections focus on these two types.

Chapter 4 STRUCTURED TYPES

4.d Dynamic arrays

We considered the difficulty of declaring a `shortstring` to receive user-name input from a user when the length of her name is not known in advance. If only there were a string type that could automatically be set to the required length! This is such a common programming requirement that modern Pascal provides exactly such a string type: the `ansistring`, and provided the compiler directive `{$H+}` is included in your program, then `ansistrings` are the default string type, used whenever you declare a string type with the generic type name `string`. To introduce the (`ansistring`) string type we shall first explore the more general **dynamic array**. This flexible type is declared thus:

```
type dynamicArrayName = array of typeName;
```

and it is used by first setting the array length using the `SetLength` procedure. For example:

```
type TDynamicIntegerArray = array of integer;
var dynIntArray: TDynamicIntegerArray;
begin
  SetLength(dynIntArray, 3);
  dynIntArray[0] := 39;
  dynIntArray[1] := 102;
  dynIntArray[2] := -7;
end;
```

This creates an array of three integers, filling the array with the values 39, 102, and -7.

The advantage of this dynamic array over a static array of integers declared as

```
var staticIntArray = array[0..2] of integer;
```

is that we can expand the dynamic array in code if we need the array to be larger for some reason, by a further call to `SetLength`:

```
SetLength(dynIntArray, 10);
```

This adds a further 7 empty 'slots' in the array, ready to have 7 integers assigned to them. Note that the value of each of the 7 new slots is undefined. If you want the value to be zero, you have to explicitly assign zero to each array element:

```
dynIntArray[3] := 0; dynIntArray[4] := 0; etc.
```

The static array is fixed: it can never have more than three integer elements. To work with a larger static array we have to rewrite the code that declares the array, giving it a larger original dimension.

The memory allocated for a dynamic array is automatically freed when the procedure or function using the array exits. However, you can do this manually as well using the call

```
SetLength(dynIntArray, 0);
```

This ensures that the memory the compiler allocated for the dynamic array is released immediately.

4.e Ansistrings

Variables declared as `string` in units or programs containing `{$H+}` are `ansistrings`. These strings behave like dynamic arrays of characters, and can contain as many or as few characters as needed for text data. Moreover, the compiler manages string variables for us, so the varying memory they require is allocated as needed, and **automatically** freed after use to release that memory for other uses. `Ansistrings` are guaranteed to be empty when first created by the compiler, so they do not require a specific

```
aString := '';
```

assignment to initialise them. This all happens transparently, and the programmer can just enjoy and benefit from Pascal `ansistring` behaviour without having to worry about managing strings herself.

Chapter 4 STRUCTURED TYPES

By default the Lazarus Editor encodes all text as utf8 (*a convenient one-byte Unicode encoding scheme, widespread in the Unix world, but not the default encoding for most Windows applications*). The utf8 default encoding for characters in Lazarus strings means the programmer can (*using default Editor settings*) freely deal with and mix standard European alphabetical characters (s, ü, ß) and ligatures (æ), mathematical and currency symbols (±, £, ¥) and most non-European characters (Ë, , κ) without anxiety.

The main places where care has to be taken over string encoding issues is the passing of 'unusual' characters to **system** routines such as procedures to open files which may recognise only characters from the system codepage encoding, or retrieving strings from, or posting strings to (*mainly older*) databases.

If you find that RTL system routines you use are not behaving correctly with filenames, add the units `lazutf8` and `fileutil` to your project, and use a `UTF8xxx` routine that corresponds to the system routine. Eventually, when a robust Unicode string implementation is fully implemented throughout the RTL, FCL and LCL these codepage-related issues will be a thing of the past. A string variable can be indexed just as if it were an array. If `st` is a string then `st[j]` is the *j*th character in `st`, provided that *j* is positive and no larger than `Length(st)`. Each character in `st` is of type `ansiChar`. Assignments can be made to individual characters, or to the entire string. Here are some examples:

```
var st: string;
st:= ''; //empty string, Length(st) is 0
st:= ' '; //space character, Length(st) is 1
st:= 'Lazarus ' + 'and ' + ' Free Pascal'; //Length(st) is 24
st[1]:= 'L'; //overwrites 'l' with 'L', length is unchanged
```

To introduce yourself to some of the many string functions available, start a new console project in Lazarus as you have done earlier, named `string_functions`. Change the `uses` clause so it includes `strutils` and `sysutils`, add a string variable `st`, and insert lines into the body of the program so it looks like the following:

```
program string_functions;
{$mode objfpc}{$H+}

uses strutils, sysutils;
var st: string;
begin
  WriteLn('Enter a word or phrase ([Enter] completes the entry)');
  readln(st);
  WriteLn('You typed: ', st);
  WriteLn('Your text converted to uppercase: ', UpperCase(st));
  WriteLn('Your text converted to lowercase: ', LowerCase(st));
  WriteLn('Your text converted to proper case: ',
    AnsiProperCase(st, StdWordDelims));
  WriteLn('Your text reversed: ', ReverseString(st));
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Here we are using predefined functions that are not in the system unit, and so we need to explicitly refer to those units in a formal `uses` clause. The functions `UpperCase()` and `LowerCase()` are found in the `sysutils` unit, and the functions `AnsiProperCase()` and `ReverseString()` are found in the `strutils` unit. How do you know what unit a function is found in when you come to write the `uses` clause? If you have an idea of the routine's name (*but can't remember which unit contains its declaration*) one way to discover this information is to type the name of the routine in the Lazarus Editor, and then with the cursor located somewhere in the word press the [F1] key.

Chapter 4 STRUCTURED TYPES

Often the help system will locate the routine, and load help information which includes details of which unit contains that routine. Provided the spelling of the routine's name is correct the capitalisation does not matter.

4.f Records

Some structured types only combine data elements that are of identical type. Arrays are like this. Each array element is the same type as every other array element. Other structured types are more versatile, with the ability to combine either identical or disparate types in one entity. One such structured Pascal type is the **record**. Here is an example of a record declaration which stores an X and a Y coordinate value in a single point entity:

```
type TPoint = record
    X: longint;
    Y: longint;
end;
```

Here is a record declaration which combines a string and an integer, thus keeping a string and an integer always associated together:

```
type TCountry = record
    Name: string;
    Area: longint;
end;
```

Records cannot contain files, and should not be used for pointers (*or pointer-based types such as ansistrings*) if the record is to be saved, say to a disk file, and reloaded for use later, since these saved references will no longer be valid. Shortstrings can be used to persist string values in records which are to be retrieved later. Record typed constants are written in parentheses with the individual elements (*called fields*) introduced each time by the field name and a colon followed by the value, separated by a semicolon (just as in their declaration). So a TCountry record constant named Germany would be declared like this:

```
const Germany: TCountry = (Name:'Germany'; Area:356734);
```

We access specific sub-sections of a record (*which are called record fields*) using the "dot notation" RecordName.FieldName. For instance we could write:

```
var: country: TCountry;
begin
    country:= Germany;
    WriteLn('The country's name is: ',country.Name);
    WriteLn('The country's area is: ',country.Area);
end;
```

Let's write a small application that demonstrates how these versatile types can be used. Start a new console project in Lazarus named countries, saving it in a new folder called Countries. Delete the content of the uses clause Lazarus provides, and type the following content for the program.

```
program countries;
{$mode objfpc}{$H+}

uses strutils;

type
    TCountry = record
        Name: string;
        Area: longint;
    end;

const Belgium: TCountry = (Name:'Belgium'; Area:30513);
    Austria: TCountry = (Name:'Austria'; Area:83851);
    Finland: TCountry = (Name:'Finland'; Area:337032);
```

Chapter 4 STRUCTURED TYPES

```
procedure DisplayInfo(aCountry: TCountry);
var barLength: integer;
begin
  barLength:= aCountry.Area div 30000;
  writeln(aCountry.Name:8, aCountry.Area:7, ' ', DupeString('*', barLength));
end;

begin
  WriteLn(' Country   Area Relative area');
  WriteLn(' -----   ----  -----');
  DisplayInfo(Belgium);
  DisplayInfo(Austria);
  DisplayInfo(Finland);
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Having declared the TCountry record type we define three constants of that type, and then write a small helper procedure to display information from each constant. The data needed by the procedure is passed to it as a parameter named aCountry of TCountry type.

We use a formatting facility of WriteLn(), which will display data in a field N characters wide if you place the specifier :N after the data, enabling us to show the country's Name in an 8-character field, and its Area in a 7-character field. The final part of the display line is a string of * characters designed to give a rough indication of the relative sizes of the displayed countries. It uses the DupeString function found in strutils (which is why we needed a uses clause specifying strutils) which forms a string of duplicated characters. The length of the character string required is passed as the second parameter to this function. We calculate its value by a simple integer division of the Area field.

If you compile and run the program, you should see output something like the following:

```
Country   Area Relative area
-----   ----  -----
Belgium   30513  *
Austria   83851  **
Finland   337032  *****
```

4.g The with . . . do statement

An alternative to the "dot notation" for referring to a specific field within a record is the with statement. Given the record definition used above:

```
type
  TCountry = record
    Name: string;
    Area: longint;
  end;
```

the compiler knows of two field names, Name and Area.

Given a record variable aCountry, the with aCountry do statement marks out a block of code within which use of those field names is assumed to refer to aCountry without needing to repeat the record name explicitly every time.

In other words, the DisplayInfo(aCountry: TCountry) procedure shown above as

```
procedure DisplayInfo(aCountry: TCountry);
var barLength: integer;
begin
  barLength:= aCountry.Area div 30000;
  writeln(aCountry.Name:8, aCountry.Area:7, ' ',
  DupeString('*', barLength));
end;
```

Chapter 4 STRUCTURED TYPES

can also be written using a "with" statement as follows:

```
procedure DisplayInfo(aCountry: TCountry);
var barLength: integer;
begin
  with aCountry do
    begin
      barLength:= Area div 30000;
      writeln(Name:8, Area:7, ' ',
              DupeString('*', barLength));
    end;
end;
```

Used judiciously the **with** statement can reduce the verbosity of code, but it is best avoided when it leads to ambiguity in understanding the correct referent of a name.

Although the compiler is not confused by **with** (*the rules for determining the scope and resolving name references correctly are clearly stated in the FPC documentation*) in writing complex statements using with you can mislead yourself about the true referent of a name used. Thus a bug creeps in because a field is altered which you did not expect.

The **with . . . do** construct can also be used in referring to **class** and **interface** and **object** fields and properties as an alternative to using the explicit dot notation. It is frequently used to avoid inserting a temporary variable that is otherwise needed for a newly constructed class. The following example will be understood better after reading Chapter 8 about classes, but is typical of the sort of constructs you will encounter in the LCL:

```
procedure Form1.Button1Click(Sender: TObject);
begin
  with TreeView1.Items.AddFirst(aNode,
  'First Node') do
    ShowMessageFmt (
    'This new node is at level %d', [Level]);
end;
```

Here the new node inserted into `TreeView1` is anonymous. The **with** construct allows you to refer to it nevertheless and access its `Level` property. Without the **with** construct you would have to create a temporary `TTreeNode` variable and use dot notation to access the `Level` property:

```
procedure Form1.Button1Click(Sender: TObject);
var tmpNode: TTreeNode;
begin
  tmpNode:= TreeView1.Items.AddFirst(aNode, 'First Node');
  ShowMessageFmt ('This new node is at level %d', [tmpNode.Level]);
end;
```

Chapter 4 STRUCTURED TYPES

4.h Set types

A further type encountered frequently in the LCL is the **set** type, which is based on some ordinal type, often an enumerated or subrange type, though it can be a `Char` or `integer`. Sets contain zero, one or more values of their base type. They are simple collections containing elements all of one base ordinal type, and they are written between square brackets *[{set elements go here}]*. Sets are declared in this way:

```
type TSetName = set of TBaseType;
```

Pascal sets are quite limited in size, and can contain no more than 256 elements.

The elements in a set have no order, and any given set element is either present in the set, or absent. There is no meaning to a set element being present more than once in a set.

The only numbers associated with a set are

- its possible maximum capacity (*which depends on the range of its base type*), that is, the potential number of elements, if all possible elements are present.
- the number of current members of the set (*from which you can always determine the number of elements that are not currently members of the set*).

If you are familiar with a language that uses bitmasks (*because it lacks sets*) you will find that sets provide similar functionality, but are often easier to read.

The FPC compiler allocates 32 bytes of actual storage for set types, unless there are less than 32 elements in the base type, when a longint is used. So small sets are handled very efficiently, as well as having full language support. A set is constructed using `[]` to delimit the range of base type elements in the set. For example:

```
type TCharSet = set of Char;
const vowels: TCharSet = ['a', 'e', 'i', 'o', 'u'];
```

A further example comes from a very commonly used LCL class: `TFont`. This class includes a `Style` property of type `TFontStyles` which is a set of values based on the enumerated type `TFontStyle`. It is a common naming convention for a base enumerated type to be a **singular** word, and the set type which uses it as a base type to be the same word in the **plural**.

Here are the declarations of these types:

```
type TFontStyle = (fsBold, fsItalic, fsStrikeOut, fsUnderline); //enumerated type
TFontStyles = set of TFontStyle; //set of enumerated type
```

Some examples of how to employ this useful set type follow. (*Note that any property of a class is accessed using the "dot notation" class.property, which we saw used earlier to specify a field of a record*).

```
Font.Style:= []; //an empty set property - a 'normal' font style
Font.Style:= [fsBold]; //font is bold
Font.Style:= [fsBold, fsItalic]; //bold + italic font
Font.Style:= [fsItalic, fsUnderline]; //italic + underlined font
```

You see that the `Style` property (*which is a set property*) is simply written as a comma-separated list of the elements in the set, enclosed by square brackets `[element1, element2, ...]`.

If the set is empty the square brackets enclose nothing. Notice the convention (*used throughout the LCL*) of a two-character prefix prepended to each set element to indicate that it is a `Font.Style` set member (*the fs is for Font Style*).

There is built-in support in Pascal for handling various operations on sets.

This support includes:

- The **in** operator which tests for the presence of a base type element in a set. (*anElement in aSet*) is a boolean expression returning `True` or `False` depending on whether `anElement` is present in `aSet` or not.
- The operators `+` and `-` for adding element(s) to a set, or subtracting element(s) from a set.

Chapter 4 STRUCTURED TYPES

For example:

```
Font.Style:= Font.Style + [fsBold];  
Font.Style:= Font.Style - [fsUnderline];  
Font.Style:= Font.Style - [fsBold] + [fsItalic, fsStrikeOut];
```

- The procedures `Include()` and `Exclude()` which add or remove a single element of the base type of a set to (or from) a set. Because class properties cannot be passed as `var` parameters these two procedures cannot be used for set properties.

Here is an example of their use:

```
type TByteSet = set of byte;  
var lowPrimes: TByteSet = [2, 3, 5, 7, 11, 13];  
begin  
  Include(lowPrimes, 17);  
end;
```

The `*` operator which with two set operands yields the union of the two sets, i.e. a set which contains only elements which are present in both operands (which may yield an empty set).

```
type TByteSet = set of byte;  
var setA, setB, setC: TByteSet;  
begin  
  setA:= [0, 1, 2, 3, 4];  
  setB:= [3, 4, 5, 6, 7];  
  setC:= setA*setB; // setC is [3, 4]  
end;
```

- The `=` operator tests two sets for exact equality (*they contain exactly the same elements*) and the `<>` operator tests two sets for inequality (*they differ by at least one element*). The expression `(setA <> setB)` is `True` for the two sets as defined above.
- The `>=` operator which tests whether all members of the right hand set are also members of the left hand set. The `<=` operator tests whether all members of the left hand set are also members of the right hand set. The expression `(setC <= setA)` is `True` for the sets as defined above. Whereas the `in` operator tests inclusion of a single element within a set, these comparison operators test for the inclusion of an entire set within another set. So the expression `(setC <= setA)` can be read as “setC is completely included within setA”. For these two sets as defined above this statement (*and thus the expression in parentheses*) is `True`.

4.i Binary files

File types represent physical disk files, or files stored on some non-disk persistent medium.

A file is a binary sequence of a known type, the **base type** of the file (*which cannot be a pointer, dynamic array, ansistring, file or variant*). Binary files are usually based either on a simple type, an array type or a record type (*which can include shortstrings*). The compiler represents files internally as records.

Text files differ from binary files in that they are not based on a fixed-size base type, but rather on lines of text of variable length. Pascal does not provide support for random access to individual lines of text. Text files must be read sequentially. You must use the `TextFile` type for text files (*see the next Section*).

RTL file routines use the system encoding for file and pathnames, based on current system codepage settings, whereas the LCL uses UTF8 encoding throughout. This means encoding conversion routines are often needed when dealing with file and path names when calling RTL file routines from Lazarus programs. See Chapter 16, Section **d** for more details. The LCL/FCL also provides `LoadFromFile` and `SaveToFile` methods for many commonly used classes, and offers easy access to system file `Open` and file `Save` dialogs for GUI programs, which saves the programmer having to deal with encoding issues. Later chapters enlarge on this.

Chapter 4 STRUCTURED TYPES

There are two ways of declaring binary files:

```
var aTypedFile: file of {base type};
    aBinaryFile: file;
```

Typed files declared as file of {base type} are often based on byte or a record type.

Specifying the base type determines the size of data transferred during file reads and writes.

Several standard routines and the global variable `FileMode` apply to binary files.

The global system variable `FileMode` specifies whether files are opened only for reading (0), only for writing (1), or for both reading and writing (2).

The default value is 2 which means all files are opened for both reading and writing.

Commonly used file-related routines include:

```
Append, AssignFile, CloseFile, Eof, Erase, FilePos, FileExists,
FileSize, IOResult, Read, Rename, Reset, Rewrite, Seek, Write.
```

You use `AssignFile` to associate a file variable with the name of a disk file, and `Reset` to open an existing file, or `Rewrite` to create a new file. Programs should be protected in the event that file operations fail. The original Pascal method for this was to check the value of the `IOResult` function. A more modern approach to ensuring robust handling of file-related errors is to wrap file operations in a `try...finally...end` block (See *Chapter 6.h* and *Chapter 17.b* for more about exception handling).

Untyped files are declared as

```
var anUntypedFile: file;
```

No base type is specified in this declaration, and it is treated as a sequence of fixed-size records.

The record size is determined from the opening call to one of the file functions `Rewrite`

or `Reset`. The default record size is assumed to be 128 bytes unless specified otherwise.

All typed-file routines also apply to untyped files except for `Read` and `Write`.

These are replaced by `BlockRead` and `BlockWrite`, used for fast data transfer between untyped files and a memory buffer. For untyped files `Reset` and `Rewrite` allow an additional parameter specifying the size of the record used in data transfers (e.g. `Reset(f, 1)`).

Note that there are two `FileSize` functions. `Sysutils.FileSize()` returns the size of a file in bytes (as you would probably expect), whereas `System.FileSize()` returns the size in record units. For a file of `byte` the result is identical, but for a file of records or a file of `double` the result will differ.

A simple example of a program that stores customer data locally, and uses a procedure `ProcessCustomer()` to report on currently outstanding customer invoices (declared in another unit processing) is given here. It uses a `while` statement (covered in the following chapter) to loop through every record stored in the customer file.

Chapter 4 STRUCTURED TYPES

```
program binary_file;
{$mode objfpc}{$H+}

uses processing;

type TCustomerRecord = record
    ID: int64;
    Name: shortstring;
    BalanceOwed: currency;
    LastTransactionDate: TDateTime;
end;

TCustomerFile = file of TCustomerRecord;

var f: TCustomerFile;
    c: TCustomerRecord;
begin
    AssignFile(f, 'customer.dta');
    Reset(f);
    try
        while not EOF(f) do
            begin
                Read(f, c);
                ProcessCustomer(c);
            end;
        finally
            CloseFile(f);
        end;
    end.
end.
```

4.j Text files

Text files differ from binary files in that text files contain only lines of text where the length of each line can vary, and can consist only of ansichars. They are not formed from a repeating sequence of binary records as are binary files. Text files are declared as

```
var aTextFile: TextFile;
```

In addition to the typed file routines given above, text files have a few specialised routines including

`EOLn`, `ReadLn`, `SeekEOLn`, `SeekEOF`, `SetTextBuf`, `SetTextLineEnding`, `WriteLn`

Line endings differ between OSs, with Windows using #13#10, Unix using #10 and Mac using #13. Consequently you should always use the predefined `LineEnding` constant which takes care of these differences for you, rather than hard-coding values, which will make your code non-portable.

Pascal text files are not limited to data transfers to and from storage media.

Predefined text files `Input` (*read-only*), `StdErr` and `Output` (*write-only*) are defined in the system unit and refer to the standard input/output device which is normally the console, automatically opened under Unix. The `Input`, `Output` and `StdErr` textfiles are opened in the system unit's file initialization code.

Using `ReadLn` and `WriteLn` without specifying a `TextFile` variable will then read text from (*and write text to*) the console, as we have done in all console program examples so far.

To avoid errors under Windows you have to set console mode via the `{$apptype console}` compiler directive, placed in the main program file, if the Program project type has not been set at the time the project was created.

The following short console program demonstrates simple text file access.

It depends on you naming your console project `text_file.lpr`. Create a new console project in Lazarus, saving it with the program name `text_file`. Adapt the skeleton code to look like the following, and then compile and run the program.

Chapter 4 STRUCTURED TYPES

```
program text_file;
{$mode objfpc}{$H+}

uses sysutils;

var txtF: TextFile;
    s: String;

procedure ReadTenLines;
const lineNo: integer = 0;
var linesRead: integer = 0;
begin
    while not EOF(txtf) and (linesRead < 10) do
        begin
            ReadLn(txtF, s);
            Inc(linesRead);
            Inc(lineNo);
            s:= Format('Line %d: %s',[lineNo, s]);
            WriteLn(s);
        end;
    end;
end;

begin
    AssignFile(txtF, 'text_file.lpr');
    Reset(txtF);
    WriteLn('Lines from text_file.lpr will be displayed 10 at a time');
    WriteLn;
    try
        while not EOF(txtF) do
            begin
                ReadTenLines;
                WriteLn;
                WriteLn('Press [Enter] to continue');
                ReadLn;
            end;
        finally
            CloseFile(txtF);
        end;
        Write('End of file reached. Press [Enter]
            to finish');
        ReadLn;
    end.
```

As well as the **while** statement discussed in the next chapter this program uses the `Format()` function (from the `sysutils` unit) which takes a string value followed by an array of parameters. Each occurrence of the format specifiers `%d` or `%s` in the string is replaced by successive parameters from the array which follows it. For example:

```
aString:= Format('This string has the number %d inserted into it',[7]);
```

The variable `aString` will now contain *This string has the number 7 inserted into it.*

This is a useful way of formatting non-text-variable's data so it is converted correctly to a string fragment that is inserted at the correct place in the string. The Free Pascal documentation gives full details on possible format specifiers and their corresponding parameter types. The program also illustrates how a local typed constant in a procedure (or function) will act as a 'static variable', retaining its value when the procedure is called repeatedly.

4.k Review Questions

1. What are the main differences between a static data structure and a dynamic one?
2. Write a record structure that would be suitable for recording information about books you refer to.
3. Write a program that cleans a text file, replacing any tab, double or triple space by a single space character.



Chapter 5 EXPRESSIONS AND OPERATORS

5.a Operators: forming Pascal expressions

In addition to assigning single values to a variable, you can use **expressions** to combine several values into a single assignment using **operators** to make those combinations. The values operated on are often given the mathematical description of **operand**.

An **expression** is any valid combination of literal value, constant, variable, or function result using appropriate operator(s). Functions are dealt with more fully in the following chapter.

Operators include both symbols familiar from school arithmetic such as +, = and -, and combination symbols such as << and <> that may be less familiar, as well as short words such as `div`, `mod`, `and`, `not`, and `or`. The list of possible operators available for all possible types is quite long. Most operators are applicable to just a few Pascal types, though many are **overloaded**. This means that the same operator can be used in analogous ways on quite different types. For instance, it is no surprise that the addition operator + is used to add two **numerical** values together. If `newCount` and `count` are longint variables we can assign a new value to `newCount` like this:

```
newCount := 37 + count;
```

The addition operator can also be applied to characters and strings. So a string variable `newWord` can be assigned a value like this:

```
newWord := 'O' + 'K'; //newWord now holds the string 'OK'
```

Used with set types the + operator can also be used to add two sets together. The + operator can also be used to perform arithmetic on pointers. Pascal operators are powerful and versatile. 'Addition' in Pascal covers mathematical addition, string concatenation, set addition and pointer manipulation, all using the one + symbol for ease of recognition.

Free Pascal also allows user-defined operators, though we shall not explore that in a book designed for beginners. This allows you to use the + operator to correctly combine vectors, matrices and imaginary numbers, for instance.

5.b Mathematical operators

Mathematical operators that apply to numerical values are listed in the Table 5.1.

Operator	Operation	Valid operands	Result type	Example expression
+	addition	integer, real	integer, real	m + n
-	subtraction	integer, real	integer, real	Result - m
*	multiplication	integer, real	integer, real	n * Pi
/	real division	integer, real	real	m/3
div	integer division	integer	integer	count div 10
mod	integer remainder	integer	integer	m mod 17
+	sign identity	integer, real	integer, real	+645
-	sign inversion	integer, real	integer, real	-m
**	exponentiation	int64, float	int64, float	2**3 (needs math unit)

Table 5.1 Mathematical operators applicable to numerical operands

Brackets () are used just as in algebra to group items within a numeric expression, or to override the normal rules of operator precedence, or simply to avoid ambiguity in evaluating an expression. For instance:

```
sum := 2 + 3 * 4; //14 is assigned to sum since * precedes + when evaluated
```

```
sum := (2 + 3) * 4; //operator precedence overridden by (), 20 is assigned to sum
```

Chapter 5 EXPRESSIONS AND OPERATORS

5.c Boolean operators: not, and, or, xor

The **not** operator is unary, meaning it works on only one boolean operand which it negates, reversing the value of that operand. Thus **not True** is equal to **False**, and **not False** is equal to **True**.

Here is an example of the not operator in use:

```
var active: boolean;  
begin  
    active := not True; // this is equivalent to False  
end;
```

Boolean variables are often used to monitor the state of some condition. For instance a boolean variable named **Finished** can be used to monitor a process in this way:

```
var Finished: boolean = False;  
begin  
    while not Finished do  
        begin {some action that will eventually set Finished to True} end;  
end;
```

Binary operators combine two operands in an expression. The three binary boolean operators are listed in the last three columns of Table 5.2, together with the result of evaluating the boolean expressions each operator performs, given two boolean variables A and B. The table gives outcomes for all permutations of possible values for A and B. Although you can write boolean expressions using **xor**, it is clearer for most readers to write the same expression using the “not equal to” operator, **<>**.

Operator:	Operation:	and Logical and	or Logical or	xor Logical exclusive or
A	B	A and B	A or B	A xor B {same as A <> B}
True	True	True	True	False
False	False	False	False	False
True	False	False	True	True
False	True	False	True	True

Table 5.2 Boolean operators and their effects

5.d Comparison (relational) operators

Relational operators are all binary, used to compare two operands. Their principal use is for comparison of ordinal types, but some relational operators also operate on strings, sets, class references and pointers. It is not recommended to use these operators on strings, however. This is because they make a crude comparison based on the character code representation, taking no account of case sensitivity or encoding. There are more sophisticated routines offering finer control when making string comparisons (*located in the sysutils and strutils and lazutils units*) which are preferred in nearly all situations.

When you compare ordinal operands they must be of compatible types, except that integer and real types can be mixed in comparisons. Most of the comparison operators available in Free Pascal are listed in Table 5.3.

Operator	Comparison operation	Result type	Example expression and its value
=	equality	boolean	True=False // False
<>	inequality	boolean	True<>False // True
<	strictly less than	boolean	10 < 2.0 // False
>	strictly more than	boolean	10 > 2.0 // True
<=	less than or equal to	boolean	10 <= 10 // True
>=	more than or equal to	boolean	10 >= 11 // False

Table 5.3 Relational operators applicable to ordinal types

Chapter 5 EXPRESSIONS AND OPERATORS

5.e Bitwise (logical) operators

Even the smallest integer types (`byte` and `shortint`) contain 8 bits of sub-byte information. The bitwise operators operate at a low level within these types allowing the programmer to alter individual bits within that type, giving the same access to the individual bits within a variable that is otherwise restricted to assembly language constructs.

The `not` operator is unary, and reverses the bits of its operand – any bit that was 0 becomes 1, and any bit that was 1 becomes 0. The other bitwise operators require two operands. Unless you are accustomed to thinking in terms of the underlying bit representation of integers, these operators are not immediately useful, and can lead to mysterious-looking results, because there is a clear distinction between signed and unsigned integers at the bit-storage level which is not apparent to the casual observer. For instance the expression `not 0` (*correctly*) yields the value -1, which to many people is not the obvious outcome of the operation.

Some programmers encourage the use of the shift operators `shr` and `shl` as a faster alternative to regular division and multiplication when powers of two are involved. However this is not always a good idea. If there is a speed differential it is usually insignificant, and these operators do not check for overflow errors. Any bits shifted outside the storage allocated for the variable are lost, which does not matter for right shifts, but usually indicates a programming error if it occurs during left shifts. If you use ordinary division and multiplication operators (*and range checking is turned on*) the compiler will make sure errors of this sort are picked up. Of course if you understand what you are doing these low level bitwise operators are another useful weapon in your programming armoury.

The available binary bitwise operators are listed in Table 5.4

Operator	Operation	Type of operand and result	Example expression
<code>and</code>	Bitwise and	integer	<code>0 and 1 // 0</code>
<code>or</code>	Bitwise or	integer	<code>0 or 1 // 1</code>
<code>xor</code>	Bitwise exclusive or	integer	<code>0 xor 1 // 1</code>
<code>shr, >></code>	Bit shift to the right	integer	<code>16 >> 2; 16 shr 2 // 4</code>
<code>shl, <<</code>	Bit shift to the left	integer	<code>3 << 2; 3 shl 2 // 12</code>

Table 5.4 Binary bitwise (logical) operators

5.f A program example: simple_expressions

Create a new console project in Lazarus as you did at the end of Chapter 3, naming it `simple_expressions`. Again, delete the uses clause entirely, and enter code so that your program source matches the following listing:

```
program simple_expressions;
```

```
{ $mode objfpc } { $H+ }
```

```
var b: Byte = 4; c: Byte = 3;
```

```
begin
```

```
  writeln('Initial value of b is ',b,', initial value of c is ',c, sLineBreak);
```

```
  b:= b shr 1;
```

```
  c:= c shl 4;
```

```
  writeln('After b shr 1, b is ',b,', after c shl 4, c is ',c, sLineBreak);
```

```
  writeln('c xor b = ',c xor b,', c and b = ',c and b,', c or b = ',c or b);
```

```
  writeln('not c is ',not c,', not b is ',not b);
```

```
  writeln;
```

```
  writeln('<b> is ',c>b,', c<b> is ',c<b,', c<<b> is ',c<<b,', c=b is ',c=b);
```

```
  writeln();
```

```
  writeln('c div b is ',c div b,', c mod b is ',c mod b);
```

```
  { $IFDEF WINDOWS }
```

```
  ReadLn;
```

```
  { $ENDIF }
```

```
end.
```

Check the values displayed by the expressions used in the above program. Are they what you would have predicted?

Chapter 5 EXPRESSIONS AND OPERATORS

5.g Review Questions

1. If `sugar` has the value `$00` and `spice` has the value `$FF`, what is the value of `sugar or spice`?
2. What would be the value of the expression
`((24 div 3) shr (5-3) = 27 mod 5)`?
3. In Pascal what is the difference between `a=b` and `a:=b`?



Chapter 6 PASCAL STATEMENTS

A Pascal statement prescribes some algorithmic action to be carried out. Statements are of two kinds:

- simple statements (*which do not contain other statements*)
- structured statements (*which are built from simpler statements*)

The following simple statements have already been introduced:

- **assignments**, such as `anInteger := -3;`
- **function** calls, such as `aDate := Date();`
- **procedure** calls, such as `WriteLn('This is a procedure call');`

We have also introduced the idea of structured statements occurring in the **main block** of a Pascal program where the statements are structured by being enclosed within the keywords `begin . . . end.`

```
Program ProgramName;  
begin  
    {a sequence of statements goes here}  
end.
```

Statement blocks more frequently terminate with a semicolon (*rather than the dot which ends a Pascal program*) and are part of subroutines or subsidiary units used by the main program:

```
begin  
    aFahrenheit := 9*aCelsius/5 + 32;  
    lastWeeksDate := Date() - 7;  
    WriteLn('This block of Pascal statements is executed sequentially');  
end;
```

The `with {record/class/interface/object variable} do` statement was introduced in Chapter 4 at Section g.

Two types of statement (*Goto and asm statements*) are not considered in this book for the sake of brevity, since it is possible to write almost any Pascal program without their use, and they occur only infrequently in the sources.

The FPC documentation covers them in full if you wish to learn about them and use them.

The structured statements considered in this chapter are:

- **conditional statements**
 - `if then else;`
 - `case of else end;`
- **looping (repetitive) statements**
 - `for to do ;`
 - `for downto do ;`
 - `for in do ;`
 - `while do ;`
 - `repeat until ;`
- **exception statements**
 - `raise ;`
 - `on do ;`
 - `try except end;`
 - `try finally end;`

Chapter 6 PASCAL STATEMENTS

Note: You can refer to the Free Pascal documentation for fuller details about all aspects of the Object Pascal dialect Lazarus uses. A simplified view of the basic ideas is given here, often leaving out more advanced aspects of the syntax, further options available and so on. The online documentation is continually updated to reflect the latest improvements in the FPC. The documentation often includes example code which you can try out.

You can obtain the Language Reference documentation from these and other mirror sites

<ftp://ftp.freepascal.org/pub/fpc/docs-pdf/ref.pdf>

<http://www.freepascal.org/docs-html/ref/ref.html>

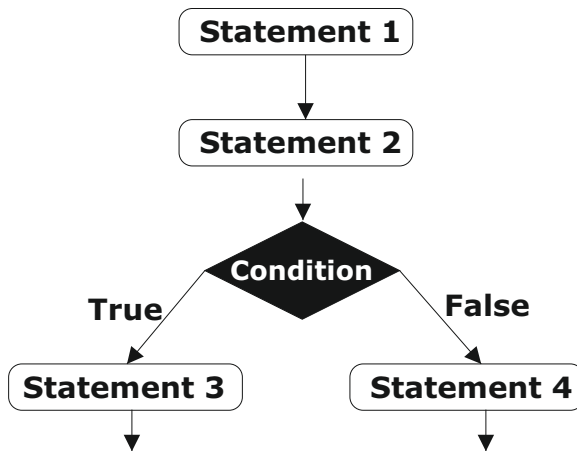
<http://sourceforge.net/projects/freepascal/files/Documentation/2.6.0/>

6.a Conditional statement: if

The order in which statements are executed in a program is termed the **flow of control**. Unless told otherwise the CPU will process the statements you write one by one in the order they are written, i.e. in a linear sequence.

Statement 1 → Statement 2 → Statement 3 → Statement 4 → Statement 5 → etc.

For the computer to choose between different paths of execution you must use a **branching** control structure in which a **condition** (a *boolean variable or expression*) is interposed which forces a selection between alternative paths of execution.



Here the flow of control is interrupted for a condition to be tested. Which statement is executed thereafter depends on the outcome of the condition. If the condition tests `True` Statement 3 is executed, otherwise Statement 4 is executed. The syntax of the `if` statement is

```
if (boolean expression)
  then statementA
  else statementB;
```

The computer chooses just one of the two alternatives `statementA` or `statementB`, according to the value of the `(boolean expression)` which does not have to be placed in parentheses, though in some cases that can aid readability. Here is a simple example:

```
if OvertimeApplies
  then pay:= hours*overtimeRate
  else pay:= hours*normalRate;
```

Chapter 6 PASCAL STATEMENTS

Note that no semicolon is allowed except at the very end of the if statement. The else part is optional.

If statements can be nested one inside another. For instance:

```
if day=1
then writeln('Sunday')
else if day=2
  then writeln('Monday')
  else if day=3
    then writeln('Tuesday')
    else if day=4
      then writeln('Wednesday')
      else if day=5
        then writeln('Thursday')
        else if day=6
          then writeln('Friday')
          else writeln('Saturday')
```

However, it soon becomes a maze to sort out which conditions apply to which parts of a complex sequence of **if** statements. In particular it can be hard to figure out when there are several **else** sections which **if** each **else** refers to.

The indentation used in the example above helps a lot. However, many programs that use multiple nested **ifs** don't provide such helpful formatting (*or it may have become messed up*). Where you need to test several conditions it is usually better to use the following conditional statement, designed for selection between several **cases**.

6.b Conditional statement: case of end

The **case** statement is a control structure that allows you to list any number of branches. It is similar to a series of nested if statements, but usually easier to read, and shorter to write. Each possible branch of code execution has a **case label**.

The **case selector** (*the initial expression which is tested*) must match one of the case labels in order for the labelled statement to be executed. It has the syntax

```
case {ordinal expression, or string expression} of
  label1: statement1;
  label2: statement2;
  . . .
else/otherwise statementElse;
end;
```

In the running program the case selector expression is evaluated and compared with each label in turn. If there is an exact match, the corresponding labelled statement is executed.

If no label matches the selector expression, and an **else** (*or otherwise*) statement is present, then the **else** is executed. If there is no **else** section, and no matching label, then none of the case statement's execution pathways is followed, and execution continues with the statement following the case construct.

As with the **if . . . then . . . else** statement, the **else** part in a **case** statement is optional. Every label constant's type must match the type of the selector expression. The label can be a range of values, which do not have to be contiguous. However no label value can be repeated explicitly within another label (nor implicitly because the label is included within the range of another label).

The statement corresponding to each case label can be a compound statement (*such as a statement sequence bounded by **begin** . . . **end***). Unlike in the **if then else** construct where no semicolon separator is allowed between the various parts, in the **case** statement semicolons are **required** to separate each labelled statement from the next one.

The recent FPC extension allowing strings (*not just single characters*) as case labels is a welcome innovation, making the **case** statement more versatile than ever.

The possibility of using **otherwise** as a synonym for **else** aligns FPC with ISO Pascal. However, the term **otherwise** is not found in the Lazarus sources (*except in comments*). After all, who would write a 9-character word when the 4-character **else** suffices nicely? Here is a short example which also demonstrates the **repeat . . . until** statement (*discussed more fully later in Section g*).

Begin a new console project in Lazarus, saving it as `keypress`. Adapt the skeleton code Lazarus writes to look like the following:

```

program keypress;

{$mode objfpc}{$H+}

procedure KeyPress(Key: Char);
begin
  case upCase(Key) of
    '0'..'9':      WriteLn('Key ',Key,' is numeric');
    'A','E','I','O','U': WriteLn('Key ',Key,' is a vowel');
    'B'..'D','F'..'H','J'..'N','P'..'T','V'..'Z':
      WriteLn('Key ',Key,' is a consonant');
    else
      WriteLn('Key ',Key,' is not alphanumeric');
    end;
end;

var s: string;
begin
  WriteLn('Press a key, then [Enter], (or [Enter] alone to finish)');
  repeat
    ReadLn(s);
    if s<>' ' then KeyPress(s[1]);
  until s='';
end.

```

Notice how we test not `Key` (*as typed by the user*), but `upCase(Key)`, capitalising all characters before testing. This means we don't have to include tests for 'a', 'b', 'c' etc. in the case labels of the `KeyPress()` routine, and tests for 'A', 'B', 'C' etc. are sufficient.

6.c Looping statement: for to do

Commonly there is a need to perform an operation repeatedly a known number of times. To repeat code instructions a specified number of times in Pascal you use the **for** loop. This requires an integer counter which is given a range from a starting value `start` to an ending value `finish`, and which is automatically incremented each time the loop executes. The **for** loop is declared like this

```

var i, start, finish: integer;
begin
  start := 1; // assign a starting value
  finish := 10; // assign a finishing value
  for i := start to finish do
    begin
      {some action we want performed ten times}
    end;
end;

```

The control variable (*i in the above example*) must be an ordinal type, and the compiler will not allow you to assign any value to it during the execution of the loop (*you can read it, but not change it*). This is because the compiler controls its value, incrementing it by one during each iteration of the loop, until it reaches the value of the upper finish bound.

In a **for to do** loop `start` must be less than `finish` if the loop is to execute at all; and the control variable is incremented at the end of each iteration.

Chapter 6 PASCAL STATEMENTS

6.d Looping statement: for downto do ; Break and Continue

In a **for downto do** loop *start* must be greater than *finish* for the loop to execute at all, and the control variable is **decremented** by one after each loop iteration.

Suppose you need a function that will return just the last word of a text string.

Such a function could be coded using a **for downto do** loop as in the following short test program which exercises such a function. Begin a new Lazarus console project named `last_word`, and adapt the program skeleton to look like the following.

```
program last_word;

{$mode objfpc}{$H+}

type TCharSet = set of Char;

function LastWord(const aPhrase: string; separators: TCharSet): string;
var L, p: integer;
begin
  L:= Length(aPhrase);
  if (L=0) or (separators=[]) then Exit('');
  for p:= L downto 1 do
    if not (aPhrase[p] in separators)
      then Result:= aPhrase[p] + Result
      else Break;
end;

var s: string;
begin
  repeat
    Write(
      'Enter a phrase (or nothing to Quit): ');
    ReadLn(s);
    WriteLn('The last word of "',s,'" is: "',LastWord(s, [' ']),'");
  until (s='');
end.
```

The compiler does not allow you to alter the value of the control variable in a **for** loop (*p* in the program above), but Free Pascal provides two useful procedures that allow you to skip some of the statements in a loop (*Continue*), or to exit completely from the loop prematurely (*Break*).

These two procedures can be used inside all loop constructs (**for**, **repeat** and **while**).

In the above program the *Break* instruction causes program execution to jump out of the **for** loop whatever the value of the control variable *p*.

6.e Looping statement: for in do

The **for to do** loop is limited to use of ordinal control variables. The **for in do** loop allows you to enumerate a number of varied types providing they have a fixed number of elements, including ordinals, arrays, sets, strings and enumerable classes. The control variable has to correspond appropriately to the base element of the type being enumerated. The following short program illustrates the **for in do** statement used with an enumerated type.

```
program religions;

{$mode objfpc}{$H+}

Type
  TReligion = (Bahai, Buddhism, Christianity, Confucianism, Hinduism, Islam,
              Jainism, Judaism, Shinto, Sikhism, Taoism);
Var r : TReligion;
begin
  WriteLn('Major world religions include the following:',LineEnding);
  for r in TReligion do
    WriteLn(r);
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

Chapter 6 PASCAL STATEMENTS

6.f Looping statement: while do

Whereas the **for** loop executes a fixed number of times, the **while** and **repeat** loops execute a variable number of times, according to a boolean control expression. In the **while** statement the loop control expression is tested before entering the loop (so the **while** loop might not run even once). In the **repeat** statement the control expression is tested at the end of each loop (so **repeat** statements execute at least once). The syntax for the **while** statement is

```
while {boolean expression} do {statement};
```

Usually the *{statement}* part of the construct is a compound statement bounded by **begin...end**. For example, the number of lines in a newly opened text file can only be known by reading the file from beginning to end, incrementing a line count as each new line is encountered, and checking before reading another line that the end of the file has not been reached yet. Given a newly opened text file, `txt`, and two procedures `ReadAFileLine()` and `ProcessAFileLine()` then a **while** loop to read and process the entire contents of the file would be as follows:

```
try
  while not EOF(txt) do
  begin
    ReadAFileLine();
    ProcessAFileLine();
  end;
finally
  CloseFile(txt);
end;
```

The text file function `EOF(txt)` returns a boolean value which indicates whether the file-read-cursor has reached the end of the file or not.

6.g Looping statement: repeat until

The **repeat** statement is analogous to the **while** statement, except the test for loop repetition is done at the end of the repeating loop, not at the beginning. While statements usually require **begin end** delimiters to mark which sequence of statements the **while** loop should process. The syntax for **repeat until** encloses the repeated statements, so no additional **begin end** is needed. The statement is written in the form

```
repeat
  {statements}
until (booleanExpression);
```

The `last_word.lpr` program given above in Section 6.d demonstrates use of the **repeat** loop.

6.h Exception statements: raise, on, try

Exceptions let you interrupt the normal flow of control in a program, and can be raised in any routine or method. The exception causes control to jump to an earlier point in the routine, or perhaps to an earlier routine further back in the call chain. Where a **try** statement is found the compiler causes that code to execute, processing the exception.

The **try . . . except** construct is used to deal with errors, and the **try . . . finally** construct is used to properly clean up resources (*usually allocated memory or opened files*) no matter what errors might occur. If no **try** constructs are found anywhere in the call stack, in GUI programs the `Application` instance handles the exception.

Most beginners do not need to consider raising exceptions, only how to respond to exceptions arising, say from divide-by-zero errors. Generally it is only low level code and utilities that need to communicate an exceptional state by raising an exception, and usually you will find that such code is already part of the RTL, FCL and LCL.

Chapter 6 PASCAL STATEMENTS

The higher level code most programmers write to do with the UI will use **try** statements (*rather than raise statements*), or perhaps an `Application.OnException` handler to catch and handle exceptions. Because raising and handling exceptions takes time and processor resources, they should be used only for truly exceptional conditions (*system file not found, for instance*). Rather than give lots of detail about the syntax of the exception statements (*which are covered in full in the FPC documentation*) an example follows illustrating their use.

To demonstrate the concepts involved, imagine you have a weather station that records data at hourly intervals, logging recorded measurements to a text file `wind_data.txt` where each line contains wind speed readings recorded that day. You write a function

```
type TIntegerArray = array of integer;
function WindSpeedArray(const aLine: string; var aDay:TDateTime):TIntegerArray;
```

that parses each line in the file, storing the date of the readings in the `aDay` variable, and returning an array containing the actual readings for that day. To make the processing of the file robust, you might code the procedure that does the processing something like this over-simple console program named `windspeed.lpi`:

```
program windspeed;
```

```
{$mode objfpc}{$H+}
```

```
uses sysutils;
```

```
type TIntegerArray = array of integer;
```

```
function WindSpeedArray(const aLine: string;
var aDay: TDate): TIntegerArray;
begin
  // actual line parsing code should go here - instead we return dummy data
  aDay:=Now;
  SetLength(Result, 3);
  Result[0]:=25; Result[1]:= 23; Result[2]:=17;
end;
```

```
procedure DisplayAverageWindSpeed(anArray:
TIntegerArray; aDay: TDate);
var sum: integer=0; i, avg: integer;
begin
  for i:= Low(anArray) to High(anArray) do Inc(sum, anArray[i]);
  avg:= sum div Length(anArray);
  writeln('Average wind speed for ',DateToStr(aDay),' is ',avg);
  SetLength(anArray, 0);
end;
```

```
var windData: TextFile; d: TDate;
s: string; wsa: TIntegerArray;
```

```
begin
  AssignFile(windData, 'wind_data.txt');
  Reset(windData);
  try
    while not EOF(windData) do
      begin
        ReadLn(windData, s);
        wsa:= WindSpeedArray(s, d);
        if Length(wsa) = 0 then
          raise Exception.CreateFmt('No data for %s: average cannot be computed',
            [DateToStr(d)])
          else DisplayAverageWindSpeed(wsa, d);
        end;
      finally
        CloseFile(windData);
      end;
    {$IFDEF WINDOWS}
    ReadLn;
    {$ENDIF}
  end.
```

Chapter 6 PASCAL STATEMENTS

If you compile and run this program in Lazarus (*with the debugger enabled*) you should get an exception “File not found” (*see Figure 6.1*). Without the debugger the outcome depends on your platform.

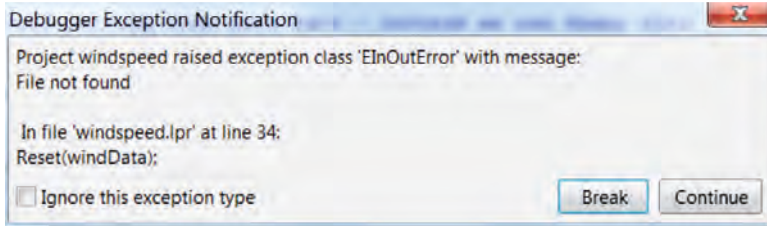


Figure 6.1 The Lazarus debugger catching an exception in windspeed.lpr

To avoid the exception, create a new text file in the same directory as the project (**File | New | Text**), and save the file as `wind_data.txt`, typing a few characters in the first line of the file in the Lazarus Editor so there is a fragment of text there for the `ReadLn()` to read. It does not matter what you type since the actual data read is discarded in this simple example, but the file can now be found.

The `DisplayAverageWindSpeed()` routine calculates an average from the array of integers passed as a parameter to the routine. If the array is empty, the routine would attempt to divide by zero. To prevent that we test the length of the array in the main program block, raising an exception if it is zero. This is to provide an (*admittedly rather artificial*) example of using the `raise` statement. An alternative (*better*) way to catch such an exception would be to wrap the calculation of the average in the `DisplayAverageWindSpeed()` routine in a **try except** statement that is designed to catch the specific exception we anticipate might arise, like this:

```
try
  avg:= sum div Length(anArray);
except on EDivByZero do
  avg:= 0;
end;
```

The `EDivByZero` exception is an exception class defined in a `sysutils` include file, along with a number of other exceptions that have self-explanatory names. Using the higher level **try except** statement rather than the lower level `raise` statement is usually a better way of working with exceptions.

The **try except** alternative, if used here, would not even cause a pause in the program, or show an error dialog. It will simply report an (*unexpected*) value of 0 for the average. This is much more user-friendly.

Chapter 6 PASCAL STATEMENTS

6.i Review Exercises

1. Write a function using a **for** loop that returns the location of the first space character
function PositionOfFirstSpace(s: **string**): integer;
in a string, or 0 if no space is found.
2. Write a routine that deletes a file
procedure DeleteFile(aFilename: **string**);
which raises an exception if the file is not deleted
(you could use the `system.Erase()` procedure).



Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

Just as Chapter 3 introducing types was mainly about **data**, so this chapter introducing routines is mainly about **code**.

7.a Routines and methods

Following the modular design philosophy of Pascal, procedures and functions are named sections of code that can be used (or *called*) simply by inserting their name at the appropriate point in your code. Collectively procedures and functions are known as **routines**. The simple statement of the named routine invokes all the code associated with that routine. It does not have to be rewritten at that point in your program. Citing the name alone is sufficient to invoke the entire code routine. Where procedures and functions occur within classes they are known collectively as **methods**.

The many predefined Pascal routines have been written by experts and debugged over many years of use. It is far more reliable (*and quite likely faster*) than any code we might write ourselves, which means that a large part of learning to program is learning the names and capabilities of the routines provided in the libraries that accompany the language you are learning. It is usually both more productive and faster to learn how to use reliable routines written by experts than to stumble through attempts at reinventing the same wheel ourselves that others have already crafted and placed in one of the three libraries that come with Lazarus (*RTL, FCL, LCL*).

Procedures (*and functions also*) may have **local** declarations of variables, constants or nested procedures (*and/or functions*) that are **private** to that procedure.

Here the terms **local** and **private** mean that the declared variables (*or other locally declared items*) are used only within the procedure itself. They are not **visible** to the rest of the program, i.e. cannot be accessed from outside of the procedure. This is the meaning of **privacy** generally in Pascal. Nothing outside is allowed to interfere with the inner working of code that is private. From the perspective of the rest of the program a procedure is a **black box**. It does its named thing, but the program does not care how. The only **interface** the procedure exposes to the program is its name and parameter list.

7.b Calling a routine

A **function** is a self-contained block of Pascal statements that returns a value at the end of its execution, and consequently all functions have a defined type which appears at the end of their declaration, following a semicolon. Since it returns a value, a function can be used as part of an expression in assignments, but can only appear on the **right hand side** of an assignment statement. For example:

```
aVariable:= SomeFunction();
```

It is also possible to invoke a function without assigning its value to a variable. In this case the value is thrown away:

```
SomeFunction();
```

A **procedure** is analogous to a function, with the difference that it does **not** return a value, so its name cannot be assigned to a variable, since it does not represent a value, only a sequence of code that is being executed. A procedure is called just as a function is called:

```
SomeProcedure();
```

Both procedures and functions can call themselves recursively, though in real life surprisingly few actually do (*which may help beginners for whom the idea of recursion can be confusing when first encountered*).

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

7.c Passing data to a routine: parameters

Both functions and procedures may have **parameter(s)**, that is, data which is processed in the course of their execution. Parameters are sometimes given the alternative designation of **arguments**. Any such required data is passed to the routine when it is called, and the data is given in a fixed sequence of parameters shown in parentheses (*brackets*) following the name of the routine.

The simplest routines have no parameters. For example, the `Pi` function which is declared (*almost*) like this

```
function Pi: double;
```

needs no parameters. You can call it (*use it*) in your code simply by writing its name, `Pi`, like this:

```
var r, circumference: double;
begin
  circumference:= 2 * Pi * r;
end;
```

Or you can call it by writing `Pi ()` as a function name followed by an empty parameter list `()`. The empty parameter list `()` emphasises the fact that `Pi` is a function (*not a predefined constant*). Most routines require parameters. Consider the need to discover the square root of a number. Rather than writing out the complex code required to determine the square root of a number repeatedly each time you need a square root, Pascal provides a routine which you can call, a function that returns the needed value. Such a function requires some input data, namely the number for which the square root is to be calculated. Accordingly the function definition includes a parameter which is a placeholder for this bit of data. The `Sqrt` function is declared (*almost*) like this:

```
function Sqrt(d: double): double;
```

A square root is meaningless without specifying what number we want the square root of, which is the number passed to the function as a parameter. The availability of the `Sqrt` function allows us to write the following:

```
var golden_section: double;
begin
  golden_section:= (1 + Sqrt(5))/2;
end;
```

7.d Picking up the value returned from a function

There are literally thousands of useful functions ready to be used in the LCL, FCL and RTL. However, you will always come across a situation where there is no pre-existing function that does what you want (*or perhaps there is, but you cannot find it*). Suppose we need to calculate the volume of a sphere. We could write our own function `VolumeOfSphere ()` which would look like this:

```
function VolumeOfSphere(aRadius: double): double;
begin
  VolumeOfSphere:= 4*Pi()*aRadius*aRadius*aRadius/3;
end;
```

Notice how the expression evaluating the volume is **assigned to the function name** to give it its final value. An alternative is to use the predefined `Result` value, which FPC provides for any function you write (it is automatically of the right type). `Result` behaves as if it were a hidden `var` parameter which is undefined before entry to the function. The section following this one provides more details about `var` parameters.

So the same function could also be written as:

```
function VolumeOfSphere(aRadius: double): double;
begin
  Result := 4 * Pi * aRadius * aRadius * aRadius / 3;
end;
```

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

The very slight differences in coding style are a matter of personal preference. Both styles are valid Pascal (*and of course there are other stylistic variations which are also valid*).

If a function requires more than one parameter, then each parameter is separated from its successor by – you guessed it – a semicolon. There is an “a” prepended to “Radius” to make the parameter name `aRadius`. This is a common convention which enables you, as you read through the code, to distinguish very easily the parameters supplied to a function from other variables used in the function body.

7.e Parameter classification: var, const, out

Parameters passed to functions and procedures are either **value**, **variable**, **constant**, or **out** parameters. The reserved words `var`, `const` and `out` indicate variable, constant and out parameters and, if used, come immediately before the name of the parameter.

Value parameters are the default and have no special specification. Parameters are assumed to be value parameters unless specified otherwise (*this is the meaning of “default”*). Value parameters transfer a copy of an actual value to a routine. Such as in this declaration of the power function:

```
function power(base, exponent: float): float;
```

Here the two parameters required are passed as value parameters. The values could be passed as literal numbers. However, let's assume that we have two float variables in our program named `bas` and `exp`. They acquire values and are then passed to the `power` function which is used to assign a new value to a third floating point variable `floatVar` as follows:

```
floatVar:= power(bas, exp);
```

Inside the power function the two parameters `bas` and `exp` (*since they are value parameters*) are copied to temporary variables. Inside the function these copies of the parameters may or may not be changed. However, when the function completes and returns its value (*which is assigned to floatVar*), the temporary copies of `bas` and `exp` are lost. The original variables `bas` and `exp` passed as parameters remain unchanged. This is the meaning of value parameters. Within the called routine the parameters can be changed, but if any such changes are made they actually happen to **copies** of the original parameters (*created inside the routine by the compiler*), so the value parameters themselves are always unchanged after the function call.

Var (or variable) parameters are passed by **reference**, meaning that no copy is made but the routine **acts on the original value** which might be changed by the routine as it executes. This allows a procedure to return value(s) as existing variables that the procedure might alter, and it allows a function to return value(s) in addition to its `Result`.

Consider the function `OffsetRect` which moves a rectangular area of type `TRect` (*specified by the var parameter ARect*) by an X delta `dx`, and a Y delta `dy`. Its declaration is as follows:

```
function OffsetRect(var ARect: TRect; dx, dy: Integer): Boolean;
```

After the function call `ARect` has changed – it is now in the new position (*unless the function result is False, which means you used a dx or a dy value that would have moved it to negative coordinates*).

Constant, `const`, parameters are like value parameters with the restriction that they cannot be assigned a value in the body of the routine (*or passed as var parameters to another routine*). This sometimes allows for certain compiler optimisations to be performed on code using `const` parameters. For large structured types such as strings this can save time-consuming copies from being made.

Out parameters, `out`, are `var` parameters passed only as output containers, i.e. they can be **uninitialised** variables. Their purpose is to pass information **out** of a procedure (*like the Result variable of a function*) rather than to pass information **in** to a procedure.

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

7.f Default parameters

Pascal allows you to declare default parameter values for simple types (*and also for string types*) by appending a `=value` phrase to the parameter type. If you call the routine without giving the parameter a value, the compiler then supplies the default value provided in the declaration.

This means you can declare an apparently overloaded procedure such as the following:

```
uses Dialogs, SysUtils;
```

```
procedure Show3Msg(s1: string; s2: string=''; s3: string='');
```

implementation

```
procedure Show3Msg(s1: string; s2: string; s3: string);
```

```
begin
```

```
  case s3=EmptyStr of
```

```
    True : case s2=EmptyStr of
```

```
      True : ShowMessage(s1);
```

```
      False: ShowMessage(s1+LineEnding+s2);
```

```
    end;
```

```
    False: ShowMessage(s1+LineEnding+s2+LineEnding+s3);
```

```
  end;
```

```
end;
```

You can then call the procedure with one, two or three string parameters, e.g.:

```
Show3Msg('one', 'two', 'three');
```

```
Show3Msg('one', 'two');
```

```
Show3Msg('one');
```

Without default parameters the last two calls would be disallowed by the compiler, giving the message `Error: Wrong number of parameters specified for call to "Show3Msg"`.

Chapter 9 gives more details about true overloading which refers to different routines which share the same name.

7.g Declaring procedures and functions

The format of a **procedure declaration** is as follows:

```
procedure ProcedureName(parameterList);  
  localDeclarations;  
begin  
  statements;  
end;
```

For example there is a procedure in the LCL called `ShowMessage` which is declared thus:

```
procedure ShowMessage(const aMsg: string);
```

This takes a single string parameter, which it displays in a modal dialog (*a modal dialog is displayed on top of other windows, keeping the focus until it is dismissed with a key press or mouse click*).

The format of a function declaration is very similar:

```
function FunctionName(parameterList): returnType;  
  localDeclarations;  
begin  
  statements;  
  Result:= {a returnType value calculated by the foregoing statements};  
end;
```

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

An alternative to the `Result:=` assignment as the last statement of a function is a `FunctionName:=assignment`. Here is an example from the `SysUtils` unit:

```
Function RenameFile(const OldName, NewName : String): Boolean;
```

This function lets you rename a file. If the renaming succeeds, the function returns `True`. If for some reason the file could not be renamed (*perhaps the file never existed, or has recently been deleted, or is a protected system file*) the function returns `False`.

7.h A program example: function_procedure

Here is a short program example of declaring and using functions and procedures. Create a new console project in Lazarus, and save it with the program name `function_procedure`. Change the program code to the following, which includes writing a short procedure called `DisplayMessage()`. The program uses two string functions from the `strutils` unit, so that must be included in a `uses` clause.

```
program function_procedure;

{$mode objfpc}{$H+}

uses strutils;

procedure DisplayMessage(const aMsg: string);
begin
  WriteLn(DupeString('-', Length(aMsg)));
  WriteLn(aMsg);
  WriteLn(DupeString('=', Length(aMsg)));
  WriteLn;
end;

const LazDescription = 'Lazarus is a very powerful IDE';
begin
  DisplayMessage(LazDescription);
  DisplayMessage('The message above will now be shown backwards');
  DisplayMessage(ReverseString(LazDescription));
  {$IFDEF WINDOWS}
  ReadLn;
  {$ENDIF}
end.
```

To reverse the characters in our string `LazDescription` we use the `ReverseString()` function found in the RTL `strutils` unit, and to form a crude border to the message display we use the `DupeString()` function from the same unit. Their names should give you an idea of the functionality you can expect them to provide, which the result of compiling and running the program probably confirmed. Notice how one procedure or function can call another function directly as in the line:

```
  DisplayMessage(ReverseString(LazDescription));
```

7.i The Exit() procedure

Free Pascal supports the `Exit` procedure which allows you to leave a routine at any point. It is equivalent to jumping directly to the last `end` in the routine, skipping any intervening statements. If the routine is a function, `Exit` can take a single optional parameter (*of any appropriate type*) which sets the return value of the function. For example, a function `NumericCharCount()` that counts the number of numeric characters in a string could be coded as in the following program:

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

```
program char_count;

{$mode objfpc}{$H+}

function NumericCharCount(const s: string): integer;
var
    c: Char;
begin
    if (s = '') then Exit(0)
    else
        begin
            Result:= 0;
            for c in s do
                if c in ['0'..'9'] then Inc(Result);
            end;
        end;
end;

var st: string;
begin
    Write('Enter text for a character count: ');
    readln(st);
    WriteLn('The text "',st,'" has ',NumericCharCount(st), ' numeric characters');
    {$IFDEF WINDOWS}
    ReadLn;
    {$ENDIF}
end.
```

Chapter 7 ROUTINES: FUNCTIONS AND PROCEDURES

7.j Review Questions

1. What does the following procedure do?

```
procedure WhatDoesThisProcedureDo (var i1, i2: integer);  
var h: Integer;  
  begin if (i1>i2) then begin  
    h:=i1;  
    i1:=i2;  
    i2:=h;  
  end;  
  
end;
```

2. Write the body of a function declared as follows
function Fahrenheit (aCelsiusTemp: single): single;
that converts a temperature in Celsius (centigrade) to Fahrenheit.
3. Suggest what is wrong with this procedure declaration:

```
procedure CalculateCircleArea (radius: integer; area: single);  
Amend it, and write a working implementation.
```

4. Write a conversion function
function BooleanToString (aBool: boolean): **string**;
which takes a boolean value and returns its string representation.



Chapter 8 CLASS: AN ELABORATE TYPE

It is time to consider GUI programming, and to do that we must first look at the elaborate type called a `class` that in Pascal is a sort of record-on-steroids, combining both disparate data types (*as the record type does*) and procedure and function code (*which traditional records do not*). Procedures and functions embedded in a class are known as **methods** to distinguish them from non-class procedures and functions which are known collectively as **routines** (*though not all authors maintain this distinction*). Additionally the Pascal `class` introduces various object oriented concepts, including

- inheritance
- data hiding
- polymorphism

all of which are important for the robust versatility and reusability of the `class` type. These ideas are discussed briefly in the sections which follow. Use of classes requires the presence of the compiler directive `{$mode objfpc}` or `{$mode Delphi}` or `{$mode Macpas}`.

8.a Generations of classes

Classes exhibit a **hierarchy**, starting with a simple **base** class, from which descendant classes can **inherit**, so becoming ever larger. The child class contains everything in its parent class. Parental features do not need to be redeclared – they are present by definition from the declaration

```
uses parentClass; // parentClass contains the TParent declaration
type TChild = class(TParent)
    {all the extra features of the TChild class are put here}
end;
```

`TParent` is a class which here is declared in the separate unit called `parentClass` (*which therefore has to be mentioned in its child's uses clause*).

The simple syntax of citing `(TParent)` in parentheses after the keyword `class` gives `TChild` every single feature declared in `TParent`. The declaration of `TChild` declares additional features beyond those in `TParent`. A `TChild` cannot remove any features it inherits from its `TParent`.

All classes inherit eventually from an ultimate pre-declared parent called `TObject` which provides certain basic class functionality – `TObject`'s properties and methods are available to all classes everywhere (*because every class inherits them*).

Declaring a class like this:

```
type TNewClass = class
    end;
```

is the same as if you had declared `TNewClass` like this:

```
type TNewClass = class(TObject)
    end;
```

The declared class is identical in the two cases. (*In MacPas mode the keyword `class` is replaced by the keyword `object`, activated using the compiler directive `{$MODE MACPAS}`*).

Note: This inconsistent naming scheme is a Delphi legacy retained for compatibility (*since `TObject` would have been better called `TClass`*).

Chapter 8 CLASS: AN ELABORATE TYPE

8.b Class data fields

Like records, classes can contain **data fields**. (By convention private fields are named *FSomething* with an initial *F* for *Field*. Beginning a private field name with *F...* is merely a convention. You are not forced to name class fields like this).

Here is a class that could almost pass as a record (though it is not a record, and is not type-compatible with a record):

```
type TMultiClass = class(TObject)
    Name: string;
    ID: integer;
    Date: TDateTime;
    Euros: double;
    InUse: boolean;
end;
```

This looks almost like an analogous record type:

```
type TMultiRecord = record
    Name: string;
    ID: integer;
    Date: TDateTime;
    Euros: double;
    InUse: boolean;
end;
```

If we had two variables:

```
var multiClass: TMultiClass;
    multiRecord: TMultiRecord;
```

we can access the `Date` field of `MultiClass` by writing `multiClass.Date := Now;`

just as we can access the `Date` field of `MultiRecord` by writing `multiRecord.Date := Now;`

Note that field access is also possible using the **with do** construct that also applies to records:

```
with multiClass do
begin
    Date := Now;
end;
```

However, although the above code assigning the result of the `Now()` function to the `Date` field of `multiClass` is valid Pascal, as written it would produce a severe error when the program containing it runs (see Figure 8.1) because of its attempt to access unallocated memory.

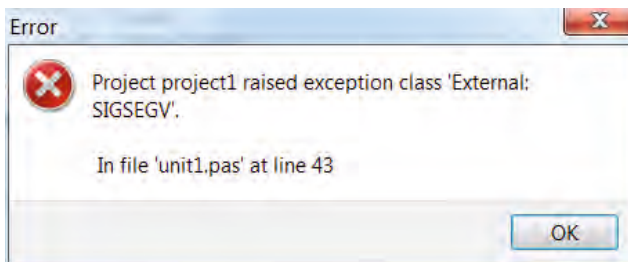


Figure 8.1 Trying to use a class before it has been constructed

Chapter 8 CLASS: AN ELABORATE TYPE

The memory management of classes is quite different from that of records. The memory needed for a record variable is set aside for you by the compiler when you declare any record variable. Usually such a record variable would have memory set aside for it on the stack, a last-in first-out (*LIFO*) memory region managed behind the scenes by the compiler.

Declaring a record variable hands the compiler full responsibility for ensuring that the record's memory requirements are met during its lifetime, and cleaned up when the variable is no longer used (*whether that memory is stack memory or located elsewhere*).

This kind of automatic memory resource allocation performed by the compiler for simple types, records, sets, files and static arrays gives rise to the description **statically allocated** variable to describe such variables' memory management. The programmer does not have to think about this memory housekeeping at all. With classes the situation is different. Automatic memory management of this sort does **not** happen for classes simply by declaring a **class** variable (*though exception classes are – pardon the expression – an exception to this rule*).

8.c Class memory management

Declaring a **class** variable

```
var multiClass: TMultiClass;
```

sets aside enough memory for a pointer (*multiClass is actually a pointer*) and after the declaration that pointer's value is unknown (*though if it is declared as a global variable it will be initialised to nil*). This means that `multiClass` **cannot be used** yet. For its fields alone `multiClass` actually requires enough memory for a **string**, an integer, a `TDateTime`, a double and a boolean. The record `multiRecord` requires 32 bytes, which the compiler sets aside for it when it is declared.

The class `TMultiClass` will need just as much memory for its equivalent fields. Classes also require some extra memory because of the fields, methods and properties they inherit from their ancestor(s), as well as some additional memory overhead required for their full functionality as classes (*functionality that is missing in a standard record*).

All classes need to have memory allocated for them from a large memory area called the heap. The compiler does not do this automatically (*though there are a few exceptions to this rule*). Instead, each class inherits two memory-management methods from `TObject` that must be used to allocate the memory needed for the class when it is created, and de-allocate that memory when the class is destroyed. The methods are named `Create` and `Destroy`. Usually a slight variation on `Destroy` (*called Free*) is used rather than calling `Destroy` directly. The memory allocation method has to be declared using the keyword **constructor**, and the memory deallocation method has to be declared using the keyword **destructor**.

So before the declared variable `multiClass` is ready for use, with enough memory allocated to it to hold all its contents, it has to be **constructed** manually in code by a call to its **constructor** which is by convention named `Create`.

```
begin
  multiClass := TMultiClass.Create; // multiClass is now constructed properly
  multiClass.FDate := Now; // we can safely make an assignment to a field
end;
```

The `Create` method was not declared in the declaration of `TMultiClass` (*which merely declared five data fields*). Like several other methods and properties of `TMultiClass`, the `Create` constructor is **inherited** from the ancestor class `TObject`. If we check the source for `TObject`'s declaration we see it includes the following (*this is a simplified listing, omitting quite a lot*):

Chapter 8 CLASS: AN ELABORATE TYPE

```
TObject = class
  public
    constructor Create;
    destructor Destroy; virtual;
    procedure Free;
    class function ClassType: tclass;
    class function ClassName: shortstring;
    class function ClassParent: tclass;
    class function InstanceSize: SizeInt;
    class function UnitName: ansistring;
end;
```

In using classes, you always have to bear in mind the 'weight' of inherited riches, ready and waiting to be used. Classes are not lightweight programming constructs, they come with considerable baggage which their parents (*and possibly grandparents, great-grandparents and so on*) have contributed to their arsenal of methods and properties.

This makes classes almost unusable if you don't have access to the source code for them. The simplified `TObject` declaration shows the `Create` method declared as a constructor, and the `Destroy` method declared as a destructor, and also a number of other methods declared as `class function` methods. Class methods, declared either as `class function` or `class procedure` can act not only on **instances** of the class in memory, they can also act on the **type** of that class (*which is a sort of template for the class*). Ordinary (non-class) methods require a constructed instance of the class in order to be called (*or else you will see an exception like Figure 8.1*).

Once a class has been constructed (*which allocates the memory it needs*) it is known as an **instance** of the class. The class type (`TMultiClass` in the above example) is a template for how the actual instance will look once it exists in memory. All references to the constructed class are via the `multiClass` variable, which is actually a pointer to the instance of the class which exists somewhere in memory.

Earlier we encountered the `SizeOf()` function which can be used to discover the size in bytes of any variable or any type. `SizeOf(boolean)` returns 1, for instance. In the same way `class` methods can be used on both class variables (*instances of the class*) and class types (*templates for the construction of the class*). Calling `SizeOf()` on a class instance or class type always returns 4 (on a 32-bit system), the size of a `pointer` variable. It does not actually give you much information about the class!

Returning to the `multiClass` example introduced above: after the program using `multiClass` has finished using the class instance which `multiClass` points to, its memory must be released for use elsewhere, otherwise a **memory leak** will occur.

This is done by a call to the `destructor` of the class, via the `Free` method:

```
multiClass.Free; // this releases all the memory the instance occupies
```

It is not obvious that class instance variables are actually **pointers** to class structures. There is no mention of pointer in the variable declaration

```
var multiClass: TMultiClass;
```

to inform you that it is a pointer, and there is correspondingly no need to use the caret `^` symbol anywhere in the declaration. This compiler subterfuge is a simplifying extension to traditional Pascal syntax. It is a concession to beginning programmers, introduced by Borland with Delphi 1 to simplify the somewhat more complex pointer syntax that would be required if classes were declared and implemented explicitly using traditional pointer syntax. Instead, the compiler has inside knowledge of class instance variables, and their memory requirements.

Chapter 8 CLASS: AN ELABORATE TYPE

Class references are dereferenced appropriately as needed, so programmers can largely forget that they are using pointers and dynamically allocated memory structures (*except at the initial moment of class construction and at the final moment of class destruction*). To use the `class` keyword in a declaration you must also include the `{$mode objfpc}` compiler directive.

8.d Exercising simple class methods

To give you an introductory feel for classes (*there is much more to learn about them yet, and this book will by no means treat classes exhaustively*) start a new Lazarus console project, save the program as `class_intro` and adapt the skeleton code provided to match the following program listing:

```
program class_intro;

{$mode objfpc}{$H+}

type TGrandParent = class(TObject)
    procedure WriteAboutMyself;
end;

procedure TGrandParent.WriteAboutMyself;
begin
    WriteLn();
    WriteLn('The name of this class is ', self.ClassName);
    WriteLn(' its parent is ', self.ClassParent.ClassName);
    WriteLn(' its InstanceSize is ', self.InstanceSize);
    WriteLn(' its declaration is located in ', Self.UnitName);
end;

type TParent = class(TGrandParent)
    FintField: integer;
end;

    TChild = class(TParent)
        FsetField: set of byte;
end;

var exampleClass: TGrandParent;
begin
    exampleClass:= TGrandParent.Create;
    exampleClass.WriteAboutMyself;
    exampleClass.Free;

    exampleClass:= TParent.Create;
    exampleClass.WriteAboutMyself;
    exampleClass.Free;

    exampleClass:= TChild.Create;
    exampleClass.WriteAboutMyself;
    exampleClass.Free;

    {$IFDEF WINDOWS}
    ReadLn;
    {$ENDIF}
end.
```

Chapter 8 CLASS: AN ELABORATE TYPE

Compiling and running `class_intro` should yield output similar to the following:

```
The name of this class is TGrandParent
its parent is TObject
its InstanceSize is 4
  its declaration is located in class_intro
  the size of the exampleClass variable is 4
```

```
The name of this class is TParent
its parent is TGrandParent
its InstanceSize is 8
  its declaration is located in class_intro
  the size of the exampleClass variable is 4
```

```
The name of this class is TChild
its parent is TParent
its InstanceSize is 64
  its declaration is located in class_intro
  the size of the exampleClass variable is 4
```

Here we see several aspects of class usage and behaviour:

- Each class (`TGrandParent`, `TParent` and `TChild`) can use the `WriteAboutMyself` method, though it is only declared once. The two descendant classes `TParent` and `TChild` inherit this method. It does not need to be redeclared in the descendants – it is there implicitly. You can only know the full extent of a class's capabilities by looking back through its ancestry. As in real life genealogies, sometimes it is surprising what you find there!
- The `WriteAboutMyself` method uses a variable called `Self` which is predeclared and available for any class. It refers, as the name would lead you to suspect, to that very instance of the class itself. It is a useful way to refer unambiguously to the class itself. You would be well advised to use the `Self` identifier even when it may not be strictly necessary, because it aids in identifying exactly which class instance's methods are being called. In a hierarchy of similar classes this is not always clear (*to the reader*) otherwise.
- We use a single variable, `exampleClass` of type `TGrandParent` to refer not only to a `TGrandParent`, instance but also to a `TParent` instance, and a `TChild` instance. Ancestor variables are type-compatible with **any** of their descendants. It does not work in reverse. If we had declared `exampleClass` to be of type `TParent` (or `TChild`) the program would not have compiled, failing with the error `Incompatible types: got "TGrandParent" expected "TParent"` because you cannot assign a parent class to a variable of its child type, only the other way round.
- The added fields (*an integer in `TParent` and an integer plus a set of byte in `TChild`*) increase the `InstanceSize` of the class in memory. The `InstanceSize` function returns only the size of the data in the class (*the size it would be if it were a record, not a class*). The actual memory footprint of the class will be larger than this figure, since there is additional overhead associated with all classes. The size of the reference variable `exampleClass` which is made to point in turn to a `TGrandParent`, `TParent` and `TChild` instance remains constant at 4 bytes (*the size of a pointer on a 32-bit system*).
- The methods invoked in `WriteAboutMyself` (*such as `ClassName`*) are implemented in `TObject`, the highest class in the hierarchy. All classes can use these inherited methods.

Chapter 8 CLASS: AN ELABORATE TYPE

- The `WriteAboutMyself` method is declared in the `TGrandParent` class. It then has to be defined, and the body of the procedure has to be written **outside** the class declaration to implement that method's functionality. The code that does this always begins with the type name of the class whose method is being written followed by a dot, then the method name:

```
procedure TGrandParent.WriteAboutMyself;  
begin . . .  
end;
```

- Usually a class will be declared in a **unit** separate from the main program. In this case the class declaration is normally placed in the first half of the unit (*the interface section*), and the implementation of any methods is placed in the second part of the unit (*the implementation section*). See the following chapter for more about units and unit organisation.

8.e Properties: special access to class data and events

Although you can happily declare and use plain data fields in a class just as you would in an analogous record, in most cases programmers rarely do this, because classes provide somewhat more sophisticated access to their data than is possible with a record. **Properties** act as normal fields, i.e. they provide read and write access to the class's data. However, they also permit data access to happen only through appropriate methods (*rather than directly reading or writing a data value – though that is possible too*). These data access methods enable the programmer to include 'side effects' as part of the data access: perhaps validating the data, filtering it or associating the access with some action such as updating the display, or causing some other knock-on effect. Properties can also be **read-only** or **write-only**; and there are also **array** properties which provide an indexed structure for array-like properties with numerous elements. Individual property elements of array properties are accessed via an index which can be an ordinal or a string type. There is also provision for **event** properties which are properties of procedural type (*not of data type*) which allow for events to be handled and generated by classes. This is a vital feature for GUI programming which is fundamentally **event-driven**, and not procedural (*the simple console examples given in the preceding pages are all procedural programs*). This topic is considered at greater length in Section 8.g.

A property declaration inside a class declaration is made in one of two ways.

- one way gives unmediated access to the data storage field supporting the property like this:

```
APropertiedClass = class  
  Fdata: TdataType;  
  . . .  
  property propertyName: TdataType read Fdata write Fdata;  
  . . .  
end;
```

- the other way provides two methods – a read **getter** function, and a write **setter** procedure which takes a data parameter. This alternative way to declare properties looks like this:

```
APropertiedClass = class  
  Fdata: TdataType; // some setter & getter methods may not need a data field  
  function GetData: TdataType;  
  procedure SetData(var aDatum: TdataType);  
  . . .  
  property propertyName: TdataType read GetData write SetData;  
  . . .  
end;
```

Chapter 8 CLASS: AN ELABORATE TYPE

Omitting the `write` part of the property makes it read-only. Less commonly the `read` part of the property declaration is omitted to make the property write-only. Outside the class declaration, but somewhere in the program or unit where it is declared, the two property access methods of the second type of property declaration must be implemented. Suppose `TdataType` is `integer`. This means that the assignment `AProptertiedClass.propertyName := 10;` is translated by the compiler to the call `AProptertiedClass.SetData(10);`

Likewise, retrieving the value of `propertyName` by assigning it to an integer variable `i`, `i := AProptertiedClass.propertyName;` is translated by the compiler into the call `i := AProptertiedClass.GetData;`

As an example of the use of **read-only** properties for data that will not change during the course of program execution, consider the following `person_class` console project, which also demonstrates the use of an **array property**, here an indexed list of `Awards[]`.

```
program person_class;

{$mode objfpc}{$H+}

uses sysutils, dateutils;

type
  TPerson = class
    FAwards: array of string;
    FBirth, FDeath: TDate;
    FName, FNationality, FRole: string;

    constructor Create(aName, aNationality, aRole, aBirth, aDeath: string);
    destructor Destroy; override;
    function GetAwards(index: integer): string;
    function GetLifespan: integer;
    function GetNumberOfAwards: integer;
    procedure DisplayInfo;
    procedure SetAwards(index: integer; AValue: string);

    property Awards[index: integer]: string read GetAwards write SetAwards;
    property Birth: TDate read FBirth;
    property Death: TDate read FDeath;
    property Lifespan: integer read GetLifespan;
    property Name: string read FName;
    property NumberOfAwards: integer read GetNumberOfAwards;
    property Role: string read FRole;
  end;

function TPerson.GetAwards(index: integer): string;
begin
  if Length(FAwards) > index
  then Result := FAwards[index]
  else Result := EmptyStr;
end;

function TPerson.GetNumberOfAwards: integer;
begin
  Result := Length(FAwards);
end;

procedure TPerson.SetAwards(index: integer; AValue: string);
begin
  SetLength(FAwards, index+1);
  FAwards[index] := AValue;
end;
```

Chapter 8 CLASS: AN ELABORATE TYPE

```
function TPerson.GetLifespan: integer;
begin
    if FDeath = 0 then Result:= YearsBetween(Now, FBirth)
    else result:= YearsBetween(FDeath, FBirth);
end;

constructor TPerson.Create(aName, aNationality, aRole, aBirth, aDeath:
                           string);
begin
    inherited Create;
    FName:= aName;
    FNationality:= aNationality;
    FRole:= aRole;
    FBirth:= StrToDate(aBirth);
    if (aDeath = EmptyStr) then FDeath:= 0
    else FDeath:= StrToDate(aDeath);
end;

destructor TPerson.Destroy;
begin
    SetLength(FAwards, 0);
    inherited Destroy;
end;

procedure TPerson.DisplayInfo;
var n: integer;
begin
    WriteLn('The ', FNationality, ' ', FRole, ' ', Name, ' was born on ',
            DateToStr(Birth));
    case FDeath = 0 of
        True: WriteLn(' is alive today and is ', Lifespan, ' years old');
        False: WriteLn(' and died on ', DateToStr(Death), ' aged ',
                       Lifespan, ' years');
    end;
    if NumberOfAwards > 0
    then for n:= Low(FAwards) to High(FAwards)
        do WriteLn('    achieved in ', Awards[n]);
    WriteLn;
end;

var person: TPerson;
begin
    person:= TPerson.Create('John Lennon', 'British',
                            'songwriter', '09/10/1940', '08/12/1980');

    person.DisplayInfo;
    person.Free;
    person:= TPerson.Create('Arvo Pärt', 'Estonian', 'composer', '11/09/1935', '');
    person.DisplayInfo;
    person.Free;
    person:= TPerson.Create('Usain Bolt', 'Jamaican',
                            'sprinter', '21/08/1986', '');

    person.Awards[0]:= 'Berlin 2009 100m world record of 9.58s';
    person.Awards[1]:= 'Berlin 2009 200m world record of 19.19s';
    person.Awards[2]:= 'London 2012 100m Olympic record of 9.69s';
    person.Awards[3]:= 'London 2012 200m Olympic record of 19.30s';
    person.DisplayInfo;
    person.Free;
    {$IFDEF WINDOWS}
    ReadLn;
    {$ENDIF}
end.
```


Chapter 8 CLASS: AN ELABORATE TYPE

There are several points to note about this example. The main program block follows the sequence

```
Create(. . .);  
DisplayInfo;  
Free;
```

for each new instantiation of the class. Memory is allocated by the constructor `Create`, used in the worker procedure `DisplayInfo`, and then freed in the indirect call to the destructor via `Free`. All uses of classes have to keep to this basic outline: Create the instance, use the instance, `Free` the instance. Neglecting to attend to freeing memory objects inevitably causes memory leaks.

`TPerson` is basically a data class holding fixed (*reference*) data that will not change. It makes sense then to protect that data, making the properties read-only by omitting any property `write` section. How then do we get data into the class? Here this is done via the constructor, `Create`. In its implementation `Create` is redeclared, this time with a parameter list designed to pass initial data to the class. In the constructor's implementation we first call the inherited constructor (*so the memory required is allocated correctly*), and then fill the data fields from the appropriate parameter. Of course a proper program would obtain its data from a database, and not hard-code data values as here in this simple example.

The program has to deal with the possibility that the people whose lives are summarised may not have died, so their death date is meaningless. Accordingly it uses a trick, and if the death date is empty, the value zero is assigned to the `FDeath` field.

The `DisplayInfo` procedure checks for this value, and formats `FDeath` for display, or ignores it altogether as appropriate.

Because this program has an array property (`Awards[]`) for which it reserves heap memory in a dynamic array (`FAwards`), we have to implement a destructor to ensure this memory is deallocated after use. Classes are able to use a single method name to achieve customised effects at different levels of the inheritance hierarchy through **virtual** methods. In `TObject` the destructor named `Destroy` is declared as `virtual`. This means we can redeclare an identically named destructor in a descendent class, and if we declare it using the `override` keyword, the compiler will generate code to ensure that at runtime the correct custom destructor is called, even though its name does not appear to show at which level of the hierarchy the correct method lies (*since the name is identical at different levels*). This scheme of using a single method name to effect different tasks appropriate to the different levels in the class hierarchy is known as **polymorphism**. There is more about this topic in Chapter 9.

The class declaration is separated into three sections: fields, methods and properties. This is for clarity, and that clarity is enhanced by listing the lines **alphabetically** within each section. For small classes like `TPerson` this is not too important. However, for larger classes (*you will find the Lazarus sources are full of them*) good organisation of class declarations aids readability enormously, and alphabetical listing of methods and properties makes it far easier to locate items when you study a new class to discover what its capabilities are. In fact sections in classes are almost always arranged according to **visibility specifiers**, which are the subject of the next section. Lazarus provides a tool (*the Code Observer*) that among its many capabilities can check the alphabetical listing of class methods. Its use is briefly detailed in Chapter 19, Section f.

Chapter 8 CLASS: AN ELABORATE TYPE

8.f Private, protected, public and published

Global variables are visible from everywhere. They live on the top of the program hill, and can be seen on the skyline from wherever you happen to be. If you have a gun it does not matter where you are, you can always aim at a global variable. If you are also a good shot, you will hit it when you fire. Generally there is a need for variables which are better protected, which are known only in their locality, and are **invisible** from far away (*invisible, say, from some distant unit that is part of the project*).

Information hiding (*i.e. exposing information strictly on a need-to-know basis*) is a cornerstone of the object oriented design underpinning Lazarus/FPC. Its class syntax provides five keywords that allow class designers to protect sections of a class from view, or to expose them. These five **visibility specifiers** allow you to guard from view (*or publish everywhere in your program*) the contents of the section in the class declaration headed by that visibility specifier.

In increasing order of accessibility a section in a class can be:

- **strict private**
- **private**
- **strict protected**
- **protected**
- **public**
- **published**

Public and **published** fields, variables, properties and methods are available globally. They present the public face of the class. The class's **constructor** and **destructor** must be declared in a **public** section (*they are not allowed in **private** or **protected** sections*). The difference between **public** and **published** is that **published** properties have extra information generated for them by the compiler, offering Run Time Type Information (RTTI) that **public** properties lack. This extra type information is needed by the Object Inspector (OI) to enable it to offer a suitable editor where you can edit the published property during the design phase to set it to a different value.

One page of the Lazarus Component Palette (*the tab labelled RTTI*) is devoted to RTTI components that exploit this published information so you can hook up these components to each other at design time in a way that enables you to create RAD applications with almost **no code**. There is lots of code in such an application, of course, it is just that a great deal of it has already been placed inside the RTTI components for you.

Note: To explore the RTTI components, choose **Tools | Example Projects...** and in the resulting Example Projects dialog click on the down-arrow and choose the folder

C:\lazarus\components\rtticontrols\examples

(*or its equivalent on your OS*) from the drop-down list beside the *Lazarus source* radio button.

Then choose an example project from the listbox labelled *Projects*, which also has a filter field above it. You may need to drag the listbox rightwards using the dotted splitter grip icon on its right in order to see the full pathnames of the various projects.

Clicking the [**Open first selected**] button opens that project in the IDE ready for you to explore, compile and run. You will see that there are many other example programs here that you might want to explore.

Other visibility sections are **protected**, **private** and **strict private**. The **protected** section of a class is accessible only by the methods of that class and its descendants. Its main usefulness comes when writing a hierarchy of classes (*particularly when developing new components*). The **strict protected** specifier is also possible, but is used only rarely in the sources.

It prevents access to the class from other classes and variables which happen to be declared in the same unit.

Chapter 8 CLASS: AN ELABORATE TYPE

The private section introduces fields, properties and methods that are not accessible by other classes (*unless they are declared in the same unit*).

The **strict private** section introduces fields, properties and methods that are not accessible to any other class, even other classes declared in the same unit. This is the ultimate in privacy. **Private** data cannot be accessed outside of its class, which guards it from unwanted accidental damage. Usually if data needs to be accessible outside the class it is declared in a **property**, which is placed in a public section of the class (*available to the entire program*) or in a published section of the class (*available publicly, and published additionally in the OI, where it can be edited both manually at design-time and in code at runtime*).

A class can have zero or more sections, each introduced by a visibility specifier.

There can be any number of sections, and they can be in any order, and can be repeated.

Data fields must precede method and property declarations within a section.

A common convention is to list the sections in order of increasing access, i.e. a **private** section ... ending with a **published** section, though if you want to annoy readers of your code, you have the freedom to list sections in some less logical order.

If no visibility specifier is present the default is **public**, unless the class has RTTI (*either because it inherits from an RTTI-enabled class, or because you include the {\$M+} or {\$TYPEINFO} compiler directive*) when the default becomes **published**.

You can increase the visibility of (**strict**) **private** and (**strict**) **protected** sections in descendant classes, but you cannot reduce the visibility of sections previously declared **public** or **published**. For this reason the LCL declares many TCustomXXX classes as immediate ancestors of many of the controls found on the Palette. TCustomXXX classes have many public and no published properties. The TXXX class descending from TCustomXXX publishes many of these public properties, for OI access when the controls are manipulated in the Designer. The TCustomXXX class remains as an ancestor for other descendants, which may not want to publish all the public properties, since adding RTTI imposes additional overhead that is not always needed or desired.

Here is a much simplified class declaration taken from the RTL, to illustrate some of these ideas.

```
TComponent = class(TPersistent)
  private
    FOwner: TComponent;
    FName: TComponentName;
    FTag: PtrInt;
    FComponents: TfpList;
  function GetComponent(AIndex: Integer): TComponent;
protected
  FComponentStyle: TComponentStyle;
  procedure SetName(const NewName: TComponentName); virtual;
public
  constructor Create(AOwner: TComponent); virtual;
  destructor Destroy; override;
  property Components[Index: Integer]: TComponent read GetComponent;
  property ComponentStyle: TComponentStyle read FComponentStyle;
  property Owner: TComponent read FOwner;
published
  property Name: TComponentName read FName write SetName;
  property Tag: PtrInt read FTag write FTag;
end;
```

Chapter 8 CLASS: AN ELABORATE TYPE

`TComponent` is a basic building block for much of the functionality of Lazarus. Here you can see that it has three read-only **public** properties. The `Components[]` array property is accessed via the private `GetComponent` method, whereas `ComponentStyle` and `Owner` read their respective private fields (`FComponentStyle` and `FOwner`) directly. It has two **published** properties `Name` and `Tag`. `Tag` has direct read/write access to its private field (`FTag`), while `Name` has direct read access to its private `FName` field, but mediated write access through the protected `SetName` procedure. The `SetName` procedure allows for **validation** of the component's name. If you drop a button on a form and try to name it "1stButton", this procedure will interrupt your action. You will see a message as in Figure 8.2

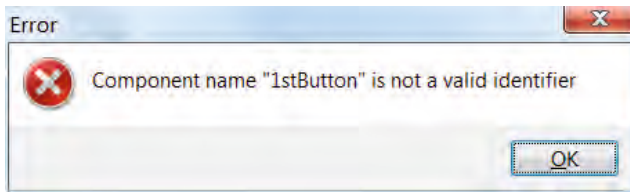


Figure 8.2 The protected `SetName` method pointing out a mistake

Valid Pascal names can contain digits, but not as the first character. The `Name` property of any `TComponent` descendant has this built-in protection, via a protected procedure, from ever being assigned an invalid name.

You notice that the message says "Component name ..." rather than "Button name ..." because the validation routine, `SetName` is inherited from its ancestor `TComponent`. Also because `SetName` is **protected** (not **public**) it cannot be called directly in a program. Likewise `FName`, (the field where the actual string value is stored) is **private** and inaccessible. The only access a programmer has to any component's name is via its `Name` property. This makes for safer programming.

8.g Events

An event is some discrete happening within the world of the computer. In programming terms the event refers to a communication about whatever happened. It may be a system event or a user-originated event (a key press, a mouse move, a timer firing) or some internal change of state of a widget (a line in a memo getting selected, a database connection being dropped, an edit losing the focus).

Procedural programming has **code sequence** as the ordering principle. For instance, consider this excerpt from a program presented in Chapter 4:

```
begin
  AssignFile(f, 'customer.dta');
  Reset(f);
  try
    while not EOF(f) do
      begin
        Read(f, c);
        ProcessCustomer(c);
      end;
  finally
    CloseFile(f);
  end;
end.
```

Chapter 8 CLASS: AN ELABORATE TYPE

Here there is a clear, pre-arranged sequence of code execution from the opening `AssignFile()` call, through the `while` loop which repeats a `begin Read(); ProcessCustomer(); end`; sequence for each customer record, `c`, in the data file, until the concluding call to `CloseFile()`.

GUI programming is based on a different, **event-driven** model in which there is no predetermined sequence of events through the lifetime of the program. Events will happen (or 'pop up') in an unpredictable way throughout the program. We do not know before the program runs exactly which events will be encountered, how many there will be, nor in what sequence they will appear. The GUI program has to be designed to deal with these events (to *handle them*, in programming jargon) as and when they arise.

Operating systems differ in the mechanisms they provide for notifying the programmer of system events, however the various implementations all use the windowing system and the window handle (or its equivalent) to receive notification of events. Windowed controls, then, are the ones that can receive such messages.

The fundamental engine of a GUI program is the **message loop**, a means by which your program can continually watch for system and Lazarus-generated events, and feed them to the correct windowed control to be handled. In Lazarus the unique instance of the `TApplication` class called `Application` has this responsibility, delegated to a method called `ProcessMessages`.

The Lazarus program loop is operated by the `Application` instance which is created for each Lazarus project. `TApplication` is a complex class, but its `Run` method which is called in every main GUI program file begins by showing the main form (i.e. the first form created if there is more than one). Then it repeatedly calls the `ProcessMessages` procedure. This varies according to OS, but in essence it queries the windowing system to see if any messages are pending for the running process. If there are any, `ProcessMessages` dispatches the message(s) to their appropriate destination control(s).

Some messages are handled automatically by controls as a result of the way the LCL is programmed. For instance a resize message leads to the control resizing itself (which may have knock-on effects leading to further messages). Other messages have effects programmed by the application developer. These are messages generated by the events available on the OI Events page, such as a form's `OnCreate` event or a button's `OnClick` event. Here is the source for `TApplication.Run`

```
procedure TApplication.Run;
begin
  if (FMainForm <> nil) and FShowMainForm then FMainForm.Show;
  WidgetSet.AppRun (@RunLoop);
end;
```

Essentially the `WidgetSet.AppRun (@RunLoop)` call invokes a widgetset-specific `ProcessMessages` until the program terminates (which happens when the main form is closed). So in pseudocode the main Lazarus program loop is

```
begin
  TMainForm.Show;
  while not Terminated do ProcessMessages;
end;
```

To see how Lazarus implements this, start a new Lazarus GUI project (**Project | New Project...**, [OK]). Then select the main program file by choosing **Project | View Project Source**. The project's `.lpr` file is displayed in the Editor, and you'll see the main body of the program is as follows:

```
begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Chapter 8 CLASS: AN ELABORATE TYPE

Based on the above we can expand the `Application.Run` call to get a feel for the overall GUI program functionality which then reads thus:

```
begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Form1.Show;
  while not Terminated
    do ProcessMessages;
end.
```

8.h Event properties

The name of the `ProcessMessages` method indicates how Lazarus/FPC approaches events: they are messages, i.e. discrete packets of information signalling that some event, some “action” has occurred. The `Application.Run` method (*via its continuously running `ProcessMessages` loop*) is committed to gathering all such “messages” and processing them, which usually means distributing them among the GUI program windows so that at least one window (*most often it will be the one with the current focus*) can handle that message.

Event properties are designed to give applications a straightforward way to respond to events with code appropriate to the event.

Events rely on provision of a **procedural type**, a type not introduced in the earlier chapters on types, since at that stage there was no context to discuss their use. A procedural type is a type that allows you to refer to a procedure - or function - through a variable, i.e. via an assignment statement. Procedure type variables store the address of a routine to call (*it can be a procedure or a function*), and are specific to routines with matching parameter lists. So they are pointer types, however they cannot be dereferenced.

A procedural type declaration is similar to the declaration of a function or procedure header, omitting the procedure name that is usually part of such a header, and instead naming the type itself. For example:

```
type TOneStringParamProc = procedure(aString: string);
```

Given this procedural type, a variable of that type can be defined and used as in the following program:

```
1 program procedural_type;
2
3 type TOneStringParamProc = procedure(aString: string);
4
5 var stringProc: TOneStringParamProc = nil;
6
7 procedure ShowStr(s: string);
8 begin
9   WriteLn('The string passed to this procedure is ',s);
10 end;
11
12 begin
13   stringProc:= @ShowStr;
14   stringProc("string parameter");
15   {$IFDEF WINDOWS}
16   ReadLn;
17   {$ENDIF}
18 end.
```

Chapter 8 CLASS: AN ELABORATE TYPE

Create a new Lazarus console project called `procedural_type` and test this out (*omitting the line numbers!*). The `ShowStr()` procedure can be called simply by making the `stringProc` variable a statement (*used in conjunction with the appropriate parameter that `ShowStr()` was declared as needing*). This is done in line 14 of the program listing above. Note that procedural types must be global, they cannot be declared within another routine.

Analogous to the procedural variable is the **method pointer**, which enables you to call a particular method of a given class instance at runtime. Syntax-wise the only difference between a procedural type and a method pointer type is the addition of the phrase “of object” required for the method pointer declaration following the method prototype header.

```
type TOneStringParamMethod = procedure(s: string) of object;
```

Once a method pointer type like this has been declared you can declare a variable of this type, and assign a compatible method to it. “Compatible” here means any method with the same parameters, declared in the same order. Method pointers are not compatible with global procedural types, because they carry a reference not only to the method whose address is assigned to them, but they also bear a reference to the specific instance of the referenced method. Procedural types do not need and cannot receive this information.

The Object Pascal syntax for classes supports events through use of event properties which are implemented using method pointers. They are special properties in that you can only assign methods to them. You cannot assign data to event properties as you do with ordinary properties. As with procedural types each event property is specifically compatible with a particular method signature (*i.e. a particular method parameter list*). Many of the most commonly used event properties in Lazarus are of the type `TNotifyEvent`. This is declared in the `Classes` unit as follows:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Here is an example of a `TNotifyEvent` procedure. It is an event handler which has been assigned to the `OnClick` event property of a `TButton` control named `Button1`:

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

This procedure is a `TNotifyEvent` procedure on account of its parameter list (*a single `TObject` parameter passed by value*). In this particular case the parameter is ignored in the body of the procedure. When the user clicks `Button1` it causes an `OnClick` event to be generated which the `ProcessMessages` method will dispatch to the receiving form of type `Form1`. Either at design time using the OI, or in code the following assignment will have been made:

```
Button1.OnClick := Form1.Button1Click;
```

The `OnClick` event property of `Button1` allows you to hook up this event to a specific action in code simply by assigning that code action (the `ButtonClick` procedure, which here is coded so it closes the current window) to a method pointer of `Button1` referenced by its `OnClick` property. (*The actual field supporting the property that stores the value is called `FOnClick`, and it is of type `TNotifyEvent`*).

Chapter 8 CLASS: AN ELABORATE TYPE

Because this `TNotifyEvent` procedure can be assigned straightforwardly like this it makes it possible for a single event handling procedure to be shared among different events. The handler used can also be changed at runtime by a similar sort of assignment. Or it can be disconnected from the code it previously invoked by setting its value to `nil`.

The OI makes it easy to reassign event handlers listed on the Events page. If you click to the right of an event name on the down-arrow alongside it, a drop-down list of potential event handling procedures is shown. These are guaranteed to be compatible with the event – they will all be procedures with the correct number, type and order of parameters to be assignable to that event. (*This assumes, of course, that you have written procedures of this sort which you can choose from – if not the list will be blank*).

There are quite a number of standard events that all GUI programs need to handle, that programmers can respond to by writing suitable event handling procedures (*like the above `Button1Click` procedure that reacted to the click event by closing its window*). Event properties that relate to system events are usually given names beginning with `On` such as `OnKeyUp`, `OnMouseDown`, `OnResize`. Events arising from changes of state are also often named `OnXXX`, such as the `OnChange` event of editable components that arises when their `Text` property is modified.

The prefixes `BeforeXXX` and `AfterXXX` are also used. You will find database-related classes that have events such as `BeforeInsert` and `AfterPost`. Using these events you can enhance, cancel or override the default behaviour of the classes and components that publish them, using standard components and widgets, but customising their behaviour to suit the needs of your project. The following chapter explores this further.

8.i Object oriented design

Object oriented programming, **OOP**, is the design philosophy underpinning the FCL, LCL and design of the Lazarus IDE. Key concepts are **encapsulation** of functionality, **inheritance** as a way to facilitate code reuse, **data hiding** to protect data from unwanted access, and **polymorphism** as a way of reusing a single programming term to adapt reliably to related but subtly different situations in a hierarchy of classes linked by their ancestry.

Classes can certainly be complex, but that complexity is only a result of the object oriented design that also yields protective data hiding, extensive reuse of well-tested code via inheritable classes, and polymorphic method calls that offer elegant interfaces to class functionality, and consistency in method naming across related classes, yet calls which adapt at run time to ensure that different (*yet correct*) methods get executed. The next chapter looks in detail at a number of aspects of polymorphism.

8.j Review Exercises

1. This chapter did not detail the use of **indexed properties**.

Look at the `lazarus\tools\debugserver\frmoptions.pp` source for a simple and elegant illustration of how to design a form class (*used in a modal dialog*) that returns five boolean properties which collect user settings in one location with an easy-to-use interface.

2. For a good example of a simple but elegant customised class (*descending directly from `TObject`*) for listing a particular item (*`TMethod` in this case*) look at the source for `lazmethodlist.pas` found at `lazarus\components\lazutils\lazmethodlist.pas`.



Chapter 9 POLYMORPHISM

Polymorphism is a cornerstone of object oriented programming, and Lazarus takes the concept to new heights with its cross-platform capabilities. Polymorphism is a way of simplifying complexity by letting a single term mean something different in different situations. The difference, of course, must be appropriate to the differing situation that calls for the different meaning. Indeed, often the difference in the programmed response is almost unnoticed because it is logical and completely fitting.

The need for polymorphic constructs (such as classes, and the LCL, and overloading as a language feature, and Lazarus's cross-platform capabilities) arises from the non-standard nature of the realities that software attempts to mirror or emulate. Although generalities can be abstracted to impose a sort of uniformity onto real-life variety, many realities (*even if similar*) are not truly compatible.

9.a Cross-platform polymorphism

Consider the car industry. Every car has a need for windscreen wipers to aid visibility in inclement weather, or just to clear dead flies and accumulated dirt from the windscreen. However, there is little or no standardisation between manufacturers. Some cars are made with a single windscreen wiper mounted centrally with a telescopic action. The majority of cars have two windscreen wipers. Larger vehicles may have three or more. American cars have wiper blades that vary in length between 10 and 31 inches. European cars have metric blade lengths (300, 340, 380, 425 mm etc.). Some cars have a single motor driving both blades through a gearing mechanism, others have two synchronised motors, one for each blade. If you need a replacement wiper blade for your BMW it is most unlikely that a Honda spare part (*even if nominally of the same size*) will be suitable.

Polymorphism is a “one size fits all” solution to the variety that programmers encounter. It is an artificial imposition of simplicity and apparent uniformity on what is in reality a more complex picture. The complexity is not removed, it is just masked. Consequently there may be the possibility of ambiguity arising (at least in the mind of a code reader, if not in the reasoning of the compiler).

Just as each car manufacturer specifies different defaults for each new car design, so computer manufacturers and operating system developers have succeeded in generating a plethora of standards and styles and interfaces in the technology that has emerged to dominate the market in the last 30 years. Lazarus, as a cross-platform tool, attempts to impose a simplifying standardisation on this variety by programming the LCL so that each Palette component you use in your GUI program behaves identically **on whichever platform** your program is built. A button behaves identically on Linux, Mac, Windows, FreeBSD, etc. It will not **look** the same – indeed the Lazarus philosophy has been precisely to use native widgets and system dialogs as far as possible, preserving this platform-specific variety of UI appearance – but it will **behave** the same. This is achieved through inclusion of largely hidden platform-specific code called via the Interfaces unit. The Interfaces unit is required in the uses clause of every LCL main program file (the `.lpr` file of each project).

Note: There are emerging options, the *CustomDrawn* Palette page, as well as the fpGUI and KOL libraries (*not currently bundled with Lazarus*) each of which attempts to make all GUI widgets look and behave identically on all supported platforms. These custom widgets supplant the native widgets provided on each platform. There is not space in this book to consider these approaches.

9.b Polymorphic methods in classes

The idea behind polymorphism in a class hierarchy is that a single verb (*say, Speak*) can be used appropriately of various nouns, and the effect in each case will be somewhat different.

Consider a base class, TPerson, containing a method `Speak`:

```
type TPerson = class
    procedure Speak; virtual; abstract;
end;
```

The keyword **abstract** simply means that this method has no implementation in this class, and so cannot be called in an instance of TPerson – the class is designed as an ancestor of further classes which will implement the method. Although you can create instances of TPerson, the compiler will warn you that the class has an abstract (*uncallable*) method.

The keyword **virtual** indicates that in a descendent class the `Speak` method can be **overridden**, i.e. a new implementation can be provided which is customised for that descendant, and that `Speak` will (*probably*) mean something different in each child class that overrides the method.

Start a new Lazarus project in a folder named `polymorphic class`. Name the project `polymorphic.lpi` and the form `mainform.pas`. Create a new unit (*not a new form*) called `person.pas`. This unit has code as follows:

```
unit person;

{$mode objfpc}{$H+}

interface

uses Dialogs;

type
    TPerson = class
        procedure Speak; virtual; abstract;
    end;

    { TBeckham }
    TBeckham = class(TPerson)
        procedure Speak; override;
    end;

    { TShakespeare }
    TShakespeare = class(TPerson)
        procedure Speak; override;
    end;

    { TWest }
    TWest = class(TPerson)
        procedure Speak; override;
    end;

    { TBlaise }
    TBlaise = class(TPerson)
        procedure Speak; override;
    end;

implementation

{ TBlaise }
procedure TBlaise.Speak;
begin
    ShowMessage('Le coeur a ses raisons que la raison ne connaît point');
end;

{ TWest }
procedure TWest.Speak;
begin
    ShowMessage('I used to be Snow White... but I drifted');
end;

{ TShakespeare }
procedure TShakespeare.Speak;
begin
    ShowMessage('The robbed that smiles steals something from the thief');
end;

{ TBeckham }
procedure TBeckham.Speak;
begin
    ShowMessage('I've got more clothes than Victoria!');
end;

end.
```

Chapter 9 POLYMORPHISM

If we have a variable `p` of type `TPerson`, then what will `p.Speak` produce? In the polymorphic world of OOP the answer is that it depends on whether `p` points to an instance of a `TBeckham`, `TShakespeare`, `TWest`, or a `TBlaise`. The variable `p`, being of the base type of all those descendants is type-compatible with any of them. At runtime, then, `p` can be assigned to any one of those descendant types. And FPC generates code which determines at runtime what the actual type of `p` is, and calls the correct `Speak` method depending on whether `p` is a `TBeckham`, a `TShakespeare`, or whatever. This depends on Object Pascal's inheritance mechanism, and on two reserved words **virtual** (*used in the ancestor class*) and **override** (*used in the descendent class*). To complete the project, drop a radio-group component on the main form named `rgPeople`. Set its `Caption` to `Choose a TPerson instance` and use the OI to add four items to it as follows (*click on the ellipsis [...] button by the Items property*):

```
David Beckham
William Shakespeare
Mae West
Blaise Pascal
```

Generate an `OnClick` event handler for the radio-group and complete the skeleton implementation so the finished `unitmain` looks like the following:

```
unit unitmain;

{$mode objfpc}{$H+}

interface

uses Forms, ExtCtrls, person;

type

  { TForm1 }

  TForm1 = class(TForm)
    rgPeople: TRadioGroup;
    procedure rgPeopleClick(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.fpm}

{ TForm1 }

procedure TForm1.rgPeopleClick(Sender: TObject);
var p: TPerson;
begin
  if rgPeople.ItemIndex < 0 then Exit;
  case rgPeople.ItemIndex of
    0: p:= TBeckham.Create;
    1: p:= TShakespeare.Create;
    2: p:= TWest.Create;
    3: p:= TBlaise.Create;
  end;
  p.Speak;
  p.Free;
end;

end.
```

Chapter 9 POLYMORPHISM

When you compile and run this program you will see how the single `TPerson` pointer `p` can be used to refer to any of the four descendant classes. The call `p.Speak` will always be appropriate to the actual class of the instance `p` points to, because `Speak` was declared `virtual` in `TPerson`, and then overridden in each descendant (*if you fail to specify `override` in descendant methods, the polymorphic mechanism does not work at all*).

9.c Polymorphic graphic classes

We've seen an example of polymorphic text in the `Speak` procedure which produced varying text strings for each `TPerson` descendant. To illustrate polymorphic graphical behaviour, consider a base class called `TDrawing`, declared like this:

```
type TDrawing=class(TGraphicControl)
  private
    FExtent: integer;
  public
    constructor Create(theOwner: TComponent; anExtent: integer);
    property Extent: integer read FExtent write FExtent;
end;
```

`Tdrawing` descends from an LCL class called `TGraphicControl`. This component is an excellent base component for drawing on the screen. It is provided with a `Canvas` property (*which gives it a surface to draw on*), and a `virtual Paint` method which we can `override` appropriately to paint the correct `TDrawing` on the screen. Lazarus provides the `Paint` method to automatically redraw the `TGraphicControl` whenever needed.

Obviously when the control is first created, `Paint` will be called to draw it. Thereafter Lazarus, working with the host OS, ensures that the `Paint` method will be called in all other circumstances in which the control needs drawing. Say while the drawing application is running you play a game of Doom, and cover the screen with all sorts of other images.

When you tire of the game and close it, the application running underneath it needs to be redrawn on the screen. Provided our drawing code has been put into the `Paint` method of `TGraphicControl` then Lazarus ensures that `Paint` will be called in this situation, so the redrawing takes place automatically.

This is why `TGraphicControl` was chosen as the base class for `TDrawing` – it has the right functionality already programmed into it. All we need to do is add a private field called `FExtent` to tell the `TDrawing` class how big it should be. Based on this integer field we define a public property `Extent` which gives us controlled access to the otherwise private `FExtent` data field.

Start a new Lazarus project named `drawingdemo.lpr`, saving it in a new folder, and naming the first (*main*) form unit `drawingdisplay.pas`. In the OI set the `Caption` of this form to **Polymorphic drawing example**. This project will develop a base `TDrawing` class, and declare two descendant classes one named `TSquare`, and one named `TCircle`, which know how to draw themselves. We will declare a variable of type `TDrawing`, and get this variable to draw itself. According to whether the `TDrawing` instance is a `TSquare` or a `TCircle`, it will draw itself differently. The one method call (`Paint`) will adapt polymorphically to suit its instance's actual type.

Chapter 9 POLYMORPHISM

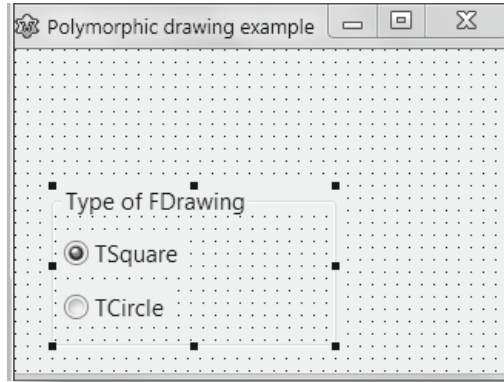


Figure 9.1 The drawingdemo UI

Drop a `TRadioGroup` control towards the bottom of the main form, naming it `rgShape`. With `rgShape` selected, in the OI set its Caption to *Type of FDrawing*, and then click on the [...] ellipsis button for the `Items` property. This opens a *String Editor* dialog where you can type labels for new radio buttons. These are added to the control when you press [OK]. Type `TSquare` and `TCircle` on separate lines in the memo of this dialog as new radio button labels. Click [OK] to accept these button labels. In the OI set the `ItemIndex` property to 0 (*delete the -1 that was there*). Your form should now look something like Figure 9.1.

In the Lazarus IDE choose **File | New Unit**, and save the new unit file as `drawing.pas`. Open the Project Inspector (**Project | Project Inspector**) and make sure that `drawing.pas` is listed in the *Files* section of the Inspector's treeview (*if for some reason it is not, [Add] it using the toolbutton of that name*).

We base our `TDrawing` class on the LCL class `TGraphicControl`, which is declared in the `Controls` unit. The actual drawing code we write also uses items declared in the `Graphics` unit. By default Lazarus has not added either of these units to the `uses` clause of our new drawing unit, which is fairly empty with a `uses` clause of just two units (`Classes` and `SysUtils`). Add `Controls` and `Graphics` to the `uses` clause. Once the `uses` clause refers to the sources for all the classes and variables we want to use, we can take advantage of Identifier Completion to speed our typing.

Below the `uses` clause start a new line for the type declarations by writing:

```
type TDrawing = class(TGraph
```

At this point, instead of completing the typing yourself press [Ctrl][Space]. A small window pops up (*see Figure 9.2*).

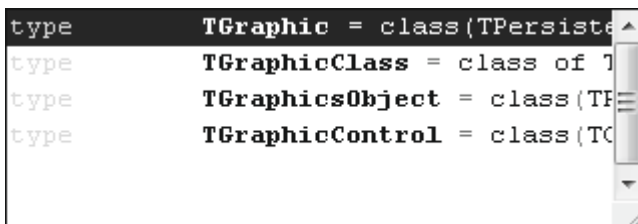


Figure 9.2 Identifier completion, invoked by [Ctrl][Space]

Chapter 9 POLYMORPHISM

The IDE has found four identifiers in the LCL, of which one should be what we want. Here it is the fourth one. Sometimes you are lucky and the first highlighted item found is the very one you want. You can either double-click on the relevant line to insert that identifier, or use the arrow keys to select it, followed by the [Enter] key. Now all you have to do is type the closing bracket. Complete the class declaration, and also declarations for `TSquare` and `TCircle` so the `type` section reads as follows:

```
type
  TDrawing = class(TGraphicControl)
  private
    FExtent: integer;
  public
    constructor Create(theOwner: TComponent; anExtent: integer);
    property Extent: integer read FExtent write FExtent;
  end;

  TSquare = class(TDrawing)
  public
    procedure Paint; override;
  end;

  TCircle = class(TDrawing)
  public
    procedure Paint; override;
  end;
```

You see that `TSquare` and `TCircle` are identical classes, except for their individual `Paint` methods, which are declared with the `override` keyword. This implements the polymorphic behaviour, and can only be used where an ancestor class has declared a method with exactly the same name, and also declared it as `virtual`. In fact `TGraphicControl` has exactly that: a `protected` `Paint` procedure that is declared as `virtual`.

Note: It is easy to check this for yourself. Place the cursor somewhere in the word `TGraphicControl` in the definition of `TDrawing`, then press [Alt][UpArrow]. This opens the Controls unit and jumps the cursor to the declaration of `TGraphicControl`. It is quite a small class (*by LCL standards*). You will locate the `Paint` method easily.

Now we need to write the `Create` and `Paint` methods. Click the `drawing` tab in the Editor and place the cursor in the declaration of the `Create` constructor for `TDrawing` and press [Shift][Ctrl][C] to invoke Code Completion. Fill out the code skeleton Lazarus generates in the unit's `implementation` section so the body of the method looks like this:

```
constructor TDrawing.Create(theOwner: TComponent; anExtent: integer);
begin
  inherited Create(theOwner);
  FExtent:= anExtent;
  Width:= FExtent;
  Height:= FExtent;
end;
```

Chapter 9 POLYMORPHISM

First we call the **inherited** constructor with a parameter called `theOwner`. The **inherited** constructor is the constructor of the parent class (`TGraphicControl` in this case). This sets aside the memory needed for the class, and sets the `Owner` of the class to the value passed in the parameter.

The LCL has a useful **automatic memory deallocation** system for components (*i.e. descendants of* `TComponent` – *this feature applies only to components, not to all classes*). Provided the `Owner` property of a component is not `nil` the `Owner` will see to freeing the memory allocated by the component's constructor when it was created, at the time when the component has to be destroyed and disposed of. For us this means that `TDrawing` and its descendants will be automatically freed after use (*provided we pass the correct `theOwner` parameter at the time they are created*). Chapter 14 gives fuller details of this (*see Section 14.c*).

Once the `TDrawing` class is created in its constructor code, we then initialise some data, set the `FExtent` data field to the value passed in via the `anExtent` parameter, and set the dimensions of the control. If the `Height` and `Width` properties were left at the zero value given them at creation, the control would appear to be invisible, a mere dot without size.

Use Code Completion to create skeleton bodies for the two `Paint` procedures and complete them as follows:

```
{ TCircle }

procedure TCircle.Paint;
begin
  Canvas.Brush.Color := clBtnFace;
  Canvas.FillRect(0, 0, FExtent, FExtent);
  Canvas.Brush.Color := clYellow;
  Canvas.Ellipse(0, 0, FExtent, FExtent);
end;

{ TSquare }

procedure TSquare.Paint;
begin
  Canvas.Brush.Color := clSilver;
  Canvas.FillRect(0, 0, FExtent, FExtent);
  Canvas.Rectangle(0, 0, FExtent, FExtent);
end;
```

The `Paint` procedure has to paint the entire surface of the control, so we first use the `FillRect` canvas method to fill in the background of the control, and then use specialised canvas methods to draw either a `Rectangle` or `Ellipse`. Passing symmetrical parameters to these drawing methods yields a square rectangle and a circular ellipse. These drawing methods are part of the `Canvas` class that comes with `TGraphicControl`. We can just call them, and not have to worry about implementing them ourselves.

Having written the three classes, `TDrawing`, `TSquare` and `TCircle`, all we need to do now is make use of them. To give `Form1` access to these classes we must add `drawing` to the **uses** clause of the `drawingdisplay` unit. Click the *drawingdisplay* Editor tab to focus that unit.

Chapter 9 POLYMORPHISM

Rather than type the alteration to **uses** manually, we'll use the IDE tool to do it. Press [Alt][F11] (or choose *Source | Add Unit to Uses section...*) and in the *Add unit to Uses Section* dialog just one unit will be suggested: `drawing`. Double-click on this line and the dialog closes, adding `drawing` to the **uses** clause.

Add just below the **uses** clause a **const** declaration:

```
const extent = 50;
```

and in the **private** section of the `TForm1` class declaration add these two lines:

```
private
    FDrawing: TDrawing;
    procedure CreateANewDrawing;
```

This adds a new variable of type `TDrawing` which we name `FDrawing`, and a new procedure `CreateANewDrawing` which will free and erase any existing drawing, and draw a new `FDrawing`. According to whether the radio button selects `TSquare` or `TCircle`, `FDrawing` is made to be appropriately a `TSquare` or a `TCircle`.

Use code completion to create a new skeleton body for `CreateANewDrawing`, and fill it out as follows:

```
procedure TForm1.CreateANewDrawing;
begin
    if (rgShape.ItemIndex < 0) then Exit;
    FDrawing.Free;
    case rgShape.ItemIndex of
        0: begin
            FDrawing := TSquare.Create(Self, extent);
            FDrawing.Left := 10;
        end;
        1: begin
            FDrawing := TCircle.Create(Self, extent);
            FDrawing.Left := 60;
        end;
    end;
    FDrawing.Top := 10;
    FDrawing.Parent := Self;
end;
```

This starts out by freeing any existing `FDrawing` (erasing it from the screen). Then if the radiogroup's `ItemIndex` property is 0 (*TSquare is selected*) a `TSquare` is instantiated and assigned to `FDrawing`. If the radiogroup's `ItemIndex` property is 1 (*TCircle is selected*) a `TCircle` is instantiated and assigned to `FDrawing`. In either case `FDrawing`'s `Left` property is varied to give an obvious visual sign that it has changed.

Lastly `FDrawing`'s `Top` property is assigned, and its `Parent` property is set. The `Parent` is an important property for visual controls, connecting them to the underlying OS widget drawing routines for correct display. If the `Parent` is not correctly assigned the control will not be displayed on the form. (*Self here refers to Form1*). Chapter 14 enlarges on this topic (see Section 14.c).

When do we want `CreateANewDrawing` to be called? On two occasions: when the form is first created (we use the form's `OnCreate` event for this), and when the selection in the radio-group is changed (we use the radio-group's `OnClick` event for this). So to complete the project, select the form, then click the OI's *Events* tab. Double-click beside the `OnCreate` event, and fill out the resulting method as follows:

Chapter 9 POLYMORPHISM

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

Select the radio-group, and then click the *Events* tab of the OI. Double-click beside the `OnClick` event, and fill out the resulting method as follows

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

Compile and Run the project by pressing [F9]. See what happens when you select the unselected radio button.

When you select the unselected radio button, the radiogroup's `OnClick` handler calls `CreateANewDrawing`. This creates a new instance of `TShape` or `TCircle`, and assigns the new instance to `FDrawing`. The Lazarus application then calls `FDrawing.Paint` to display the new class instance on the screen (*this is an automatic internal call, you do not see it in any code you have written*). And depending on whether `FDrawing` is pointing to a `TShape` or a `TCircle` instance, so the correct `Paint` method is called. This is what polymorphism is: a single verb (`Paint`) that acts differently according to the type of the instance making the call.

9.d Overloading

The kind of polymorphism exhibited by related classes that declare a method `virtual` and then in descendants declare the same method using `override` is a scheme limited just to classes. There is a more general polymorphic syntax scheme that applies to procedures and functions, whether inside classes (*methods*) or outside classes (*plain procedural routines*). This involves use of the keyword `overload`, and procedures or functions that use this keyword are described as being **overloaded**.

Normally you cannot declare a variable or procedure or function in the same unit (*within the same scope*) which has the same name as another variable or procedure or function. If you do inadvertently reuse the same name for a new procedure, the compiler will balk with a message:

```
Duplicate Identifier "offendingName"
```

where `offendingName` is the name you have tried to use twice. Identifiers must be unique. Overloaded routines all share the same **name** (*by definition*), and they must therefore differ in some other respect for the compiler to be able to distinguish them and recognise which code to use when that single name is making a call that can be interpreted in several ways. So overloaded routines must differ either in the **number** or in the **type** of the **parameters** they specify. The simplest way to introduce the idea of overloading is to give a working example. Consider the following `Add` functions, declared together in one unit interface section:

```
Add(a, b: integer): int64; overload;
Add(a, b: single): double; overload;
Add(a, b: string): string; overload;
Add(a, b: boolean): boolean; overload;
```

Because these functions have different signatures (*although the number of parameters is identical in each function, their types are not*) the compiler can distinguish them, and even though they bear the same name `Add` they are completely distinct, and when the compiler encounters

```
Add('Monty ', 'Python');
Add(34.56, -203.651);
```

there is no doubt which `Add` function needs to be called in each case.

Chapter 9 POLYMORPHISM

Note: Because the differences in the signatures of the routines is sufficient for the compiler to distinguish the `Add` routines successfully, there is no **need** for the `overload` keyword in most instances from the point of view of the compiler (*the only exception is when routines in different units need to be overloaded*). So it is possible to write the above four routines in the same unit **without** using the `overload` keyword, and the unit will compile successfully.

However, it is helpful to readers of the code to include the `overload` keyword even when not strictly necessary to make explicit the programmer's intention to reuse the same name for differing routines. This is also Delphi compatible, and allows the code to be compiled by Delphi.

9.e Default parameters

Object Pascal supports the use of default parameters for simple types such as booleans, enumerated types, integers, pointers (`nil` is the only default allowed), characters and also for the `string` structured type. In some situations using a default parameter is preferable to providing two overloaded routines. If you consider the two floating-point-to-string conversion routines:

```
function DoubleToStr(aDouble: double): string; overload;
function DoubleToStr(aDouble: double; aMinWidth: integer): string; overload;
```

The first overloaded function probably provides a predefined `aMinWidth` value (say 2), and is possibly written by calling the second function with that value. So it makes more sense to replace these two overloaded functions with a single function that provides a default parameter:

```
function DoubleToStr(aDouble: double; aMinWidth: integer = 2): string;
```

Here's a useful function for display of currency values stored in pence (cents) based on the same idea:

```
uses math;
function PenceToString(pennies: int64; aPrecision: integer = 2): string;
begin
  Result := Format('%.' + IntToStr(aPrecision) + 'F', [pennies / (10 ** aPrecision)]);
end;
```

If no `aPrecision` argument is given, the function returns what is normally needed, a currency value giving two decimal places. Otherwise it gives a string with the specified number of digits following the decimal separator.

9.f Review Questions

1. How does overriding differ from overloading?
2. Extend the `DrawingDemo` project with a third descendant of `TDrawing` called `TTriangle`, writing an overridden `Paint` method for it (hint: `Canvas` has a `LineTo` method), and adapt `CreateANewDrawing` and the `rgShape` control to display all three shapes.
3. Write an implementation of the four overloaded `Add` functions given above and repeated here, and put them in a simple application that lets you test their use.

```
Add(a, b: integer): int64; overload;
Add(a, b: single): double; overload;
Add(a, b: string): string; overload;
Add(a, b: boolean): boolean; overload;
```



Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

The console programs we have produced up to now have mostly been short, and monolithic, i.e. everything we wrote was included in a **single** program file, and no other Pascal files were written, though we did reference other files (*written by FPC/Lazarus developers*) as needed, adding them to the uses clause near the beginning of the program. Many console programs will involve you in creating and writing additional unit files, but our examples so far were short enough that this was not required.

GUI projects **always** require at least **two** Pascal source files as part of a program:

- the **main program** file, `program_name.lpr`
- one or more **unit files** conventionally named `unit_name.pas` or `unit_name.pp`

Unit files are of two sorts:

- Plain Pascal unit source files applicable to console and GUI projects alike. You create such a new unit file from the main menu by choosing **File | New Unit**, whereupon Lazarus inserts a new Pascal unit source template into the Editor, naming it by default `Unit2` (*if you only had a single `Unit1` before*).
- A form file unit, not usually relevant for console programs. You create a new form file from the main menu by choosing **File | New Form**, whereupon Lazarus inserts a considerably more complex unit source template into the Editor (*it might be named `Unit3` by default if `Unit1` and `Unit2` were already present*), together with a Designer window showing an empty form named `Form2` (*if there was an existing form named `Form1`*).

In addition to the Pascal source for each form (*that is, each window*) in a GUI project, there is also a corresponding resource file, the **form definition** file, always named `unit_name.lfm` (*where the corresponding Pascal source is called `unit_name.pas`*). This file is created and maintained automatically by Lazarus, and although it is a text-format file, and editable, beginners are advised to leave maintenance of form definition files to Lazarus, since manual alteration of the file is somewhat risky (*because you are interfering with an automated maintenance process*).

Units are a mechanism for dividing the code needed by the program over several modules saved as files, rather than packing everything into one enormous file. Units are the main file-based way in which complexity is modularised and functionality separated into sections of mutually accessible code. They provide both privacy (*through the **implementation** section which is invisible from outside the unit*) and public interface (*through the **interface** section which any other part of the program can reference simply by including the name of the unit in its uses clause*).

The keyword for defining such modules is **unit**, just as **program** is the keyword for defining the overall program source file.

10.a Unit structure and scope

The structure and syntax of unit organisation is identical whether the unit contains code for a GUI form or component, or is an assemblage of standard procedural Pascal code. Its syntax is modelled on that for the main program, but is enhanced in several ways to provide for encapsulation of code and data, and to limit visibility of code and data declared within that unit. The **scope** of an identifier is the region of code in which that identifier (*variable, type or routine*) is accessible. Outside its scope the identifier is unknown and inaccessible. If you name it in an inaccessible region, trying to access it, the compiler will stop with an “unknown identifier” message (*even though if you name the identifier within its proper scope the compiler knows the identifier perfectly well*). The unit structure deliberately imposes these visibility limits in order to protect (*or expose*) data and routines appropriately (*as you decide by placing them accordingly*).

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

If you start a new GUI project (**Project | New Project, Application, [OK]**), and then immediately create a new unit (**File | New Unit**) Lazarus creates a new – almost empty – Pascal source file named `Unit2`, and you will see the following code already written in the opened file in the Editor:

```
unit Unit2;
```

```
{$mode objfpc}{$H+}
```

```
interface
```

```
uses
```

```
    Classes, SysUtils;
```

```
implementation
```

```
end.
```

The new keywords used, **interface** and **implementation** define the two sections present in every unit:

- Everything in the **interface** section (*after the word **interface** and up to the word **implementation***) is visible to other units that use this unit and to the main program file.
- Everything in the **implementation** section (*after the word **implementation***) is invisible outside the unit, hidden from view (*analogous to the **private** visibility specifier that can be used in **class** declarations*).

Units can optionally incorporate two further sections at the end of their implementation section:

- an **initialization** section whose code is executed when the program is first loaded into memory
- a **finalization** section whose code is executed just before the program terminates.

A more complete unit template layout would look like this (*though most of these elements are optional*):

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

```
unit UnitName; // the unit heading is mandatory

interface // the interface keyword is mandatory

uses unit1, unit2, ... unitN;

type // global type declarations
  TExampleType = ...;
var // global variable declarations
  anExample: TExampleType;
const // global constants
  GoldenSection = 1.62;

procedure ExampleProc; // global routines

implementation // the implementation keyword is mandatory

uses unitX, unitY; // units referred to only in the implementation

type // hidden type declarations employed only in the implementation
  TPrivateType = ...;
var // hidden variable declarations employed only in the implementation
  aPrivateVar: TExampleType;
const // hidden constant declaration employed only in the implementation
  password = 'hiddenPassword';

procedure ExampleProc;

  procedure HiddenProc;
  begin
    // code for the nested procedure HiddenProc
  end;

begin
  // code for the main body of global procedure
  HiddenProc;
end;

initialization
  // optional initialisation code goes here
finalization
  // optional final (clean-up) code goes here
end. // the final end. is mandatory
```

Note in particular the separation between the declaration of the `ExampleProc` procedure (which is visible to and exported to other units) in the `interface` section, and its implementation in the `implementation` section. The code that actually implements the procedure is thus hidden from other units – they can see only the interface, i.e. the way they need to call the procedure (its name, and the parameters it requires).

The keywords `interface` and `implementation` must be present in every unit, even if the `implementation` section is empty. Any identifier declared in the `interface` section of a used unit is available to any unit that uses it, just as any identifier declared in the `implementation` section of a used unit is unavailable (*unknown*) to any unit that uses it.

The `system` unit is **automatically** used by all units (and programs), and its interface identifiers are therefore globally available (e.g. `LineEnding`, `Odd()`, `FillChar()`). Consequently it is an error (“Duplicate identifier”) to add the `system` unit to any `uses` clause.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

10.b The GUI program skeleton

Choose **Project | New Project ...** and select *Application* from the *Create a new project* dialog to cause Lazarus to create a main program file named `project1.lpr`, a form unit named `unit1.pas` (together with a form definition file named `unit1.lfm`) along with the `project1.lpi` and `project1.lps` files (see *Figure 10.1*).

Open the Project Inspector (via **Project | Project Inspector**) and double-click on the treeview node in the Project Inspector named `project1.lpr`. This loads the main program file into the Editor (see *Figure 10.2*). You will now see two tabs at the top of the Editor: `unit1` and `project1`. `Project1` has the current focus, and the `unit1` tab is darker to indicate that page is hidden. Clicking either Editor tab brings the appropriate page to the front, and the cursor will be positioned where you last left it when editing that page (or at the beginning of the file if you have not edited it before).

Note: Another way to edit the main project file is to choose **Project | View Project Source**.

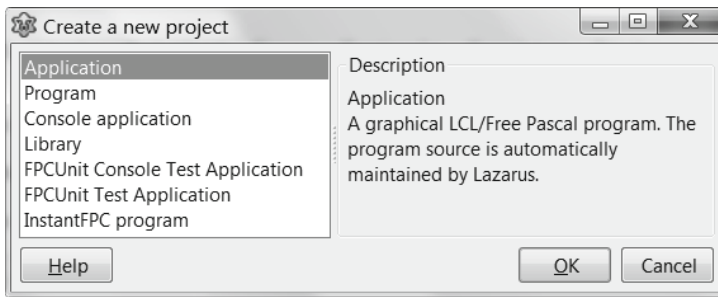


Figure 10.1 Creating a new GUI project in Lazarus

Place the cursor outside all words, somewhere in the white space, and press the space bar to insert a space. Notice that the tab for the file that is current now has an asterisk beside it (`*unit1` or `*project1`). The file has not been renamed! The asterisk merely indicates that the file has changed since it was loaded from disk, and that the changed file will be saved automatically (*overwriting the original*) if you recompile to see the effect of your changes. A second indicator of editing changes unsaved so far is the word *Modified* which appears in the status bar at the bottom of the Editor.

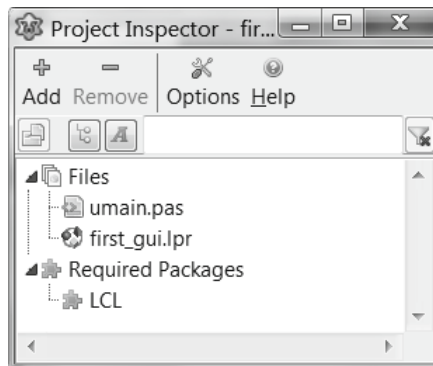


Figure 10.2 The Project Inspector showing `project1` renamed to `first_gui`

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

Rename the project to `first_gui.lpi`, and rename `unit1.pas` to `umain.pas`. You should find that in your project directory (*along with a backup subdirectory*) Lazarus has created a set of files with the following names :

- `first_gui.ico`
- `first_gui.lpi`
- `first_gui.lpr`
- `first_gui.lps`
- `first_gui.res`
- `umain.lfm`
- `umain.pas`

The `first_gui.lpr` main program file when loaded into the Editor looks like this:

```
program first_gui;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, umain
  { you can add units after this };

{$R *.res}

begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The `uses` clause which cites `cthreads` (*on Unixes*), `Interfaces`, `Forms` and the form unit `umain` is longer than we have seen for console programs.

The main program block employs two global variables (*predefined in the LCL*), `RequireDerivedFormResource` (*a boolean which is set to True*), and `Application`, a global class instance which represents the project itself. After initialising the `Application` instance, the program creates and displays the window called `Form1` (*declared in the `umain` unit*), and runs the main program message loop. This loop waits on all the events arising during the program, ending the program when the main form (*window*) is closed.

The `{R *.res}` directive causes all program resources to be read during compilation and linking. This adds the program icon (*by default a Lazarus glyph*) so the OS has more than just a text name for your project, and you can identify it with an icon too.

The Project Inspector gives immediate access to all the files in your project, allowing you to add files (*they don't all have to be Pascal files – you may want image files, database files, .po files, a ToDo file etc. defined as part of your project*) and dependencies (*i.e. requirements for particular libraries or packages*). You can also remove files that are no longer required using the Project Inspector's Remove toolbutton, and call up the *Project Options* dialog for this project easily using the Options toolbutton (*see Figure 10.2*).

The lower part of the Project Inspector is a treeview listing all the Pascal files in your project, and any dependencies. Lazarus has already included the LCL under *Required Packages*, since this is a GUI project that necessarily depends on the LCL.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

10.c Packages

You can see in the Project Inspector treeview that the *Required Packages* branch lists the LCL. Most of the functionality we exploit in the `first_gui` application has been coded by the Lazarus and FPC teams, and that code is stored in the LCL library (*and its dependencies*), which is why the LCL is a **required** package (*in addition to the underlying OS code, of course*).

The program will not compile or run without it, since all its functionality depends on code somewhere inside the LCL or the other libraries considered in the following paragraphs.

A **package** is a named collection of code-related files, which includes information (*metadata*) about where Lazarus can find those files, how they should be compiled, and which other packages are needed to do this. A package may be an entire library, or a smaller logical module encapsulating certain functionality needed for a project. While packages are relevant for console programs, packages really come into their own in helping organise GUI programs, which are nearly always more complex.

Package filenames are given an `.lpk` extension. Packages Lazarus knows about are pre-compiled, and (*assuming they have not changed*) can be used by the IDE without needing to be compiled again, which saves development time. All the GUI programs in this book are based on the LCL, which is a huge library of useful, well-debugged code which in turn depends on two other large libraries: the **FCL** (*Free Pascal Component Library*) and **RTL** (*Free Pascal RunTime Library*). Since a GUI application cannot run without reference to the LCL, Lazarus adds this as a required package automatically.

If you load a new Lazarus project (`project1.lpi`) and compile it without adding anything to it you will have a do-nothing executable file (*on Windows named `project1.exe`, or on Linux named `project1`*). This is quite large (*typically 15 MB or more*) because of all the 'hidden' code which Lazarus includes via the LCL dependency. So although a do-nothing Lazarus executable starts big, as you add functionality to it with code you write, it grows only slowly in size, since much of the 'internal' code you might use (*called by the additional functionality you write*) is already present, compiled into the LCL dependency. Actually 'do-nothing' is not completely correct. The program window can be resized, minimised, dragged around the screen, displays a custom title and icon, has clickable icons, and so on. This 'empty program' functionality is part of the 15 MB binary.

10.d Changing the program icon

To change the icon associated with your project from the default Lazarus cheetah paw-print, click on the Options toolbutton in the Project Inspector to open the *Options for Project: first_gui* dialog box (*see Figure 10.3*). The opening page of this dialog has an Application Settings section where you will see the `Title` set to `first_gui`. You can edit this to give your project a different `Title`. By default Lazarus sets `Title` to be the same as the name of the program.

A `TImage` control below the `Title` edit control displays the program icon, which always defaults to the Lazarus paw print. The [Load Icon] button lets you choose a different icon, and the `TTrackBar` control (*slider*) below the icon lets you set the size of the icon (*between 16x16 and 256x256*). This is the icon shown by your OS file browser when the executable for your project is listed or selected.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

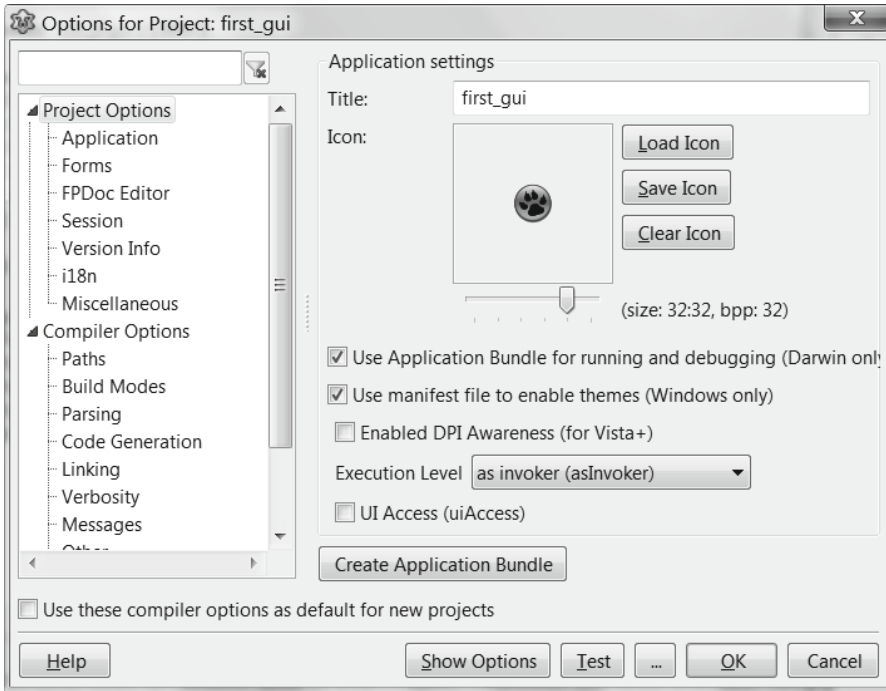


Figure 10.3 The Options for Project:
... dialog where the project's icon can be changed

10.e The main form file

If you now double-click on `umain.pas` in the Project Inspector (or click the appropriate tab in the Editor) the Pascal source for the main form is displayed, which looks like the following:

```
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

Lazarus has written this code for us, declaring a new form class (*which descends directly from TForm*) named `TForm1`. It has empty private and public sections, ready to be customised for our use. A global variable of type `TForm1` is also declared, named `Form1`. This variable is instantiated by the earlier call we saw in `first_gui.lpr`:

```
Application.CreateForm(TForm1, Form1);
```

The `Application` method `CreateForm` instantiates a `TForm1` instance named `Form1` and also shows the form. At design time the form is seen in the Designer, not exactly as it will be shown in the running program. The form we see in the Designer corresponds exactly with the code shown in the Editor, and the form properties as currently set in the OI. You can toggle focus between viewing the Editor or the Designer using the shortcut [F12], or via **View | Toggle Form/Unit View**. If the two windows are small enough, you can view them both on screen at once, as well as viewing the OI.

The `CreateForm` call not only creates a new instance of `TForm1` named `Form1`, but it also makes `Application` the **owner** of this form. This means that `Form1` will be automatically freed when `Application` is freed. This automatic destruction mechanism applies to all descendants of `TComponent` (like `TForm1`, and `TApplication`). **Ownership** (*which requires provision of a non-nil TComponent descendant as a parameter to the Create call*) frees the programmer from having to remember to free the form and any controls dropped onto it herself. This is done for her by the owning component at the time of its destruction.

Lazarus has written a generous **uses** clause, thinking that we will need to use seven LCL units to write a GUI program. For many short programs this is an over-generous list. However, that does not matter. Units which are not used are ignored by the compiler (*except that it emits Hints when they are not needed, mentioning that fact*), and will not be linked into the final executable. In fact the presence of potentially unneeded units is actually helpful in that it enables the **Code Completion** feature to work much of the time (*when otherwise it would not unless you added a needed unit*).

10.f Editor Auto-completion

The Lazarus Editor has numerous cunning and time-saving features built in to it, and it is a model example of an editor customised for a specific purpose – the rapid typing of Pascal code constructs. To illustrate how one such feature helps in the typing of code, click the `umain` tab in the Editor to bring it to the fore (*and press [F12] if you do not see Form1 in the Designer, to toggle its display to the fore*).

Click the Standard tab of the Component Palette to select it, and then click the button icon (*fourth from the left, with ok written on it*) to select it. Now click somewhere within `Form1` in the Designer. Lazarus creates a new button control which it inserts at the point where you clicked, naming it `Button1`.

Double-click on this button. A new event handler is generated, the Designer disappears to the background, the Editor displays `umain` in the foreground again, and the cursor jumps to the implementation of the new event handler which Lazarus has named `TForm1.Button1Click()` – you are free to rename it if you wish. Lazarus gives it this name only because it has to have a name, so a suitable default name is generated for you.

The cursor is positioned in a new empty line immediately following the **begin** of the procedure body. This is an empty procedure skeleton – we have to customise it to do something useful. Without moving the cursor type `"showm"` (*without the quotation marks*), and then press [Ctrl][Space], the shortcut to invoke Identifier Completion. A popup menu appears (*see Figure 10.4*)

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE



Figure 10.4 Identifier Completion offering appropriate options

Press the down-arrow key to select the second entry in the popup list (`ShowMessage()`) and press [Enter] to insert this selection in the Editor. In this case, if we had typed “showme” rather than “showm” the list would have been filtered to three entries (*rather than four*), and the first entry would have been the one we were after, so a single [Enter] keypress would have completed the desired word. Finish the entry so the procedure body becomes the following:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ShowMessage('Hi!');  
end;
```

Press [F9] to compile and run the program, and click on `Button1` to see the effect of the `ShowMessage` procedure call (see *Figure 10.5*).

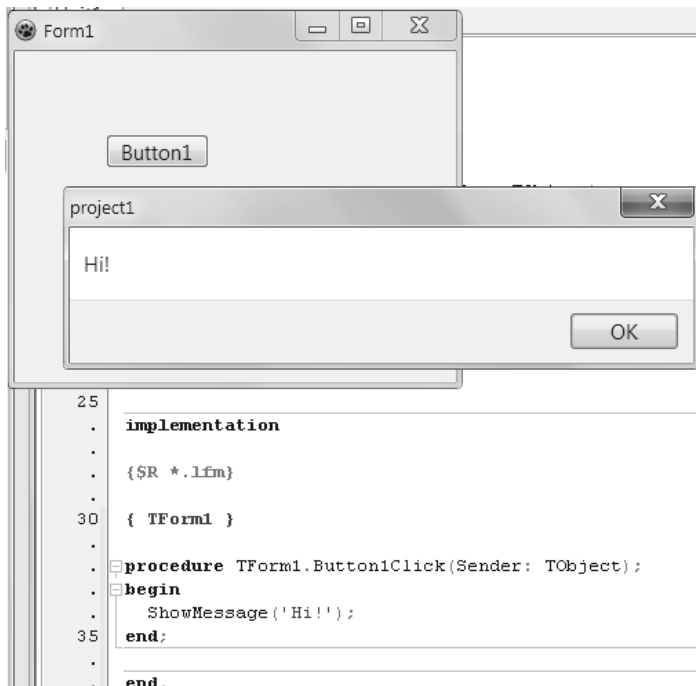


Figure 10.5 `ShowMessage('Hi!')` displaying text in a program window above the code that calls it

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

Note: Identifier Completion works for all identifiers declared in all units which are in scope. The 'extra' units Lazarus puts by default in the uses clause of every GUI project are useful in this respect. The procedure call we wanted (`ShowMessage`) is declared in the `Dialogs` unit. Because this is one of the units already specified in the **uses** clause, the Identifier Completion feature has parsed `Dialogs` in the background already and so can add `ShowMessage` to the popup list.

If you remove the `Dialogs` unit from the **uses** clause Identifier Completion will fail to find `ShowMessage`. In fact it will find only one identifier matching what you have typed ("showm") which is `ShowModal` from the `Forms` unit. Consequently if you invoke Identifier Completion in this situation, `ShowModal` (*as the only possibility*) will be inserted automatically by Lazarus. So it definitely helps you to write code faster if you become familiar with the main LCL units and the sort of routines and identifiers they contain. You can then add them to the **uses** clause early on in your project development. This will not only aid in the compiler finding the routines you need (*avoiding the "identifier not found" error*), but Identifier Completion can then help you to spell them correctly.

A further set of auto-completion wheezes are termed **Code Completion**. To give a short example, return to the `first_gui` project, and add a **private** function to the form class type declaration, overwriting the comment there, as follows:

```
TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
  private
    function TodayAsString: string; // <- this is the line to add
  public
    { public declarations }
end;
```

With the cursor somewhere in the new function declaration line press the shortcut [Shift][Ctrl][C]. Lazarus will construct a new function body in the **implementation** section of the unit (*where it is needed*) and the cursor will jump to the position waiting for you to type code to complete the provided skeleton. Type `Result:= formatd` (*without the quotation marks*) and press [Ctrl][Space], then press [Enter]. Lazarus will write in the name of the `FormatDateTime` function. Continue typing until the completed function body looks like the following:

```
function TForm1.TodayAsString: string;
begin
  Result:= FormatDateTime('dddd, d mmm yyyy', Now);
end;
```

Move to the `Button1Click` procedure you created earlier, and change it so it looks as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessageFmt('Today is %s', [TodayAsString]);
end;
```

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

Press [F9] to compile and run the program. You should see a small information dialog reporting the current date in a user-friendly format when you click on `Button1`.

In this short example we used a date formatting routine from the `SysUtils` unit (*which Lazarus has helpfully already included in the `uses` clause it wrote for us*) named `FormatDateTime`. We also called another function, `Now`, from the same unit. This returns the current date as a `TDateTime` value, which is why we needed to format it as a string in order to display it. The displayed string is made up from two parts, a constant `'Today is '` with the string value of `TodayAsString` appended to it. There are several ways to accomplish this string concatenation. For instance we could have written the amended `Button1Click` procedure like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Today is ' + TodayAsString);
end;
```

This uses the `+` operator to join the two strings. The alternative given first is by way of introduction to the `Format()` function (*which is combined with the `ShowMessage` procedure in `ShowMessageFmt`*). The `Format()` function is hardly needed in this case for concatenating strings (*when the `'%s'` placeholder gets replaced by the string in the array that follows the string*), but it is a very versatile formatting routine for handling all kinds of non-string values. Well worth looking up in the documentation to learn about its many capabilities for converting non-string types to formatted strings.

10.g Using the Designer

The Designer window is a very capable visual editor. Notice how you can select `Button1` and drag it around the Designer to place it where you want. If you alter the `AutoSize` property of `Button1` in the OI to `False`, you will find you can drag the little sizing grips at the edges of `Button1` (*visible only when it is selected*) to resize it in any direction. Right-clicking on the button or elsewhere on the form gives access to several dialog-based tools for changing aspects of the form or button design. Experiment – you can always exit the project, choosing not to save it, if you mess things up.

The popup menu appearing when you right-click in the Designer is context-sensitive, so some functionality may be greyed-out where it is not appropriate to the component(s) you right-clicked.

You can **group** several components by holding down [Shift] and clicking on each component in turn. If you then select say **Align...** from the context menu, the alignment options you then choose will apply to all selected components, not just the one you clicked first. This lets you make many changes very quickly.

Likewise, a group of selected components can have common properties changed in the OI, affecting every component with one edit. Be aware though that these changes cannot usually be undone by pressing [Ctrl][Z]; whereas this is always a reversion option available to you in the Source Editor.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

10.h The Object Inspector

Lazarus offers two ways to assign values to the properties of controls you drop onto a form from the Component Palette. Either you can write code specifying the name of the control followed by a dot and the property name, and use the assignment operator `:=` to give the property a new value, like this:

```
Button1.Caption := 'Show Date';
```

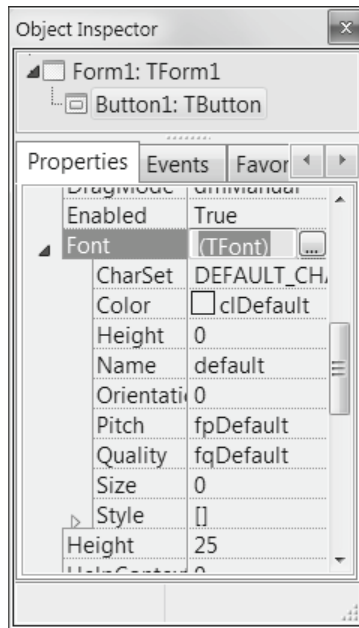


Figure 10.6 The OI showing Font properties expanded

Or you can use the OI (*Object Inspector*) which provides a variety of editors to set property values manually at design time. Either way is preferable to leaving the `Caption` of `Button1` as `'Button1'` – a hopelessly uninformative caption for any control, not giving the user the least clue about what might happen if they click it. It is obvious that the control is a button, so a `Caption` identifying it as such is useless. The UI question is: What does this button **do**?

A sensible programmer would already have renamed `Button1` to something more meaningful, say, `btnDate` or `BDate` or `DateButton`. It is good to adopt a consistent naming scheme for all the code you write. It will help everyone who reads your code later (*including yourself*).

A simple naming scheme used by many Lazarus people is to prepend the name of each control used with a letter (*or a few letters*) indicating what sort of control it is (*B for Button, E for Edit, L for Label, M for Memo etc.*). There are no hard and fast rules about naming. It is very much a matter of style and personal preference.

However it is more important than many realise, not only to provide clarity as you scan your own code identifiers to appreciate immediately what they refer to, but also to help others reading your code. Inconsistent naming, or lazy acceptance of all the default names Lazarus supplies quickly proves confusing to fresh readers of your code who, even if not confused, will constantly have to refer back to declarations remote from the code section they are reading if you do not choose names intelligently to make them self-explanatory. The name `Button2` gives away nothing about the button's purpose.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

In the `first_gui` project select the button named `Button1`, clicking it if necessary (*you may have to press [F12] first to show the Designer*), and make sure that the Properties page is selected in the OI (*the leftmost of the four tabs heading the lower grid section of the OI*).

The OI Properties page has a **gutter** at the left for the small icons used to expand and collapse structured property values. The two columns of the grid section are not titled, but if they were, they would say *Property name* (left column) and *Property value* (right column). It is the right hand column that is editable. You can type directly in the right hand column to change or delete values of some properties (*generally string, boolean, ordinal and numeric properties*). Double-clicking on the value of an enumerated-type property will enumerate the various values, cycling through them one by one.

Click the small grey arrow in the gutter just to the left of the `Font` property on the OI Properties page to expand the `Font` property to expose its own properties (*some platforms use a + expansion icon rather than a grey ► icon*). See Figure 10.6.

Note: If you are not a mouse-lover (*or have trouble with RSI*) a useful keyboard shortcut for expanding class and set properties to show their sub-properties is `[Alt][RightArrow]`. Conversely `[Alt][LeftArrow]` collapses an already expanded property.

Here is a class within a class. The `TButton` class has numerous properties, of which one is the `Font` property which (*being itself a class*) has properties of its own. When expanded in the OI, you can see that `Button1`'s `Font` exposes nine properties. They are indented slightly, to show that they belong just to `TButton1.Font`. Once a class property has been expanded the grey right-pointing expansion triangle becomes black and points downwards slightly. `Font`'s expanded properties are listed alphabetically, and the last one, `Style`, (*a set property*) is empty by default, showing as `[]`.

This is the only one of `Font`'s nine properties that has a grey expand triangle beside it. Click this triangle to expand the `Style` property, and below it (*you may need to scroll downwards to see them all*) are listed the four possible elements in the `Style` set property, all with the value `False` (*meaning that each of those elements is absent from the set*).

If you double-click on any of the `TFontStyle` values – say on `fsBold` – it toggles from `False` to `True`, and the `Style` property line jumps from empty `[]` to `[fsBold]`. A further double-click reverses the assignment. Double-clicking on a different element adds that element (or subtracts it, if it was there to begin with). As you set or unset these property values you can see the caption `Button1` immediately reflecting the change in style in the button displayed in the Designer. The form is actually in a Designer window which continuously repaints the form and any controls it contains (*such as Button1*) to reflect the current state of its properties.

Other properties display an **ellipsis** [...] button that when clicked opens a new editor window for editing of a more complex property (*e.g. the Anchor Editor window for the `Anchor` property, or the Font window for the `Font` property*). Many single line property editors when clicked allow either the typing of a new value, or have a small arrow at the rightmost edge which when clicked opens a drop-down list of possible values to choose from. The key combination `[Alt][DownArrow]` also opens such **drop-down lists** which can then be navigated with the arrow keys, and pressing `[Enter]` on a selection inserts the selected item as the new property value.

The **treeview** box at the top of the OI lists the currently selected form together with all the controls the form contains, both visual (*e.g. edits, labels*) and non-visual (*e.g. dialogs, timers*). Individual controls can be selected by clicking in the treeview, as well as clicking on the form. Sometimes the treeview is the only way to select a control with the mouse. For instance, panels are not selectable on a form if their child components are aligned to fill the panel client space. An alternative for selecting controls is to press the `[Tab]` key.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

This cycles among the form's various child controls (*but does not ever select the form itself*).

[Shift][Tab] cycles in the reverse order among child controls.

Try editing other values in the OI to see the visual effect painted on the form immediately in the Designer window. With `Button1` still selected scroll upwards and check that the `AutoSize` property is `False` (*change it from `True` if necessary*). The `Width` property will be set at 75. Delete that value, replacing it by 100. As soon as you press the [Enter] key you will see the button broaden in width. Similarly if you change the `Left` or `Top` property you can make the button jump around the form. You can easily set the `Left` or `Top` properties to values that move the control outside the visible area of the form, making the button appear to be invisible (*set either to -100 for example*).

Likewise, if you click outside the button to select the form, and expand the `BorderIcons` property, by setting all of the values to `False` you can change the title bar of the form to show only the [X] close icon (*this border icon cannot be removed, otherwise users might be left with no way of closing the application*).

10.i OI Favorites and shortcuts

Once you start to program in earnest, and are spending a lot of time dropping new controls on forms and setting their properties in the OI you may find the OI *Favorites* page useful. The Properties grid is really long for some complex controls, and navigating up and down the rows to get to the property you want can be annoying (*the [UpArrow] and [DownArrow] keys work as well as the mouse and scrollbar*). If you right-click on an often-used property name and choose *Add to Favorites* from the popup menu, this adds it to the shortened property list on the *Favorites* page where you can more quickly locate the properties you use most often.

There are several other useful keyboard shortcut keys for heavy OI users.

- [F11] focuses the OI if you are in the Designer or Editor.
- [Ctrl][Enter] cycles through the options in a drop-down list with keyboard as an alternative to using the mouse.
- [Tab] moves you between the two columns in the property grid

When you are in the left (*Name*) column you can navigate to a remote property by typing the **first letter** of its name. Say you have clicked on a button in the Designer. Pressing [F11] takes you to the OI *Value* column of the property grid, to the first property *Action*. If this is not the property you want to edit, press [Tab] to move to the *Name* column. Pressing any character key will jump to the first property whose name begins with that key, and will focus that property's *Value* field ready for editing. Often this will be the property you want. If not it will only be a few rows lower.

You can now type the new property value (*or choose from a drop-down list of values, where appropriate, or click the [...] ellipsis button to open an appropriate editor dialog*). Pressing [Enter] or [Tab] saves the chosen property value.

Note: Two further helpful OI features are available, which are not on by default: OI **hints**, and the OI **information box**. These are basically designed as alternatives (*though you can enable them both if you wish*). You turn them on via **Tools | Options...** ([Shift][Ctrl][O]). In the resulting *IDE Options* dialog, from the *Environment* branch in the treeview on the left click on the *Object Inspector* node to open the OI options page, and tick one (or both) of the options *Show hints* and *Show information box*.

The hints are documenting windows that appear fleetingly (*with information about the property or event under the mouse cursor*) when you hover briefly above a property or event name. The information box is a box permanently appended to the bottom of the OI which displays the same information as the fleeting hint window would do for each selected property or event as you click on it. You may have to resize the information box by dragging the small splitter control at the top of the box upwards (*it has a faint line of gray dots to indicate where to grab it*).

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

10.j The OI Restricted page

Because there is only limited standardisation between operating systems, you sometimes find that some properties or features of a control are unavailable on certain platforms. The OI has a *Restricted* page which lists these restrictions (*you may need to click the horizontal arrow buttons beside the OI tabs to move this page into view*). See Figure 10.7, and check it out in Lazarus for yourself.

For `TButton` this page shows that the `Color` property is restricted on Windows. Try this out: select `Button1` and on the `Color` row click either the arrow button to open the drop-down list of colours, or the [...] ellipsis button to open a *Color* dialog. Change the `Color` of `Button1` from `clDefault` to something else, pressing [Enter] to confirm your selection. If you are running on Windows the colour of `Button1` does not change, whatever choice you make (*because of this Windows restriction*)!

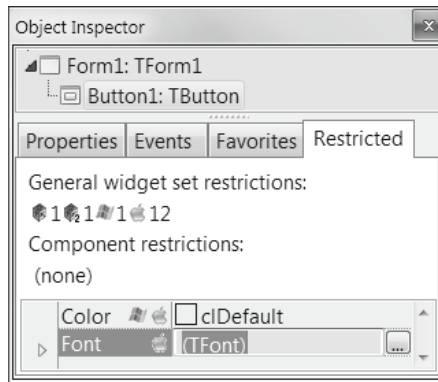


Figure 10.7 The OI Restricted page for `TButton`

If you are running on Linux or on a Mac you will see the button change colour as expected. These restrictions are OS restrictions, not LCL restrictions. Since the LCL uses the underlying OS widgets to paint the various aspects of LCL Palette controls and since the LCL is cross-platform, it provides access to most of the commonly used widget attributes (*such as colour, via the `Color` property*). Where a particular OS lacks a widget attribute the LCL is stuck. It provides the facility for the OSs that offer it, but obviously has to leave a 'hole' in the functionality where the OS widget cannot provide the hoped-for functionality.

Of course you may find that even among a couple of hundred controls there does not appear to be one that does exactly what you want. Sometimes this is because of an OS limitation. LCL controls (*apart from the upcoming `CustomDrawn` set*) are based on native controls. We have seen that on the Windows platform button controls cannot be coloured differently from the theme colour. This is an unfortunate Windows limitation.

In future Lazarus versions it is likely that the overall number of Palette components will increase still further, since cool new components may be added (*such as the `CustomDrawn` controls currently being developed, for which the design incentive has been creation of controls that work well on tablets and mobile devices with touch screens*). It is unlikely that any will be removed, to avoid compromising backwards compatibility. For instance the `TDbf` component (*on the Data Access page*), was for a time marked deprecated (*since at that time it was not actively maintained or improved*) but it was not removed, thereby keeping it available for older projects that used it.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

10.k The Component Palette

The Component Palette in Lazarus version 1.0 has 14 tabs (*Standard, Additional, Common Controls, Dialogs, Data Controls, System, Misc, Data Access, SynEdit, LazControls, SQLdb, RTTI, IPro and Chart*). These 14 pages contain just over 200 components altogether! This is an embarrassment of riches. Each of these components is itself quite a complex class, with specific functionality. How can we tame all this complexity? How can you know, when you come to design the UI of your next application which of these 203 components are the appropriate ones for your purposes?

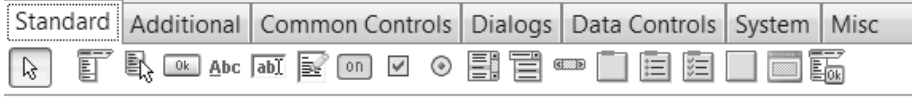


Figure 10.8 The Component Palette of the Lazarus IDE open at the Standard page

The point of a Palette of components, as with a palette of colours held in a painter's hand, is to give you a wide choice among possibilities. However, unlike blobs of white and burnt sienna on an artist's palette which can be mixed in an infinite number of ways to give all manner of subtle shades, the components on the Palette cannot be mixed together.

Each component is a discrete class which shares some functionality (*such as the ability to be dropped on a form in the Designer, properties such as `Left` and `Top`, and possession of a list of event properties such as `OnClick`*), but each component is completely distinct from the other available components.

Components have a place on a Palette page because they encapsulate certain tried-and-tested functionality which other programmers have found is often needed. Visual components often wrap underlying OS widgets, sometimes extending their functionality, and sometimes (*e.g.* `TStringGrid`, `TChart`) providing a control not available as an OS widget.

Palette components are a testament to one of the principles of object oriented programming (*OOP*): reusability. Once a particular well-designed wheel has been tested and debugged it can be 'fossilised' and placed on a Palette page for reuse whenever needed. That particular wheel won't ever again need to be reinvented.

10.l Finding a Palette component

For historical reasons (*to do partly with Delphi compatibility*) the arrangement of controls on the various Palette pages is not always as helpful as it might be when you are trying to find a control that you "know is there somewhere". To save you clicking on every Palette tab and searching each page for a component you are sure exists somewhere, the IDE provides a quick-locate dialog named *Components*, accessed via **View | Components** or ([Ctrl][Alt][P]). See Figure 10.9.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

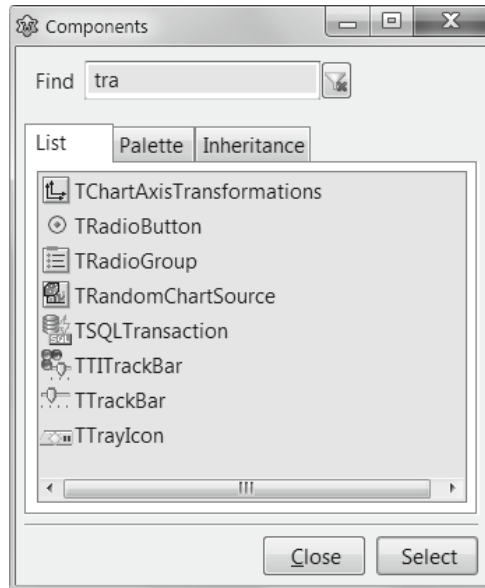


Figure 10.9 The IDE Components List page after typing "tra" in the search filter

A search filter field enables you to narrow the search among listed components. With each typed character the list of matching components is reduced. The list has three pages. The *List* page is simply an alphabetical list of all 203 components. The *Palette* page lists components according to Palette page, and the *Inheritance* page shows part of the class ancestry for each component (*more useful for component writers than casual users, since it necessarily shows many classes not available on the Palette such as the TCustomXXX classes*).

Double-clicking on a listed item inserts that component on a form in the Designer located near the top, left of the form, from where you can drag it to your desired location. If you (*single*) click a listed component to highlight it and click the [Select] button, the *Components* dialog is closed and the appropriate *Palette* page is opened with the indicated component already selected.

10.m Regular, DB and RTTI component types

There is apparent duplication on the *Palette* in that several standard components come in three flavours. In addition to the 'basic' component there is often a DB and a RTTI version as well. The visual DB controls are all located on the *Data Controls* page (*14 components*) and the RTTI controls as you would expect on the *RTTI* page (*22 components*). For example there is a `TLabel`, a `TDBText` and a `TTILabel`. There is a `TEdit`, a `TDBEdit`, and a `TTIEdit`. Why this 'duplication'? The reason is to provide specialised, enhanced functionality.

All the `TDBxxx` controls are very similar to their non-DB counterparts, except they have added `DataSource` and (*except for TDBGrid*) `DataField` properties which allow them to be hooked, via a `TDataSource` instance, to a `TDataSet` descendant. This links them (*once the dataset is Active*) to a specific field in the dataset (*or in the case of TDBGrid to some or all of the fields of the database*). The DB component then displays the content of the linked field of the current database record, and most DB components also allow the displayed value to be edited as well. This gives rapid and almost code-less programmable access to many databases.

Chapter 10 UNITS, GUI PROGRAMS AND THE IDE

Lazarus provides components to interface with dBase (.dbf) and .csv files (*on the Data Access page*) and through various `TSQLConnection` descendants can also connect to Firebird, SQLite, Postgres, MySQL, Oracle, and MS SQLServer. These databases can then be queried using the `TSQLQuery` and `TSQLTransaction` components. These SQL-enabled components are all found on the SQLdb Palette page. This book does not explore database programming (*which is a book in itself*), but you will find Lazarus comes with several database example programs you can study. The following paragraph explains how to locate them.

The **RTTI components**, likewise are very similar to their non-RTTI counterparts, except for the addition of a `Link` property. The link is itself a class with sub-properties. The most important of these are `TIOObject`, `TIPropertyName` and `TIElementName`. Connecting the `Link` through these properties allows, again, almost code-less access to the properties of other GUI components at runtime. There is an example of one use of an RTTI component in Chapter 12, Section a: *Editing short phrases*. You will find several further examples in the Lazarus example sources. See **Tools | Example Projects...**, and in the **Example Projects** dialog make sure the *Include Examples* checkbox is ticked, and that *Lazarus Source* is selected in the *Search projects from* radio-group. You may have to wait while the IDE searches the source paths, since there are over 100 example projects shown there for you to browse.

10.n Non-visual LCL and FCL support classes

In addition to the Palette components registered with the IDE which can be dropped onto a form for immediate use, the FCL and LCL contain a great many highly useful classes (*not all of which are components*) that are not available on the Component Palette. Learning to use Lazarus as a development tool means becoming familiar with what is on offer in this 'hidden' pool of classes which mostly do not have a visual representation, as well as becoming familiar with the Palette components the majority of which are both visual and have mouse/keyboard interactivity built in. Among these vital 'hidden' support classes are the important classes `TList`, `TStrings` (*and its descendant* `TStringList`), `TCollection` and `TStream`, as well as many other component and non-component classes. These support classes have a chapter (*Chapter 15*) devoted to them, since you need to understand them to be able to use many of the Palette controls effectively.

The next chapter looks at some of the most useful Palette display-only controls.

Succeeding chapters look at a selection of editing controls, and how to tackle GUI projects.

10.o Review Exercises

1. Start a new GUI project, and drop several controls from several Palette pages on the form. Explore the components' properties using the OI. Try changing various properties, and explore the different editors used for setting string, integer, enumerated, set and `Anchor`s properties.
2. Try [Shift]-clicking two or three form controls and explore what properties and events they have in common. Try setting common properties or a common event to see what happens.
3. Right-click on the form in the Designer and choose *View Source (.lfm)*. If you haven't yet saved the form you will have to do that first. Lazarus will open the `unitName.lfm` file, and you can see the text format in which the published properties of the form and all its child controls are stored.



Chapter 11 DISPLAY CONTROLS

Certain components have been designed principally to **report** information, rather than to **edit** it; or are useful for visual layout and arrangement. These controls tend to be the simplest to use and understand, since editing capabilities add a level of complexity that display-only controls do not need. This chapter offers an overview of several display-only controls, starting with perhaps the most commonly used control of all, the label.

11.a Display controls: TLabel

A **label** displays read-only text on the screen, optionally associating it with another control placed on the same form, via its `FocusControl` property combined with an accelerator key. A label cannot receive focus itself (*i.e. you cannot get the cursor to select or highlight a label in a running application, nor can you move to a label by pressing [Tab]*).

To get a feel for `TLabel`, start a new Lazarus GUI project, naming it `labels`, and rename `unit1` to `umain`. Make sure the Designer is visible showing `Form1` (press [F12] to bring it to the foreground if it is not visible), and from the Standard Palette page drop a button and an edit control onto `Form1` followed by a label (*these controls are all found on the same Palette page*). Select the label, and in the OI click the down-arrow in the empty field beside the label's `FocusControl` property. The drop-down list will have the options: (none), `Edit1`, `Button1`. Choose `Edit1`. To complete the functionality requires adding an accelerator key to the label. In the label's `Caption` property type `&Edit1` and press [Enter]. Notice that in the Designer the label's `Caption` now reads `Edit1` – the `&` character causes the letter following it to be underlined (*if it does not display an underline, change the `ShowAccelChar` property to `True`. It may be that your platform uses emboldening rather than underlining to mark the accelerator key*).

Set the `TabOrder` property of `Edit1` to 1 (*if it is 0*), and check that both the edit and the button have their `TabStop` property set to `True`, so that `Button1` will receive the focus when the form is first displayed. Then compile and run the program by pressing [F9]. You should see that `Button1` has a focus rectangle round it (*or however your platform marks a focused button*), and pressing [Spacebar] will depress the button. Press [Alt][E], which will then shift the focus from the button to the edit control. Although `Label1` cannot receive the focus itself, it can trigger the change of focus from button to edit through use of its accelerator key.

Close the running program, and go back to the Designer, selecting one of the components you earlier dropped on the form. Enlarge the OI so it occupies the full screen height, and scroll the *Properties* grid so you can see both the `Left` and the `Top` property of the selected control in the OI (*you may have to use the splitter between the upper treeview and lower grid to reduce the size of the OI treeview*).

Now drag the selected component round the form, and notice how the values of `Left` and `Top` are continuously updated to reflect the control's new position. The OI keeps its displayed values continuously synchronised with the Designer form display. Lazarus also ensures that the underlying property storage (*in the `umain.lfm` file*) is also updated to reflect changes in property values, and also reflect any additions or removals of form controls.

Chapter 11 DISPLAY CONTROLS

11.b Display controls: exploring TLabel properties

Often the best way to understand what a particular component property does is to see it changing in a program. The following demonstration exercises four display properties of `TLabel` (*Color*, *Alignment*, *Layout* and *Font.Style*) to give you a feel for how you can use this apparently simple component to good effect.

Start a new Lazarus project in a new project folder named `LabelProperties` and save the main project file as `label_properties.lpr`, and save the form unit as `label_form.pas`. Running the project once you have completed it will look something like Figure 11.1.

A grid of adjacent labels is drawn (*using code*) in the upper part of the form, and the lower part of the form contains three groupbox controls – two radio-groups and a check-group. Radio controls are designed for choosing between mutually exclusive options, and checkboxes are designed for editing on/off property types. Checkboxes can optionally also have a third **greyed** state (*neither On nor Off*).

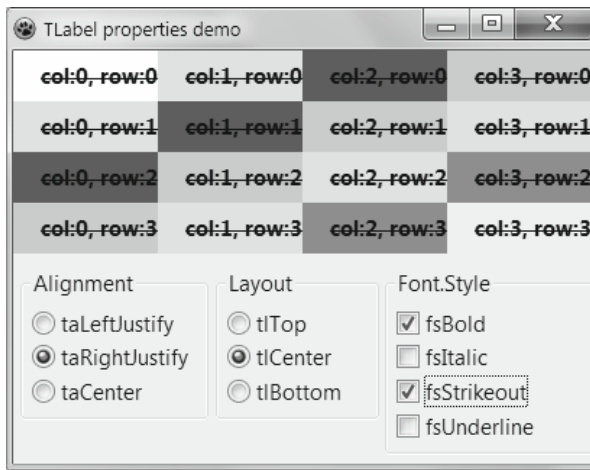


Figure 11.1 label_properties user interface

This demonstration will affect the following properties of `TLabel`, which are described here:

- `Color` is an integer subrange type, combining bytes for Red, Green and Blue values in a single integer.
- `Alignment` and `Layout` are enumerated types which affect the positioning of a label's `Caption` within its borders. Don't confuse `Alignment` (*which affects the interior of a label*) with the `Align` property, which affects how the label is positioned on its parent control (*usually a form*).
- `Font.Style` affects the appearance of the typeface used to draw the label's `Caption`.

The code we write in this example program is concerned with setting up a grid of differently coloured labels, together with providing event handlers for the three groupbox components so that changed selections in the groupbox get reflected in the labels' properties and display. First click in your new `Form1` and rename it (*use the Name property*) in the OI to `propertiesForm`. Set its `Height` to 285, its `Width` to 400 and its `Caption` to `TLabel properties demo`. Then from the Standard Palette page drop two radio-groups and one check-group on the form.

Chapter 11 DISPLAY CONTROLS

Hold down the [Shift] key and click on each group-box component in turn so that all three are selected (*their tiny black square sizing grips become grey, and a new enclosing rubber band with grips appears*). In the OI you'll see that some property values are now blank, and others shared in common are visible.

Set the common `Top` property to 150, `Width` to 125 and `Height` to 130. Check that the controls do not overlap horizontally, and right-click on the common selection and choose *Align...* from the popup menu. Select the *Space equally* radio button in the *Horizontal* groupbox. *No change* is already selected in the *Vertical* groupbox. Click [OK] to align the groupboxes and close the *Alignment* dialog (see *Figure 11.2*).

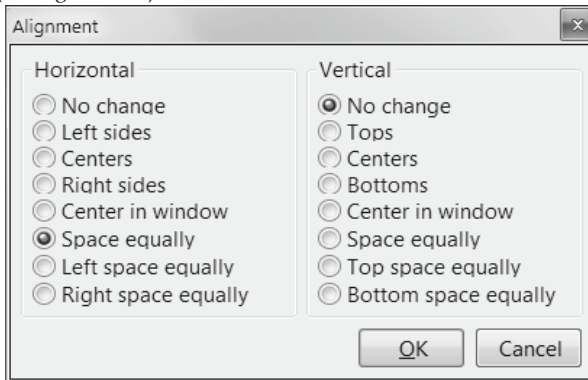


Figure 11.2 The Designer's Alignment tool

In the OI change the name of one radio-group to `rgAlignment`. Click the ellipsis button in the value column for the `Items` property. In the Strings Editor add the three items:

```
taLeftJustify
taRightJustify
taCenter
```

Set the `ItemIndex` property to 0, and the `Caption` property to `Alignment`.

Change the name of the other radio-group to `rgLayout`. Click the ellipsis button in the value column for the `Items` property. In the Strings Editor add the three items:

```
tlTop
tlCenter
tlBottom
```

Set the `ItemIndex` property to 0, and the `Caption` property to `Layout`.

Change the name of the check-group to `cgFontStyle`. Click the ellipsis button in the value column for the `Items` property. In the Strings Editor add the four values:

```
fsBold
fsItalic
fsStrikeout
fsUnderline
```

Set the `Caption` property to `Font.Style`.

We are going to create several instances of `TLabel`, a class declared in `stdctrls`, so add `stdctrls` to the `uses` clause of the unit. (When you drop a label on a form Lazarus adds `stdctrls` to the `uses` clause automatically. We have to do the same thing manually here).

In the code for unit `labelForm`, add a `const` declaration below the `uses` clause:

```
const limit = 3;
      cWidth = 100;
      cHeight = 35;
```


Chapter 11 DISPLAY CONTROLS

Then, with the form selected, click on the OI's *Events* tab, and double-click on the blank space beside the *OnCreate* event property, and in the skeleton Lazarus writes for the new event handler type `SetupLabelGrid`; so it looks like this:

```
procedure TpropertiesForm.FormCreate(Sender: TObject);
begin
    SetupLabelGrid;
end;
```

Now we need to write the code for `SetupLabelGrid`. In the **private** section of the form declaration add new lines so that it looks as follows (*after deleting the public section*):

```
TpropertiesForm = class(TForm)
    cgFontStyle: TCheckGroup;
    rgAlignment: TRadioGroup;
    rgLayout: TRadioGroup;
private
    labelGrid: TLabelGrid;
    procedure SetupLabelGrid;
    function CreateLabel(aCol, aRow: integer): TLabel;
end;
```

Lazarus has already added the names and declarations of the check-group and radio-group controls we dropped on the form. They have been added at the top of the **class** declaration in a **published** section of the class (*this initial section is managed by Lazarus to keep it in sync with the controls shown in the Designer, and the controls are all published, though the section does not explicitly state that*).

We have added a new **private** data field (`labelGrid`), and two new **private** methods, a function `CreateLabel`, which returns a new instance of a label located at a grid coordinate, and a procedure `SetupLabelGrid` (*called by the OnCreate handler*). The new `labelGrid` field is of type `TLabelGrid`. We need to declare this new type as follows (*after type and before TpropertiesForm*):

```
TLabelGrid = array[0..limit, 0..limit] of TLabel;
```

This is a two-dimensional array (*columns and rows*) of labels. Arranging the array in two dimensions like this aids in specifying the (*Top, Left*) data for each label arranged in a grid pattern.

Place the cursor within the `CreateLabel()` function declaration and press [Shift][Ctrl][C] to invoke Code Completion. Lazarus writes skeleton bodies for the two methods we have declared (`SetupLabelGrid` and `CreateLabel`) which have no implementation so far. Fill out the code skeleton for `CreateLabel` as follows:

```
function TpropertiesForm.CreateLabel(aCol, aRow: integer): TLabel;
var sum: integer;
begin
    result := TLabel.Create(Self);
    result.Parent := Self;
    result.Caption := Format('col:%d, row:%d', [aCol, aRow]);
    result.AutoSize := False;
    result.SetBounds(aCol*cWidth, aRow*cHeight, cWidth, cHeight);
    sum := aCol + aRow;
    if (sum = 1) then sum := 4;
    result.Color := clInfoBk + sum;
end;
```


Chapter 11 DISPLAY CONTROLS

The last three lines simply ensure that adjacent labels in the grid are coloured differently, so the boundary of each label is clearly visible (*this saves writing considerably more code to draw separating grid lines between each label*). If all the labels have the same background colour they merge into each other when adjacent, and the layout and alignment changes we want to demonstrate are much harder to see.

After creating a new label its `Parent` property is set to the form (`Self`) to ensure correct display, its `Caption` is written identifying the label by its column and row, its `AutoSize` property is turned off, and its dimensions are set to fit its position in the grid, using the `SetBounds` procedure that is available to all components.

Next, fill out the `SetupLabelGrid` procedure which simply calls `CreateLabel()` for each 'cell' position in the grid, as follows:

```
procedure TpropertiesForm.SetupLabelGrid;
var c, r : Integer;
begin
  for c := 0 to limit do
    for r := 0 to limit do
      begin
        labelGrid[c, r] := CreateLabel(c, r);
      end;
    end;
  end;
```

Lastly we need to add three event handlers for the three groupboxes to ensure that changed selections there propagate to change the labels.

Start with `rgAlignment`. Click on it in the Designer to select it, and in the OI Events page double-click beside the `OnClick` event. Complete the generated skeleton code as follows:

```
procedure TpropertiesForm.rgAlignmentClick(Sender: TObject);
var rg: TRadioGroup;
    c, r, idx: integer;
    algnment: TAlignment;
begin
  rg := TRadioGroup(Sender);
  idx := rg.ItemIndex;
  algnment := TAlignment(idx);
  for c := 0 to limit do
    for r := 0 to limit do
      labelGrid[c, r].Alignment := algnment;
    end;
  end;
```

The radio-group sends a parameter in its `OnClick` event identifying itself. We cast this `TObject` parameter to be a `TRadioGroup` (*since we know its origin, this is a safe cast*) and then read its `ItemIndex` property. `ItemIndex` specifies which item in the radio-group has just been selected. Since the items have been deliberately added to the radio-group in the order they are declared in the enumerated type `TAlignment`, we can simply cast `idx` to be `TAlignment` to get a correct `TAlignment` value, which is then applied to every label in the grid in nested `for do` loops.

Generate a similar `OnClick` event handler for `rgLayout` and complete it as follows:

Chapter 11 DISPLAY CONTROLS

```
procedure TpropertiesForm.rgLayoutClick(Sender: TObject);
var rg: TRadioGroup;
    c, r, idx: integer;
    layout: TTextLayout;
begin
    rg := TRadioGroup(Sender);
    idx := rg.ItemIndex;
    layout := TTextLayout(idx);
    for c := 0 to limit do
        for r := 0 to limit do
            labelGrid[c, r].Layout := layout;
        end;
    end;
```

This is very similar to `rgAlignmentClick()`, but here we cast the radio-group's `ItemIndex` property to a different enumerated type, `TTextLayout`. Again the `Items` in the radio-group are arranged to mirror the declaration of this type. Take care with the event handler for the `Font.Style` check-group. In the *Events* tab of the OI we do not this time want the `OnClick` event, but rather the `OnItemClick` event. Fill out the skeleton for it as follows:

```
procedure TForm1.cgFontStyleItemClick(Sender: TObject; Index: integer);
var c, r: integer;
    fs: TFontStyle;
begin
    fs := TFontStyle(Index);
    for c := 0 to limit do
        for r := 0 to limit do
            if cgFontStyle.Checked[Index]
            then labelGrid[c, r].Font.Style := labelGrid[c, r].Font.Style + [fs]
            else labelGrid[c, r].Font.Style := labelGrid[c, r].Font.Style - [fs];
        end;
    end;
```

The `OnItemClick` event of a checkgroup provides the `Index` of the currently clicked checkbox as a parameter we can use directly. We cast this to the `TFontStyle` type, and then loop through each label applying this set element, adding it to the `Font.Style` set or removing it from the set as appropriate.

Note: You cannot use the `Include()` and `Exclude()` procedures with set properties, only with set variables.

Press [F9] to compile and run the application. Clicking on any selection in one of the groupboxes will immediately reformat the grid of labels accordingly, giving you immediate feedback about the purpose and functionality of these label properties. Many other LCL components which display text have similarly named properties, though `Color` and `Font` are much more widely implemented in text controls than `Alignment` or `Layout`.

`TLabel` has many other useful properties, among them being the ability to display multiple lines of text that wrap to successive lines. You set the `WordWrap` property to `True` for this functionality, and also have to set `AutoSize` to `False`.

11.c Display controls: `TStaticText`

In addition to `TLabel`, there is a further text-display component called `TStaticText`, located on the Additional page of the Palette. Why have two such components? `TLabel` is a lightweight control, consuming little memory, and it is not windowed, meaning it cannot receive focus, and cannot serve as a **container** for other controls dropped on it. `TStaticText` is a windowed control, and it can accept other controls, and can participate in the tab order of the form (*it has a `TabStop` and `TabOrder` property, unlike a `TLabel`*).

Chapter 11 DISPLAY CONTROLS

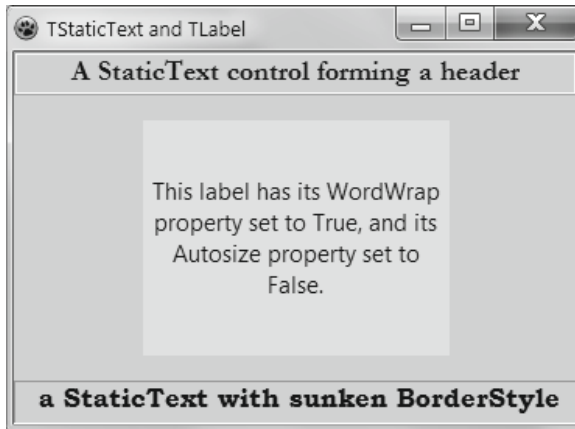


Figure 11.3 TStaticText instances and a label used for display

Although it lacks the `WordWrap` and `Layout` properties of `TLabel`, `TStaticText` does have a `BorderStyle` property, and so can serve as a useful static header-type control. `TStaticText` has an `AutoSize` property like `TLabel`, but unlike `TLabel` this is `False` by default. You would rarely want to set the `Align` property of a `TLabel` to anything other than its default, but it might make sense to do that for a `TStaticText`. Figure 11.3 shows a form with two static text controls aligned top and bottom. Experiment yourself with the behaviour of this text control.

11.d Display controls: TBevel and TDividerBevel

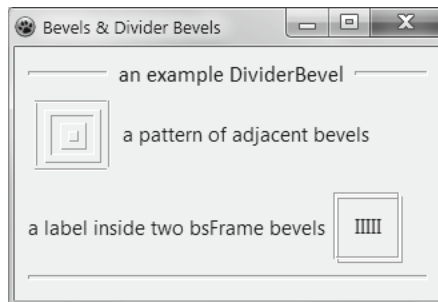


Figure 11.4 Bevels and divider bevels in use

`TBevel` and `TDividerBevel` are 'cosmetic' serving simply to help group or compartmentalise other controls on a form. They cannot receive the focus or have other controls dropped on them. They are lightweight controls compared to labels and panels, and so use fewer resources (*a bevel has 30, and a divider-bevel 31 published properties; whereas a label has 56, and a panel 71 published properties*).

A bevel (*on the Additional Palette page*) is a visual rectangular frame, and its 'inside' is empty. Through the `Shape` property you choose which sides of the rectangle are displayed, whether it is a 'box' or a 'frame', and whether it has a raised `Style` (*giving an embossed look*) or a lowered `Style` (*giving an engraved look*). It is useful for providing the impression of 3-D shading in areas of a form.

A `TDividerBevel` is a similar component found on the `LazControls` Palette page. It differs mainly in having `Caption` and `CaptionSpacing` properties rather than `Shape` and `Style` properties. This lets you easily name a section within a form. It forms a horizontal divider rather than a frame, and in Lazarus 1.1 can also be used as a vertical divider. If I had been asked to name it, I think I would have called it `TCaptionBevel`. See Figure 11.4.

11.e Display controls: TListBox

The *Standard* page's listbox is the simplest control for displaying lists of text items which are selectable. The list of strings is held in a `TStrings` property called `Items` (see Chapter 15 for more about the `TStrings` class and its descendants).

Listboxes have a `Columns` property whose default value is 0, meaning it is turned off. Setting this property to values higher than 1 arranges the list of items in the corresponding number of columns (not all platforms support the column functionality). Clearly this suits shortish items, and works best if each item is approximately the same width. Over-long items are clipped by items in adjacent columns if there is not sufficient room to display them, and the listbox automatically displays scrollbars if required.

The listbox can be set to have only one item selectable, or if `MultiSelect` is `True` it will allow many items to be selected. It will sort items alphabetically if the `Sorted` property is set to `True`. There is an example in which a `TListBox` is used in Chapter 12 (see Section c: *Editing integers...*)

11.f Display controls: TStatusBar

We conclude this chapter by considering the statusbar, a control that looks similar on most operating systems. You find it on the *Common Controls* Palette page. Drop a statusbar on an empty form in a new GUI project. Notice that it aligns itself to the bottom of the form automatically (just as a main menu component aligns itself to the top of a form automatically).

The simplest type of statusbar is obtained by setting its `SimplePanel` property to `True` (the default). In this state the control provides a single panel capped by a triangular sizing grip (the `SizeGrip` property is `True` by default, but can be turned off). Text assigned to the `SimpleText` property (either using the OI or in code) is shown in a single line at the bottom. If the text is too long to fit, it does not scroll, it simply gets truncated. Nor does over-long text get wrapped if you set the `AutoSize` property to `False` and increase the `Height`.

To have several separate sections (*panels*) in the statusbar, as you find in most word processors and the IDE's own Editor, you have to add panels either in code, or more easily by clicking on the [...] ellipsis button beside the `Panels` property to open a Panels Editor. Clicking repeatedly on the [+ Add] button adds as many panels as you wish. Each panel has `Alignment`, `Bevel`, `Style`, `Text` and `Width` properties. The default `Width` is 50, which does not accept much text (that is 50 pixels, not 50 characters).

Let's construct a simple demonstration example. Start a new Lazarus project named `statusbar_demo.lpr`, with a form unit named `statusbar_form.pas`. Set the form's `Caption` to `StatusBar demonstration`, and its `Width` to 400.

From the *Common Controls* Palette page drop a `TStatusBar` on the form naming it `statusbar`. Edit its `Panels` property as described above, adding four panels. Scroll the OI treeview until you see all four panels listed (the list starts with 0 - `TStatusPanel`). Hold down [Ctrl] and click on the first two panels to select them. On the *Properties* page of the OI type 110 as the value for the `Width` of these two panels, and press [Enter] to set the value at 110. In a similar way set the `Width` of the last two panels to 90.

Next add a selection of controls to the form, a `TLabel`, a `TCheckBox`, a `TProgressBar`, a `TBevel`, a `TBitBtn`, a `TEdit`, a `TSpinEdit`, and a `TButtonPanel`. Use the *Components* dialog ([Ctrl][Alt][P]) to locate these if you need to. Now select **all** the dropped controls except the statusbar by holding down [Shift] and clicking on each control in turn.

Chapter 11 DISPLAY CONTROLS

Click the *Events* tab in the OI, and you will see just a few events listed (*these are the events common to all the selected controls*). Double-click on the row beside `OnMouseMove` to generate an event handler that will be connected to each of the selected controls. Lazarus will name the event handler after the first control you selected, and move the cursor to the skeleton code in the Editor.

We want to rename this common event handler, so click the name in the *Events* page of the OI and shorten the name to just `MouseMove`, and press [Enter] to set that as the name. Complete this event handler skeleton as follows:

```
procedure TForm1.MouseMove(Sender: TObject;
                           Shift: TShiftState; X, Y: Integer);
var ctrl: TControl;
begin
    statusBar.Panels[0].Text:= Sender.ClassName;
    ctrl := TControl(Sender);
    statusBar.Panels[1].Text:= ctrl.Name;
    statusBar.Panels[2].Text:= Format('Top: %d', [ctrl.Top]);
    statusBar.Panels[3].Text:= Format('Left: %d', [ctrl.Left]);
end;
```

Compile and run this project. As you pass the mouse cursor over the different controls on the form you should see information about that control displayed in the various statusbar panels. If the information does not appear for one or more of the controls, close the running project and click on the control that is not yielding information.

In the OI *Events* page check that its `OnMouseMove` event is set to the (*only*) value that appears in the drop-down list when you click the down-arrow (*because sometimes when assigning an event to several selected controls Lazarus seems to miss one or two*). Then run the project again. It will look something like Figure 11.5, though of course the way you have laid out the various controls will be rather different.

Hopefully you are starting to get a feel for some of the common properties shared by components, and learning where to find them on the Palette. Also note the difference (*in the first two statusbar panels*) between the `ClassName` and the instance `Name` of each component. The `ClassName` is fixed, and belongs to the class itself. The instance `Name` is the name you give that instance (*which might be the default name Lazarus provides, as in this case – or did you name each component yourself?*).

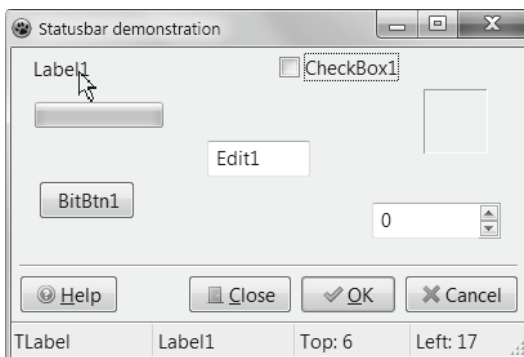


Figure 11.5 Statusbar panels displaying Label1 information

Chapter 11 DISPLAY CONTROLS

If you have two labels, each has the identical `ClassName` of `TLabel`, but each will have a different `Name` (*Lazarus does not let you name two components identically on the same form – try it and see what happens*). The `ClassName` method is a special type of function called a **class method** which you can invoke on the class type itself (*as well as on instances of the class*).

11.g Display controls: further options

Lazarus offers several other selection and display-oriented controls, and makes wide use itself in the IDE of the `TTreeView` component, and to a lesser extent the `TListView` component (*found on the Common Controls Palette page*). These are slightly more complex controls to understand and use. The `TTreeView` is demonstrated in an example in Chapter 12 (*Section e, A component browser*).

11.h Review Questions

1. What makes the difference between a control that can accept focus and one that cannot?
2. Compare several Palette components. What properties do they share in common? What events do they have in common? To check your answer, drop them on a form, select all of them, and see what common properties are showing in the *OI Properties* and *Events* pages.
3. Build a project modelled on the `label_properties` project that shows the effect of changing the properties of a `TPanel`. You won't need a grid of panels, just a single one will suffice. Your project should be able to change the `BevelInner`, `BevelOuter`, `BevelWidth`, `BorderStyle` and `BorderWidth` properties, and display the effects of changes. (*Hint: the `TSpinEdit` component is useful for editing integer values*).



Chapter 12 GUI EDIT CONTROLS

All programs need ways for the user to interact with them, whether to choose settings, enter names or other data, or to accept user-determined changes. Controls that accept key-presses (*or mouse clicks or gestures/touch*) as a means of entering or altering data are generally termed editors. They are not restricted just to text entry.

Editing controls must be able to receive the **focus** in an application, so the user is clear which control is the one currently receiving user input. Small- to medium-size controls usually indicate which is the currently-focused component in a multi-component form by means of a **focus rectangle**.

Larger GUI controls such as word processor windows and the IDE's Source Editor indicate focus usually by colouring a tab or page differently from other tabs and pages, and sporting a flashing cursor to indicate where in a line of text the focus of text entry will be. Often the current entry field in an editing control is highlighted in some way to narrow the focus appropriately.

Lazarus controls use the boolean `TabStop` and integer `TabOrder` properties to manage focus, as well as several methods including `SetFocus`, `RemoveFocus`, `CanFocus`, `Focused`, `SelectNext`, and `PerformTab`. All editing components that have this functionality are direct or indirect descendants of `TWinControl` (*a key LCL class that is the first in the component class hierarchy to provide specific focus functionality*).

If you are not familiar with the idea of focus moving from one control to the next, set up a new Lazarus project, drop several controls (*such as `TEdit` or `TMemo`*) on the main form and experiment with the `TabOrder` and `TabStop` properties to see how changing these properties affects the behaviour of the compiled project when you press the [Tab] or [Shift][Tab] keys. In some OSs the [Enter] key works like the [Tab] key for changing focus, or can be configured to do so.

12.a Editing short phrases: `TEdit` and `TLabelledEdit`

The most commonly encountered editor component is the `TEdit` class and its sister `TLabelledEdit`. `TEdit` lives on the *Standard*, and `TLabelledEdit` on the *Additional Palette* page. The editing functionality of these two edit components is identical, however the `TLabelledEdit` has additional properties relating to its attached label.

Both flavours of edit are **single-line** editors only. Although you can set an edit's `Height` much higher than its default value (*first setting `AutoSize` to `False`*), the text field never wraps round to fill the larger region. To edit multiple lines you need a `TMemo` or `TSynMemo` or `TSynEdit`.

The most important property is `Text`, which holds the string being edited, which is often set to the empty string `''` before use, but can be set, of course, to some other default value.

A program accepting address information via edit fields might have an `edtCountry` edit field set to the country of the user, as a very commonly needed default.

Several other properties relate to the editing functionality. `ReadOnly` (*when `True`*) prevents editing, though gives no visual indication that [BkSpace], [Delete] and character keys have no effect – the cursor moves as usual, and the current `Text` is displayed. Whereas setting `Enabled` to `False` greys out the entire control, and prevents it gaining focus, giving a clear visual indication of the unresponsiveness of the edit in that state.

Setting `EchoMode` to `emPassword`, and `PasswordChar` to a value other than `#0` masks key entry. If you set it to `'*'` then the edit displays only asterisk characters as you type in the edit field.

However the `Text` property contains the characters as typed (*i.e. no asterisk characters, unless you happen to have typed one*).

Chapter 12 GUI EDIT CONTROLS

Alignment affects the edited Text in a similar way to a label's Alignment. MaxLength prevents entry of characters beyond an upper limit (*a 0 value turns the property off*). If CharCase is set to a value other than ecNormal then you can force the edited Text to be all upper case or all lower case.

To try out the effects of alterations in these properties, start a new Lazarus project named edit_props and enlarge the form size to be about 600x600.

Drop a TEdit and a TToggleBox from the *Standard* page, and a TLabelledEdit from the *Additional* page on the left of the form, below each other (*the order is not critical*). Set the Caption of the togglebox to 'Point grid to LabeledEdit'. You'll need to enlarge the togglebox to show this four-word caption.

Drop a TTIPropertyGrid component (*second from the rightmost on the RTTI Palette page*) on the form and set its Align property to alRight. Drag its left border leftwards to make the grid wider. Click on the down-arrow beside the TIOBJECT property and set it to Edit1. Double-click on the toggle-box and in the ToggleBox1Change() event handler skeleton type the following:

```
procedure TForm1.ToggleBox1Change(Sender: TObject);
begin
  case ToggleBox1.Checked of
    False : begin
      TIPropertyGrid1.TIOBJECT := Edit1;
      Edit1.SetFocus;
      ToggleBox1.Caption:= 'Point grid to LabeledEdit';
    end;
    True: begin
      TIPropertyGrid1.TIOBJECT := LabeledEdit1;
      LabeledEdit1.SetFocus;
      ToggleBox1.Caption:= 'Point grid to Edit';
    end;
  end;
end;
```

Compile and run the program. Clicking the toggle-box sets the property grid to display properties from the edit that is not the one currently shown in the grid. You can use this **runtime** property grid just as you use the OI at design time to alter the current edit's properties, and you see the effects immediately reflected in the behaviour of the running program. Try changing the Color, Cursor, CharCase ... and in the labeledEdit see the effect of changing the values of LabelPosition and LabelSpacing. **Notice** that the EditLabel property (*being a subcomponent*) is greyed out, and cannot have its properties edited in this otherwise pretty amazing grid-style runtime property editor.

Compared to the amount of code we wrote for the event handlers in examples given earlier (e.g. label_properties, in Chapter 10, Section 10.b) use of the RTTI component leads to almost code-free programming. Why did we not use RTTI components earlier, then? It would certainly have introduced you to several new components, and reduced the amount of code you had to write. But 'quicker and easier' is not our goal here. Learning to program involves typing code, just as learning to write a novel involves practice writing as well as lots of reading.

Chapter 12 GUI EDIT CONTROLS

12.b Editing or choosing short phrases: TComboBox

Like a `TEdit` the `TComboBox` allows you to edit text in its edit field, but it also offers a list of text items to choose from. The `Style` property gives the option for the list either to be dropped down by clicking on a down-arrow icon, or to be a list of choices that is shown permanently. As in `TListBox` the text strings in the drop-down are stored in a stringlist `Items` property.

If `ReadOnly` is `True` no text can be typed in the edit field, but any item selected from the drop-down list is returned in the `Text` property, which reflects the content of the edit field when focus leaves the control. This allows you to control whether the user can enter new text to be returned, or only be permitted to choose from the predefined `Items`. You can set `Text` directly to a default string value to be shown on entry to the control, or the `ItemIndex` value can be set to a positive value which selects the item in `Items` with that index, and displays it in the edit field. Figure 12.1 provides a typical example.

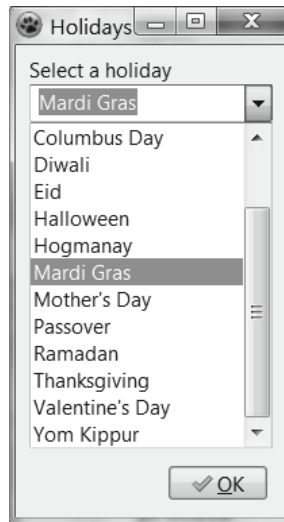


Figure 12.1 A `TComboBox` used for data entry

12.c Editing integers and floating point numbers

Two editors, `TSpinEdit` and `TFloatSpinEdit` found on the *Misc* Palette page provide simple number-specific editing abilities, and prevent input of anything non-numeric. The numeric values are entered either by directly typing digits into the text number field, or by incrementing (or decrementing) the existing number in the field using up/down arrow keys or mouse clicks on the up/down arrow icons embedded in the control. Each OS gives its spinedit widget a slightly different appearance, but whatever the platform you will recognise this control easily.

One point of note is that the published `Value` property is not a string but numeric, an integer for `TSpinEdit`, and a double for `TFloatSpinEdit` (there is a public `Text` string property – but you will rarely if ever need to use it). This is the principal advantage of using these controls: there is no need to make manual type conversions between string and numerical values. User input can be read directly as an integer or float in the `Value` property, even when entered as text in the number field. These controls also provide **validated** numeric user input. If you use text-entry `TEdits` to obtain numeric data from users you will have to validate the entered strings yourself.

Chapter 12 GUI EDIT CONTROLS

We will exercise the `SpinEdit` control with a simple arithmetical example that also demonstrates use of the `math` unit. Start a new project named `numberediting` with a main form named `numeditmain.pas`. Set the form's `Caption` to `SpinEdit example`.

Add `math` to this unit's `uses` clause. Drop two labels at the top of the form named `lblA` and `lblB`, and set their captions to `integer A` and `integer B`. Below the labels drop two `SpinEdit` controls named `seA` and `seB`. Increase the width of these two controls so they can accept large numbers easily, and set their properties as follows:

```
MinValue      -10000
MaxValue      10000
Value         10
```

Below the `SpinEdits` drop a `TDividerBevel` and set its `Caption` to `calculated results`. Below the bevel drop a listbox named `lbResults`. The form will look similar to Figure 12.2

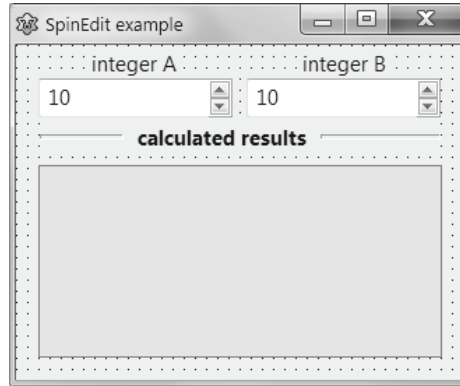


Figure 12.2 The `SpinEdit` example main form design

Click on the first `SpinEdit`, and shift-click the second `SpinEdit` to select them both. In the OI Events page double-click beside the `OnChange` event to create new event handler, then click there again to select the name, and rename it from the Lazarus default to `ChangeAorB`. In the Editor complete the skeleton for this event handler as follows:

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

We are calling a procedure `UpdateCalculations`, which we now need to write. In the `private` section of the form class declare the `procedure` `UpdateCalculations`, use Code Completion to generate the procedure skeleton and complete it as follows:

Chapter 12 GUI EDIT CONTROLS

```
procedure TForm1.UpdateCalculations;
var OK: boolean;
    n: Int64;
begin
  lbResults.Items.Clear;
  lbResults.Items.Add('A + B : ' + IntToStr(CalcSum));
  n := CalcIntDivision(OK);
  if OK then
    begin
      lbResults.Items.Add('A DIV B : ' + IntToStr(n));
      lbResults.Items.Add('A MOD B : ' + IntToStr(CalcModulus(OK)));
    end
  else lbResults.Items.Add('A DIV B, A MOD B are not calculable');
  lbResults.Items.Add('A x B : ' + IntToStr(CalcProduct));
  lbResults.Items.Add('A*A + B*B : ' + IntToStr(CalcSumOfSquares));
  lbResults.Items.Add('A to POWER B : ' +
    FloatToStr(CalcPower));
end;
```

In this procedure we call several small procedures we also need to write. So in the **private** section of the form class add the declarations for these procedures, then use Code Completion to generate the required skeletons and complete them as follows:

.....

```
private
  procedure UpdateCalculations;
  function CalcSum: int64;
  function CalcProduct: int64;
  function CalcPower: double;
  function CalcIntDivision(var isValid: boolean): int64;
  function CalcModulus(var isValid: boolean): int64;
  function CalcSumOfSquares: Int64;
end;
...
implementation

function TForm1.CalcSum: int64;
begin
  Result := seA.Value + seB.Value;
end;

function TForm1.CalcProduct: int64;
begin
  Result := seA.Value * seB.Value;
end;

function TForm1.CalcPower: double;
begin
  Result := intpower(seA.Value, seB.Value);
end;

function TForm1.CalcIntDivision(var isValid: boolean): int64;
begin
  Result := 0;
  isValid := (seB.Value <> 0);
  if isValid
  then Result := seA.Value div seB.Value;
end;
```

Chapter 12 GUI EDIT CONTROLS

```
function TForm1.CalcModulus(var isValid: boolean): int64;
begin
    Result := 0;
    isValid := (seB.Value <> 0);
    if isValid
    then Result := seA.Value mod seB.Value;
end;

function TForm1.CalcSumOfSquares: Int64;
begin
    result := seA.Value*seA.Value + seB.Value*seB.Value;
end;
```

Every time either spinedit value changes we call `UpdateCalculations` which simply writes a new set of `Items` strings in the listbox. We need to ensure this listbox update happens when the program is first displayed. We use the `OnActivate` event to do this. Select the main form, and in the *OI Events* page double-click beside the `OnActivate` event. Complete the event handler as follows (*it calls `ChangeAorB` with the dummy parameter `nil`, the event handler as we are using it ignores the `Sender` parameter it receives, so the value of the parameter does not matter*):

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    ChangeAorB(nil);
end;
```

Compile and run this project, and see the effect of altering the values of integer A and integer B. A typical program run is shown in Figure 12.3.

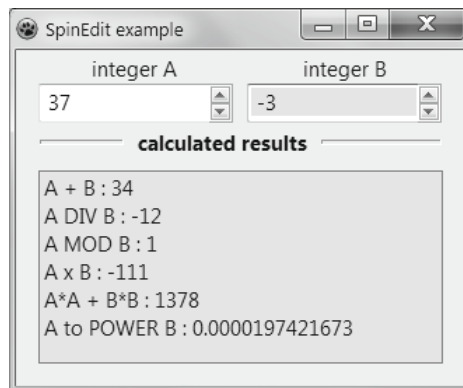


Figure 12.3 The running numberediting project

12.d Multiple-line editors

For editing short sections of text such as names to be stored in databases a single-line `TEdit` or `TDBEdit` usually suffices. However where a line of text is too long to fit within an edit control (*without wrapping round*) a more sophisticated control is needed. You may have struggled, like me, when trying to edit a section of a long path in a `TEdit`-style system widget where the single line scrolls because it is too long to be shown in the available screen width... and you lose track of where you are in a line whose two ends have scrolled out of view.

Chapter 12 GUI EDIT CONTROLS

Lazarus has several controls that offer multiple line editing capabilities, including `TMemo`, `TSynEdit` and `TSynMemo`. All these controls accept text typed into them, and will automatically display scrollbars if necessary. However the first, `TMemo`, is quite restricted in display terms: the text is shown only in one colour, in a single font (*typeface*), and any enhancement you make to the text shown (*say to underline it*) applies to the entire text. There are no Lazarus Palette components that have built-in `.rtf` capabilities, though there is a rich text component available in the code repositories (*which has not yet been deemed good enough to be packaged with Lazarus*).

The two synedit controls are more capable (*TSynEdit has over 100 published properties*), not lightweight components at all. Consequently there is more of a learning curve to be able to use these latter two components to highlight individual words in different colours and so on.

The Source Editor in Lazarus is based on `TSynEdit`, which gives you some idea of the sophistication possible with these components. However this also underlines the orientation towards source code editing rather than word processing as such, and their functionality is based on the use of mono-spaced fonts (*though this might be extended in future, to allow use of proportional fonts*). Also every line has the same height.

We conclude this chapter with a component browser example designed to let you examine the ancestry and published properties of some commonly used Palette components. It introduces the `TTreeView`, the most complex display and item-selection component encountered so far, which was omitted from the discussion of display components in Chapter 11 on account of its complexity.

12.e A component browser

Start a new Lazarus project and save it as `compbrowser`, and its main form unit as `comp_browser_main.pas`. Set the main form's `Caption` to `Component Browser`, its `Height` to 420 and its `Width` to 680. Drop a treeview from the Common Controls Palette page on the main form, setting its `Name` to `tvComps`, its `Align` to `alLeft` and its `Width` to 210.

To make a browser to cover the entire 203 Palette components would require a lot of typing of names, so here we will restrict ourselves to the first two Palette pages (*code for a full browser is included on the CD accompanying this book*). At the end of this project you will have become a lot more familiar with those first two Palette pages and their contents.

In order for the compiler to find both the components listed in the browser and various other LCL routines we need to specify the units where they are declared in our `uses` clause. So add the following units to those already in the uses of `compBrowserMain.pas`:

```
..., Menus, Buttons, StdCtrls, ExtCtrls, ActnList, MaskEdit, grids,
CheckLst, PairSplitter, ColorBox, ValEdit, SynHighlighterPosition, strutils,
typinfo;
```

In order to refer to individual Palette pages we define an enumerated type as follows:

type

```
TPalettePage = (ppStandard, ppAdditional, ppOtherPagesNotListedHere);
```

Alongside this enumerated type we define a constant array of strings corresponding to these three enumerations:

const

```
MaxComponentsOnAPage = 21; // Additional page
PageNames: array[TPalettePage] of shortstring =
  ('Standard', 'Additional', 'Other unlisted pages');
```

Chapter 12 GUI EDIT CONTROLS

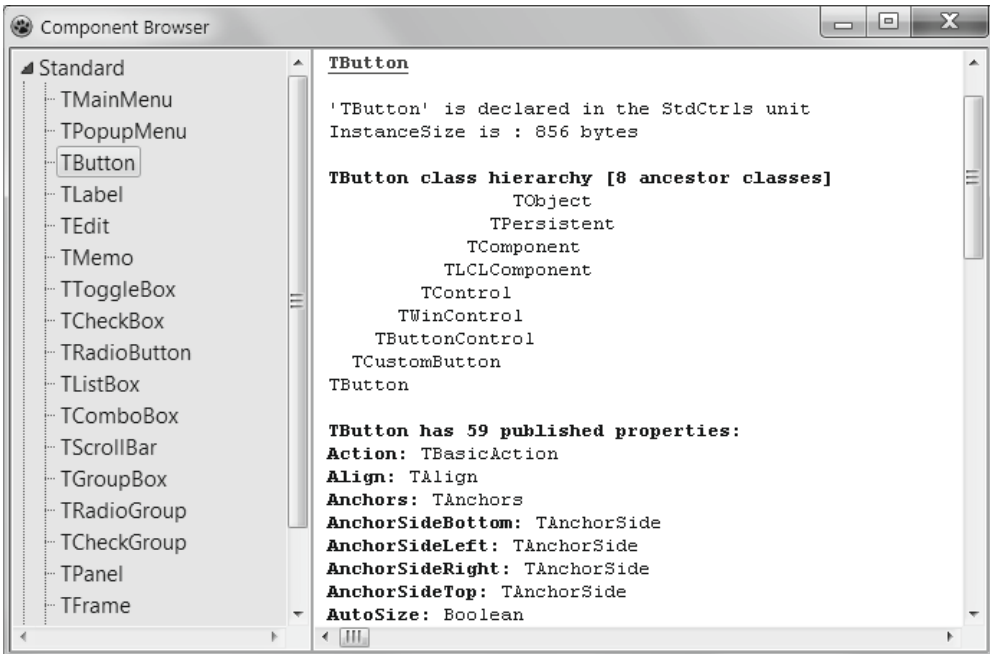


Figure 12.4 The Component Browser in action

We want our treeview to display nodes for the Palette pages, and sub-nodes for each component. Selecting a component should generate information about it using RTTI, which gets displayed in a control alongside the treeview. We will use a `TSynEdit` control for this display so that it is possible to highlight only certain aspects of the information (*which is not possible with a memo or listbox, though they are considerably simpler to use and program*).

Drop a `TSynEdit` component from the `SynEdit` Palette page next to the treeview, and set its properties as follows:

```
Name          seViewer
Gutter.Visible False
ReadOnly       True
Align          alClient
ScrollBars     ssAutoBoth
```

To pick an arbitrary component from a list and display its properties we need a variable to refer to it which is assignment-compatible with any component. The LCL declares such a type, and it is called `TComponentClass`. We declare such a variable here. We also need a procedure that will load the treeview with the initial list of Palette pages and components. In the form's **private** section add the `TComponentClass` variable, and a new procedure as follows:

```
private
  compClass: TComponentClass;
  procedure LoadTreeView;
```

Chapter 12 GUI EDIT CONTROLS

Use Code Completion to write out a skeleton for the `LoadTreeView` procedure, and complete it as follows:

```
procedure TForm1.LoadTreeView;
var aNode: TTreeNode;
    palPage: TPalettePage;
    i: integer;
begin
    tvComps.BeginUpdate;
    for palPage := High(TPalettePage) downto Low(TPalettePage) do
    begin
        aNode := tvComps.Items.AddFirst(nil, PageNames[palPage]);
        for i := 1 to MaxComponentsOnAPage do
        begin
            compClass := GetComponentClass(palPage, i);
            if Assigned(compClass) then
                tvComps.Items.AddChildObject(aNode, compClass.ClassName,
                    TObject(compClass));
        end;
    end;
    tvComps.EndUpdate;
    tvComps.Selected := tvComps.Items[0];
end;
```

To fill a treeview with items it is possible to use the treeview editor in the OI. If you select the treeview in the Designer, and navigate to the `Items` property and then click on the [...] ellipsis button you will open an editor designed for adding nodes and sub-nodes to a treeview. Once you have got used to the way this editor works (*you have to keep careful track of which node is selected before you click any buttons*) it is a convenient way of quickly adding text items to any treeview. Unfortunately in our case the treeview *Items Editor* will not suffice, because we want to add to the treeview not just text items identifying a component, but a reference to the actual component class itself, i.e. we need to add a string **and** a class reference. The `Items` list of a treeview has a function to add two such elements at once to a node:

```
function TTreeView.Items.AddChildObject(ParentNode: TTreeNode;
    const S: string; Data: Pointer): TTreeNode;
```

and the function returns a reference to the new node added, which is of type `TTreeNode`

You see that the function requires three parameters: a reference to the parent node in the treeview to which the new node is to be added, and the text description of the new item together with its associated `Data` (*in our case this will be a `TComponentClass`*).

We determine the parent node by the assignment

```
aNode := tvComps.Items.AddFirst(nil, PageNames[palPage]);
```

The root node of any treeview is `nil`, and the first (*three in our case*) identifiable nodes are generated by `AddFirst(nil, 'text')` where we use entries in the `PageNames` array (*indexed by `palPage`*) for the text entry.

The `LoadTreeView` procedure consists of two nested for loops, the outer loop steps through the values in the `TPalettePage` enumeration, and the inner loop steps through each component on the given Palette page. Since the number of components on each page varies considerably we set the inner for loop to iterate the maximum required number of times, and test the value of the `GetComponentClass()` function for `nil` (*using the `Assigned()` function*) to discard upper values where there are no components with those indices.

Which brings us to the `GetComponentClass()` function. This is not an LCL routine – we have to write it ourselves. It is a function which returns the correct component class (*or `nil`, as the case may be*) given a Palette page and component index. In the form's **private** section add the following function declaration:

Chapter 12 GUI EDIT CONTROLS

```
function GetComponentClass(aPage: TPalettePage; anIndex: word): TComponentClass;
```

Use code completion to have the IDE write the skeleton for this function, and complete it as follows:

```
function TForm1.GetComponentClass(aPage:TpalettePage;anIndex: word):TComponentClass;
begin
  case aPage of
    ppStandard: case anIndex of
      1: result := TMainMenu;
      2: result := TPopupMenu;
      3: result := TButton;
      4: result := TLabel;
      5: result := TEdit;
      6: result := TMemo;
      7: result := TToggleBox;
      8: result := TCheckBox;
      9: result := TRadioButton;
      10: result := TListBox;
      11: result := TComboBox;
      12: result := TScrollBar;
      13: result := TGroupBox;
      14: result := TRadioGroup;
      15: result := TCheckGroup;
      16: result := TPanel;
      17: result := TFrame;
      18: result := TActionList;
      else result := nil;
    end;
    ppAdditional: case anIndex of
      1: result := TBitBtn;
      2: result := TSpeedButton;
      3: result := TStaticText;
      4: result := TImage;
      5: result := TShape;
      6: result := TBevel;
      7: result := TPaintBox;
      8: result := TNotebook;
      9: result := TLabeledEdit;
      10: result := TSplitter;
      11: result := TTrayIcon;
      12: result := TMaskEdit;
      13: result := TCheckListBox;
      14: result := TScrollBox;
      15: result := TApplicationProperties;
      16: result := TStringGrid;
      17: result := TDrawGrid;
      18: result := TPairSplitter;
      19: result := TColorBox;
      20: result := TColorListBox;
      21: result := TValueListEditor;
      else result := nil;
    end;
  else result := nil;
end;
end;
```

The function has two nested **case** statements which return either a class reference or **nil** according to the parameters passed to the function. Notice in passing that we wrapped the `LoadTreeView` procedure in calls to `BeginUpdate` and `EndUpdate`. This is often done for complex visual controls both as an optimisation, and to prevent possible flicker caused by adding scores of text values to the treeview and redrawing them one by one. `BeginUpdate` postpones repainting of the control until the `EndUpdate` call, meaning it is redrawn only once at the end rather than scores of times during iterations of the for loops. When should `LoadTreeView` be called? At the start of the program – i.e. as soon as the main form has been created. Double-click beside the form's `OnCreate` event in the OI and complete the event handler the IDE writes for you as follows:

Chapter 12 GUI EDIT CONTROLS

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

Press [F9] to check your typing and test the project compilation so far. You should see the program displaying a treeview with three branches, the first two of which can be expanded to reveal a list of component names appropriate to their Palette page location. We now need to add the ability to display information about a selected component in the synEdit control. We chose the synEdit component because it enables us to mark words or phrases in bold, underline, colour etc. To do this we need a highlighter class instance, and we have to create attributes for the particular highlightings required. We also need a counter to keep track of the lines added to the synEdit. So to the **private** section of the form add the following fields:

```
Hiliter: TSynPositionHighlighter;  
atrUL, atrBD: TtkTokenKind;  
lineNo: integer;
```

Expand the form's OnCreate method body to read as follows:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    hiliter := TSynPositionHighlighter.Create(Self);  
    seViewer.Highlighter := hiliter;  
    atrUL := hiliter.CreateTokenID('atrUL', clBlue, clNone, [fsBold,  
                                                                    fsUnderline]);  
    atrBD := hiliter.CreateTokenID('atrBD', clBlack, clNone, [fsBold]);  
    LoadTreeView;  
end;
```

We need a procedure that can gather and display the component information for each node. Let's call it **procedure** DisplayComponentInfo (aNode: TTreeNode); Add this procedure to the form's **private** section, and generate a skeleton body by invoking Code Completion.

Fill it out like this:

```
procedure TForm1.DisplayComponentInfo(aNode: TTreeNode);  
begin  
    seViewer.Lines.Clear;  
    hiliter.ClearAllTokens;  
    lineNo := 0;  
    case aNode.Level of  
        0: begin  
            seViewer.Lines.Add('');  
            seViewer.Lines.Add(' (' + aNode.Text + ' Page)');  
        end;  
        else  
            begin  
                compClass := TComponentClass(aNode.Data);  
                DisplayPageInfo(aNode);  
                // DisplayComponentData;  
                // DisplayComponentHierarchy(aNode);  
                // DisplayComponentProperties;  
            end;  
    end;  
end;
```

Chapter 12 GUI EDIT CONTROLS

Notice we are including three procedures here which are commented out. The idea is to write them one by one, un-commenting each one as it is written. We can then compile and test these procedures incrementally. To the form's **private** section add a declaration for

```
procedure DisplayPageInfo (aNode: TTreeNode);
```

Generate an implementation body for this procedure and complete it as follows:

```
procedure TForm1.DisplayPageInfo (aNode: TTreeNode);  
var s: string;  
begin  
    seViewer.Lines.Add(''); inc (lineNo);  
    s := ' Palette Page: ' + aNode.Parent.Text;  
    hiliter.AddToken (lineNo, 1, tkText);  
    hiliter.AddToken (lineNo, Length (s), atrUL);  
    seViewer.Lines.Add (s); inc (lineNo);  
    seViewer.Lines.Add (''); inc (lineNo);  
end;
```

The synEdit highlighter has a rather verbose syntax, requiring both the line number to highlight, the position in the line where the highlight attribute is to be applied, and the attribute to use. Because the highlighter requires line number information, we have to track the line number as we go. Next select the treeview in the OI and double-click on the *Events* page beside the treeview's *OnChange* event. Complete this event by calling the display procedure as below, and compile and run to test the program so far.

```
procedure TForm1.tvCompsChange (Sender: TObject; Node: TTreeNode);  
begin  
    DisplayComponentInfo (Node);  
end;
```

If all is well, you can proceed, or you may need to fix a few typos.

12.f Getting RTTI information for a component

For each component listed, the program so far simply writes its Palette page in a blue underlined typeface. How do we obtain more information about a component class? Some of it is contained in properties of *TComponentClass*. Add a further procedure in the form's **private** section:

```
procedure DisplayComponentData;
```

Generate an implementation body for this procedure and complete it as follows:

```
procedure TForm1.DisplayComponentData;  
var st: string;  
begin  
    st := '' + compClass.ClassName;  
    HiLiter.AddToken (lineNo, 1, tkText);  
    HiLiter.AddToken (lineNo, Length (st), atrUL);  
    seViewer.Lines.Add (st); inc (lineNo);  
    seViewer.Lines.Add (''); inc (lineNo);  
    seViewer.Lines.Add (Format (' ''%s'' is declared in the %s unit',  
                                [compClass.ClassName, compClass.UnitName]));  
  
    inc (lineNo);  
    seViewer.Lines.Add (Format (' InstanceSize is : %d bytes',  
                                [compClass.InstanceSize]));  
  
    inc (lineNo);  
    seViewer.Lines.Add (''); inc (lineNo);  
end;
```

Chapter 12 GUI EDIT CONTROLS

Now uncomment the line in `DisplayComponentInfo` that calls this procedure we have just written, and compile and run the project to test the effect. If all works satisfactorily, we can proceed to getting and displaying information about the component's ancestry.

For this we require a helper function, `GetAncestorCount`, that counts how many levels of inheritance apply to a particular component. Declare a further function in the form's private section:

```
function GetAncestorCount (aClass: TClass): integer;
```

Generate an implementation skeleton body for it, and complete it as shown below. Note that we have to use a general `TClass` parameter here (*not* `TComponentClass`) since many ancestors higher up in the hierarchy will be classes but not components.

```
function TForm1.GetAncestorCount (aClass: TClass): integer;
begin
  result := 0;
  if not Assigned(aClass.ClassParent)
  then Exit
  else
  begin
    while Assigned(aClass.ClassParent) do
      begin
        inc(result);
        aClass:= aClass.ClassParent;
      end;
    end;
  end;
end;
```

In this loop we step up through the class's ancestry, incrementing a counter as we do so, until we reach a class whose `ClassParent` is `nil`. This means we have reached `TObject`, and can go no further. In the form class declaration add a further **private** procedure:

```
procedure DisplayComponentHierarchy (aNode: TTreeNode);
```

Generate an implementation method skeleton and complete it as follows:

```
procedure TForm1.DisplayComponentHierarchy (aNode: TTreeNode);
var sl: TStringList;
    step: integer = 1;
    ancestorCount: integer = 0;
    i: integer;
    s: string;
    aClass: TClass;

  function Plural (aCount: integer): string;
  begin
    case aCount of
      1: result := '';
      else result := 'es';
    end;
  end;
```

Chapter 12 GUI EDIT CONTROLS

```
begin
  ancestorCount:= GetAncestorCount(compClass);
  s:= Format(' %s class hierarchy [%d ancestor class%s]',
            [compClass.ClassName, ancestorCount, Plural(ancestorCount)]);
  hiliter.AddToken(lineNo, 1, tkText);
  hiliter.AddToken(lineNo, Length(s), atrBD);
  seViewer.Lines.Add(s); inc(lineNo);
  aClass:= TClass(aNode.Data);
  if Assigned(aClass.ClassParent) then
    begin
      sl := TStringList.Create;
      try
        while Assigned(aClass.ClassParent) do
          begin
            sl.Add(DupeString(' ', step) + aClass.ClassName);
            aClass:= aClass.ClassParent;
            inc(step, 2);
          end;
        sl.Add(DupeString(' ', step) + aClass.ClassName);
        for i := sl.Count-1 downto 0 do
          begin
            seViewer.Lines.Add(sl[i]);
            inc(lineNo);
          end;
        finally
          sl.Free;
        end;
      end
    else begin seViewer.Lines.Add('      (No parent class)'); inc(lineNo); end;
end;
```

We are using the very useful `TStringList` class here that is discussed more fully in Chapter 15 (see section c) in order to list the names of the ancestor classes (*whose number is not known in advance*). Moreover we indent the class names successively as we move up through the hierarchy using the `DupeString` function we first encountered in Chapter 7 (Section 7.h).

We also use the `try ... finally ... end;` construct to make sure that if something goes wrong after we create the needed stringlist that the memory allocated for the stringlist will still be freed, and no memory leak will occur. Note how we initialised the step variable at the point of declaration, and included a nested helper `Plural()` function to get the correct spelling distinction between *class* and *classes*. Uncomment the `DisplayComponentHierarchy` procedure in `DisplayComponentInfo`, and compile and run the program to check your typing so far. Whew! We come to the final and most complex procedure: `DisplayComponentProperties`, which uses a number of types and functions declared in the `typinfo` unit, some of which are slightly abstruse, and make heavy use of pointers. You may wish just to accept the following code for now and come back to study it later. Declare this last procedure in the private section of the form declaration:

```
procedure DisplayComponentProperties;
```

Generate an implementation skeleton for it, and complete it is as follows:

Chapter 12 GUI EDIT CONTROLS

```
procedure TForm1.DisplayComponentProperties;
var aPPI: PPropInfo;
    aPTI: PTypeInfo;
    aPTD: PTypeData;
    aPropList: PPropList;
    sortSL: TStringList;
    i: integer;
    s: string;
begin
    seViewer.Lines.Add(''); inc(lineNo);
    aPTI:= PTypeInfo(compClass.ClassInfo);
    aPTD := GetTypeData(aPTI);
    s := Format(' %s has %d published properties:',
               [aPTI^.Name, aPTD^.PropCount]);
    hiliter.AddToken(lineNo, 1, tkText);
    hiliter.AddToken(lineNo, Length(s), atrBD);
    seViewer.Lines.Add(s); inc(lineNo);
    if (aPTD^.PropCount = 0)
    then seViewer.Lines.Add(' (no published properties)')
    else
    begin
        GetMem(aPropList, SizeOf(PPropInfo)* aPTD^.PropCount);
        sortSL := TStringList.Create;
        sortSL.Sorted:= true;
        try
            GetPropInfos(aPTI, aPropList);
            for i := 0 to aPTD^.PropCount - 1 do
                begin
                    aPPI := aPropList^[i];
                    sortSL.AddObject(Format(' %s: %s',
                                             [aPPI^.Name, aPPI^.PropType^.Name]),
                                     TObject(Pointer(Length(aPPI^.Name))));
                end;
            for i := 0 to sortSL.Count - 1 do
                begin
                    seViewer.Lines.Add(sortSL[i]);
                    hiliter.AddToken(lineNo, Succ(Integer(sortSL.Objects[i])), atrBD);
                    hiliter.AddToken(lineNo, Length(sortSL[i]), tkText);
                    inc(lineNo);
                end;
            finally
                FreeMem(aPropList, SizeOf(PPropInfo)* aPTD^.PropCount);
                sortSL.Free;
            end;
        end;
    end;
end;
```

Uncomment the final commented procedure in `DisplayComponentInfo` and compile and run the program, which will appear similar to Figure 12.4. You should have a working component browser that can be extended to cover all LCL Palette components, if you wish. You might want to add it to the IDE as an external tool you can call up to use when developing other projects (*see the next Section*).

12.g Adding external tools to the IDE

Lazarus provides a way to add any executable to the IDE main menu. We'll use the executable you just built as an example of how you can extend the IDE in this way. From the main menu choose **Tools | Configure External Tools...** to bring up the *External Tools* dialog. If you have not yet added any extra tools to the Tools menu this dialog will be an empty box with a toolbar at the top and [Help], [OK] and [Cancel] buttons at the bottom. If you have already installed one or more tools in the menu they will be listed in this dialog. Click on the **+ Add** toolbutton (*the leftmost button*).

Chapter 12 GUI EDIT CONTROLS

This opens a further *Edit Tool* dialog. Type a suitable name for the tool you are adding in the *Title* field, then press the [...] ellipsis button in the *Program Filename* field.

This brings up a system *Open File* dialog which you can use to locate the executable you want added as a tool. In Figure 12.5 you can see the entry for Component Browser as located on the author's computer. Note the checkbox highlighted in grey in the illustration. For some reason this is always checked by default. Make sure you **uncheck** this option (*otherwise the tool you execute will be invisible*).

There are options for command-line parameters that may be needed for some tools (for which a variety of macros are provided), and the possibility to assign a shortcut key you can use to execute the tool. Click [OK], and [OK] again to dismiss the *External Tools* dialog.

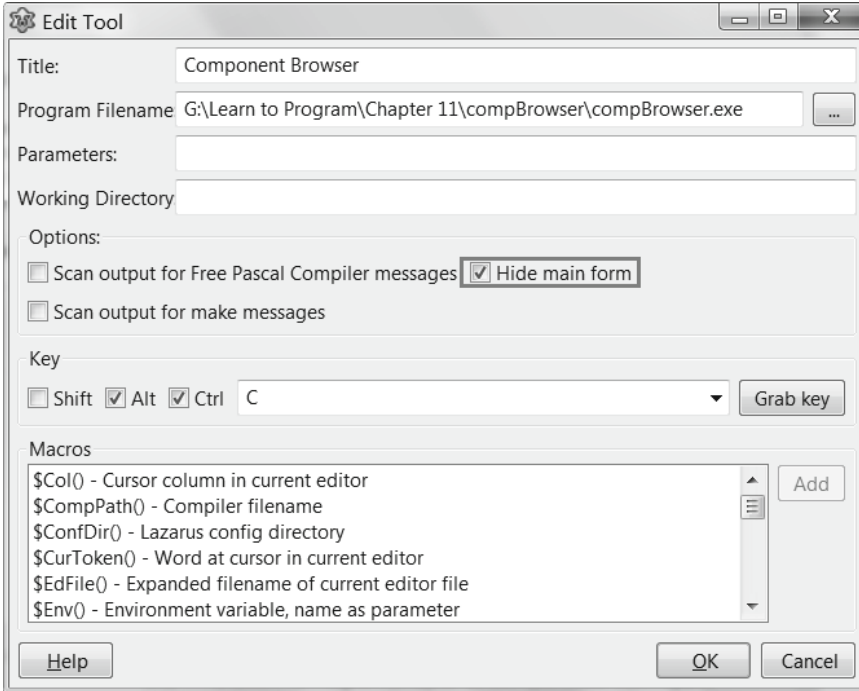


Figure 12.5 The Edit Tool dialog for adding an external tool to the IDE

Chapter 12 GUI EDIT CONTROLS

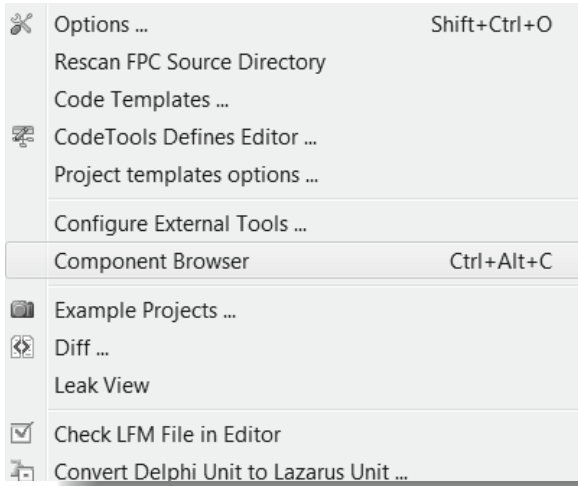


Figure 12.6 The Tools menu with a new Component Browser entry

The tool you have added will now have its own place in the Tools menu (see Figure 12.6). You can execute the tool either by clicking on its menu item, or by using its shortcut key combination (if you assigned one).

12.h Review Questions

1. Try extending the Component Browser to show further Palette pages. You need to extend just one function, `GetComponentClass`. Can you figure out how to do this? If you're stuck, look at the project included in the code that accompanies this book.
2. When would you prefer a `TComboBox` over a `TEdit` for accepting user input?



Chapter 13 LAZARUS GUI PROJECTS

We have enough Pascal under our belts at this stage that several of the provided examples will not be throw-away programs but rather small projects that could be extended to be genuinely useful: worth naming and keeping (*such as the last chapter's Component Browser*). This is also a stage at which it is worth reviewing some important aspects of project development.

13.a Planning a project

A programming project starts with a goal or goals you want to achieve, and as you ponder these a plan for achieving your goal may crystallise. Simple projects often develop via a trial-and-error method, and Lazarus as a RAD environment often enables you to code the outline of a project quite quickly. From the early sketch program you can develop different parts individually to shape the overall project to fit your goal. The modular nature of Pascal facilitates such a divide-and-conquer approach. However, starting with the UI can constrain you by focusing on **how** you will achieve a goal, rather than identifying exactly **what** functionality you need to create. To someone with a UI hammer, everything can seem to be a potential nail.

Once the principal functionality the project will provide has become clear you might sketch out the user interface (*UI*) on paper. Then ask yourself questions such as the following to help you delineate the tasks ahead.

- What are the main aims of this software? Can I see developing a few modules that together will meet these aims? What will I **name** them?
- What are the **essential** features, and which parts are decorative?
- Can I envisage developing a new **class** that will encapsulate the functionality I need? If so, what will the class be called?
- Does the program need a database? If so, which one(s)?
- How many forms (*windows*) are needed? Where I see several alternative ways to implement a feature how shall I choose between them?
- Are custom dialogs needed for particular functionality? How will I test them?
- How will I go about testing the functionality and UI?
- Which parts take me into areas where I need to research or learn new techniques? Who can help me with areas where I am not fully competent? Can I pay someone to code parts that are beyond my skills, or which I don't have time to get up to speed on?
- What time/money constraints (*if any*) impinge on this project?

13.b Creating a project task list

It is helpful to develop a checklist of tasks that will have to be completed by the end of the project development process. It is rewarding to see such list items getting ticked off as you complete them, and it also helps you remember tasks and ideas that you might otherwise think of as you write code, but later overlook or forget. The list also gives a basic sort of project development history. The IDE has good support for a task list comprising `ToDo` comments, which is explained in the following section.

Some people find comments (*even short ones*) distracting clutter in source files, and are averse to inserting `ToDo` lines for the same reason. If you feel like that you might prefer simply to maintain a `ProjectTasks.txt` file in your project directory to keep track of this aspect. You could format this file however you wanted, and use the Project Inspector to add it to the project (*it is then simple to double-click it in the Project Inspector to open the file in the Editor to tick off completed tasks, add new ones and so on*). The IDE's `ToDo` functionality is available in any text file you edit in Lazarus, so you might combine this approach with the next one, simply keeping your `ToDo` list in a separate file from your Pascal sources.

Chapter 13 LAZARUS GUI PROJECTS

13.c ToDo functionality

Lazarus provides **ToDo** functionality which meets most programmers' needs for a tasklist. The shortcut for this when in the Editor is [Shift][Ctrl][T], (or *right-click to show the context popup menu and choose **Insert ToDo***). The **ToDo** functionality works by inserting a simple **comment** in your source code (or into any text file open in the Editor).

Code comments were mentioned earlier, and you will recall that curly brackets { } are used for enclosing comments and that the compiler understands comments beginning { \$... } as compiler directives, and that the Editor colours such directives differently to make them immediately distinctive visually. The IDE recognises two special comment categories: **ToDo** comments and **Done** comments. In addition to colouring `ToDo` comments blue in the Editor there is a *ToDo List* dialog (accessed via **View | ToDo List**) which collects all `ToDo` entries from units listed in the main project file into a single list. Clicking on a list entry jumps straight to that `ToDo` in the relevant file.

An alternative listing of `ToDo` items is provided by the Code Explorer (provided you specify this option) which is accessed via **View | Code Explorer**. The Code Explorer is a sophisticated and helpful tool. Some of its features are explored in Chapter 19 (Section f).

The IDE considers any comment a `ToDo` if it begins with `{todo: or {#todo: or {done: or {#done:` and spaces and capitalisation are ignored, so

```
{ToDo: 2 This is a ToDo with priority 2 -oJane -cKillerFeature}
```

is a valid `ToDo` comment. `ToDo`s that you have changed into `done`s are shown in the *ToDo List* dialog with an **X** in the *Done* column. `ToDo` comments can include an **integer** to indicate

priority, a **-oOwner** tag to identify the Owner and a **-cCategory** tag to indicate the Category, in addition to the main text of the comment. These elements are parsed when you export a `ToDo` list via the *ToDo List* dialog toolbar button marked *Export*. The resulting file is a `.csv` (comma separated values) text file, named with an appended date thus: `ToDoList_YYYY_MM_DD.csv`.

If you open this file in a spreadsheet program you will see that each `ToDo` item has been parsed into seven columns headed *Done, Description, Priority, Module, Line, Owner, and Category*.

13.d Version control

Working with a version control system (VCS) is helpful even if you are not collaborating with colleagues in producing software (when a version control system is essential). Lazarus and FPC developers use SVN, and many in the Lazarus and FPC communities use SVN, Git, Mercurial or a combination of these. What you use is obviously up to you. This book will not say more about this topic, since it is beyond the scope of these pages, except to emphasise the desirability of version control even for hobby programmers and beginners. The facility to collaborate in an organised way with other programmers, as well as allowing reversion to earlier designs if an initially promising branch of development turns out to be a dead end, and merging of bug fixes or new features once they are debugged are just some of the advantages of a VCS, apart from the basic backup facility.

13.e Test-driven software development

Commercial software development requires a more rigorous approach, and constant testing needs to be built in to the development process. Lazarus and FPC provide for this via the *FPCUnit Test Application* and *FPCUnit Console Test Application* project types (the last two options in the *Create a new project dialog* accessed via **Project | New Project...**). **Test-driven development** is beyond the scope of this chapter, however it is an important topic which you will want to explore if you go very far in the world of programming, and if you explore the labyrinth of Lazarus/FPC `svn` subdirectories you will see that quite a few are devoted to tests – one reason Lazarus and FPC is high-quality software.

This topic is treated briefly later – see Section 18.f of Chapter 18, Algorithms and Unit Tests.

Chapter 13 LAZARUS GUI PROJECTS

13.f Naming

What will your new project be called? Try to distil the essence of its purpose or function into a meaningful name, and if your source code becomes big enough (*as it soon might*), consider the best names for the sub-divisions of your code which will be your Pascal **units**. Almost any names you come up with will be an improvement on the default names that will be used otherwise (`project1.lpr`, `unit1.pas`, `unit2.pas` and so on).

Also remember if you work on a case-insensitive platform like Windows that other potential users may have problems if you call your main unit `UnitMain.pas` and share it. If the name gets changed inadvertently to `Unitmain.pas`, a Mac or Linux filesystem will recognise that as a different file, and give an **Error: UnitMain.pas not found** message.

Best to use **all lowercase names at first naming** if there is ever any likelihood of cross-platform usage. Since comprehensive cross-platform capability is one of the most attractive features of Lazarus/FPC, that possibility always needs to be borne in mind. It is impossible to know where files you write today may be running someday in the future.

The filenames you choose are restricted not only by operating system rules (e. g. no '?' or ':' characters on Windows, no null or '/' characters on Linuxes), but by Pascal naming rules.

As mentioned above this prevents use of a numeral as the first character of a name, and also prevents use of non-English alphanumeric characters such as 'ø', space, '+' and '-' in names.

The underscore '_' is OK, and it can also be the first character in a filename.

Lazarus prevents you from using illegal Pascal names for filenames when projects and units are first saved (*you can circumvent this by renaming files later, of course, outside Lazarus – but if you give Pascal source files illegal names they will not compile*). If you save a project or unit with an all-lowercase name (say `mainform.pas`) you can manually edit the name of the unit inside the source to change its capitalisation (*say to `MainForm`*) and this is perfectly OK, and Lazarus accepts this without a murmur. However, it is probably wisest to keep unit and project names identical in capitalisation with their filenames.

13.g Project directory structure

Where will the project be kept? A new folder (*directory*) must be created and named, and a directory structure must be considered so that logically related file groups are stored together. At all costs avoid the laziness of lumping all project files into one folder. Lazarus does its best to prevent you doing this by creating an output directory where compiled files (*compiled unit .ppu files and various intermediate compiled resource and object files*) are placed, as well as a backup folder where it places copies of the source files you generate (*unless you unwisely turn this backup feature off*).

A chest with several drawers in it is much more helpful in organising your possessions than a large single-compartment box in which everything is thrown together. Likewise, creating a sensible folder hierarchy to store your application/project files will help you organise, find and update items more easily than cramming everything into a single `project3` folder. If you're creating an application to store and play tracks from your mp3 collection, you might decide to call the application `mp3index`, and you could create a folder hierarchy named as follows:

```
mp3index
  source
  lib
  data
  images
  backup
```

This is just one of a myriad of possibilities, of course. You might need a directory called `translations`, or separate `mp3`, `wav`, `ogg` and `midi` folders rather than a single `data` folder.

Your application might not use any images (*meaning your program will look rather dull*).

The point is to organise file storage thoughtfully. This will help you in the overall management of the project. You should also consider including a `docs` folder to document your work.

Chapter 13 LAZARUS GUI PROJECTS

13.h A template project: SetDemo

This section begins the development of a short example project that illustrates some of the above ideas, which you can use as an adaptable template when you develop projects of your own. The purpose of the software end-product is to give you a hands-on feel for **sets**.

Sets are widely used throughout the LCL because they are an elegant and convenient Pascal feature that helps to simplify all sorts of GUI property programming. However, you may not have encountered or used sets since your days in a maths class at school, so the following project provides a visual illustration of what is involved in

- adding items to a set
- removing items from a set
- set **union** and **intersection**
- set **difference** and **symmetric difference**

The project is designed so that you will be able to see a clear visual representation of the changing sets as you alter them. This, after all, is one of the main advantages of GUI over console programming.

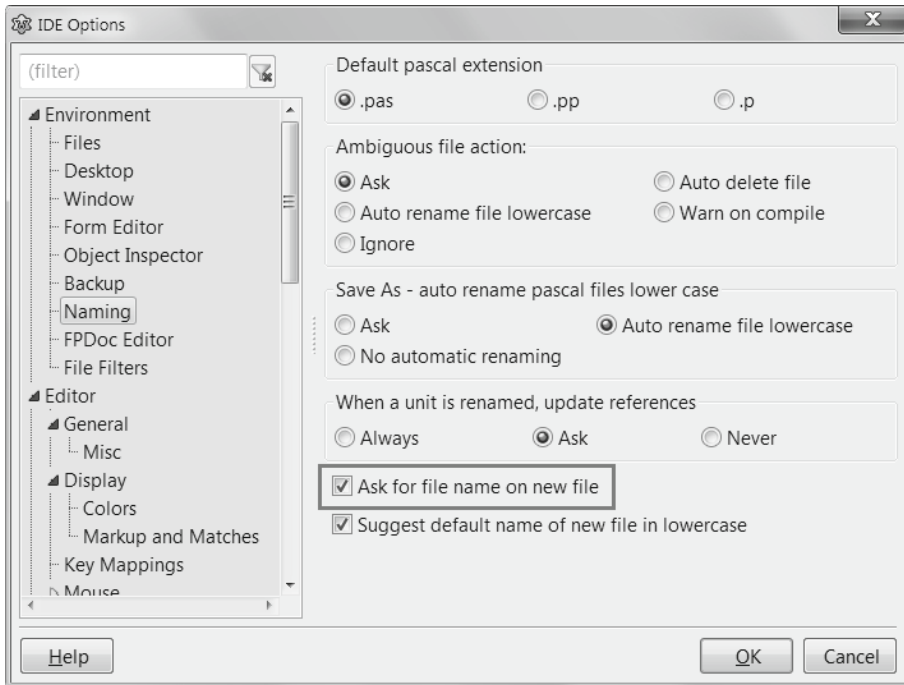


Figure 13.1 The Naming page of the IDE Options dialog, highlighting the file naming setting

Create a new project directory (*somewhere where you have file-write permission*) named `SetDemo` (for instance, on the author's system this was `G:\Learn to Program\Chapter 13\SetDemo`). Start Lazarus, and check that you have a new project (`Project1`) loaded (*not the previous project you worked on*). If you see a previous project loaded choose **Project | New Project**, [OK].

To help you avoid accepting the default names Lazarus always supplies for files, make two changes to your Lazarus settings. Open the IDE Options dialog via **Tools | Options...** and click on the `Naming` node under the `Environment` branch of the treeview at the left of the dialog (see Figure 13.1). Tick the `Ask for file name on new file` setting.

Chapter 13 LAZARUS GUI PROJECTS

Next in the *Form Editor* node of the same *Environment* treeview branch make sure the *Ask name on create* checkbox is checked (see Figure 13.2). If you don't see this checkbox, enlarge the dialog by pulling its lower border downwards. Click [OK] to accept these settings. Although you may find that enabling these features is rather annoying, they will force you to think about naming whenever it is relevant to do so.

With these two naming settings checked you cannot accept the default names Lazarus supplies without making a conscious decision to do so. A dialog will intervene every time you drop a new component on a form, suggesting the Lazarus default name, but giving you the option to over-type that name (*it is already selected, so any key other than [Enter] will delete the default name and start entry of the alternative you type – you don't need to explicitly delete the default name*).

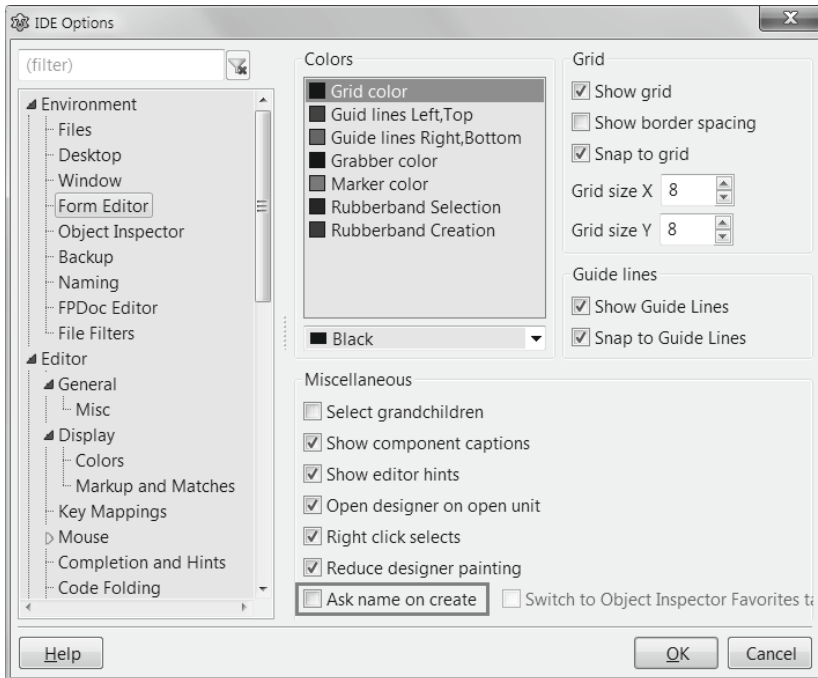


Figure 13.2 The Form Editor page, highlighting the setting for naming new components

Next pull down the **Project** menu, and choose **Save project as...**, and in the resulting two **Save** dialogs save the main project file as `setdemo.lpi` and save the unit as `setdemo_main.pas` (or `setdemo_main.pp`). When you open the **Project Inspector** (*Project | Project Inspector*) you will then see these two files listed, as in Figure 13.3.

Chapter 13 LAZARUS GUI PROJECTS

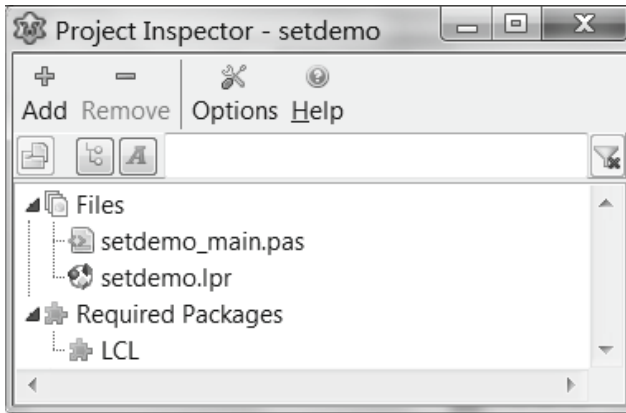


Figure 13.3 The Project Inspector displaying the setdemo project

Click on the [Options] toolbutton in the Project Inspector. The *Options for Project: setdemo* dialog opens at its first *Application* page. The `Title` field will read `setdemo`. Replace this with `Integer set demo`. Click [OK] to accept the new `Title`.

Double-click on `setdemo.lpr` in the Project Inspector's file list to focus that file in the Editor. When you look at the code you see that Lazarus has added the line:

```
Application.Title:='Integer set demo';
```

The result of typing a new `Application Title` in the *Project Options* dialog is that Lazarus has written code to effect that change in the main program file. If you look in the title bar of the Lazarus IDE you see another effect: it now reads:

```
Lazarus IDE v1.0.4 - Integer set demo
```

Your IDE version will probably differ, but Lazarus displays the `Title` of the application in the IDE title bar, rather than the name of the project (`setdemo`) if the two differ.

The project name is the name that comes after program in the main project file (*in this case it is setdemo*). The project name can differ from the `Application.Title` (*as here*) although it is perfectly OK to leave them identical, which is the default. `Application.Title` does not have the same naming restrictions that apply to Pascal program names and OS filenames, so it can contain spaces as here, or characters such as `'/'` which are not allowed in filenames on most OSs.

13.i Encapsulating set interaction within a new class

Recall that one of the key questions to ask when beginning a new project is: Can I envisage developing a new class that will encapsulate the functionality I need? For this project that means creating a **class** that will encapsulate the functions (*as mentioned above*) of

- adding items to a set
- removing items from a set
- illustrating set *union* and *intersection*
- illustrating set *difference* and *symmetric difference*

Accordingly we will create a class that offers this functionality, writing a skeleton with appropriately named functions, procedures and properties. Once we have a set-interaction 'engine' drafted out, we can then design the UI, and complete the project by hooking up the set engine class to the UI. Mixing up UI code with non-UI code in a single GUI form is a recipe for a difficult-to-maintain project. A **loosely-coupled** design is better (*to use the programming jargon*).

Chapter 13 LAZARUS GUI PROJECTS

A project design that eschews close coupling of UI code with core project functionality will force us to think about separating out non-UI functionality at an early stage. This also promotes possible later code reuse, since encapsulating functionality in a well-designed and debugged class means we have a code module that can easily be plucked out of a project for use elsewhere (*made into a component, perhaps*). Whereas, if the functionality you might need elsewhere is completely wedded to the widgets and UI code of your project, it will have to be re-written for use elsewhere. It cannot simply be decoupled and plugged into another situation.

The class we want will include two sets of digits (*set A and set Z*), read/write properties allowing addition of digits to these sets (*or removal of digits*), and methods that let us visualise not only the sets, but operations on the sets such as set union, set intersection and so on. 'Visualisation' here means that the methods will be functions returning strings that display the set contents. So we will need a `SetAsString()` function that can convert a set of digits into a displayable string. Let's call the new class `TDualAZSet`, to indicate that it holds two digit sets, labelled A and Z. To encompass the functionality summarised at the beginning of this Section, we will need the new class to look like the following:

type

```
TDigits = 0..9;
TDigitsSet = set of TDigits;

TDualAZSet = class
  private
    FsetA, FsetZ: TDigitsSet;
    function SetAsString(s: TDigitsSet): string;
  public
    property DiffAZAsText: string read GetDiffAZAsText;
    property DiffZAAsText: string read GetDiffZAAsText;
    property IntersectionAsText: string read GetIntersectionAsText;
    property SetA: TDigitsSet read FsetA write FsetA;
    property SetAasText: string read GetSetAasText;
    property SetZ: TDigitsSet read FsetZ write FsetZ;
    property SetZasText: string read GetSetZasText;
    property SymDiffAsText: string read GetSymDiffAsText;
    property UnionAsText: string read GetUnionAsText;
end;
```

Here there are two **public** read/write `TDigitSet` properties, `SetA` and `SetZ` based on **private** storage fields `FsetA` and `FsetZ`, together with a **private** conversion function `SetAsString`.

The other **public** properties are read-only string properties that report the set operation that their name indicates. The purpose of the class is to encapsulate all the set-related functions, gathering them in one location.

We will further enhance this encapsulation by putting this set-related class in a unit of its own. Create a new unit (*not a new form*) via **File | New Unit** and save it in the project directory as `dualdigitset.pas`. The Project Inspector should now show this new unit as one of the files in the project.

Lazarus has placed two units in the **uses** clause of the new unit: `Classes`, and `SysUtils`.

The `Classes` unit is not needed and can be deleted from the **uses** clause. Below the **uses** clause, add a type declaration as given above for the two ordinal types and the longer class type.

If you keep the cursor somewhere inside the class declaration you have just written and invoke Code Completion via **[Ctrl][Shift][C]** Lazarus will expand the **private** section of the class with declarations for the various getter functions required for the string properties, and also write skeleton bodies in the **implementation** section for all the declared methods. The full **class** declaration should now read as follows:

Chapter 13 LAZARUS GUI PROJECTS

```
TDualAZSet = class
  private
    FsetA, FsetZ: TDigitsSet;
    function GetDiffAZAsText: string;
    function GetDiffZAAsText: string;
    function GetIntersectionAsText: string;
    function GetSetAasText: string;
    function GetSetZasText: string;
    function GetSymDiffAsText: string;
    function GetUnionAsText: string;
    function SetAsString(s: TDigitsSet): string;
  public
    property DiffAZAsText: string read GetDiffAZAsText;
    property DiffZAAsText: string read GetDiffZAAsText;
    property IntersectionAsText: string read GetIntersectionAsText;
    property SetA: TDigitsSet read FsetA write FsetA;
    property SetAasText: string read GetSetAasText;
    property SetZ: TDigitsSet read FsetZ write FsetZ;
    property SetZasText: string read GetSetZasText;
    property SymDiffAsText: string read GetSymDiffAsText;
    property UnionAsText: string read GetUnionAsText;
end;
```

Most of the method skeletons will be filled out with one-line set-related code sections. Complete the implementation as follows:

```
function TDualAZSet.GetDiffAZAsText: string;
begin
  Result:= SetAsString(FsetA - FsetZ);
end;

function TDualAZSet.GetDiffZAAsText: string;
begin
  Result:= SetAsString(FsetZ - FsetA);
end;

function TDualAZSet.GetIntersectionAsText: string;
begin
  Result:= SetAsString(FsetA*FsetZ);
end;

function TDualAZSet.GetSetAasText: string;
begin
  Result:= SetAsString(FsetA);
end;

function TDualAZSet.GetSetZasText: string;
begin
  Result:= SetAsString(FsetZ);
end;

function TDualAZSet.GetSymDiffAsText: string;
begin
  Result:= SetAsString(FsetA+FsetZ - FsetA*FsetZ);
end;

function TDualAZSet.GetUnionAsText: string;
begin
  Result:= SetAsString(FsetA + FsetZ);
end;
```


Chapter 13 LAZARUS GUI PROJECTS

```
function TDualAZSet.SetAsString(s: TDigitsSet): string;
var d: TDigits;
begin
  Result:= EmptyStr;
  for d in TDigitsSet do
    if (d in s) then
      begin
        if Length(Result) > 0 then AppendStr(Result, ',');
        AppendStr(Result, IntToStr(d));
      end;
  Result:= Format('[%s]', [Result]);
end;
```

That completes the “set engine” class. Make sure you save the unit, before continuing with the UI development work which is considered in the next Section.

It is possible to write a `SetAsString()` function using types and routines from the `typinfo` unit. We used this unit to write RTTI-related parts of the Component Browser project in Chapter 12 (Section f). There we had no choice, since this author knows of no other way to access RTTI apart from the routines of the `typinfo` unit. In the case of getting information about sets we can obtain the information far more simply using the (`if anElement in aSet`) Pascal construct, rather than using the pointer-oriented `typinfo` types and routines. So I have preferred to use a type conversion algorithm for `SetAsString()` which is easier for beginners to understand.

13.j The setdemo UI

To visualise the contents of the two sets `SetA` and `SetZ` we will design a UI in three sections:

1. In the first section we need a set of buttons to populate or depopulate Set A, with a representation of the contents of Set A
2. The second section will have a set of buttons to populate or depopulate Set Z, with a representation of the contents of Set Z
3. The third part will have representations of the sets formed by adding, intersecting, and subtracting Sets A and Z

To provide a clear colour distinction between the three areas we shall use three panels to define these regions. Look at Figure 13.4 to see the outcome of this UI design.

Press [F12] to focus the Designer containing `Form1`, if it is not already the foremost window, and in the OI set the form properties to the following values:

Height	380
Width	400
Name	MainForm

Notice that the change to the form's `Name` also causes the form class declaration (*which was* `TForm1`), and the form variable (*which was* `Form1`) to change to the following:

```
type
  TMainForm = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  MainForm: TMainForm;
```


Chapter 13 LAZARUS GUI PROJECTS

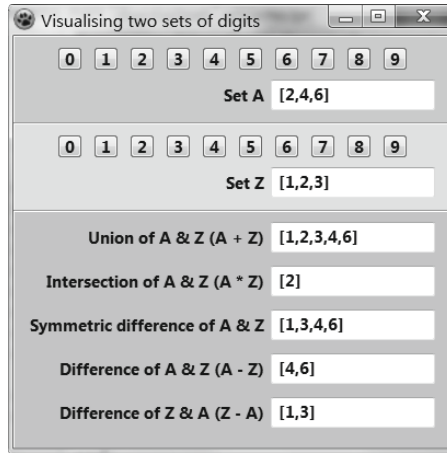


Figure 13.4 The completed setdemo project with typical entries showing

Change the Caption of the form to read *Visualising two sets of digits*. Make sure the *Standard Palette* page is open and drop a panel (*3rd icon from the right*) onto *MainForm*. Boom! Because of our earlier change to the *Form Editor* page of the IDE's Options, checking *Ask name on create*, we now have a new dialog to complete (see *Figure 13.5*).

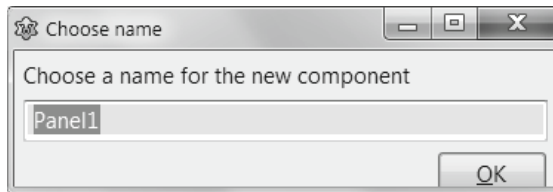


Figure 13.5 The Choose name dialog

Type `pnlA` in the name edit field, and click [OK] to close the *Choose name* dialog. In the OI set the following properties for `pnlA` (Name is now already set to `pnlA`):

Align	alTop
Caption - remove the	Caption leaving it empty
Color	clGradientActiveCaption
Font.Style	[fsBold]
Height	80

Drop a labeled edit control (*from the Additional Palette page*) named `edtSetA` on the **panel** (*not on the unused lower part of MainForm*) and set the following `edtSetA` properties:

EditLabel.Caption	Set A
LabelPosition	lpLeft
LabelSpacing	5
Left	235
ReadOnly	True
TabStop	False
Text - remove any text	to leave this empty
Top	40
Width	150

Chapter 13 LAZARUS GUI PROJECTS

Note that the `EditLabel.Caption` displaying **Set A** is now bold, because its `ParentFont` property is by default `True` and its parent, `pnlA`, has a bold font style.

Right-click on the panel (*not the edit*) and select *Copy* from the context menu.

Right-click below the panel on `MainForm` and choose *Paste* to paste a copy of `pnlA` into position. Notice this new copied panel is not a completely exact copy of `pnlA`. It is not possible to have two controls with the same name on a form (*how would the compiler know the difference between them?*) so Lazarus names the copied panel `pnlA1`, to make it distinctive. Set the properties of the new panel as follows:

```
Color          clGradientInactiveCaption
Name           pnlZ
```

Select the edit control on `pnlZ` (*by clicking on it either in the Designer or in the OI treeview*) and set its properties as follows:

```
EditLabel.Caption Set Z
Name             edtSetZ
```

Drop a new panel from the *Standard* page of the OI on the remaining clear region at the bottom of the form, naming it `pnlResultSets` and set its properties as follows:

```
Align          alClient
Caption - delete the Caption to leave the property an empty string
Color          clScrollBar
Font.Style     fsBold
```

From the *Additional Palette* page drop a `TLabelEdit` control (*10th icon from the left*) on to `pnlResultSets` naming it `edtUnion`. Set this `LabelEdit`'s properties as follows:

```
EditLabel.Caption Union of A && Z (A + Z)
LabelPosition  lpLeft
LabelSpacing   5
Left          235
ReadOnly      True
TabStop       False
Top          10
Width        150
```

Making sure `edtUnion` is selected, right-click and choose *Copy* from the context menu, then right-click on `pnlResultSets` below `edtUnion` and choose *Paste*. With the newly copied `TLabelEdit` selected, use the OI to change its properties to the following new values:

```
EditLabel.Caption Intersection of A && Z (A * Z)
Left          235
Name          edtIntersection
Top          50
```

Right-click below `edtIntersection` and choose *Paste* again, setting properties of the newly pasted `TLabelEdit` as follows:

```
EditLabel.Caption Symmetric diff. of A && Z
Left          235
Name          edtSymmetricDiff
Top          90
```

Chapter 13 LAZARUS GUI PROJECTS

Right-click below `edtSymmetricDiff` and choose *Paste* again, setting the properties of the newly pasted `TLabelEdit` as follows:

```
EditLabel.Caption  Difference of A && Z (A - Z)
Left               235
Name               edtDiffAZ
Top               130
```

Right-click below `edtDiffAZ` and choose *Paste* again, setting the properties of the newly pasted `TLabelEdit` as follows:

```
EditLabel.Caption  Difference of Z && A (Z - A)
Left               235
Name               edtDiffZA
Top               170
```

The code Lazarus has written for `setdemomain.pas` now has a longer class declaration for `TMainForm` which will look similar to the following:

type

```
{ TMainForm }
TMainForm = class(TForm)
    edtDiffZA: TLabelEdit;
    edtSetA: TLabelEdit;
    edtSetZ: TLabelEdit;
    edtUnion: TLabelEdit;
    edtIntersection: TLabelEdit;
    edtSymmetricDiff: TLabelEdit;
    edtDiffAZ: TLabelEdit;
    pnlA: TPanel;
    pnlZ: TPanel;
    pnlResultSets: TPanel;
private
    { private declarations }
public
    { public declarations }
```

Check that you have three panels, and seven labeledEdits, though they may well be listed in a different order from that above. The form in the Designer should now look something like Figure 13.6. You'll see that we have followed a naming convention that prepends a 3-letter code to the beginning of each name, identifying what type of component it is. You can ignore conventions of this sort; but on forms with scores of controls it will often save you writing nonsense code. A consistent naming scheme also helps you to remember what names you have given to controls, so you can type their names later without having to scroll elsewhere or jump about to look up names.

Note: Lazarus puts all controls dropped onto a form into an **unmarked section at the very beginning** of the form's class declaration. This section is in fact a **published** section (*although it does not say so explicitly*) so that all the listed controls are accessible in the OI. These controls' published properties and events can then be manipulated in the OI. Although you can freely add code, fields, and declarations anywhere in the form class declaration, it is advisable to leave this first **published** controls section – for which Lazarus has written all the code – well alone. Lazarus keeps this section perfectly synchronised with the data written to the form file (`.lfm`), and if the programmer inadvertently messes up this synchronisation, it can lead to strange compilation errors and erratic or mysterious Designer behaviour. It is wiser to leave that part of the form class entirely to Lazarus, unless you really know what you are doing in trying to edit it yourself.

Chapter 13 LAZARUS GUI PROJECTS

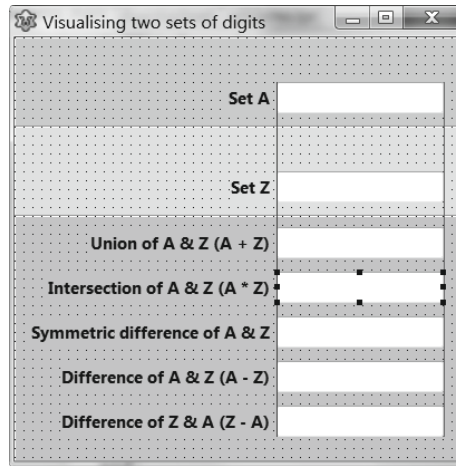


Figure 13.6 setdemo's main form showing the Designer's alignment Guide Lines

We want a row of buttons [0], [1] ... [9] – a button for each digit – in the upper two panels. Clicking on a button will add (*or remove*) that button's digit from the relevant set. There is always a choice in GUI design between placing controls by dropping them from the Palette and setting their properties in the OI, or placing them by coded declarations and property assignments in the form unit. The former approach stores the property data in a `.lfm` file, and the latter approach stores property data as code assignments in a `.pas` file.

Sometimes one approach is better than the other – often it makes little difference. Placing twenty button controls and setting their properties manually from the Palette and OI is rather tedious, though by selecting multiple components with [Shift]-click it is possible using the OI to assign a value to the properties of all selected components at once. However, we shall create these controls and assign their properties in code, to show you how this can be done (*you might also want to experiment with the manual method yourself*). We will use the `TSpeedButton` control (3rd icon from left on the Additional page of the Palette). To create a row of ten speed-buttons in code we introduce a helper type, `TButtonArr`, to keep track of the buttons, declared thus:

```
type TButtonArr = array[TDigits] of SpeedButton;
```

where `TDigits` is the subrange type declared in the `dualdigitset` unit we wrote earlier. This is the moment to add `dualdigitset` to the `uses` clause of the main form. You don't need to type its name. Press [Alt][F11], and in the *Add unit to Uses section* dialog, you will find that the very unit you want to add is the only one immediately offered! How is that? It is the only other Pascal unit in your project, and it is not yet in the unit's `uses` clause, so Lazarus assumes it is the one you want to use. Click the unit name to select it, and press [OK]. The dialog closes and Lazarus adds the required unit to the `uses` clause without the need for you to type anything. This gives the `mainform` unit access to the two ordinal types and the class declared in `dualdigitset`.

```
type TDigits = 0..9;  
    TDigitsSet = set of TDigits;  
    TDualAZSet = class ...
```

`TSpeedButton` is declared in the `Buttons` unit, which is so far missing from our `uses` clause.

Lazarus has no way to divine that we need this unit, so we have to add this unit manually to the `uses` clause. The beginning of the interface section of `setdemo_main.pas` should now look like this (*the order of units in the uses clause is not important*):

Chapter 13 LAZARUS GUI PROJECTS

uses

Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, ExtCtrls, StdCtrls, dualdigitset, Buttons;

type

```
TButtonArr = array[TDigits] of TSpeedButton;  
{ TMainForm }  
TMainForm = class(TForm)...
```

There are actually several unused units in this **uses** clause. To remove the unneeded units rightclick in the Editor and choose *Refactoring*, then choose *Unused Units ...* Click the [Remove all units] button. This feels like a rather risky button to press, but Lazarus knows what it is doing. It doesn't actually remove all the units in your **uses** clause, only the unused units (*which is all the units listed in the dialog*). You'll end up with a much-slimmer **uses** clause, which will still build and compile OK, but will no longer give you Hints in the Messages about *Unit "xyz" not used in setdemo_main.pas*. To create 2 rows of ten buttons for our form we need to write a `CreateButtons` method that will be part of our `TMainForm` class declaration, and a `ButtonClick` method that each button can use. We will also need a helper procedure `UpdateSetExpressions` to update the display of the set operations every time the content of one of the sets changes. Additionally, we also need a variable to hold an instance of the `TDualAZSet` class. So delete the form's **public** section, and place the Editor cursor just after the word **private** in the form class declaration overwriting the { private declarations } comment so that the form declaration ends up looking like this:

```
TMainForm = class(TForm)  
.  
.  
private  
  ds: TDualAZSet;  
  function CreateButtons(setID: Char): TButtonArr;  
  procedure ButtonClick(Sender: TObject);  
  procedure UpdateSetExpressions;  
end;
```

This `CreateButtons` function has been designed with a single parameter, `setID` which allows us to indicate whether the buttons are linked to Set A or Set Z. What we have just written is the *signature* for these methods – just the name, return type, and the needed parameters. This signature is placed in the **interface** part of the unit. The code that creates the array of buttons, and provides button-click functionality now needs to be written – the *body* of the function. Use Code Completion to generate the method body skeleton in the **implementation** section. Type the following into the provided skeletons:

```
function TMainForm.CreateButtons(setID: Char): TButtonArr;  
const spacing = 10;  
  aLeft = 40;  
var i: integer;  
  b: TSpeedButton;  
begin  
  for i := Low(TDigits) to High(TDigits) do  
    begin  
      b := TSpeedButton.Create(Self);  
      b.Top := spacing;  
      b.Left := aLeft + i * (b.Width + spacing);  
      b.Caption := IntToStr(i);  
      b.Tag := i;  
      b.Name := Format('%s%d', [setID, i]);  
  
      case setID of  
        'A': b.Parent := pnlA;  
        'Z': b.Parent := pnlZ;  
      end;  
      b.OnClick:= @ButtonClick;  
      Result[i]:= b;  
    end;  
end;
```

Chapter 13 LAZARUS GUI PROJECTS

This is a `for` loop that creates 10 buttons, setting various properties for each button before adding the button to the `TButtonArr` returned by the function. Towards the end of this function we've assigned `@ButtonClick` (i.e. *the address of* `ButtonClick`) to the `OnClick` event of `b`. We now need to write the `ButtonClick` procedure.

```
procedure TMainForm.ButtonClick(Sender: TObject);
var b: TSpeedButton;
begin
  if not (Sender is TSpeedButton) then Exit;
  b := TSpeedButton(Sender);
  case b.Name[1] of
    'A': begin
      if (b.Tag in ds.SetA)
        then ds.SetA:= ds.SetA - [b.Tag]
        else ds.SetA:= ds.SetA + [b.Tag];
      edtSetA.Caption := ds.SetAasText;
    end;
    'Z': begin
      if (b.Tag in ds.SetZ)
        then ds.SetZ:= ds.SetZ - [b.Tag]
        else ds.SetZ:= ds.SetZ + [b.Tag];
      edtSetZ.Caption := ds.SetZasText;
    end;
  end;
  UpdateSetExpressions;
end;
```

Fill out the skeleton Lazarus provides for `UpdateSetExpressions` as follows:

```
procedure TMainForm.UpdateSetExpressions;
begin
  edtUnion.Caption:= ds.UnionAsText;
  edtSymmetricDiff.Caption:= ds.SymDiffAsText;
  edtIntersection.Caption := ds.IntersectionAsText;
  edtDiffAZ.Caption := ds.DiffAZAsText;
  edtDiffZA.Caption := ds.DiffZAAsText;
end;
```

Note: This is one of the useful features of the OI treeview. We cannot select the main form by clicking on it in the Designer, since there is no exposed main form region to click on (*the entire form being covered in panels*). Clicking a form's borders or title bar does not select it.

With the main form selected, click on the *Events* tab in the OI, and double-click in the empty column beside the `OnCreate` event. Lazarus creates an `OnCreate` event handler, and moves the cursor to the Editor ready for you to complete the code, which should look like the following when finished:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  ds:= TDualAZSet.Create;
  CreateButtons('A');
  CreateButtons('Z');
end;
```

Chapter 13 LAZARUS GUI PROJECTS

Here we simply create the `ds` instance along with two button arrays.

Create a `TMainForm.OnDestroy` event handler in the same way as you did for the `OnCreate` handler, and complete it as follows:

```
procedure TMainForm.FormDestroy(Sender: TObject);  
begin  
    ds.Free;  
end;
```

Now press [F9] to compile and run the application, and explore its functionality. Note that the buttons created in `FormCreate` don't need freeing explicitly since they are owned by the form through the constructor call that created them:

```
b := TSpeedButton.Create(Self);
```

The `Self` parameter in this button-creation call is `MainForm`, which as `Owner` organises the freeing of all the components it owns when it is itself freed.

13.k Review Questions

1. One school of psychology suggests that everyone falls somewhere between two extremes of personality: at one end are those who plan carefully, list out the tasks that await them, and tend to think ahead. At the other extreme are the happy-go-lucky who don't worry about the future and are allergic to plans, preferring to wait and see how things turn out, keeping their options open. Where do you fit on this spectrum, and how does this affect the way you might approach the development of a software project?
2. How might you improve the `setdemo` application for someone who is visually impaired?
3. Would it be feasible to adapt the `setdemo` application to deal with the set of alphabetic characters A..Z rather than the digits 0..9? What would be involved?



Chapter 14 COMPONENT CONTAINERS

The RAD programming paradigm involves dropping components on a window to quickly design a user interface. This requires program elements that can accept components (*controls, widgets*) being dropped onto them. This chapter considers several Lazarus component containers, beginning with `TForm` which behaves rather differently from other LCL controls in several respects (*for instance, there can only be one visually designed form per unit - whereas you can have multiple instances of any other class declared in a unit*). It is helpful to keep the LCL class hierarchy in mind in considering how controls differ from one another.

14.a Non-visual RTL classes

There are four important RTL classes that descend directly from `TObject`:

- `TList`
- `TStream`
- `TPersistent`
- `TThread`

For LCL classes, `TPersistent` is the ancestor of the visual classes, being a class that provides the streaming mechanism (*RTTI*) used to store the published properties of visual LCL components (*in the .lfm form file at design time*), and to reconstruct these components and forms when their constructors are called at runtime, using RTTI stored in the compiled program executable. The next chapter looks briefly at some of the more important non-visual support classes that LCL controls depend on.

A simplified ancestry tree for `TForm` looks like this:

```
      TObject
      |
      | TPersistent
      | |
      | | TComponent
      | | |
      | | | TLCLComponent
      | | | |
      | | | | TControl
      | | | | |
      | | | | | TWinControl
      | | | | | |
      | | | | | | TForm
```

At each stage in the hierarchy (*including several stages missed out in the above `TForm` lineage*) further functionality is added to the class, making `TForm` a fairly complex class with hundreds of fields, methods and properties. Before any further controls are added to a form it has a 1116 byte `InstanceSize`.

The Lazarus Designer is a tool specifically crafted for working with the `TForm` class and the controls it can contain. Every GUI project has at least one form (*the main form*) which by default is shown in the Designer when Lazarus first starts. As controls are dragged from the Palette and dropped on the form in the Designer, so appropriate code is added to the main form unit, expanding the form declaration; and the form definition file (`unitname.lfm`) is continuously updated with details of the property and event values for each control the form currently contains.

The form declaration Lazarus writes in the main form file is always a direct descendant of `TForm` (*never `TForm` itself*). For instance, the default form type is called `TForm1`, declared as

```
type TForm1 = class(TForm)
    end;
```

Likewise the form variable by default is called `Form1`, and it is of type `TForm1` (*not of type `TForm`*). The form type is best renamed to `TMainForm` or something more appropriate. Lazarus keeps track of form naming in a way it does not track other type names.

Chapter 14 COMPONENT CONTAINERS

If you rename the form from `Form1` to `frmMain`, Lazarus will not only rename the form instance to `frmMain`, but it will also rename the **type** of the form variable to `TfrmMain`, both in the declaration of the class, and in the declaration of the global form variable. (*This automatic type renaming does not happen as a result of any other manual variable renaming*). At the same time the IDE resets the form's `Caption` to the new name of the form class variable, so the Designer title bar also reads `frmMain`. This is correct, because of course the form instance you see depicted in the Designer is indeed `frmMain`.

Every GUI program must have at least one window (*usually referred to as the main window*). Closing the main window terminates the GUI program. Among the many `TForm` events is the `OnCloseQuery` event which provides a **var** boolean `CanClose` parameter which makes it straightforward to program a check that the user really does want to close the window (*which will close the program if it is the main window*).

The LCL automatically shows the main window of a GUI application. Recall that the main program file of a default new GUI project has a program block that looks like this:

```
begin
  RequireDerivedFormResource := True;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The last statement in this program is a call to the `Application.Run` method which can be written in pseudo-code like this:

```
procedure Application.Run;
begin
  ShowMainForm; // this is Form1 in newly opened default project
  RunMainMessageLoop;
end;
```

14.b Creating new forms

If you need to create second or further forms beyond the main form, you can do this manually in code:

```
var extraForm: TExtraForm;
begin
  extraForm:= TExtraForm.Create(Application);
  . . .
end;
```

Note that the `Application` instance needs to be made the `Owner` of the form in order for it to be freed correctly at the end of the program.

However, beginners are better advised to let Lazarus manage the creation of forms automatically. If you are writing an accounts program that needs a customer list window and a supplier list window and an invoicing window, you would use the main menu **File | New Form** command to create each form. In your `accounts` project if you choose **Project | Project Options ...** ([Shift][Ctrl][F11]) and under *Project Options* in the treeview on the left click on *Forms*, you will see a dialog listing the forms auto-created by Lazarus (see Figure 14.1).

The checkbox on this page, *When creating new forms add them to auto-created forms* is checked by default, and it is best to leave it that way until you have gained more experience as a programmer.

Chapter 14 COMPONENT CONTAINERS

The forms listed in this *Options for Project* dialog correspond to the `Application.FormCreate()` statements that Lazarus has written in the main project file, `account.lpr`. In the case of the accounts program just cited, the main program block written by Lazarus would look something like:

begin

```
RequireDerivedFormResource := True;
Application.Initialize;
Application.CreateForm(TmainAccountsForm, mainAccountsForm);
Application.CreateForm(TcustomerList, customerList);
Application.CreateForm(TsupplierList, supplierList);
Application.CreateForm(TinvoiceForm, invoiceForm);
Application.Run;
```

end.

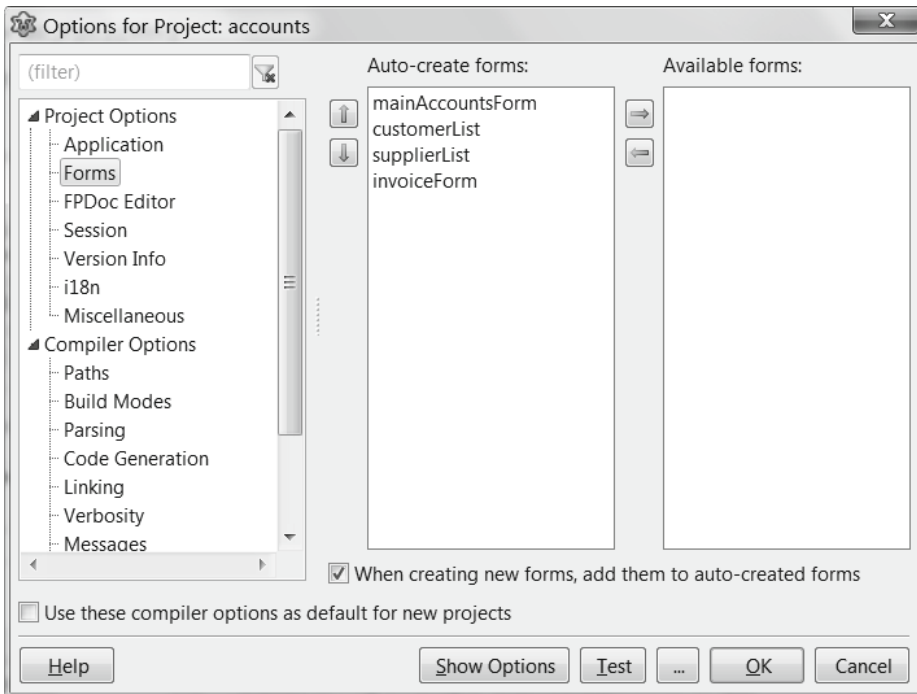


Figure 14.1 The Forms page of the Project Options dialog

If you have lots of forms in your project you probably do not want them all auto-created (*since that ties up a great deal of memory throughout the life of your program*), and you might prefer to create needed forms on demand, and free them immediately after use. In this case you would move one or more of the forms from the *Auto-create forms* listbox on the left of this dialog page to the *Available forms* listbox on the right of the dialog. You would then write code to `Create`, `Show` (or `ShowModal`), `Close` and `Free` the forms as appropriate. Section d of this chapter gives an example of on-the-fly form creation code.

Chapter 14 COMPONENT CONTAINERS

14.c Ownership and Parentage

The component containers on which you can drop controls in Lazarus (*whether forms, frames, or lighter-weight components such as `TPanel` and `TDataModule`*) all share in a scheme designed to ensure correct display of their child controls, and the correct automatic freeing of any child instances when the container is itself freed. To participate in this scheme two properties of the child control have to be set: `Owner` and `Parent`.

The child control's `Owner` is fully responsible for freeing the child instance. It is an error to attempt to free such a control manually. All containers of controls maintain a `Components[]` array listing their owned components, which can be queried by the programmer (*and which is enumerated automatically just before the container is itself destroyed to free all its children*).

When any `TComponent` descendant is created its owning control is passed as a parameter to the constructor. (You can pass `nil` as `Owner` parameter if you do not want to participate in the automatic freeing scheme). Any Palette control dropped onto a container component in the Designer automatically has its `Owner` set to the containing form (or frame) by Lazarus.

The child control's `Parent` is responsible for correct display of the child instance. Only `TWinControl` descendants can correctly display child components, and all such parents of other controls maintain a `Controls[]` array listing the child controls that are displayed on them. Parents are hooked into the system's window management and message-event system, and hence are in a position to relay system information such as `Repaint` messages to their children. Any visual Palette control dropped onto a container component in the Designer automatically has its `Parent` set to that display container by Lazarus.

For dropped controls the `Owner` is often identical to the `Parent`, being the form the control was dropped onto. However, where containing components such as panels or groupboxes have controls dropped onto them, the child controls will have the underlying form as their `Owner`, but the panel or groupbox as their `Parent`.

If you construct visual controls in code you are reminded about their need for an `Owner` through the required parameter in the call to the constructor. There is no such prompt for setting a manually constructed visual control's `Parent` property. If you forget to set it appropriately your newly constructed component will not be seen. In Delphi it may be required to set the `Parent` as the first property assignment after creation for correct display. There is no such requirement in Lazarus. In fact it is more efficient to make the `Parent` property the very last property assignment before the control is displayed.

14.d Programmatic form creation

The following short program gives an example of one way to solicit data from a user using a modal dialog, i.e. a dialog to which the user must respond before she can proceed with further program tasks. It is often the case that you need a user to enter a specific data item. A small dialog customised for this purpose can save you rewriting the same sort of routine over and over again. If you need a date to be entered, you can use the `TDateEdit` control (*located on the Misc Palette page*). This has a button which when clicked pops up a calendar for date selection. However, there are situations where three separate edits for entering day, month and year data are preferable. This short project provides this.

Create a new GUI project named `test_dateform.lpi`, with a form unit named `maintest.pas`. Create a further form (**File | New Form**) and save this as `dateform.pas`. Set the form's properties as follows:

```
TdateForm
Caption Date entry
Height 180
Name dateForm
Width 220
```

Chapter 14 COMPONENT CONTAINERS

Onto the dateform drop a `TButtonPanel` (accepting the default name), three labels and three spinedit. Set their properties as follows, so the resulting design looks like Figure 14.2:

`TButtonPanel`

```
ShowButtons [pbOK, pbCancel]
ButtonOrder boCloseCancelOK
```

`TLabel`

Caption	Day	Month	Year
Left	45	45	45
Name	LDay	LMonth	LYear
Top	20	60	100

`TSpinEdit`

Left	120	120	105
Name	seDay	seMonth	seYear
MaxValue	31	12	9999
MinValue	1	1	1
TabOrder	0	1	2
Top	16	56	96
Width	50	50	65

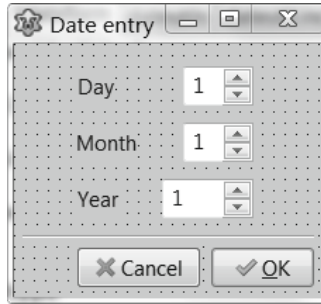


Figure 14.2 Designing a modal dialog

To the `dateform` unit add a global boolean function named `GetDate`, generate an implementation skeleton for it, and add `dateUtils` to the `uses` clause so that the completed unit looks as follows (the exact order of controls in the `TdateForm` class is immaterial):

Chapter 14 COMPONENT CONTAINERS

```
unit dateform;
```

```
{$mode objfpc}{$H+}
```

```
interface
```

```
uses
```

```
  SysUtils, Forms, Controls, ButtonPanel, Spin, StdCtrls, dateUtils;
```

```
type
```

```
TdateForm = class(TForm)
  ButtonPanel1: TButtonPanel;
  LDay: TLabel;
  LMonth: TLabel;
  LYear: TLabel;
  seDay: TSpinEdit;
  seMonth: TSpinEdit;
  seYear: TSpinEdit;
end;
```

```
function GetDate(var dt: TDateTime): boolean;
```

```
implementation
```

```
function GetDate(var dt: TDateTime): boolean;
```

```
var frm: TdateForm;
```

```
    d, m, y: word;
```

```
begin
```

```
  frm:= TdateForm.Create(nil);
```

```
  try
```

```
    if dt = 0 then dt := Now;
```

```
    DecodeDate(dt, y, m, d);
```

```
    frm.seYear.Value:= y;
```

```
    frm.seMonth.Value:= m;
```

```
    frm.seDay.Value:= d;
```

```
    if (frm.ShowModal=mrOK) then
```

```
      begin
```

```
        y:= frm.seYear.Value;
```

```
        m:= frm.seMonth.Value;
```

```
        d:= frm.seDay.Value;
```

```
        case IsValidDate(y, m, d) of
```

```
          False: Result:= False;
```

```
          else
```

```
            dt:= EncodeDate(y, m, d);
```

```
            Result:= True;
```

```
          end;
```

```
        end
```

```
      else Result:= False;
```

```
    finally
```

```
      frm.Free;
```

```
    end;
```

```
end;
```

```
{$R *.lfm}
```

```
end.
```

Chapter 14 COMPONENT CONTAINERS

The global `GetDate()` function is responsible for creating an instance of `TdateForm`, showing it via `ShowModal`, and freeing it using a `try... finally... end` construct. Data acquisition is through the `Value` property of three spinedit controls, which are set initially according to the value of the date passed as a `var` parameter to the `GetDate()` function. An invalid date, or cancelling the dialog both set the function's result value to `False`. If the function returns `True` you know the `var` date parameter passed back is a valid, user-accepted date.

To make the maintest unit a suitable test-bed for this date dialog, add `dateform` to its `uses` clause (use the `[Alt][F11]` shortcut for this), drop a button on the form, generate an `OnClick` handler for the button and complete this handler as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
var d: TDateTime=0;
begin
  if GetDate(d) then
    ShowMessageFmt('The date obtained via GetDate() was %s',
      [FormatDateTime('dd/mm/yyyy', d)])
  else ShowMessage('The date entry dialog was cancelled '+
    'or an invalid date was entered');
end;
```

Compile and run this program to check its UI and its manner of working.

You can adapt the scheme illustrated in `dateform.pas` to construct simple modal dialogs, called by a global function with a `var` data parameter, for getting validated user input about all manner of data.

14.e TGroupBox, TPanel

A bevel can be used to 'contain' other controls in the sense of providing an enclosing visual frame for them. However, `TGroupBox` and `TPanel` are containers in the full sense: they **parent** their contained controls, and if you drop a label, say, on a `TPanel` or `TGroupBox` you can drag it around the panel, but you cannot drag it **off** the panel (*try doing this to see what happens*).

Well, actually once dropped on a panel or groupbox you can drag a component off it. However, doing so does not move it to a part of the form outside the panel or groupbox. It seems to have disappeared, because it is drawn by its parent panel or groupbox, and all such drawing is clipped at the borders of the parent control. The only way to 'retrieve' such an unseen control is to select it in the OI treeview, and reset its `Left` and `Top` properties so as to bring it back inside its `Parent`.

Note that the `Left` and `Top` properties of contained (*child*) controls are given **relative to their parent container** control, which may not be the form.

The `TRadioGroup` and `TCheckGroup` controls are both based on a `TGroupBox` control. In one case the control contains a group of `TRadioButton` controls, and in the other case it groups `TCheckBox` controls. You may use `TCheckBox` controls singly on occasion (*outside of a group box*), but it never makes sense to use a single `TRadioButton`, so you will nearly always use the `TRadioGroup` control when radio button functionality is needed, rather than several individual radio buttons.

The `TGroupBox` border is rather inflexible in its appearance, always expecting a text `Caption`. If this is set to an empty string the upper border is still drawn halfway down the nonexistent text, giving an asymmetry with the bordering perimeter which means it can only sensibly be used with a non-blank `Caption`.

The `TPanel` by contrast is more versatile, not having a `Caption` drawn as part of its border – the `Caption` (*which can be blank*) is drawn by default in the centre of the control.

Chapter 14 COMPONENT CONTAINERS

The `Caption` is always centred vertically, but can be left-justified or right-justified to move it away from the horizontal centre. It can be used with or without a border.

If you want a completely border-less panel, then you will discover that setting `BorderStyle` to `bsNone` alone is not sufficient (*which is rather counter-intuitive*). You also have to set both `BevelInner` and `BevelOuter` to `bvNone`. However, panels are more commonly used with borders, and they are excellent controls to use to group other controls in sections on a form. Judicious use of the `Align` property of adjacent panels enables all manner of rectangular layouts to be set up without needing any code at all. The best way to learn about this is to experiment with three or four panels on a form and to try setting their `Align` properties to various combinations of `alNone` (*the default for a panel newly dropped in the Designer*), `alTop`, `alBottom`, `alLeft`, `alRight` and `alClient`. The last value means that the panel will occupy whatever space remains on the form. Figure 14.3 is an illustration of a layout that can be set up with panels in the Designer and OI in a couple of minutes without any coding at all.

14.f Resizeable children

More sophisticated layouts can also be designed without code using the `Anchors` property provided for each visual component. The Anchor Editor is accessed either by right-clicking a component in the Designer and choosing **Anchor Editor**, or by pressing the [...] ellipsis button beside the `Anchors` property in the OI. It is fun to explore the effects of this surprisingly powerful editor for yourself.

If you are happy to write code, the `Anchors` and `AnchorSide` properties are available for assignment, and there are several `AnchorXXX` methods also available for positioning controls in differing layouts if code-free OI settings using the `Anchors` and `ChildSizing` properties do not achieve the layout you desire.

If you need a container of two controls (*number one and number two*) whose **relative** sizes are to be user controlled, the `TSplitter` found on the *Additional Palette* page is the visual separator you are looking for. This is made to stick to one edge of control number one (*whose `Align` is set to one of `alTop`, `alBottom`, `alLeft` or `alRight`*) by having its `Align` set to the same value.

The adjacent control (*number two*) is made resizeable by having its `Align` set to `alClient`.

The user can drag the splitter (*which sports a small 'grip' of dots along its centre*) to resize these adjacent controls.

Here is a short program example demonstrating how this same effect can be achieved in code, rather than via property setting in the OI.

Create a new GUI project and name the main form unit `main_anchoring.pas`. Generate an `OnCreate` event handler for the form, and complete it so the unit looks like the following (*it will need `ExtCtrls` and `StdCtrls` in the `uses` clause for the memos and splitter that are created*).



Figure 14.3 A form of panel controls set up using the Designer/OI without writing any code

Chapter 14 COMPONENT CONTAINERS

```
unit main_anchoring;

{$mode objfpc}{$H+}
interface
uses Classes, Forms, Controls, StdCtrls, ExtCtrls;
type TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
end;
var Form1: TForm1;

implementation

{$R *.lfm}
procedure TForm1.FormCreate(Sender: TObject);
var
    Memo1, Memo2: TMemo;
    splitter: TSplitter;
    sl: TStringList;
begin
    Height:=140;
    sl:= TStringList.Create;
    try
        sl.CommaText:='Memo1 line2 line3 line4 line5';
        Memo1:=TMemo.Create(Self);
        with Memo1 do
            begin
                Lines.AddStrings(sl);
                Align:=alLeft;
                Parent:=Self;
            end;
        splitter:=TSplitter.Create(Self);
        with splitter do
            begin
                Align:=alNone;
                Left:=120;
                Parent:=Self;
                AnchorParallel(akBottom,0,Parent);
            end;
        Memo1.AnchorToNeighbour(akRight,0,splitter);
        sl.Clear;
        sl.CommaText:='"Memo 2","2nd added line","3rd added line",' +
            '"4th added line","5th added line"';
        Memo2:=TMemo.Create(Self);
        with Memo2 do
            begin
                Align:=alRight;
                AnchorToNeighbour(akLeft,0,splitter);
                Lines.AddStrings(sl);
                Parent:=Self;
            end;
        finally
            sl.Free;
        end;
    end;
end.
```


Chapter 14 COMPONENT CONTAINERS

The completed application will look something like Figure 14.4, and the two memo controls' widths are resizable using the central splitter. The stringlist component used to populate the two memos is considered in the next chapter.

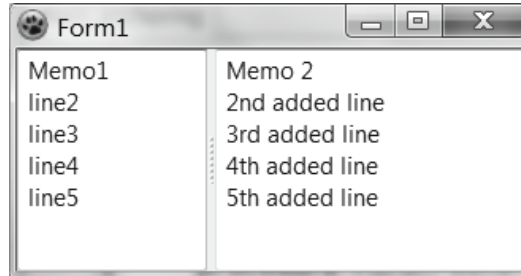


Figure 14.4 A splitter control dividing adjacent controls contained in a form

14.g TFrame

A **frame** is a further component container similar in many ways to a form (*i.e. a window*).

However, you can only have one form per unit, whereas you can have more than one frame per form.

A principal use of frames is to give a multi-window application a consistent look from window to window by using the same frame in different windows to place exactly the same components in exactly the same relative positions with exactly the same shared properties and methods.

A frame is a containing template that both positions contained controls in terms of layout and sets other non-layout property values. A single frame placed on several different forms acts as a sort of duplicator, enabling you to position components identically across several different windows, without needing to manually construct and design each one hoping it will look identical to the others.

There is no predefined `TFrame` component on the Palette. Just as you have to create a new form using **File | New Form**, so you have to create any frames you use by choosing **File | New...** and then picking *Frame* from the *Module* branch of the treeview in the ensuing *New...* dialog. This creates a new frame named `Frame1` of type `TFrame1` by default, just as the first new form is named `Form1` by default.

You develop the design of `Frame1` (or whatever you might rename it to) by dropping controls onto the frame, and writing event handlers or property setting code just as you would for a form. Lazarus provides a supporting unit for the frame (named `Unit2.pas` or similar by default) just as it does for a form, and a corresponding (automatically synchronised) `Unit2.lfm` file where the frame's property and event data is recorded. The newly designed frame has to be saved before you can use it. It then becomes part of that project. Each frame requires its own unit (which Lazarus automatically provides) since it also needs the corresponding `.lfm` file for its property data.

To use the new frame after saving it, you use the *Standard* Palette page, choosing the frame icon, (*second from the right*). Select this icon, and on clicking a form (or a panel) a *Select Frame* dialog appears which lets you select a frame from the list of frames you have so far defined in your project. The frame you select will be dropped on the form at the point where you click it, and the frame with its components can be manipulated in the OI or in code just like any other components on the form outside the frame.

Chapter 14 COMPONENT CONTAINERS

14.h TDatamodule

A datamodule is a container class you can use to centralise data access, business rules and non-visual components in an application. You can tell from their name that datamodules were first introduced to modularise the location of database objects.

However, a datamodule can also be a helpful container for non-database applications to group timers, dialogs, and other non-visual components (*even popup menus*) that relate to the application as a whole rather than to a specific form.

This is particularly true in multiple-form applications. Most often datamodules are used as containers for components from the *Data Access Palette* page (such as *TDataSource*) or from the *SQLdb* page (such as *TSQLQuery*).

Datamodules are not visible at runtime, and their visibility at design time is simply to aid in dropping the required components within the container. Any unit can have access to the objects in a datamodule simply by including that module in the unit's **uses** clause.

A datamodule behaves rather like a hidden window.

However, compared to a visual form they are helpfully lightweight containers (*an empty datamodule has an InstanceSize of just 60 bytes compared to an empty form of about 1Kb*).

You create a new datamodule by selecting **File | New...** to open the *New...* dialog, and then you choose *Data Module* from the list under the *Module* branch of the dialog's treeview. Non-visual components are dropped on a datamodule in exactly the same way that they are placed on a form or a frame.

Datamodules have just two published events (*OnCreate and OnDestroy*) that can be exploited helpfully to respectively initialise and at the end of the program to clean up the non-visual component resources your application uses.

A datamodule automatically becomes the Owner of any component dropped onto it at design time, in an analogous way to the automatic ownership a form takes of components dropped onto it. Consequently such contained components never need to be freed manually.

Since datamodules are invisible at runtime, and their contained components are non-visual, the question of parentage does not arise in this case (*except for menus, which are allowed*).

14.i Review Exercises

1. Create a new project and add several new forms to it.
Use **Project | Project Options...** to open the *Options for Project...* dialog and navigate to its *Forms* page. Use the functionality here to make some forms auto-created and some forms not. Then check the behaviour of your project at runtime.
2. Use the form methods, `Show`, `Close`, `ShowModal` to duplicate in code the behaviour of the project started in Exercise 1.



Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

There are hundreds of classes declared in the FCL and LCL which GUI programmers use whether knowingly or not, since visual controls often depend on several other classes in addition to their named ancestor. Not all classes are created equal, however, and some are far more important to understand than others in order to develop robust GUI programs. This chapter looks at some of the more important classes on which the LCL design depends, the non-visual classes that are not components (*i. e. they don't descend from TComponent*). This means they can never have a place on the Component Palette ready to be dropped on a form. Nevertheless you will use them (*directly or indirectly*) in all your GUI programs alongside (*even within*) the visual controls you use.

15.a TPersistent descendants

All the visual (*and non-visual*) controls on the Palette that can be dropped on a form in the Designer and manipulated in the OI depend on a system of meta-information called RTTI (*RunTime Type Information*) for their correct runtime behaviour. The base class that introduces this is called TPersistent. A great many classes used in Lazarus consequently descend from TPersistent, including all components (*that is, descendants of TComponent*) and all TStrings classes such as TStringList. There are two other important TPersistent descendants we need to mention to complete this chapter's survey. They are TCollection and TCollectionItem. The inter-relationship of a select few of the more important classes underpinning the LCL is depicted in Figure 15.1.

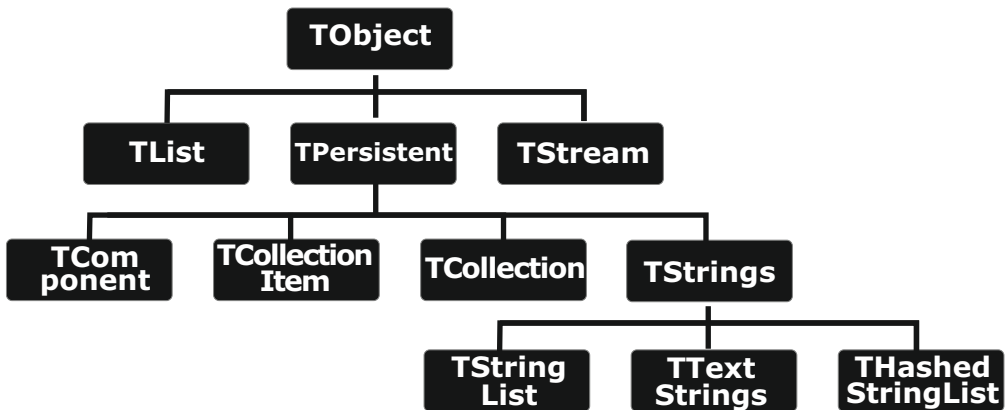


Figure 15.1 A very limited portion of the class hierarchy supporting the LCL

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Most controls of any complexity need to manage groups of identical elements. For instance a grid component is made up of columns. While each column may have different display characteristics (*different widths, positions, colours, underlining and so on*) the basic structural features of each column are the same. Likewise with the selection of styles in a chart. Each style is different, but the basic functionality of each style is the same, to be consistent across all styles. `TCollection` and `TCollectionItem` are classes designed to manage collections of such identically structured elements. To illustrate how these collection classes work an example follows that uses an RTTI component, the `TTIGrid`.

15.b A chemical `TCollection`

Start a new Lazarus project named `chemCollection` with a main form unit named `mainChemForm.pas`. In the OI name the form `ChemForm`. **Note** how Lazarus immediately renames the form class to `TChemForm`. Drop a `TTIGrid` on the form and name it `chemGrid`. For this example we will set its properties in code rather than using the OI, to demonstrate that this is sometimes a useful alternative way of initialising controls. (*For example, if the properties are public rather than published you have to set them in code since they will not be available in the OI*). The completed project will look something like Figure 15.2.

A `TCollection` always needs a paired `TCollectionItem`. Here we will use a small class called `TElementItem` which has four properties relating to a chemical element (`AtomicNo`, `Name`, `Group`, `Symbol`) based on four correspondingly named private data fields. The properties are implemented through direct read/write access to their associated data fields (*no setter or getter methods are needed here, since we are not interested in any side effects*). This means the `TCollectionItem` class needs no code in the unit's implementation section since it has no new methods to implement.

`TElementItem` is declared as a descendant of `TCollectionItem`. Since `TCollectionItem` is a descendant of `TPersistent` this gives it full RTTI functionality, and `TElementItem`'s `TCollectionItem` ancestry means it already knows how to work as part of a `TCollection` without our needing to write any further code.

In the unit's type declaration add the declaration for `TElementItem` as follows:

```
type
  TElementItem = class(TCollectionItem)
  private
    FNumber: cardinal;
    FName: string;
    FSymbol: string;
    FGroup: string;
  published
    property AtomicNo: cardinal read FNumber write FNumber;
    property Name: string read FName write FName;
    property Symbol: string read FSymbol write FSymbol;
    property Group: string read FGroup write FGroup;
  end;
```

The only new idea here is the `cardinal` type, an unsigned integer type (*since atomic numbers cannot be negative*). Although a byte would clearly be sufficient storage (*even artificial elements don't yet have atomic numbers higher than 110*) the `cardinal` type is preferred since it is a native type on 32-bit computers, and we no longer need to squeeze every available byte out of memory as was required in the 1970s. `TElementItem` is basically a data-containing class which has `TCollectionItem` functionality built in through inheritance.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Add grids to the unit's **uses** clause, and in the **private** section of the form class insert a field called `Felements` of type `TCollection`, followed by a procedure named `SetupChemGrid`, and invoke code completion to write its skeleton implementation body. This procedure will create a `TCollection` instance, populate the `Felements` collection with a few `TElementItem` instances and initialise some properties of `chemGrid`, the `TTIGrid`. The tasks carried out by `SetupChemGrid` are separated into three sub-procedures to make the logic flow clear. Here is the code to insert:

```
procedure TChemForm.SetupChemGrid;

  procedure AddElement(aNumber: cardinal; const aName, aSymbol,
                      aGroup: string);
  var ei: TElementItem;
  begin
    ei := TElementItem(Felements.Add);
    ei.AtomicNo:= aNumber;
    ei.Name:= aName;
    ei.Symbol:= aSymbol;
    ei.Group:= aGroup;
  end;

  procedure AddElements;
  begin
    AddElement(1, 'Hydrogen', 'H', 'I');
    AddElement(2, 'Helium', 'He', 'II');
    AddElement(3, 'Lithium', 'Li', 'I');
    AddElement(4, 'Beryllium', 'Be', 'II');
    AddElement(5, 'Boron', 'B', 'III');
    AddElement(6, 'Carbon', 'C', 'IV');
    AddElement(7, 'Nitrogen', 'N', 'V');
    AddElement(8, 'Oxygen', 'O', 'VI');
    AddElement(9, 'Fluorine', 'F', 'VII');
    AddElement(10, 'Neon', 'Ne', 'VIII');
  end;

  procedure InitialiseChemGrid;
  begin
    Self.Height:= 310;
    Self.Width:= 340;
    chemGrid.Align:= alClient;
    chemGrid.TIOptions:= chemGrid.TIOptions + [tgoStartIndexAtOne];
    chemGrid.Options:=chemGrid.Options + [goDrawFocusSelected];
    chemGrid.FixedColor:=clMoneyGreen;
    chemGrid.DefaultDrawing:=True;
    chemGrid.AutoFillColumns:=True;
  end;

begin
  Felements := TCollection.Create(TElementItem);
  AddElements;
  InitialiseChemGrid;
  chemGrid.ListObject := Felements;
end;
```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

You will notice here two unusual aspects of working with collection classes.

Firstly you instantiate a collection by invoking its `Create()` constructor to which you pass a type parameter specifying what sort of `TCollectionItem` class this collection will be collecting. So here:

```
Felements := TCollection.Create(TElementItem);
```

Secondly, rather than explicitly creating each `TElementItem` instance with a `Create` call we create each instance (for which we reuse a variable called `ei`) by using the owning collection's `Add` method:

```
ei := TElementItem(Felements.Add);
```

Although it does not look as though there is a constructor call here, in fact there is, and `FElements.Add` both instantiates a new `TCollectionItem` and adds it to the collection.

Because this is a general `Add` scheme designed for any `TCollectionItem` descendant we have to cast the newly instantiated `TCollectionItem` returned by the `Add()` call as a `TElementItem`. When should we call `SetupChemGrid`? At the very start of the program, once the main form is instantiated. In other words we use the form's `OnCreate` event handler to call this setup routine. In the OI, with `chemForm` selected, click the *Events* tab, and double-click beside `OnCreate` to get Lazarus to generate the event handler. Complete it with the simple call:

```
procedure TChemForm.FormCreate(Sender: TObject);  
begin  
    SetupChemGrid;  
end;
```

One last consideration. In `SetupChemGrid` we called a `TCollection` constructor, instantiating a new collection. What happens to that collection instance in memory at the end of the program? If we don't free the memory, nothing else will. It will remain orphaned, out of reach and unusable by any other program. This is a very common bug: forgetting to free memory we have allocated which leads to memory 'leaks' (*as they are called*). Actually leak is not a very suitable term here, because although the computer will behave as if its memory were leaking away in fact what the memory has is not a leak but constipation! A build up of inaccessible garbage in memory that is not being disposed of or thrown away.

It is simple enough to deallocate the memory allocated in the `Felements` constructor.

In the OI's *Events* tab double-click next to `OnDestroy` to create an event handler that will be invoked just before the form is destroyed. Complete it as follows:

```
procedure TChemForm.FormDestroy(Sender: TObject);  
begin  
    Felements.Free;  
end;
```

Why do we not need to also free all the `TElementItem` instances we created? Because the collection instance, `Felements`, frees everything in its collection when it is itself freed. That is part of the `TCollectionItem` functionality we inherited when we declared `TElementItem` as a `TCollectionItem` descendant. The RTL classes come with a great deal of intelligence built in to them already. We are benefiting from the experience of a team of skilled programmers who have walked this way before us. Which is why we will nearly always do better to seek a solution within the RTL/FCL/LCL than try to reinvent such wheels ourselves.

Studying the source code is one of the best ways to learn about this. So users of open source proprietary development tools are better off than programmers who struggle with closed source proprietary development software and libraries, which always remain unknowable 'black boxes' (*unless fees are paid for access to the source, and NDAs are signed when you hand the money over*).

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Added to which, FPC and Lazarus users can get bugs fixed often within days, and submit patches themselves to address problems they encounter. This usually makes for a far quicker response from the developers than you find with many commercial software vendors.

Nothing is guaranteed, of course. Actually that is true in both scenarios.

You cannot buy bug-free software, and “more expensive” does not necessarily equate to “has fewer bugs”. You can only try to make your software as stable and bug-free as possible.

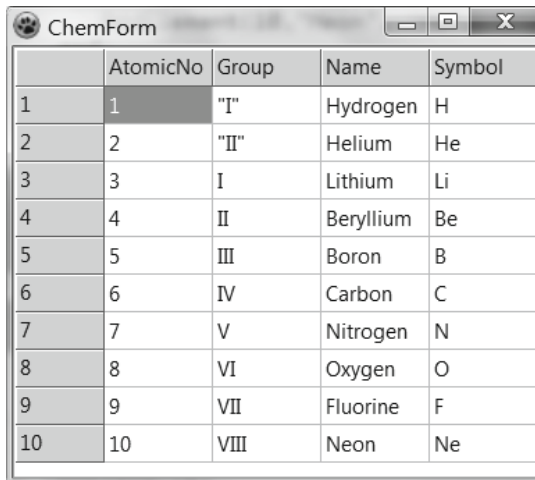
People who claim their software is absolutely bug-free are self-deluded. All that can be claimed (*if it is actually true*) is that all presently-known bugs have been fixed.

If you compile and run the `chemCollection` project you should see a grid presenting the first few elements of the Periodic Table. Through the 'magic' of RTTI and use of `TPersistent` descendants as classes the `TTIGrid` has taken the `TElements` collection and

- determined that each item has four properties
- constructed a grid to display those four properties
- added a row for each item in the collection
- arranged the columns in alphabetical order by property name
- numbered each item according to its row

We achieved this with less than 100 lines of our own code. You can look at the source for

`TPersistent` and `TTIGrid` to see how much other code is also required to produce these effects!



	AtomicNo	Group	Name	Symbol
1	1	"I"	Hydrogen	H
2	2	"II"	Helium	He
3	3	I	Lithium	Li
4	4	II	Beryllium	Be
5	5	III	Boron	B
6	6	IV	Carbon	C
7	7	V	Nitrogen	N
8	8	VI	Oxygen	O
9	9	VII	Fluorine	F
10	10	VIII	Neon	Ne

Figure 15.2 Viewing a chemical `TCollection`

15.c The `TStringList` class

`TList`, depicted in the class hierarchy diagram shown in Section a (*see Figure 15.1*), is a class maintaining a list of pointers. It is a sort of model for a number of similar list classes which offer methods with identical (*or analogous*) names such as `Add`, `Assign`, `Clear`, `Delete`, `Insert`; and similar properties such as `Capacity`, `Count` and a default array property (`Items[]` for `TList`, `Strings[]` for `TStringList`). All these list classes have a zero-based default array property for easy access to the principal data contained in the list.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

The first pointer in a `TList` named `aList` is `aList[0]` and the last pointer is `aList[Count-1]`. The first string in a stringlist named `sList` is `sList[0]`, and the last string in the stringlist is `sList[Count-1]`. There are numerous such correspondences between list classes in the Lazarus libraries, so getting familiar with the methods and properties of one class means you will know a good deal about many of the similarly-named methods and properties of the other list classes. `TStringList` is one of the most widely used RTL classes used in LCL components.

It is not a visual control (*so not available on the Palette*); nevertheless it is one of the fundamental classes you need to become familiar with in order to use Lazarus effectively. It is a direct descendant of the `TStrings` class, which is an abstract class, designed to manage and manipulate a list of strings without actually implementing any storage for them. An *abstract* class is a class which should not have any actual instances but rather is designed as the immediate ancestor for a family of more specialised related classes. For instance, the `TMemo` component uses a `TStrings` descendant very similar to a `TStringList` to implement its `Lines` property. So you will not be surprised to know that the first line in a `memo1` is `memo1.Lines[0]`, and the last line is `memo1.Lines[Count-1]`. A listbox also uses a `TStrings` descendant for its `Items`. Another example of the use of string lists in the LCL is the `TRadioGroup` control. If you drop such a control on a form it is initially empty. The radio buttons are added as `Items` in the OI. Initially the `Items` stringlist is empty (*Items.Count is zero*), and new items are added as new text lines are typed in the OI Strings Editor. In the Project Options dialog (**Project | Project Options...**) the *Session* page includes a setting for session storage (see Figure 15.3).

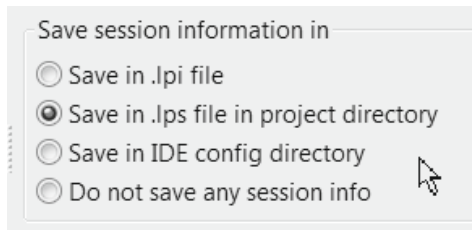


Figure 15.3 A radio group used in the IDE

There are four session options, of which only one can be active. This is exactly the situation where radio button choices are ideally suited. Internally the labels for each choice are stored in a stringlist. These session choices are *Project options* (*not IDE options affecting the whole IDE environment*) so you can set different options here for each project you write, if you wish.

15.d Sorting lines in a text file

Unlike its ancestor `TStrings`, a `TStringList` provides storage for the strings it contains (*in a special array that can contain a theoretical maximum of 134,217,728 strings*), and it adds methods to sort its strings and to find a particular string. `TStrings` itself is a surprisingly capable class able to save and read strings from disk files, associate any arbitrary `TObject` with each string, and provide specialised handling of two types of structured text: text where each line is of the form *name = value*, and text where each line is a series of comma-separated phrases.

The following example illustrates some of `TStringList`'s capabilities in a short application, included so that you can extend the ideas here more usefully. The project analyses text, reporting both the word count, and the count of unique words.

Begin a new Lazarus project called `slistexample`, with a main form unit called `slistform.pas`.

Add a `const` declaration between the `uses` and `type` sections:

```
const specimen: string = 'Enter by the narrow gate; for the wide gate has a '+  
'broad road which leads to destruction and there are many people going that '+  
'way. The narrow gate and the hard road lead to life and only a few find it.';
```


Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Set the form's properties as follows:

Caption	Stringlist example
Height	320
Width	480

Drop a label on the form, and set its properties as follows:

Left	10
Name	LCount
Top	10

Drop a listbox on the form, and set its properties as follows:

Align	alBottom
Columns	4
Height	200
Left	10
Name	LBWords

Drop a memo on the form above the listbox, and set its properties as follows:

Align	alBottom
Height	80
Left	10
Name	Mdisplay

The form should look something like Figure 15.3.

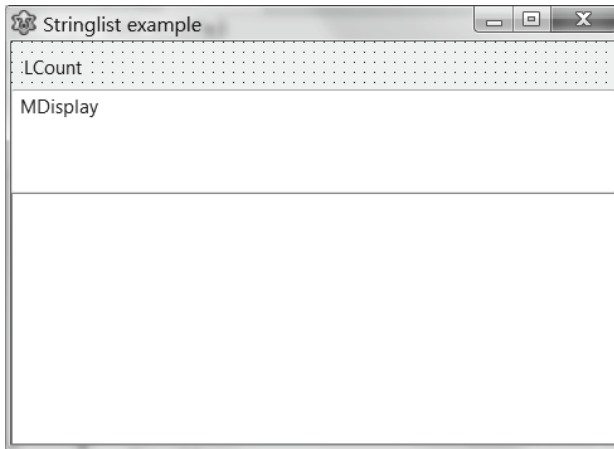


Figure 15.3 Stringlist example in the Designer

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Double-click the form to create an `OnCreate` handler and fill out the skeleton routine thus:

```
procedure TForm1.FormCreate(Sender: TObject);
var sl, noDuplicates: TStringList;
    j: integer;
begin
  MDisplay.Text:= specimen;
  sl := TStringList.Create;
  noDuplicates := TStringList.Create;
  try
    sl.CommaText:= specimen;
    sl.Sort;
    for j := 0 to sl.Count-1 do
      if (noDuplicates.IndexOf(sl[j]) < 0)
      then noDuplicates.Add(sl[j]);
    LBWords.Items.AddStrings(noDuplicates);
    LCount.Caption:=
      Format('Specimen text has %d words of which %d are unique',
            [sl.Count, noDuplicates.Count]);
  finally
    sl.Free;
    noDuplicates.Free;
  end;
end;
```

Compile and run this project, and see the instantaneous parsing undertaken by the two stringlist instances. Here we have used two stringlists. The first one, `sl`, has its `CommaText` property set to the specimen text. (**Note** that on some OSs the `Listbox`'s `Columns` property has no effect, since the native widget does not implement this functionality).

By default `TStringList` treats the space character ' ' as a separator (*delimiter*) and the assignment of `specimen` to `CommaText` causes the stringlist to parse the assigned text at each separator adding each parsed phrase to the list of strings. The value of the separator recognised for this process can be changed using the (Char) `Delimiter` and (boolean) `StrictDelimiter` properties. `TStringList` has a convenient `Sort` method we can call. An equivalent would be to use the `Sorted` property, setting it to `True`.

The second stringlist, `noDuplicates`, is used to filter out any duplicate entries in `sl`. After creation `noDuplicates` is an empty list. We step through each string in `sl`, and provided it is absent from `noDuplicates` we add it. If it is already present it is not added again. This test is performed using the `IndexOf()` method which returns a positive index value for a string's position in the list if it is present, and -1 if the string is absent.

The resulting stringlist of unique strings is copied to the listbox's `Items` property using the `AddStrings()` method. This takes a single `TStrings` parameter. Since `noDuplicates` is a `TStringList` it is also a `TStrings`, so is assignment-compatible with the required parameter. The `try...finally...end` block ensures that the memory required to instantiate the two string lists is freed correctly after use, even if something goes wrong.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

15.e Streams

`TStream` (depicted above in the class hierarchy in Figure 15.1) is FPC's object oriented encapsulation of a general sequences of bytes. The `TStream` class descendants cater for a large variety of sequential data such as files, areas of memory, program resources, strings, database BLOB fields, sockets and named pipes.

A stream is a useful abstraction when dealing with sequential data storage and the movement of that data to and from its storage. The idea behind a stream is that you are moving along the data while you read it, so a stream is based on the idea of a flow of items being handled in sequence. A stream can contain any data, in any order. Streams are often designed to handle fixed-sized data units (*since this makes for simpler programming*), however they are not in any way restricted to containing fixed-size data as dynamic arrays are.

Streams are 'indexed' only in terms of numbered bytes, whereas dynamic arrays are indexed according to the size of the base type of the array. A stream can grow in memory up to the size of memory available on the computer running the stream program, which is usually a very high capacity value on today's computers.

`TStream` is an abstract class, meaning it has no actual implementation (*so you cannot create an instance of `TStream`*). Rather, `TStream` specifies a programming interface and is designed solely as a parent for actual implementations such as `TFileStream` (*which encapsulates reading and writing operations on files*). `TStream` declares generic methods and two properties that are implemented in descendant classes, and it uses exceptions (*not run-time errors*) for handling errors. `TStream` and most of its descendants are declared in the `Classes` unit of the RTL.

`TStream`'s two properties are:

- `Position`
- `Size`

`Position` denotes the current location in the stream where `Read` or `Write` operations will be executed. `Size` denotes the current stream size in bytes. If you write additional data at the end of the stream, the stream grows and `Size` increases accordingly.

Note that some stream descendants (*e.g. pipes and compressed/decompressed streams*) lack these two properties since they are not applicable to all streams.

`TStream`'s principal methods are `CopyFrom`, `Read`, `Seek` and `Write`. As you would expect `CopyFrom` is used to copy data from one stream to another. `Read` reads data from the stream, starting at `Position`. `Seek` is used to change the `Position`. `Write` writes data to the stream starting at `Position`. Stream syntax is particularly simple and elegant.

A big advantage in working with streams rather than traditional Pascal file routines is that any stream descendant can be passed as a `TStream` parameter to routines that utilise streams, giving very versatile interoperability. Of course the calling routine has to pass the correct type of `TStream` descendant (*just as routines passing `TStrings` parameters need to pass the correct `TStrings` descendant*). In fact many LCL components and FCL classes have a `SaveToStream()` or `LoadFromStream()` method, so understanding `TStream` is as essential as understanding `TStrings` if you are going to exploit the full functionality of these libraries in your own code.

Streams were introduced into Object Pascal partly to aid in storing class data, and restoring a newly created class instance to the state in which it was last saved (*i.e. with all its properties set to their correct values*). Consequently `TStream` has several component-related methods such as `ReadComponent()` and `WriteComponent()` which are a specific and perhaps surprising capability that all RTL streams inherit. In fact all `TComponent` descendants have the built-in capability to stream themselves to and from data storage.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

15.f TFileStream

To create a file stream you need to supply two parameters for the `Create()` constructor: the name of the file, and a flag indicating the file mode.

Possible values for the file mode parameter are:

- `fmOpenRead`
- `fmOpenWrite`
- `fmOpenReadWrite`
- and five further share modes.

We give here a simple file copying example program utilising `TFileStream` showing how to copy any file to the same directory as the original with “Copy of” prepended to the filename (optionally setting the copied file's date to the original file's date).

Start a new Lazarus project named `filecopy` with a main form unit named `maincopy`.

Set the form's `Caption` to `File copying using streams`, and its `Width` to `680`. Drop a label, `lblCopyFrom` in the top left-hand corner of the form, with its `Caption` set to `Copy from file:`, and drop beside it a `TFileNameEdit` named `edtCopyFrom` setting its `Width` to `515`.

Below the first label drop another label named `lblCopyToFile` with its `Caption` set to `File will be copied to:`, and below that label drop a `TCheckBox` named `cbDate` with its `Caption` set to `Copy original file's date stamp`. Complete the GUI with a button named `btnCopy`, with its `Caption` set to `Copy file`.

Delete the form's `public` section, and add the following fields and method to the `private` section of the `TForm1` class declaration:

```
private
    SourceFileName: string;
    CopiedFileName: string;
    procedure CopyFile(sourceName, destinationName: string;
                      copyDateToo: boolean);
end;
```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Generate event handlers for the button's `OnClick` event, and the `FileNameEdit`'s `OnAcceptFileName` event. Press `[Shift][Ctrl][C]` on the appropriate line to generate the skeleton for `CopyFile`. Complete these three methods as follows:

```
procedure TForm1.edtCopyFromAcceptFileName(Sender: TObject; var Value: String);  
var path, fName: string;  
begin  
  if (Value = EmptyStr) then Exit;  
  path := ExtractFilePath(Value);  
  fName:= ExtractFileName(Value);  
  copiedFileName:= path + 'Copy of ' + fName;  
  case FileExistsUTF8(copiedFileName) of  
    False: begin  
      lblCopyToFile.Caption := 'File will be copied to: '+  
        copiedFileName;  
  
      btnCopy.Enabled:= True;  
      SourceFileName:= Value;  
    end;  
    True : case QuestionDlg('Warning', copiedFileName + ' already exists'  
      +sLineBreak+'Overwrite existing file?', mtWarning,  
      [mrYes,'Overwrite file', mrNo,'Cancel file copy'],0) of  
      mrYes: begin  
        lblCopyToFile.Caption := 'File will be copied to: '  
          + copiedFileName;  
  
        btnCopy.Enabled:= True;  
        SourceFileName:= Value;  
      end;  
      else begin  
        Value:= EmptyStr;  
        btnCopy.Enabled:= False;  
        SourceFileName:= EmptyStr;  
        CopiedFileName:= EmptyStr;  
      end;  
    end;  
  
  end;  
end;  
  
procedure TForm1.btnCopyClick(Sender: TObject);  
var append: string;  
begin  
  CopyFile(SourceFileName, CopiedFileName, cbDate.Checked);  
  btnCopy.Enabled:=False;  
  edtCopyFrom.Text:= EmptyStr;  
  case cbDate.Checked of  
    False: append := ' created with current date';  
    True: append := ' created with original date';  
  end;  
  lblCopyToFile.Caption:= CopiedFileName + append;  
end;
```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

```
procedure TForm1.CopyFile(sourceName, destinationName: string;
  copyDateToo: boolean);
var src: TFileStream = nil;
    dest: TFileStream = nil;
begin
  if SameText(sourceName, destinationName) then Exit;
  src := TFileStream.Create(UTF8ToSys(sourceName), fmOpenRead);
  try
    dest := TFileStream.Create(UTF8ToSys(destinationName),
                               fmOpenWrite or fmCreate);

    try
      dest.CopyFrom(src, src.Size);
      if copyDateToo
      then FileSetDate(dest.Handle, FileGetDate(src.Handle));
    finally
      dest.Free;
    end
  finally
    src.Free;
  end;
end;
```

The `CopyFile` procedure uses `TFileStream.Create`, and `TFileStream.CopyFrom` to perform the actual file copy. The button `OnClick` procedure calls `CopyFile`, disables the `Copy` file button, clears the edit field of the `TFileNameEdit` for subsequent use, and reports on the result of the current copy operation.

The `AcceptFilename` event of the `TFileNameEdit` checks for an existing file of the same name as the about-to-be-created file copy, and proceeds accordingly, on the basis of user input if the filename to be used does indeed exist. User input is solicited, if necessary, using the `QuestionDlg()` function, one of several user-input dialogs found in the `dialogs` unit (which is one of the units Lazarus adds automatically to every new GUI project).

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

15.g TMemoryStream, TStringStream and Blowfish

We conclude this chapter with an example that uses a memory stream to hold a variable number of randomly generated records. The stream can be saved to and restored from a disk file, and the program's UI is designed so you can move along the stream, backwards and forwards, examining individual records in the context of the surrounding records.

The application introduces two LCL components we have not used so far (*TUpDown*, and *TTrackBar*) and also shows slightly more sophisticated use of some formatting functions to display string and date/time data clearly.

The example uses a `TMemoryStream`-derived class named `TdemoStream`, rather than a plain `TMemoryStream`. This is to illustrate how even such a simple class as `TdemoStream` can be a useful encapsulation of code and data.

The stream used in this demonstration is a stream of fixed-size records, since having fixed-size records makes navigating up and down the stream very straightforward.

These records are associated only with this stream, and because the stream handles no other kinds of data it makes sense in this application to wrap the record definition and record-data display code within the derived memory stream class. For this reason we declare the record definition and new memory stream class in a unit of their own, separate from the main form unit. The main form unit then refers to the stream unit in its `uses` clause.

This example also exercises the encrypting/decrypting streams provided in the FCL which are based on the public domain Blowfish algorithm. The application of Blowfish here is completely pointless except to illustrate one way to program string and cipher streams (see *Figure 15.4*).

Start a new Lazarus project named `stream_visualise`, with a main form unit named `main_stream`. Via **File | New Unit** generate a new unit for the project named `demo_stream`. You'll see that Lazarus gives a non-form unit a `uses` clause with only two units in it: `Classes` and `SysUtils`. Of course it may happen that you don't need anything from either of these units, but in our case we will use both of them, so can happily accept this default `uses` clause. The memory stream we shall employ will hold record entries of a type named `TdemoRec` which for demonstration purposes consists of a character, an integer, a shortstring, and a `TDateTime` value. In the virgin `demo_stream` unit, declare these types first, as follows:

```
type
  Ts20 = string[20];

  TdemoRec = record
    aChar: Char;
    anInt: integer;
    aString: Ts20;
    aDate: TDateTime;
  end;
```

These two types are used by our `TMemoryStream` descendant class, named `TdemoStream` whose declaration follows. It is very little different from a basic `TMemoryStream` class, but has a new constructor, which takes a parameter specifying how many new records to insert in the newly created stream. It has a `private` helper function used by this constructor, and also a `public` string function used to display an individual record in the stream.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

There is also a `public rec` field which will hold the record at the current stream's Position (if there is one). This is declared as a field rather than a property since it is merely used as a placeholder for data. We could have declared a `Rec` property with setter and getter methods that always reflected the contents of the record at the current `Position`, but this is not needed in this application, so we went for a simpler implementation with fewer lines of code.

```
TdemoStream = class(TMemoryStream)
  private
    function GenerateRandomRec(num: integer): TdemoRec;
  public
    rec: TdemoRec;
    constructor Create(numRecords: integer);
    function RecAsString: string;
end;
```

In the **implementation** section add `uses math`; and then use Code Completion to generate the method skeletons required for the `TdemoStream` class. Complete them as follows:

```
function TdemoStream.GenerateRandomRec(num: integer): TdemoRec;
begin
  Result.aChar:= Chr(65 + Random(26));
  Result.aDate:= TDateTime(RandomRange(100, trunc(Now)) + Random/10000);
  Result.anInt:= num;
  Result.aString:= Format('example string %d',[num]);
end;

constructor TdemoStream.Create(numRecords: integer);
var i: integer;
begin
  Randomize;
  inherited Create;
  Position:= 0;
  for i := 0 to numRecords-1 do
    Write(GenerateRandomRec(i), SizeOf(TdemoRec));
end;

function TdemoStream.RecAsString: string;
begin
  Read(rec, SizeOf(rec));
  Result := Format('%5d %s %17s %s',
    [rec.anInt, rec.aChar, rec.aString,
    FormatDateTime('dd-mm-yyyy:mm:ss', rec.aDate)]);
end;
```

The constructor initialises the FPC random number generator by calling `Randomize`, and then calls the inherited `TMemoryStream` constructor to instantiate a new instance of this class. It then proceeds to write to the newly created stream the specified number of `TdemoRec` records with data suitably supplied for each record via a call to `GenerateRandomRec()`.

The `RecAsString: string` function reads a record from the stream (at the current `Position`) and formats this composite char-integer-string-date record as a single string for display.

Note that after each `Read` operation required to produce `RecAsString` that the stream `Position` is advanced automatically ready to read the subsequent record. Streams therefore don't need a manual `Seek` following a `Read`. This completes the `TdemoStream` declaration and implementation, so we can move on to the program's GUI.

How can we best visualise a stream of records?

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

15.h Visualising a stream

The remaining work is to build a GUI to exercise the stream class we have just written. The finished application will look something like Figure 15.4. Start building the GUI by setting the main form's properties thus:

Caption	Visualising a stream of records
Height	670
Width	700

Drop a `TTrackBar` on the form, setting its properties as follows:

Align	alTop
Name	tbPosition
TabStop	False

Below the trackbar drop three labels. The leftmost label, `lblOrigin` has its `Caption` set to 0, and has its top anchored to the trackbar. Open the `Anchors` Editor by clicking the [...] ellipsis button next to the `Anchors` property in the OI to set this (*in the Top Anchoring groupbox choose `tbPosition` in the drop-down list, and click the middle of the three buttons to set the label's anchoring to the bottom side of the trackbar*).

The centre label, `lblPositionDesc` has its `Caption` set to < the trackbar pointer above illustrates the stream's `Position` between origin and `Size` >.

The rightmost label, `lblStreamSize`, is anchored to the bottom of the trackbar, and to the right edge of the form. Its `Alignment` is set to `taRightJustify`, and it has an empty `Caption`.

Below the labels drop an edit named `edtRecord`, and when you set its `AutoSize` property to `False` you can stretch it to fit the window. Set its `Alignment` property to `taCenter`.

Below the edit drop a memo, and set its properties as follows:

Font	(set to a mono-spaced typeface such as Courier)
Height	300
HideSelection	False
Name	memoDisplay
ScrollBars	ssAutoBoth
Width	450

In the section to the right of `memoDisplay` drop a label named `lblAdjustPosition` with the `Caption` set to Show previous/next record, and beside it drop a `TUpDown` named `udMover` with its `Orientation` set to `udHorizontal`.

With the `TUpDown` selected, on the *Events* page of the OI double-click beside the `OnClick` event to generate an event handler, and complete this as follows:

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

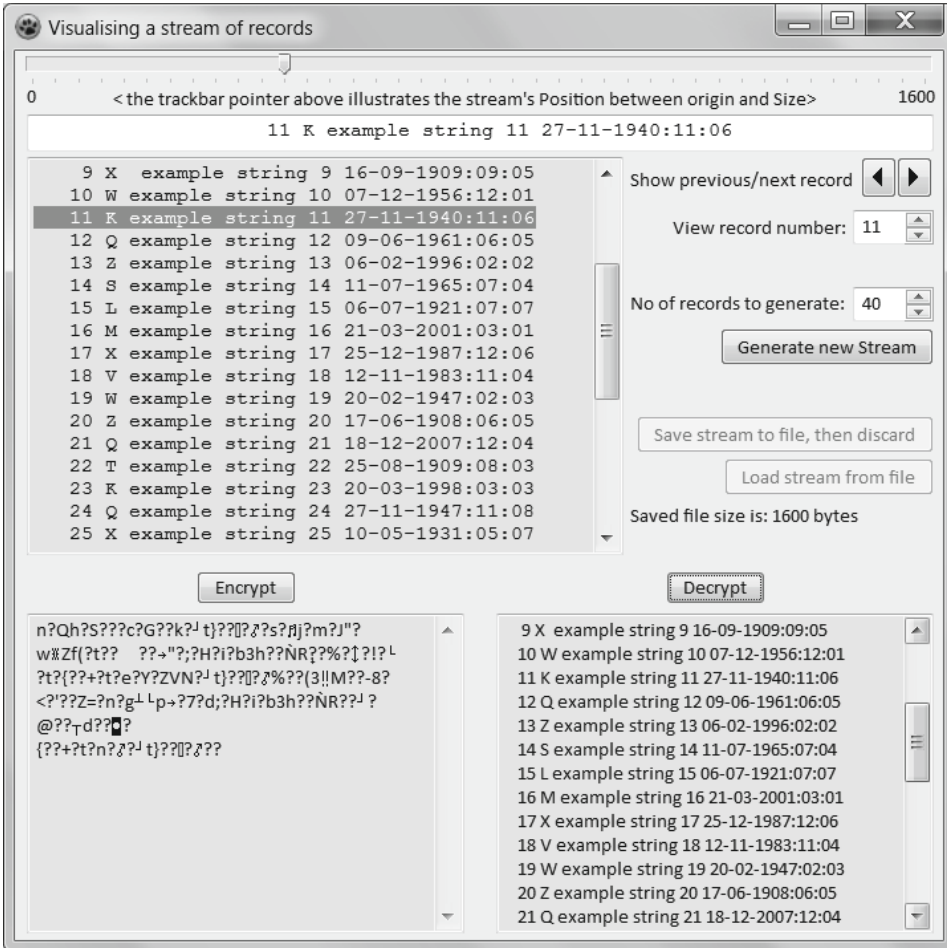


Figure 15.4 The stream_visualise project displaying a short stream

```

procedure TForm1.udMoverClick(Sender: TObject; Button: TUDBtnType);
begin
  if not Assigned(ds) then Exit;
  case Button of
    btPrev: if (ds.Position >= SizeOf(ds.rec)) then
      begin
        ds.Seek(- 2*SizeOf(ds.rec), soFromCurrent);
        if (ds.Position < 0) then ds.Position:= 0;
        edtRecord.Text:= ds.RecAsString;
        seCurrentRecNo.Value:= seCurrentRecNo.Value-1;
      end;
    btNext: if (ds.Position < ds.Size) then
      begin
        edtRecord.Text:= ds.RecAsString;
        seCurrentRecNo.Value:= seCurrentRecNo.Value+1;
      end;
  end;
  seCurrentRecNoChange(nil);
end;

```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

Below the TUpDown drop a label with View record number: as its Caption, and beside it drop a spinedit named seCurrentRecNo with its Value set to 30.

Below these two components drop a further label with No. of records to generate: as its Caption, and beside it a spinedit named seNoOfRecords, with its MaxValue set to 10000, its MinValue to 1 and its Value to 40.

Below these place three buttons all with AutoSize set to True.

- Firstly btnGenerateStream with Caption set to Generate new Stream.
- Secondly btnSaveStream with Caption set to Save stream to file, then discard and Enabled set to False.
- Thirdly btnLoadStream with Caption set to Load stream from file with Enabled set to False.

Below the buttons place a further label named lblSavedFileSize with an empty Caption.

Lastly place two memos at the bottom of the form named memoEncrypted and memoDecrypted and delete all text from these memos. Drop a button above memoEncrypted named btnEncrypt with its Caption set to Encrypt, and drop a final button above memoDecrypted named btnDecrypt with its Caption set to Decrypt.

The required supporting code now has to be added. Add Blowfish to the main form's uses clause, then add four fields to the main form class's private section, and a public procedure as follows:

```
...
private
    ds: TdemoStream;
    sOriginal, sEncrypted: TStringStream;
    key: string;
public
    procedure DisplayStream;
end;
```

We declare a variable to point to an instance of TdemoStream, two TStringStream instance variables, and a string which will be the Blowfish encryption/decryption key.

Next generate OnClick handlers for btnEncrypt and btnDecrypt, and a body for the DisplayStream procedure as follows:

```
procedure TForm1.btnEncryptClick(Sender: TObject);
var be: TBlowFishEncryptStream;
begin
    if memoDisplay.Lines.Count = 0 then Exit;
    key:= 'example cipher key for Blowfish';
    sOriginal:= TStringStream.Create(EmptyStr);
    be:= TBlowFishEncryptStream.Create(key, sOriginal);
    be.WriteAnsiString(memoDisplay.Text);
    be.Free;
    memoEncrypted.Text:= sOriginal.DataString;
end;
```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

```
procedure TForm1.btnDecryptClick(Sender: TObject);  
var bd: TBlowFishDeCryptStream;  
begin  
    sEncrypted:= TStringStream.Create(sOriginal.DataString);  
    bd:= TBlowFishDeCryptStream.Create(key, sEncrypted);  
    memoDecrypted.Text:= bd.ReadAnsiString;  
    bd.Free;  
    sEncrypted.Free;  
    sOriginal.Free;  
end;  
  
procedure TForm1.DisplayStream;  
begin  
    lblStreamSize.Caption:= Format('%d ',[ds.Size]);  
    tbPosition.Max:= ds.Size div SizeOf(ds.rec) - 1;  
  
    memoDisplay.Lines.BeginUpdate;  
    memoDisplay.Clear;  
    ds.Position:= 0;  
    while (ds.Position < ds.Size) do  
    begin  
        memoDisplay.Lines.Add(ds.RecAsString);  
        tbPosition.Position:= (ds.Size * seNoOfRecords.Value) div ds.Position;  
    end;  
    memoDisplay.Lines.EndUpdate;  
  
    seCurrentRecNo.MaxValue:= seNoOfRecords.Value;  
    ds.Seek(seCurrentRecNo.Value * SizeOf(ds.rec), soFromBeginning);  
    edtRecord.Text:= ds.RecAsString;  
    tbPosition.Position:= (ds.Size * tbPosition.Max) div ds.Position;  
end;
```

Now we need to create `OnClick` handlers for the three buttons in the centre of the UI, which generate the demonstration memory stream, save it to file and load it again from a saved file. These three handlers are as follows:

```
procedure TForm1.btnGenerateStreamClick(Sender: TObject);  
begin  
    ds.Free;  
    ds:= TdemoStream.Create(seNoOfRecords.Value);  
    seCurrentRecNo.MaxValue:= seNoOfRecords.Value-1;  
    DisplayStream;  
    seCurrentRecNoChange(nil);  
    btnSaveStream.Enabled:= True;  
    lblSavedFileSize.Caption:= EmptyStr;  
    memoDecrypted.Lines.Clear;  
    memoEncrypted.Lines.Clear;  
end;  
  
procedure TForm1.btnLoadStreamClick(Sender: TObject);  
begin  
    FreeAndNil(ds);  
    ds := TdemoStream.Create(0);  
    ds.LoadFromFile(savedFilename);  
    seNoOfRecords.Value:= ds.Size div SizeOf(ds.rec);  
    seCurrentRecNo.MaxValue:= seNoOfRecords.Value-1;  
    DisplayStream;  
    btnLoadStream.Enabled:= False;  
    seCurrentRecNoChange(nil);  
    memoDecrypted.Lines.Clear;  
    memoEncrypted.Lines.Clear;  
end;
```

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

```
procedure TForm1.btnSaveStreamClick(Sender: TObject);
begin
  if Assigned(ds) then
  begin
    memoDisplay.Lines.Clear;
    edtRecord.Text:= EmptyStr;
    ds.SaveToFile(savedFilename);
    lblSavedFileSize.Caption:= Format('Saved file size is: %d bytes',
                                     [FileSize(savedFilename)]);

    btnLoadStream.Enabled:= True;
    btnSaveStream.Enabled:= False;
    FreeAndNil(ds);
  end;
end;
```

Lastly we need to set the constant `savedFilename`, ensure that the `TdemoStream` is freed at the end of the program life, and write an `OnChange` handler for the spinedit `seCurrentRecNo` to synchronise changes there with scrolling in `memoDisplay` and movement of the trackbar thumb. Before the form's `type` declarations add a `const` declaration thus:

```
const savedFilename: TFilename = 'stream.dat';
```

Double-click beside the form's `OnDestroy` event and add this implementation:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  ds.Free;
end;
```

Lastly generate a `seCurrentRecNo OnChange` event and complete it as follows:

```
procedure TForm1.seCurrentRecNoChange(Sender: TObject);
var L: integer;
begin
  if not Assigned(ds) then Exit;
  ds.Seek(seCurrentRecNo.Value*SizeOf(ds.rec), soFromBeginning);
  edtRecord.Text:= ds.RecAsString;
  L := Length(edtRecord.Text) + Length(LineEnding);
  tbPosition.Position:= tbPosition.Max*ds.Position div ds.Size;
  memoDisplay.SelStart:= (tbPosition.Position)*L;
  memoDisplay.SelLength:= L;
end;
```

If all is typed correctly you should be able to compile and run the application and explore its features. The memo displaying the encrypted text should be taken with a pinch of salt – although the encrypted bytes are all correctly saved in its `Lines` property, they cannot be displayed correctly since some of the byte values stored there are not viewable characters.

Note how in setting the highlighting in `memoDisplay` we have to take account of the unseen line ending character(s) at the end of each line by adding `Length(LineEnding)` to the length of each displayed string. Use of the predefined `LineEnding` constant ensures cross-platform consistency.

Chapter 15 NON-VISUAL GUI SUPPORT CLASSES

15.i Review Exercises

1. Adapt the `chemCollection` project. Try altering `TElementItem` by adding another field (or deleting one) and see how the `TTIGrid` adapts to the change.
You will need to alter the `AddElement()` routine accordingly, and also the `AddElements` procedure.
2. What are the principal RTL classes required for building the LCL, and (after `TObject`) which is the class all participants in the RTTI system must descend from?
3. Design a `TPlanetItem` class that would form the basis for a grid display in a `TTIGrid` of planetary information.
Then try implementing it to give you a working program.
To check that it is truly flexible at runtime, add an extra property – the planet name in a language that is not your native language.
Adapt your program to provide this extra information.
Does the `TTIGrid` adapt itself at runtime as well?
4. The `stream_visualise` project could be extended in a number of ways.
How would you alter the project so that dragging the trackbar pointer caused movement through the records in the stream, and so would update the record display in the memo, current record number and text fields? (Hint: look at the available events for `TTrackBar`).
5. Having seen how to utilise a Blowfish stream in the above code, design and implement an application that uses the analogous `TCompressionStream` and `TDecompressionStream` classes to load a text file into a `TStringList` and save the compressed data to a file with a different extension (you'll need to add `uses zstream;` to your application).



Chapter 16 FILES AND ERRORS

Every GUI program you write will deal with data, and data demands to be **stored**. Data that is truly transient and temporary is of little long-term interest. Important data that needs to be referred to more than once needs to be stored somewhere accessible. The computer's memory (RAM) is OK for storage during the running of your program, but thereafter whatever data was held in memory is lost. If not saved in some persistent medium your data will be lost for ever, unless it can be recreated.

All the data describing the size, colour, text fonts and so on of the widgets used in your program are stored as resources within the executable **file**. The main purpose of files is to provide a means by which data can persist over time. Lazarus has its own internal mechanism for reading and storing the component data your program needs, and for making sure your executable program holds that unvarying data securely so it can be accessed every time your program runs. This ensures that the look and feel of your GUI is identical each time. The executable file itself holds this data.

An executable file, however, is not a suitable place to store other kinds of data, data that varies over time. Indeed modern OSs prevent you writing to executable files or storing anything in them that differs from their original contents. This is one ploy in the ever more complex battle against viruses and malware. To a large degree, then, your program's UI is fixed at compilation time. An upgrade to a later version will overwrite the original executable (*replacing it*), but user settings and configuration of the program's functionality and appearance have to be stored **outside** the executable in some other file.

16.a File access in Pascal

There are two possible approaches to file access in Pascal. The classic approach (*incorporated into the language by Wirth before the GUI era*) uses a number of independent routines, and relies on **run-time errors** for error handling. Run-time errors can occur for manifold reasons (*such as an invalid pointer, or stack memory overflow*), but for beginners it is usually file-related run-time errors that are more commonly encountered.

Pascal provides two basic file types: `file` and `TextFile`. Text file routines can access data from the console, and write to the console as well as to disk files, and so are a natural choice for writing command-line programs. Files of this sort were considered briefly in two Sections at the close of Chapter 6. Most programs use the `SysUtils` unit whether for file-related routines (such as `FileOpen`, `DirectoryExists`) or for type conversions such as turning a string into a `TDateTime` or encoding three word values (`year`, `month`, `day`) into a `TDateTime`.

The `SysUtils` unit converts all run-time errors into exceptions, about which there is more detail in the following Section.

More recently an object oriented approach to file handling has been added to the Pascal RTL.

It is based on descendants of the `TStream` class (*a class which offers aptly-named methods and properties*) which relies on exceptions for error handling. This is a more versatile file-type scheme embracing more than simply disk files, and commonly used `TStream` descendants include `TFileStream`, `TMemoryStream`, `TStringStream` and `TResourceStream` (*and there are several others declared in the libraries; and you can of course write your own*). Object oriented file access of this sort was introduced in the last few Sections of the previous chapter.

16.b Run-time errors and exceptions

Oscar Wilde wrote that experience is the name people give to their mistakes, and you probably know that "to err is human, but to really foul things up takes a computer!" Dealing well with errors that arise in the course of running software must be a consideration for any programmer. Run-time errors are the result of routines that fail. Code that can potentially trigger an error must take account of all possible eventualities. If programmed for run-time error handling, library routines return a predefined value if the routine fails to execute.

Chapter 16 FILES AND ERRORS

Frequently, however, run-time errors are automatically converted into exceptions. Exceptions let you interrupt a program's normal flow of control. When an exception occurs your application is notified. The compiler generates code that knows how to interrupt your code's processing, and deal with locally allocated memory on the stack (*your error-handling routine must specifically free any heap-based memory allocations*), and it also generates an error instance which stays in memory until it is freed. The call stack begins to unwind from the point of the interruption. The unwinding process continues up the call stack tree until either a **try** . . . **except** construct is encountered, or a global exception handler is encountered. Freeing the error instance is called "handling the exception".

Possible causes of error are manifold. A poorly coded routine might corrupt memory, or access an unavailable device, or some resource limit may be exceeded or an entity may be absent ('*out of memory*', '*disk full*', '*file does not exist*' etc.). Exceptions can also arise from database drivers, network .dlls or shared objects, or the OS itself.

Any exceptions you don't specifically handle yourself get passed to the Application to handle. If you want to intervene at that level, one way to do so is to drop a `TApplicationProperties` component on your main form, and write a handler for its `OnException` event.

To illustrate the two approaches to error handling (*run-time errors and exceptions*) consider the simple case in which you code an `OverwriteCharAtPos()` function which replaces the character at a specified position in a string. You might program to defend against errors in the following way.

16.c An example of string error-handling

Start a new Lazarus project called `string_error` with a main form called `error_form.pas`.

Set the form's `Caption` to `String Error Example` and drop on the form

- a labelled edit named `edtString` with its `EditLabel.Caption` set to `String for method to call`, its `LabelPosition` set to `lpLeft` and its `Text` set to `example`
- a spinedit named `edtPosition` with its `Value` set to `8`
- a label named `lblPosition` in front of the spinedit with its `Caption` set to `Position of Character to overwrite:`
- a label named `lblResultString` with its `Font.Style` set to `[fsBold]`
- a button named `btnCallFunctionRTE`, with its `Caption` set to `Call Function (Run-time Error)`, and its `AutoSize` property set to `True`
- a button named `btnCallProcedureExcept`, with its `Caption` set to `Call Procedure (Exception)`, and its `AutoSize` property set to `True`

The resulting form in the Designer will look something like Figure 16.1.

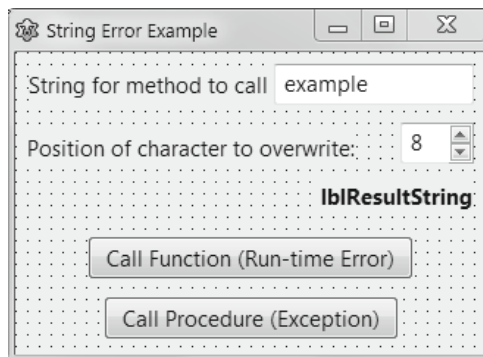


Figure 16.1 The `string_error` project in the Designer

Chapter 16 FILES AND ERRORS

Add two method declarations to your form's **private** section:

```
function OverwriteCharAtPosRTE(aChar: Char; var aString: string;  
                               aPos: integer): boolean;  
procedure OverwriteCharAtPosExcept(aChar: Char; var aString: string;  
                                   aPos: integer);
```

Generate implementation skeletons for these methods and complete them as follows:

```
function TForm1.OverwriteCharAtPosRTE(aChar: Char; var aString: string;  
                                       aPos: integer): boolean;  
  
var len: integer;  
begin  
    len := Length(aString);  
    Result := (len <> 0) and (aPos > 0) and (aPos <= len);  
    if Result then aString[aPos] := aChar;  
end;
```

```
procedure TForm1.OverwriteCharAtPosExcept(aChar: Char; var aString: string;  
                                          aPos: integer);  
  
var len: integer;  
begin  
    len := Length(aString);  
    if (len <> 0) and (aPos > 0) and (aPos <= len)  
        then aString[aPos] := aChar  
        else raise Exception.CreateFmt(  
            'Index (%d) out of range error in string "%s"', [aPos, aString]);  
end;
```

Double-click btnCallFunctionRTE's `OnClick` event to generate an event handler and complete it as follows:

```
procedure TForm1.btnCallFunctionRTEClick(Sender: TObject);  
var s: string;  
begin  
    s := edtString.Text;  
    if OverwriteCharAtPosRTE('Z', s, edtPosition.Value)  
        then lblResultString.Caption := s  
        else RunError(999);  
end;
```

Double-click btnCallProcedureExcept's `OnClick` event to generate an event handler and complete it as follows:

```
procedure TForm1.btnCallProcedureExceptClick(Sender: TObject);  
var s: string;  
begin  
    s := edtString.Text;  
    OverwriteCharAtPosExcept('Z', s, edtPosition.Value);  
    lblResultString.Caption := s;  
end;
```

Chapter 16 FILES AND ERRORS

Build this program (**Run | Build** or [Shift][F9]). Run the executable using your OS's file explorer (*i.e. not using the IDE, which runs it under the debugger, unless you prefer to turn the debugger off and use the IDE to run it without the debugger*). You can cause a run-time error or exception in several ways. Experiment with different values for the string and the character position. Catching the error using the run-time error method will pop up one or perhaps two dialogs similar to Figure 16.2, and close the application. Precisely how run-time errors are displayed is rather OS dependent, and also depends in Windows on whether you have set the application build as *{\$apptypc console}* so you see a 'DOS box' window as well as your GUI windowed application. The intervention of the debugger (*and Project Options settings for debugging and display of line number information*) also affect the output.

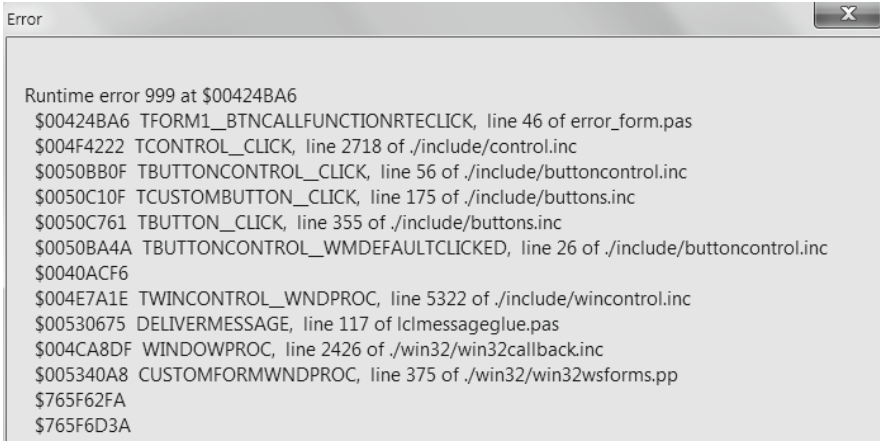


Figure 16.2 Part of a run-time error dialog presented when the string_error project bombs

Catching errors using an exception is usually the preferred way to handle errors, giving a dialog similar to Figure 16.3. This lets you recover from the error by pressing [OK], or close the application by pressing [Cancel], which is far more user-friendly.

Exceptions also allow you to separate normal program logic from error-handling more effectively. Since you can create exception objects which can have as much data as you wish, the error message displayed when the exception is handled can be far more informative for users, who are not interested at all in the hexadecimal stack addresses and line number information of the run-time error message. While they may not be interested in the *Index (8) out of range error in string "example"* message shown below, it will be far more useful when relayed to you the programmer, to understand what has gone wrong. It would be simple to adapt the message to include, say, the name of the failing routine.

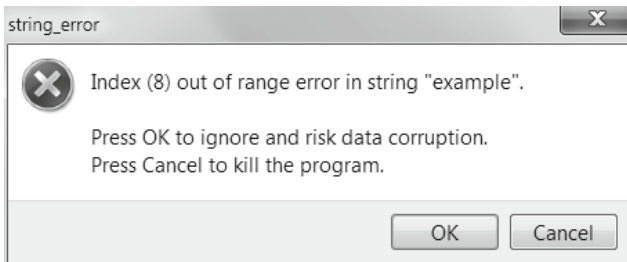


Figure 16.3 string_error dialog showing an exception, allowing you either to recover from it or exit

Chapter 16 FILES AND ERRORS

The short excursion into error handling given above is needed because dealing with files is the first major area encountered so far where robust error-handling is essential. You don't want your program to crash simply because a user types in an invalid file name, or attempts to write to a read-only CD-ROM drive.

16.d File name encoding issues

Internally Lazarus works with the UTF8 (*Unicode*) character set (*both in LCL code, and in the IDE's Editor*), and some OSs (*notably Windows*) do not. The FPC RTL routines use system encoding for file routines (*not UTF8*). Consequently we need to use encoding conversion routines in Lazarus when using traditional Pascal file routines (*which are all implemented internally in the compiler or in the RTL*).

Suppose you have a text file where you list useful software shortcut keys, called `Abkürzung.txt`. To open this text file named `AbkText` in your program and read it into a memo component we have to write:

```
procedure TForm1.LoadShortcutFileIntoMemo1;
var AbkText: TextFile;
    fName: string = 'Abkürzung.txt';
    s: string;
begin
  AssignFile(AbkText, UTF8ToSys(fName));
  try
    Reset(AbkText);
    while not Eof(AbkText) do
      begin
        readln(AbkText, s);
        Memo1.Lines.Add(s);
      end;
  except
    ShowMessageFmt('Text file %s not found', [fName]);
  end;
end;
```

If we fail to include the encoding conversion function `UTF8ToSys` which wraps `fName` in the above code, the application will not find the file since its name contains an umlaut, and the call to `Reset` will fail, triggering an exception. Actually it is not a good idea to hard-code file names in a routine like this, but this is given to illustrate the principal point about needing correct encoding of system names.

Note here the simple syntax required to catch exceptions that a section of code might generate.

Just as we created an exception for an error condition earlier with the line

```
raise Exception.CreateFmt(' . . . ', [ . . . ]);
```

so we catch an exception, should it arise, with a `try ... except ... end;` code block.

Actually we can avoid using an explicit `while` loop to read individual lines from the file altogether. The `TMemo.Lines` property (*a TStrings descendant*) has its own `LoadFromFile` method. A simple call,

```
Memo1.LoadFromFile(UTF8ToSys(fName));
```

is all that is needed.

Chapter 16 FILES AND ERRORS

16.e User-directed file searching and naming – the Dialogs Palette page

It was pointed out above that hard-coding file names is generally poor programming style. Lazarus provides a number of useful file dialogs on the *Dialogs Palette* page which wrap system dialog controls from each platform's widgetset to give simple platform-independent ways of opening and saving files.

The various dialogs share commonly named properties, of which the two most important are `DefaultExt` and `FileName`, and a boolean method called `Execute` which shows the dialog, and if the dialog is not cancelled returns `True`, with the chosen file in the `FileName` property.

`FileName` can be pre-assigned to a default value. `TOpenDialog` and `TSaveDialog` are complemented by `TOpenPictureDialog` and `TSavePictureDialog` which are more specialised file pickers and savers designed for graphic-format files. Their `Filter` property is pre-populated with a generous selection of common graphic file formats Lazarus supports, and they also provide (*optional*) picture previews.

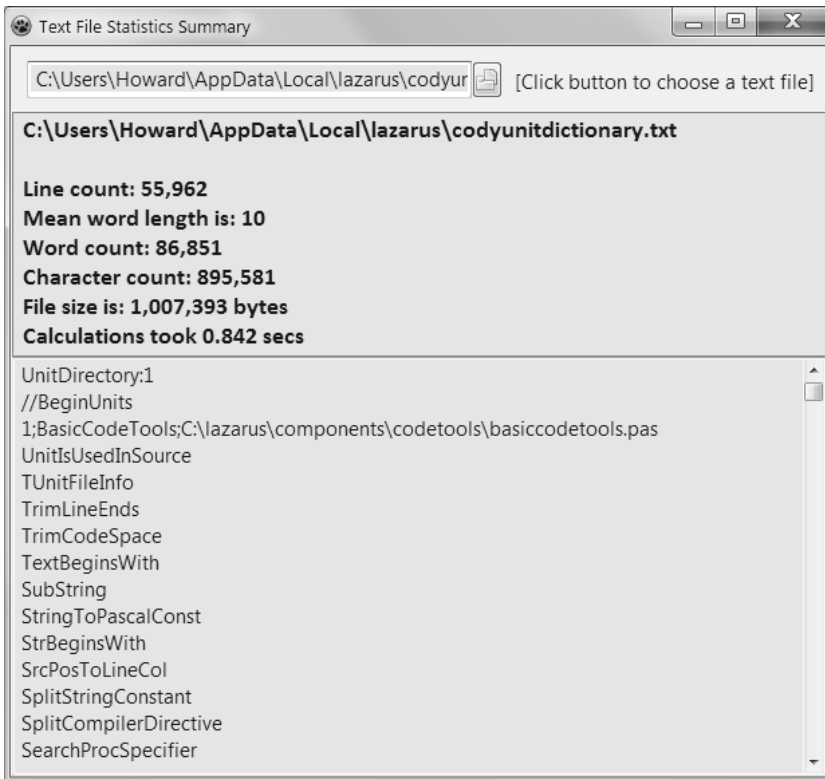


Figure 16.4 The `text_stats` project analysing a large text file

Using the file dialog components makes it quite straightforward to develop a text file statistics application that analyses the contents of a text file. Start a new Lazarus project named `text_stats` with a main form named `stats_main`. Set the form's `Caption` to `Text File Statistics Summary`, its `Height` to 540 and its `Width` to 610.

Chapter 16 FILES AND ERRORS

Rather than use the file dialogs from the *Dialogs* page, we will use the `TFileNameEdit` component from the *Misc* Palette page, which conveniently combines a file open/save dialog with a button to invoke it. The completed text file statistics tools will look like Figure 16.4 when complete. Drop a `TFileNameEdit` at the top of the form, naming it `fnEdit`, setting its `Width` to 330 and its `DefaultExt` to `txt`. Click on the ellipsis [...] button beside the `Filter` property and use the Filter Editor to add three lines of file selections as shown in Figure 16.5, using semicolons to separate multiple filters entered on a single line. It is safer not to allow the third line (`*.*`) – so you can choose whether you add it or not.

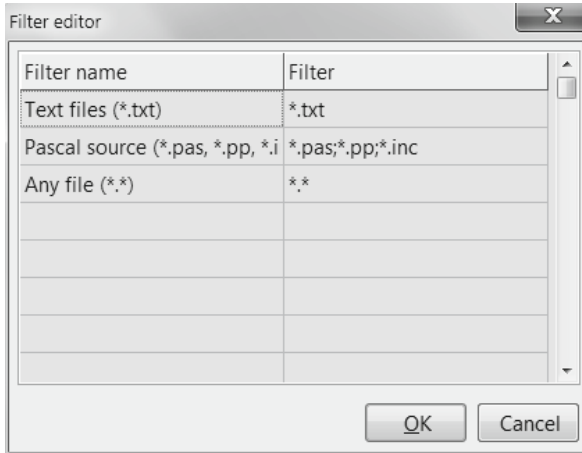


Figure 16.5 The file dialog Filter Editor

Drop a `TLabel` beside the file-edit, named `lblDirections`, setting its `Caption` to [Click on the button to open a text file]. Drop a `TMemo` on the form named `memoPreview`, setting its `Align` to `alBottom`, its `Height` to 310, its `scrollbars` to `ssAutoBoth` and deleting its name from the `Lines` property.

Drop a `TListBox` named `lbStats` on the form, and set its `Align` to `alBottom`. Drag its top border upwards so your form looks similar to Figure 16.4. Add `strutils` to the form's `uses` clause.

Delete the `TForm1`'s `public` section, and in the `private` section add the following items:

```
private
  FFileName: string;
  function CharCount(const aText: string): integer;
  procedure ComputeStats;
  procedure DisplayFileInfo;
  function WordCount(const aText: string): integer;
  function ParseToWords(const aString: string; separatorsSet: TSysCharSet):
TStringList;
  function IsTextFile(aFileName: string): boolean;
```

Select `fnEdit` in the Designer (or *OI treeview*) and double-click on the *OI Events* page beside the `OnAcceptFileName` event. Lazarus will generate an implementation skeleton for this event, and also generate skeletons for all the other methods you have just declared. Complete the `OnAcceptFileName` event handler as follows:

Chapter 16 FILES AND ERRORS

```
procedure TForm1.fnEditAcceptFileName(Sender: TObject; var Value: String);
begin
  FFileName:= UTF8ToSys(Value);
  if not IsTextFile(FFileName)
then case QuestionDlg('Caution!',
                      Value+' does not look like a text file'+sLineBreak+
                      'What do you want to do?',
                      mtWarning,
                      [mrYes,'Open file anyway',
                      mrNo,'Look for a different file', mrCancel], 0) of
    mrNo: Exit;
    mrYes: DisplayFileInfo;
  end // case QuestionDlg
  else DisplayFileInfo;
end;
```

Although we have set the file filter to default to a .txt extension, it is possible that files with .txt extensions are not text files. So we will write a (*very simple-minded*) `IsTextFile()` function to test whether a file is text or binary (*even though it is not possible to write a function that always makes this distinction correctly*).

Our function will test for ASCII characters, BOMs for unicode encodings and xml files, and also reject files with successive repeated zeros (*which are almost certainly binary*). This is not very thorough or reliable, but perhaps will cover 95% of cases.

If the 'text' file fails this test, we give the user the option to open and read the file anyway, by using the `QuestionDlg()` function from the `dialogs` unit. This is a very useful and versatile function for eliciting Yes/No/Cancel sort of answers, with eminently configurable parameters which let you specify which buttons you want the dialog to carry, and what the button captions should be.

Modal dialogs return a `TModalResult` value which tells you which button the user clicked. We test this with a case statement to determine whether to `Exit` this procedure, or call the `DisplayInfo` procedure.

Chapter 16 FILES AND ERRORS

16.f Discriminating between text and binary files

Complete the code for the `IsTextFile()` function (which has three nested functions) as follows:

```
function TForm1.IsTextFile(aFileName: string): boolean;
const BufSize = 1024;
var inf: file;
    buffer: array[1..BufSize] of byte;
    numRead: int64;

function IsUTF8OrXML: boolean;
begin
    Result := False;
    if buffer[1] = $3C then Result:= True;
    if (buffer[1] = $00) and (buffer[2] = $3C) then Result := True;
    if (buffer[1] = $FF) and (buffer[2] = $FE) then Result := True;
    if (buffer[1] = $FE) and (buffer[2] = $FF) then Result := True;
    if (buffer[1] = $EF) and (buffer[2] = $BB) and (buffer[3] = $BF)
    then Result := True;
end;

function IsASCIIfile: boolean;
var index: integer;
begin
    Result := True;
    for index := 1 to numRead do if Result then case buffer[index] of
        9, 10, 12, 13: ;
        32..126: ;
        else Result := False;
        end
    else Break;
end;

function IsBinary: boolean;
var zeroCount: integer = 0; currZero: boolean; index: integer;
begin
    Result := False;
    if (buffer[1] = Ord('M')) and (buffer[2] = Ord('Z')) then Exit(True);
    for index := 1 to numread do
        begin
            currZero:= buffer[index] = 0;
            case currZero of
                False: zeroCount:= 0;
                True: inc(zeroCount);
            end;
            if (zeroCount >= 4) then
                begin
                    Result := True;
                    Break;
                end;
        end;
    end;
end;

begin
    Filemode := 0;
    AssignFile(inf, aFileName);
    try
        Reset(inf, 1);
    except
        begin ShowMessageFmt('File %s could not be opened',[aFileName]);
            Result := False;
            Exit;
        end;
    end;
    BlockRead(inf, buffer, SizeOf(buffer), numRead);
    CloseFile(inf);
    result := IsASCIIfile or IsUTF8OrXML and not IsBinary;
end;
```

Chapter 16 FILES AND ERRORS

This function would not pass muster in a commercial application, but suffices for our purposes here, examining only the first few bytes of the file (*or up to 1024 bytes for ASCII files*). The code for `DisplayFileInfo` is somewhat shorter:

```
procedure TForm1.DisplayFileInfo;  
begin  
    memoPreview.Lines.Clear;  
    memoPreview.Lines.LoadFromFile (FFileName);  
    ComputeStats;  
end;
```

Here is the code for `ComputeStats`:

```
procedure TForm1.ComputeStats;  
var c, w, m: integer;  
    start: TDateTime;  
begin  
    if (FFileName = EmptyStr) then Exit;  
    start := Now;  
lbStats.Items.Clear;  
lbStats.Items.Add(' ' + SysToUTF8(FFileName));  
lbStats.Items.Add(EmptyStr);  
c := CharCount(memoPreview.Text);  
w := WordCount(memoPreview.Text);  
if (w = 0) then m := 0  
    else m := c div w;  
lbStats.Items.Add(' Line count: '+FormatFloat('#',  
                                                    memoPreview.Lines.Count));  
lbStats.Items.Add(Format(' Mean word length is: %d', [m]));  
lbStats.Items.Add(' Word count: ' + FormatFloat('#', w));  
lbStats.Items.Add(' Character count: ' + FormatFloat('#', c));  
lbStats.Items.Add(' File size is: ' +  
                    FormatFloat('#', FileSize(FFileName))+ ' bytes');  
lbStats.Items.Add(Format(' Calculations took %s secs',
```


Chapter 16 FILES AND ERRORS

We first check that there is an actual file to report on, if so noting the current time, and clearing any previous file statistics. The lines which follow collect and format the information, calling custom `WordCount()` and `CharCount()` functions in the process. `WordCount()` in turn calls `ParseToWords()`, a routine developed for Chapter 18:

```
function TForm1.WordCount(const aText: string): integer;
var sl: TStringList;
begin
  Result := 0;
  if aText = EmptyStr then Exit;
  try
    sl := ParseToWords(aText, []);
    Result := sl.Count;
  finally
    sl.Free;
  end;
end;

function TForm1.ParseToWords(const aString: string;
                             separatorsSet: TSysCharSet): TStringList;
var
  buildingAWord: boolean = False;
  s: string = '';
  c: Char;
begin
  if (Length(aString) = 0)
  then begin
    Result := nil;
    Exit;
  end;

  if (separatorsSet = [])
  then separatorsSet:= WordDelimiters;

result := TStringList.Create;
  for c in aString do
    case (c in separatorsSet) of
      False: begin
        if not buildingAWord
        then buildingAWord:= True;
          s := s + c;
        end;
      True: begin
        if buildingAWord
        then begin
          buildingAWord:= False;
          Result.Add(s);
          s := EmptyStr;
        end;
      end;
    end; //for-case
  if (s <> EmptyStr)
  then Result.Add(s);
end;
```

Chapter 16 FILES AND ERRORS

Lastly, type the simpler CharCount function:

```
function TForm1.CharCount(const aText: string): integer;  
  
var c: Char;  
begin  
    Result := 0;  
    for c in aText do  
        if (Ord(c) > 32) then Inc(Result);  
    end;
```

Select the `lbStats` listbox, and in the OI set its `Font` to a clear, readable typeface in bold. Compile and run the program, testing various text files. If you try to open a non-text file, notice how the customisable `QuestionDlg()` function presents a useful dialog with very little code required (see Figure 16.6), considering how many options are available for constructing the dialog.

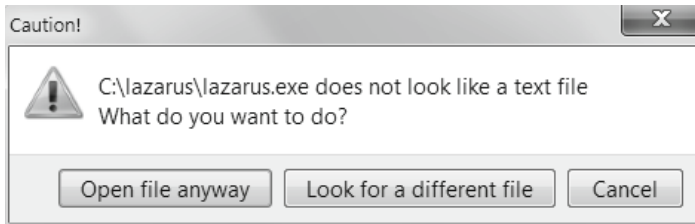


Figure 16.6 The customisable `QuestionDlg` function in action



Chapter 17 WORKING WITHIN KNOWN LIMITS

Pascal is a strongly typed language which means that every data holder (*or variable*) you use has to be specified in advance, and its size (*or potential size*) known in advance. This enforces a certain size-awareness at the design stage as you begin to write code, but does not, of course, prevent you from inadvertently exceeding the size limits of the data containers you choose when your program is running.

What happens if you pour a programming quart into a pint pot? Do the excess bytes disappear gracefully, leaving the remaining properly contained bytes undisturbed? Does your user notice anything amiss? A lot depends on how severe the 'spillage' is, and also whether you have anticipated the potential for such a problem to arise, and provided for the possibility that a variable may prove too small to hold the data being forced upon it. We'll look at one scenario based on use of a recursive routine (*a routine that calls itself*).

17.a Using recursion to evaluate factorials

The factorial function, written $N!$ is defined as the product of all non-negative integers less than or equal to N (*here "non-negative integer" is taken to include zero, since some mathematicians do not accept that inclusive meaning for the term "positive"*). The values of $0!$ and $1!$ are both defined as 1. So $4! = 4 \times 3 \times 2 \times 1 = 24$.

Most Pascal tutorials include code for a factorial function since it is one of the simplest and most straightforward illustrations of the use of recursion, in which a routine calls itself repeatedly. Successful recursion depends on making sure that the repeated calling of the routine stops at some point, so the recursion does not become infinite. An infinite loop (*or endless loop, the phrases have the same meaning*) is very easy to accomplish in programming, and completely intractable. There is no way to break out of an infinite loop. All you can do is kill the running program (*or turn the computer off*). Clearly programmers aim to avoid this situation. The value of $N!$ increases very quickly with N and $12!$ is the largest value that can be stored in a 32-bit integer, and $20!$ is the largest value that can be stored in a 64-bit integer. Let's use a 64-bit factorial function, that will be part of a project called `factorial` (*see Figure 17.1*).

Start a new Lazarus project you name `factorial`, which has a main form named `main_factorial.pas`. Set the main form's Caption to `Factorial example`, and add a new method in the form's `private` section:

```
private
    function Factorial64(aByte: byte): int64;
```

Generate a skeleton for this method and complete it as follows:

```
function TForm1.Factorial64(aByte: byte): int64;
begin
    case aByte of
        0, 1: Result:= 1;
        else Result:= aByte * Factorial64(aByte - 1);
    end;
end;
```

For parameter values greater than 1 this function generates a result which is the product of the passed parameter multiplied by the function called with a parameter one less than the current parameter. This results in a succession of calls to `Factorial64()` with successively smaller parameters. Eventually the parameter in the recursive call will have dropped to 1, at which point the recursive calls stop, and the final multiple-product result is evaluated.

Chapter 17 WORKING WITHIN KNOWN LIMITS

To create a UI for testing this function, drop a button at the top of the form named `BListFactorials`, and set its `AutoSize` to `True`, its `Height` to 500, and its `Caption` to `List Factorials up to:`. Beside the button drop a spinedit named `EParameter` with its `Value` set to 21, and its `MaxValue` set to 30.

Below the button and edit drop a memo named `MDisplay` and set its `Align` to `alBottom`, and its `ScrollBars` to `ssAutoBoth`. Drag the top of the memo upwards so it fills most of the remaining height of the form. Double-click the button to generate an `OnClick` event handler and complete it as follows:

```
procedure TForm1.BListFactorialsClick(Sender: TObject);
var i: integer;
begin
    MDisplay.Lines.Clear;
    for i := 1 to EParameter.Value do
        Mdisplay.Lines.Add(Format('%d! is %s',
                                   [i, FormatFloat(',',#, Factorial64(i))]));
end;
```

Compile and run this program, and see what happens if you specify values for factorials above 20! – the program sails on without a murmur, but gives nonsense results for those higher values (see Figure 17.1), though if we were not alerted by the negative values we might not notice!

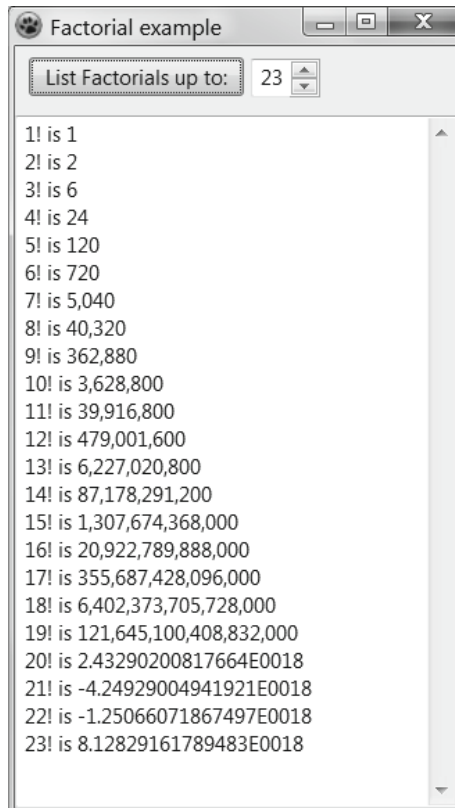


Figure 17.1 Output from the Factorial example project

Chapter 17 WORKING WITHIN KNOWN LIMITS

How can we prevent the erroneous values for 21!, 22! and higher from being displayed? There are several possible remedies. We could, for instance, limit the values accepted by the program (*knowing that 20! is the highest value that can be held in the int64, 64-bit integer*). The easiest way to do this would be to set `EParameter.MaxValue` to 20. Or we could use a higher-capacity type to store the factorial value (*say, `qword` or `extended`*), and discover what the new upper limit for the parameter became with a larger result type to store the factorial before it too overflowed. An alternative would be to use a `try ... except ... end` block to catch the overflow condition by trapping a specific exception. In this case we would be looking for an `EIntOverflow` exception. To do this we need both to adapt the code given above slightly, and also change the code generation settings to get the compiler to add a check for overflow.

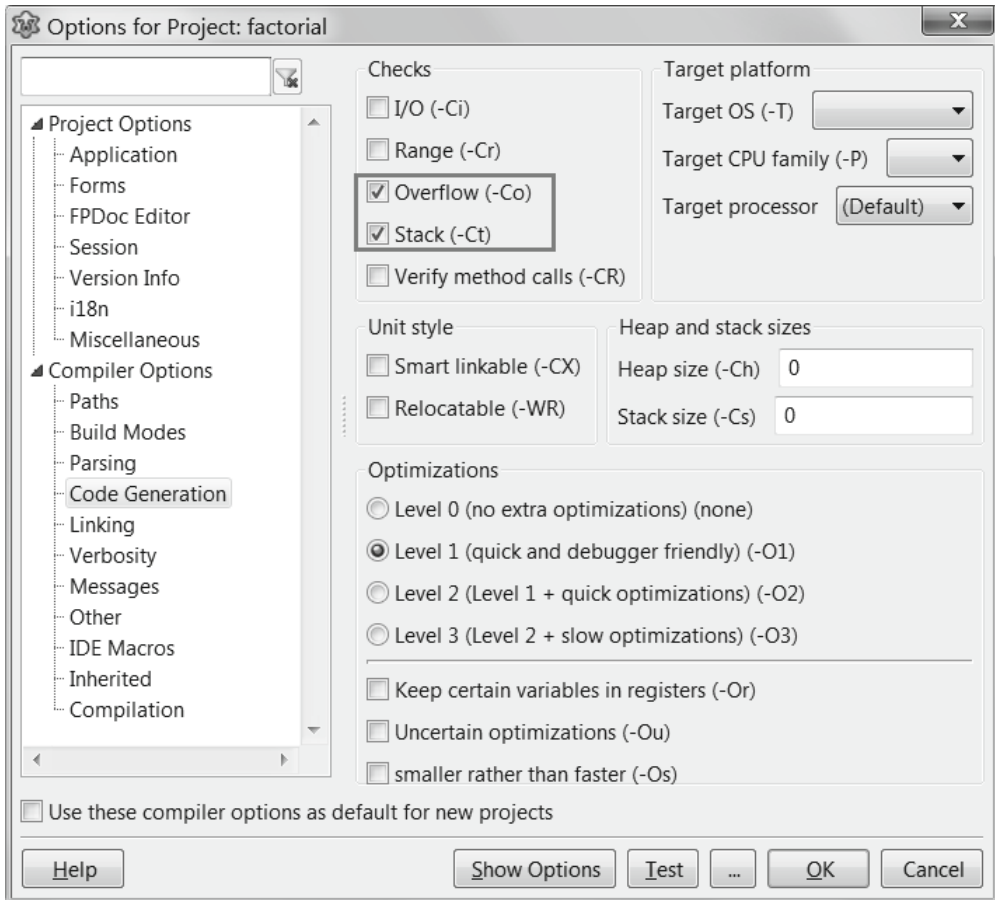


Figure 17.2 The Project Options Code Generation page with two relevant checks highlighted

To make the second alteration call up the Project Options dialog (either from the Project Inspector toolbar, or via **Project | Project Options...** or via `[Shift][Ctrl][F11]`). In the treewiew at the left of the dialog under the *Compiler Options* node click on Code Generation to open that page, and in the *Checks* groupbox make sure the *Overflow (-Co)* checkbox is checked. Since we are using a recursive routine it is also a good idea to enable the *Stack (-Ct)* checkbox as well.

Chapter 17 WORKING WITHIN KNOWN LIMITS

The stack is the memory area used for all subroutines, including recursive ones, which by their nature can use unusually large amounts of stack memory if they have very large local variables, or if the level of nested calls becomes very deep.

Modern OSs reserve large areas of memory by default for stack operations (*e. g. Windows 7 sets aside 16MB per thread, MacOS X 8MB per thread*) so that normal program usage will not cause stack overflow. Although Lazarus provides an option to set stack size on the Code Generation page, programmers will not normally need to tinker with this setting (*see Figure 17.2*).

17.b Catching a specific exception

Add a further button to the UI named `BFactorialException` with its `Caption` set to `List Factorials` using exceptions. Double-click this button to generate an `OnClick` event handler and complete it as follows (*it is identical to the first button's code except for a different function call*):

```
procedure TForm1.BFactorialExceptionClick(Sender: TObject);
var i: integer;
begin
  MDisplay.Lines.Clear;
  for i := 1 to EParameter.Value do
    MDisplay.Lines.Add(Format('%d! is %s',
                              [i, FormatFloat('#', FactorialExc(i))]));
end;
```

Now it remains to write the new `FactorialExc()` method. Declare it in the form's private section as:

```
function FactorialExc(aByte: byte): int64;
```

Use Code Completion to create a skeleton for this method, and complete it as follows:

```
function TForm1.FactorialExc(aByte: byte): int64;
begin
  Result := -1;
  case aByte of
    0, 1: Result := 1;
  else
    try
      Result := aByte * FactorialExc(aByte - 1);
    except on e: EIntOverflow do
      MessageDlg('Problem',
        Format('%s"%s" a parameter value of %d caused the above exception'+
              ' in the FactorialExc function',
              [e.Message, LineEnding, aByte]), mtWarning, [mbClose], 0);
    end;
  end;
end;
```

Here we wrap the nested call to `FactorialExc()` within a `try/except` block which checks for the specific `EIntOverflow` exception. If found it calls the `MessageDlg` function which has several parameters designed to let you construct a fairly sophisticated message dialog. Here we include a nested call to `Format()` to add the exception error message (`e.Message`) to our own message text. Note the syntax for catching a specific exception: `on ... do`.

The compiler takes care of constructing and freeing the exception instance. We can give it a name (*here "e"*) so we can reference it to display its `Message` property. In other situations exceptions are anonymous from the programmer's point of view, since only the compiler needs to be able to reference an exception that is created and freed almost straightaway.

The stack is 'unwound' to the point where the exception occurred. Initially setting `Result` to `-1` (*it is normally overwritten later in the `try` section of the function*) ensures that the displayed result will be negative if there is an exception. The negative value is to proclaim its invalidity (*in this case – in other situations some other error indication may have to be used*).

Chapter 17 WORKING WITHIN KNOWN LIMITS

You can avoid the debugger interposing intermediate error dialogs if you run this program outside the IDE (or turn off the debugger using *Tools | Options... | Debugger | General | Debugger type and path*).

The project so far will produce a screen something like Figure 17.3.

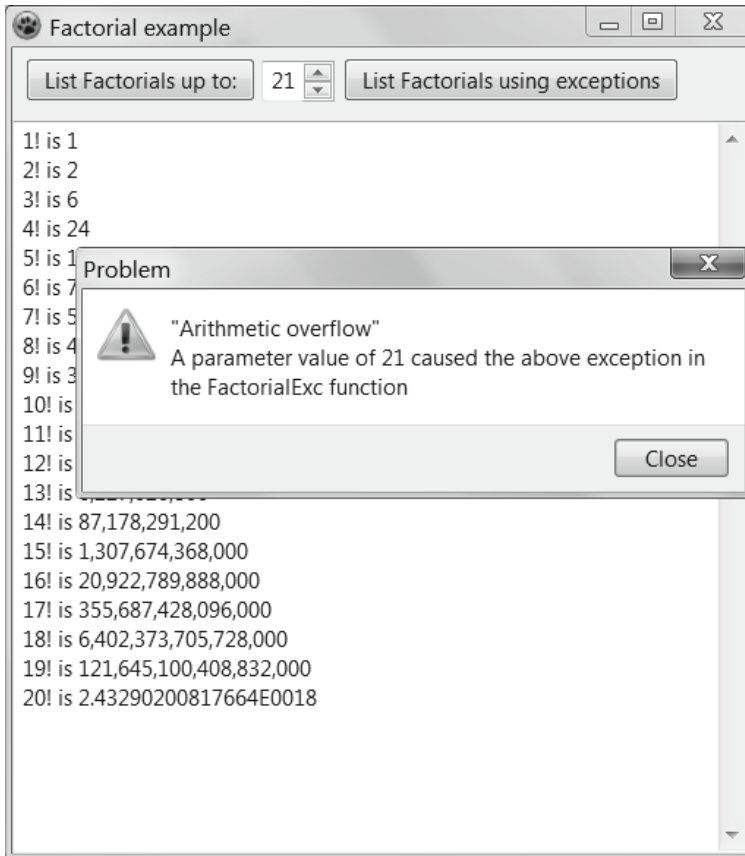


Figure 17.3 An integer overflow exception being caught

17.c Permutations

The factorial function $n!$ gives the number of ways of ordering all items from a set of n items, or (to use the mathematical term) the number of ways of **permuting** n items. Permutations of words are called anagrams. If we take the word **tea**, possible anagrams are: tae, eta, eat, aet, ate. For this 3-character word there are $3!$ permutations possible, i.e. 6 possible permutations (*tea and 5 others*).

It is relatively straightforward to design a recursive procedure to produce all the anagrams of a given word. The basic algorithm is to move each letter in turn into the first position, and then list all possible orderings of the remaining letters. To "list all possible orderings of the remaining letters" we again move each of the remaining letters into the now second position and list all possible orderings of the remaining letters. For a three-letter word there is only one possibility left, having already used two of the letters. So we have one of the possible permutations.

Chapter 17 WORKING WITHIN KNOWN LIMITS

You can see that this is a recursive algorithm that can be extended to cover words of arbitrary length. Such an algorithm is implemented in Section 17.e. We also know from the successive values of $n!$ in the preceding program that the numbers of possible permutations increases very rapidly for word lengths greater than about 5 characters. Which leads into a consideration of time limitations in programs.

17.d Time-consuming routines

Computers are extremely fast at moving data about in memory, and somewhat slower at displaying changing data on screen, slower still at reading and writing to disks and other devices. Generating and displaying all the permutations of lengthy words is likely to take measurable, noticeable time even on a very fast modern computer. In fact, designing a routine that takes an appreciable time to execute raises an oft-encountered issue: How do we best cope with a routine that is taking a long time?

Tight loops that run many millions of times before exiting tie up CPU cycles, and make UIs appear to freeze. Some OSs will post a “not responding” message in an application’s title bar if it becomes, well, unresponsive, and also grey out the offending program’s window.

This is not a good situation for users of your programs, or for you. Users may think the entire computer has frozen or crashed, and may turn it off, or kill your program, thereby losing or corrupting data. They may think your program is flawed and abandon it, especially if the steps they took to recover from your frozen program damaged other data.

This is more likely if there is no feedback warning of the possible freeze or delay, or even an hourglass cursor to indicate that something expected is going on unseen. The best feedback is a progress bar or other continually updated indication of progress towards the end of the hold-up. Users may want large files to download instantaneously, but they do know this is unrealistic, particularly on a Friday evening.

The problem partly arises from the **single-threaded** nature of a basic Lazarus program. Modern CPUs can run several threads at once, but each program or process has only one **main** thread. Unless you create further threads for your program’s routines you are restricted to the main thread which Lazarus generates and in which all its GUI components operate.

If you hog that thread with a processor-intensive routine (*such as generating and displaying 900,000 anagrams*) that takes noticeable seconds or even minutes to complete, the GUI will appear to freeze. Until the hog-routine completes, no amount of mouse activity or keyboard bashing makes any difference since the main Lazarus program loop that normally responds to such user input is stymied, and itself unresponsive.

The Lazarus program loop is operated by the `TApplication` instance which is created for each Lazarus project, and which we glanced at briefly earlier (*see Section i of Chapter 8, and Section b of Chapter 9*). `TApplication` is a complex class, but its `Run` method which is called in every Lazarus main program file begins by showing the main form (*i.e. the first form created if there is more than one*). Then it repeatedly calls the `ProcessMessages` procedure. This varies according to OS and widgetset, but in essence it queries the windowing system to see if any messages are pending for the running process.

If there are any messages, `ProcessMessages` dispatches the message(s) to their appropriate destination control(s). Some messages are handled automatically by controls as a result of the way the LCL is programmed. For instance a resize message leads to the control resizing itself. Other messages have effects programmed by the application developer. These are messages generated by the events available on the *OI Events* page, such as a form’s `OnCreate` event or a button’s `OnClick` event.

This means that sometimes we are able to make a GUI more responsive by explicitly calling `Application.ProcessMessages` in our own processor-intensive loop code.

Chapter 17 WORKING WITHIN KNOWN LIMITS

Of course this will not make a time-consuming tightly-coded-loop routine any faster (*in fact it will slow it down*). However, it will give the main program loop an opportunity to process pending messages such as giving feedback to the user about the state of the processor-intensive task.

A better option (*outside the scope of this book*) is to place time-consuming tasks in a thread of their own, and avoid bogging down the main program thread with time-consuming routines that block normal program messaging systems from operating.

Let's see how this works out in a program that generates all possible anagrams of a user-supplied word.

17.e Generating anagrams

Start a new Lazarus project named `anagrams`, with a main form named `anagram_main`, whose `Caption` is set to `Anagram Generator`, whose `Width` is 590 and whose `Height` is 460.

Drop a `TLabelledEdit` at the top of the form named `EWord`, and set its `LabelPosition` to `lpLeft`, its `EditLabel.Caption` to `Generate anagrams for this word`, its `Width` to 110, and its `MaxLength` to 12, arranging its `Left` property to display the full `EditLabel.Caption`.

Drop a label below the `EWord` named `LAnagramCount`. Double-click `EWord` to generate an `OnChange` event handler, and complete its implementation as follows:

```
procedure TForm1.EWordChange(Sender: TObject);
var w: integer;
    f: int64;

    function Plural(x: int64): char;
    begin
        Result := #0;
        if (x > 1) then Result := 's';
    end;
begin
    w := EWord.GetTextLen;
    f := Factorial(w);
    LAnagramCount.Caption := Format('%s will generate %s anagram%s',
                                   [EWord.Text, FormatFloat(',', #',', f), Plural(f)]);
end;
```

This procedure gives the user continuously updated feedback on the expected number of anagrams his/her word will generate, using a little helper function to make sure the plural of 'anagram' is applied correctly, and calling a `Factorial` function we now have to supply. In the form's private section declare the function as follows:

```
function Factorial(anInt: integer): int64;
```

Then generate an implementation skeleton and complete it thus:

```
function TForm1.Factorial(anInt: integer): int64;
begin
    result := -1;
    if (anInt < 0) then Exit;
    case anInt of
        0, 1: Result := 1;
        else Result := anInt * Factorial(Pred(anInt));
    end;
end;
```

Chapter 17 WORKING WITHIN KNOWN LIMITS

This procedure differs slightly from the one given before, including a couple of optimisations. On 32-bit OSs it is usually faster to work with the native 32-bit integer type than use byte variables, and it may also be faster to use the `Pred()` function than calculate `anInt - 1` on each recursion. A further optimisation is the dropping of an **on/except** block altogether, since in this application we are restricting the anagram word length by setting `EWord.MaxLength` to a value which prevents factorial overflow. Exception trapping adds overhead to every routine which uses it (*though in this case the effect of these optimisations may be imperceptible*).

Add two further labels, `LElapsedTime` and `LProgress` to the right of `EWord`. Below `LAnagramCount` drop two buttons. Name the first button `BSlowAnagrams` with its `AutoSize` set to `True` and its `Caption` set to `Generate Anagrams (slow)`. Name the second button `BFasterAnagrams` with its `AutoSize` set to `True` and its `Caption` set to `Generate Anagrams (faster)` (see *Figure 17.4*).

Double-click these buttons to generate `OnClick` event handlers and implement them as follows:

```
procedure TForm1.BSlowAnagramsClick(Sender: TObject);  
begin  
    LWords.Items.Clear;  
    LWords.Items.Add(['Slower']);  
    wordCount:= 0;  
    start:= Now;  
    LWords.Items.BeginUpdate;  
    GenerateAnagramsPoorly(EWord.Text, 1);  
    LWords.Items.EndUpdate;  
    LElapsedTime.Caption:= 'Elapsed time: '  
        + FormatDateTime('n"m" s:z"s"',Now-start);  
end;
```

```
procedure TForm1.BFasterAnagramsClick(Sender: TObject);  
begin  
    LWords.Items.Clear;  
    wordCount:= 0;  
    ss := TStringStream.Create(['Faster']+LineEnding);  
    ss.Seek(0, soFromEnd);  
    start:= Now;  
    LWords.Items.BeginUpdate;  
    try  
        GenerateAnagramsFaster(EWord.Text, 1);  
        LProgress.Caption:= 'Loading listbox from stream...';  
        Application.ProcessMessages;  
        ss.Seek(0, soFromBeginning);  
        LWords.Items.LoadFromStream(ss);  
    finally  
        ss.Free;  
    end;  
    LWords.Items.EndUpdate;  
    LElapsedTime.Caption:= 'Elapsed time: '  
        + FormatDateTime('n"m" s:z"s"',Now-start);  
end;
```

Chapter 17 WORKING WITHIN KNOWN LIMITS

The faster routine deals with the anagrams in two stages. First it creates a `TStringStream` and uses that to cache the anagrams as they are produced in the call to `GenerateAnagramsFaster()`. Then the listbox is filled with a call to `LBWords.Items.LoadFromStream()`. Caching the data using a stream wins hands down on speed.

The slower routine adds each anagram as it is produced to the `LBWords.Items.Text` property. When you run this routine with longer words you will see that as the `Text` property gets longer, so the routine gets slower and slower as all the string concatenations are performed.

The last UI control is a listbox named `LBWords`. Drop this at the bottom of the form, setting its `ScrollWidth` to 590, its `Columns` to 6, its `Align` to `alBottom`, and its `Height` to 350 (see Figure 17.4). **Note** that some OSs do not allow for more than one column in a listbox. If you develop on such a platform you will not see the anagrams displayed compactly in columns across the screen. Before we can run the program we need to add a few fields to the form's **private** section, and generate a few more method implementations. Delete the form's **public** section, and add **private** fields and methods as follows (the *Factorial* function should be there already):

```
private
wordCount: int64;
start: TDateTime;
ss: TStringStream;
procedure Exchange(var a, b: Char);
function Factorial(anInt: integer): int64;
procedure GenerateAnagramsFaster(aWord: string; charPos: integer);
procedure GenerateAnagramsPoorly(aWord: string; charPos: integer);
end;
```

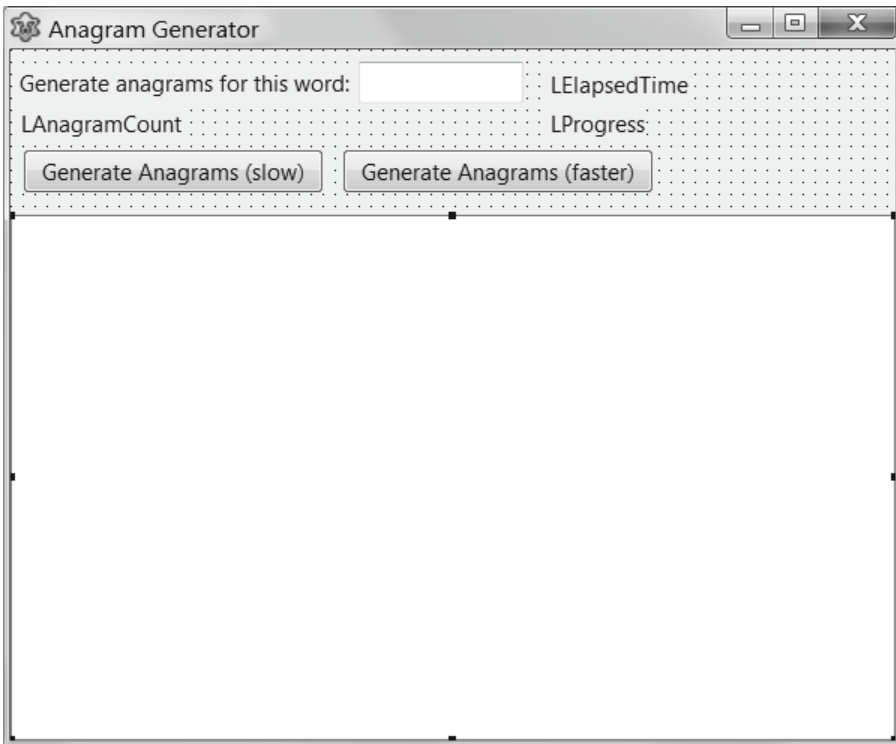


Figure 17.4 The anagram project GUI design, with listbox at the bottom

Chapter 17 WORKING WITHIN KNOWN LIMITS

Invoke Code Completion on one of the private methods to generate implementation skeletons, and complete these as follows:

```
procedure TForm1.GenerateAnagramsPoorly(aWord: string; charPos: integer);  
var le, p: integer;  
begin  
    le := Length(aWord);  
    if (charPos >= le) then  
        begin  
            LBWords.Items.Text := Format('%s%s', [LBWords.Items.Text, aWord]);  
            inc(wordCount);  
            if (wordCount mod 10) = 0 then  
                LProgress.Caption:= Format('%d anagrams generated', [wordCount]);  
            Application.ProcessMessages;  
        end  
    else for p:= charPos to le do  
        begin  
            Exchange(aWord[p], aWord[charPos]);  
            GenerateAnagramsPoorly(aWord, Succ(charPos));  
            Exchange(aWord[p], aWord[charPos]);  
        end;  
    end;  
  
procedure TForm1.Exchange(var a, b: Char);  
var tmp: Char;  
begin  
    tmp := a;  
    a := b;  
    b := tmp;  
end;  
  
procedure TForm1.GenerateAnagramsFaster(aWord: string; charPos: integer);  
var le, p: integer;  
begin  
    le := Length(aWord);  
    if (charPos >= le) then  
        begin  
            ss.WriteString(aWord + LineEnding);  
            inc(wordCount);  
            if (wordCount mod 10) = 0 then  
                LProgress.Caption:= Format('%d anagrams generated', [wordCount]);  
            Application.ProcessMessages;  
        end  
    else for p:= charPos to le do  
        begin  
            Exchange(aWord[p], aWord[charPos]);  
            GenerateAnagramsFaster(aWord, Succ(charPos));  
            Exchange(aWord[p], aWord[charPos]);  
        end;  
    end;  
end;
```

Chapter 17 WORKING WITHIN KNOWN LIMITS

Compile and run this program, for which typical output is shown in Figure 17.5. Take care not to use the slower routine on words of 7 or more letters – you'll wait all day! Notice how strategically placed calls to `Application.ProcessMessages` allow the GUI to be updated with information about progress even within intensive recursive routines. However some calls are opaque, such as the call to `LBWords.Items.LoadFromStream()` which is monolithic. We can't get 'inside' it to report on progress (*without rewriting the `TListBox` component*).

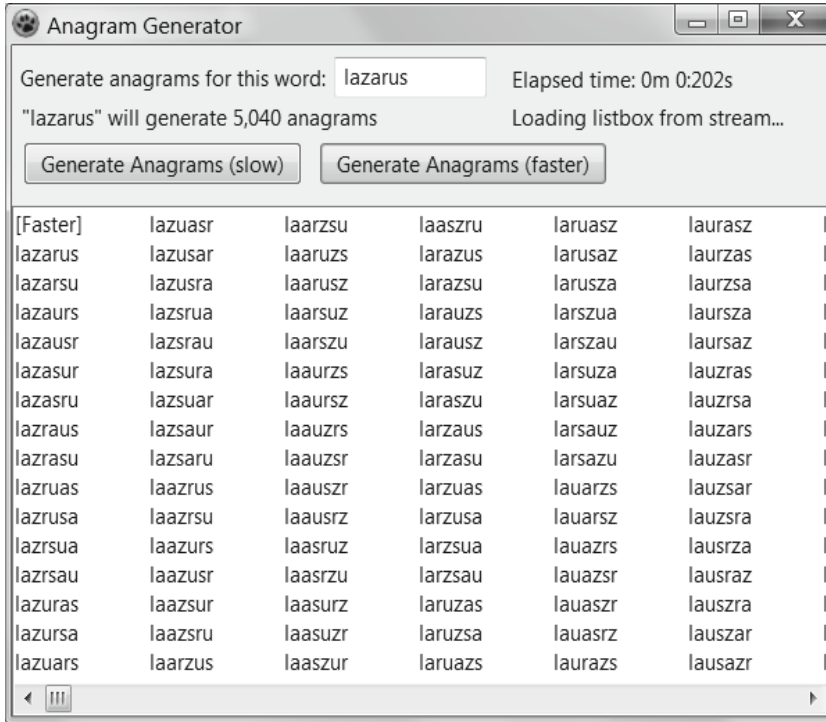


Figure 17.5 The Anagram Generator in action

The two `GenerateAnagrams...` functions are similar in that they use the same recursive algorithm for exchanging individual characters. They differ only in the way the generated anagrams are stored and displayed.

17.f Review Questions

1. It was stated blithely earlier (*in Section e*) that three optimisations applied to the `anagrams` project probably made an imperceptible difference. How would you test the truth of this assertion?
2. Extend the anagram project by designing a way to make several timed trials of the two anagram generating routines, and comparing and displaying average results for the two routines for varying word lengths.



Chapter 18 ALGORITHMS AND UNIT TESTS

18.a Collaboration

The final product of your Lazarus project will be a program that does certain things well, perhaps even excellently if you become a good programmer. You must find a good name for your program. This is not always an easy task, particularly if you are disinclined to copy an existing name, or prevented from doing so by reason of copyright. (*Can a generic term like 'Word' be a registered brand name? In some places, apparently, Yes*).

Perhaps you will write all the needed code yourself (*except, of course, the libraries like the FCL and LCL you depend on*). In most human endeavours it transpires that collaboration is a benefit, that two are better than one, especially if those two people's gifts and approaches are different, even opposite. Someone else may have the experience or skill you lack and be able to offer it where needed. To me this is one of the joys of open source programming communities – you encounter people who are willing to share something of themselves and their experience. And this is foremost a human encounter, rather than a commercial contract with a fee (*though there is a place for that as well, if programmers are to eat*).

One way to move forward in understanding and programming insight is to join a programming forum or newsgroup. You can learn a lot about programming just 'listening in' to discussions of programming topics on a mailing list between half a dozen programmers contributing to an informed debate. You may well find that such discussions, even if you are rarely a contributor, are well worth the effort of following. We all need to be stretched in our understanding, and listening to several different views debated over some moot point in programming often uncovers the subject to newcomers far more than the contributors to the debate have any awareness of.

18.b The algorithm – a specific plan

Programmers term a specific plan, a series of steps to be followed to achieve some end, an **algorithm** (*the word derives from the work of Al-Karismi, a seminal Islamic mathematical writer of the ninth century*). The phrase **pseudo-code** is also used to indicate a similar concept. In other words, in thinking about particular functionality that your software needs to encapsulate, it is often best to start, not with detailed Pascal code, but with an **outline of the steps** to be followed in implementing the desired functionality. Articulating this algorithm, firstly as a notion in your imagination, and then possibly as a sketch on paper, may clarify your thinking, and also help you name the smaller steps into which you have analysed the needed routine; steps that will be combined to provide the overall functionality.

Let's take a simple, straightforward example to illustrate the process. We find the need, in the course of a project's development for a function that takes text (*of arbitrary length, such as a paragraph from this book*), and parses it into its constituent words. We want our function to be able to specify the definition of a 'word' by stipulating what character(s) are to be regarded as word separators (in programming jargon these are often referred to as **delimiters**). The function should return the list of parsed words in a stringlist. A stringlist is preferred over a dynamic array of strings for this purpose, because it offers a count, it automatically expands to the size needed, and it offers an interface widely used in LCL controls such as `TMemo` and `TListBox`.

There are various ways to accomplish this task. Here we consider one of these many ways, not because it is the 'best', or the fastest, but because it illustrates the process of designing, refining and testing an algorithm implemented in Pascal, with a few pointers for subsequent optimisation.

Firstly: What are we to call this function? If we want it to be pretty much self-descriptive, it could be called `ParseTextToIndividualWordsList`. However, for our own use a shorter name might suffice, say `ParseToWords`. Some languages provide such a function in their main library with a shorter name (*e. g. PHP offers an Explode function*). But usually the shorter the name, the less information it conveys, and so the more cryptic the function may appear. We will aim to give fully meaningful names to routines, without being pedantically verbose.

Chapter 18 ALGORITHMS AND UNIT TESTS

18.c A parsing algorithm

Before you start to write any new function you should always ask yourself: “Is this functionality already provided in the libraries at my disposal (RTL, FCL, LCL)?” We may waste time reinventing the wheel simply because we do not know the name or location of the required wheel in the 1.6 million lines of code that come with Lazarus. Sometimes a simple search of the relevant source directories and subdirectories using the **Find in Files** tool (IDE shortcut [Shift][Ctrl][F], or **Search | Find in Files...**) will throw up exactly the routine we need, or something very close to it.

The RTL `strutils` unit has two functions, `WordCount()` and `ExtractWord()` that could be combined to do what we want. But there is nothing that precisely meets our specification (*as far as this author knows*). However, looking at the above RTL functions does reveal a useful type for our needs:

```
type TSysCharSet = Set of Char;
```

and a useful default constant:

```
const
```

```
{Default word delimiters are any characters except the core alphanumerics}
```

```
WordDelimiters: set of Char = [#0..#255]-['a'..'z','A'..'Z','0'..'9'];
```

We can use this `const` simply by specifying `strutils` in our `uses` clause. Thus the signature of the function will be:

```
function ParseToWords(const aString: string; separators: TSysCharSet):  
    TStringList;
```

We pass a text string to the function, along with a set of valid word separator characters (*such as* `['!',' ','.']` and so on), and receive a stringlist containing the parsed words. The `const` specifier for the parameter does two things. Firstly it prevents us from accidentally altering the string parameter passed in (*if our code mistakenly alters aString the compiler will warn us that aString is supposed to be const*), and secondly it opens the possibility of the compiler making some optimisations to the string-handling code in the function that would not be possible if the string were passed as an ordinary parameter that could be changed.

Since it is possible that very large strings indeed might be passed to the function, this could be a significant advantage over a non-`const` parameter. **Note** that there are possible drawbacks to using `const` parameters, which can lead to subtle bugs appearing. There has been detailed discussion of this issue on the mailing lists, so it will not be pursued here, except to underline that `const` is not a panacea to be applied everywhere.

What about an algorithm? A basic 'brute force' approach would be to examine each character in the string in turn. We ignore any characters that are separators until we find the first character that is **not** a separator. This will become the first character of a parsed word. We continue moving along the string checking characters as we go, adding successive characters to the word fragment being parsed until we encounter another separator character. At that point we pause and extract the word we've found, adding it to the `Result` list. We then repeat this individual character examination again, continuing from the point we had reached before pausing, and continue examining, pausing to extract a parsed word as needed until we reach the end of the string.

Having reached the end of the string there may be a final parsed word ready to add to the resulting stringlist which has not yet been added. If so, we add it.

Chapter 18 ALGORITHMS AND UNIT TESTS

Our algorithm should be able to cope with these situations, all of which may well exist:

- `aString` is empty
- `separators` is an empty set, `[]`. In this case we use a default (`WordDelimiters`). Unfortunately because the parameter is a set type we cannot provide a **default** value in the parameter list itself, as we could if it were a `Char` e.g.
`Parse(const aString: string; separator: Char=' '): TStringList;`
- several adjacent separators may be present (e.g. 'What?! Surely not!...')
- the points just before the beginning of the string, and just after the end of the string must be treated as separators, even though there are no string characters at these points to examine
- separators may, or may not, appear at the beginning and/or the end of the string
- `aString` may turn out to contain *only* separators

In outline our algorithm will look like this:

- begin with an empty parsed string fragment
- step through each character in `aString` in turn
- if the character is part of a word, add it to the current parsed fragment
- else, if the character is the first separator following a word, add the current fragment to the stringlist and reset the current fragment length to zero. Second or subsequent adjacent separators are ignored.
- After reaching the end of the string, if a non-zero-length fragment remains, add this to the stringlist too.

Implementing the algorithm is a matter of using the best Pascal routine for stepping through each character in a string. Since the string is known at the outset (*it is passed as a parameter*) we can use a **for** loop starting at the first character, and ending at the last character.

FPC recently provided optimised syntax to aid in using **for** loops for situations like walking through each character of a string. The syntax is as elegant as it is simple:

```
for element in aString do {something using element};
```


Chapter 18 ALGORITHMS AND UNIT TESTS

This enables us to write the following:

```
function ParseToWords(const aString: string; separators: TSysCharSet
                    ):TStringList;
var
  c: Char;
  s: string;
  buildingAWord: boolean = False;
begin
  if Length(aString)=0
  then begin
    Result := nil;
    Exit;
  end;
  if separators = []
  then separators:= WordDelimiters;

  Result := TStringList.Create;
  for c in aString do
  case (c in separators) of
    False: begin
      if not buildingAWord
      then buildingAWord:= True;
      s := s + c;
      end;
    True : begin
      if buildingAWord
      then begin
        buildingAWord:= False;
        Result.Add(s);
        s := '';
      end;
      end;
  end;
  if (Length(s) > 0) then Result.Add(s);
end;
```

Note that in addition to the simplified `for . . . in . . . do` syntax, we have also initialised the boolean variable `buildingAWord` in its declaration.

We first check for passing of any empty parameters, providing for the case that either of them is empty. Then we create the stringlist the function will return.

The `for` loop that steps character-by-character through the string contains a single `case` statement which tests the current character for membership in the `separators` set.

The character is either a separator or part of a word.

To record whether a word is currently growing or whether we are finding successive separators we maintain a record of the parsing state by means of a boolean variable, `buildingAWord`.

This history of the parsing state, held as a boolean, saves us from the need to backtrack to see whether previous characters were separators or word-characters.

As long as we find word-characters, `buildingAWord` is `True`, and the characters are added to the growing fragment. As soon as a separator is encountered, `buildingAWord` is set to `False`, and the current fragment (*now a complete word*) is stored in the `Result` stringlist, and the fragment is cleared to be empty for the next word.

Chapter 18 ALGORITHMS AND UNIT TESTS

18.d Testing the ParseToWords function: the FPCUnit Test

Lazarus provides two ready-to-run test program skeletons, one for console tests, the other for GUI tests. We will adapt the GUI test-runner application. Select **Project | New Project** and choose the last option in the listbox on the left of the *Create a New Project* dialog: *FPCUnit Test Application* and click [OK].

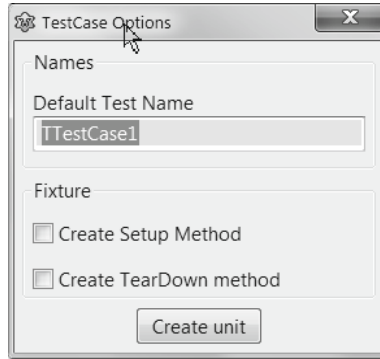


Figure 18.1 The TestCase Options dialog

This choice presents you with the *TestCase Options* dialog, and the *Default Test Name* field has the default name `TTestCase1` highlighted for you to overwrite. Change this to `TParseWordsTestCase`, and click the [Create unit] button (*leaving the two checkboxes unchecked*). See Figure 18.1.

Choose **Project | Save Project As...** naming the project `parseWordsUnitTest.lpi` and putting this project in a new directory. Name the main unit `parsewordstestcase.pas`. The skeleton code Lazarus provides contains a lot of functionality inherited from the `TTestCase` class:

```
type TParseWordsTestCase= class(TTestCase)
  published
    procedure TestHookUp;
end;

implementation

procedure TParseWordsTestCase.TestHookUp;
begin
  Fail('Write your own test');
end;
```

We shall adapt the `TestHookUp` method of the `TTestCase` descendant to perform the tests we devise. First we have to declare the `ParseToWords()` function. Choose **File | New Unit** and save the new unit Lazarus generates as `ParseWords.pas`. Add `strutils` to the **uses** clause, and add a type section with the declaration of the `ParseToWords` function as above.

Chapter 18 ALGORITHMS AND UNIT TESTS

With the cursor in that function declaration line press [Shift][Ctrl][C] to get Lazarus to write a code skeleton in the **implementation** section of the unit. Then copy the body of the function as given in Section e, and save your work.

Move back to the `ParseWordsTestCase` main unit, and press [Alt][F11] to open the *Add unit to Uses section* dialog. Click the *Interface* radio button, and double-click `parseWords` in the list (*it should be the only unit listed*). The dialog closes and Lazarus adds `parseWords` to the **uses** statement.

To complete the test runner program we simply need to add tests to the `TestHookUp` method. Place the cursor within the class name `TTestCase`, and press [Alt][Up arrow]. This jumps you to the declaration of `TTestCase`, and you will see that it descends from `TAssert`. Place the cursor in `TAssert` and press [Alt][Up arrow] to reach the declaration of `TAssert`. You will see that it consists of about 40 overloaded procedures named `AssertXXX...`. These are the procedures we can invoke to test our assertions about the tests we write.

For example, if we pass an empty string to `ParseToWords` it should return a `nil` value, not a stringlist. So one of our tests can be:

```
AssertNull( 'Empty string returns nil function result', ParseToWords('', []) );
```

The first parameter to the `AssertNull` procedure is a string describing the nub of the test, and the second parameter is a `nil` value (*hopefully*). If we wrote:

```
AssertNull( 'This test is bound to succeed', nil );
```

then when we run the test program, this test cannot fail to succeed – not only are we asserting that the parameter passed is `nil`, we actually pass a `nil` value, so the outcome of the test is now a certainty rather than merely an assertion.

Move back to the main unit `ParseWordsTestCase`, and change the name of the procedure `TestHookUp` to `BoundToSucceed`. Press [Shift][Ctrl][C] to get Lazarus to change the name in the **implementation** section. Replace the line

```
Fail( 'write your own test' );
```

with the line

```
AssertNull( 'This test is bound to succeed', nil );
```

Compile and run the program. If all goes well, when you click on the [Run] button you should see green success icons as in Figure 18.2.

Whereas, if you change the line of code to the following and run the program, you will see purple fail icons when you [Run] the test as in Figure 18.3:

```
AssertNull( 'This test is bound to fail', pointer(1) );
```

Here we have put in a pointer value that is not `nil`, and so (*in spite of the procedure's name*) this one is bound to fail. You may find when running the unit test program under the debugger that an exception dialog is raised. If so, click on [Continue] (*rather than [Break]*) and use the taskbar if necessary to focus the program to the foreground again. Hopefully that little exercise has given you a feel for how the FPCUnit Test application is designed and works.

Chapter 18 ALGORITHMS AND UNIT TESTS

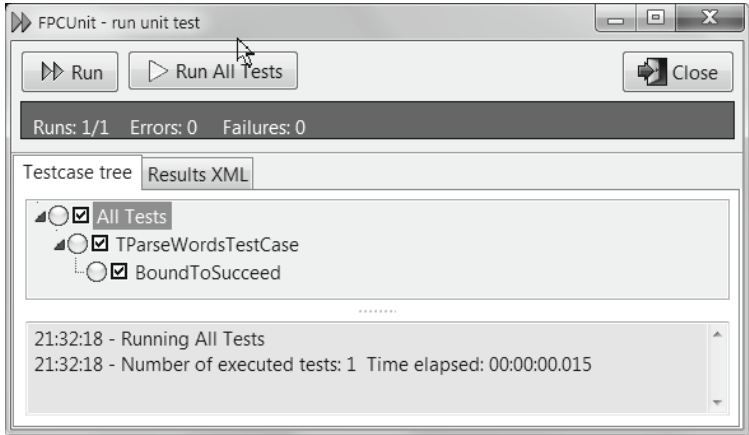


Figure 18.2 A successful run of BoundToSucceed

The important question is: What tests should we write? It is hard to give a satisfactory answer, since you can **never have too many** software tests, and unfortunately many important tests of GUI programs are very hard to write (*because testing GUI functionality and interactivity in an effective and automated way is not easy*).

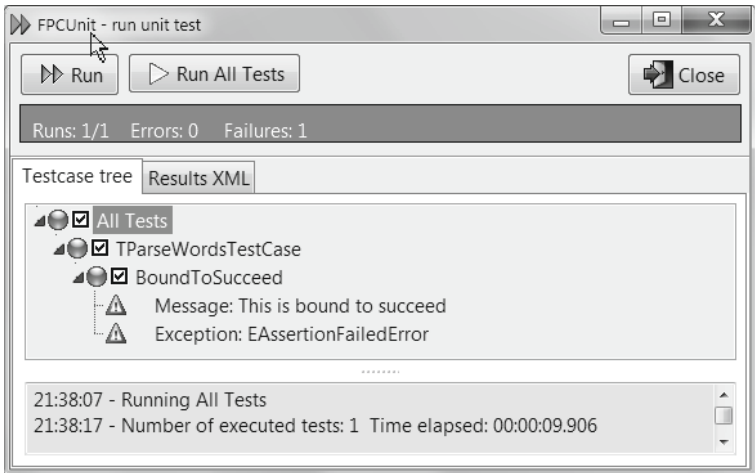


Figure 18.3 A failed run of BoundToSucceed

Chapter 18 ALGORITHMS AND UNIT TESTS

18.e Example tests

The easiest tests to write are not of the GUI parts of software, but of procedural code such as function results. We test the 'extremes' of parameters that might be passed to the tested function, and devise tests that provide as wide a variety of possible inputs to the function as we can come up with (*and have time to write*).

Once written, the Unit Test program can be run again very quickly if the function is ever patched or altered, to make sure that the core functionality has not been compromised (*or perhaps to discover that it has been compromised*). Designing a function that passes all the valid tests we can devise does **not** guarantee that it is bug-free. However it does reduce the likelihood that the most glaring and commonly encountered bugs have slipped past us.

We give here a few more tests that might be applied to the `ParseToWords()` function. It is not a comprehensive list by any means, but a selection of the type of tests you should consider applying: simple straightforward tests; tests of 'corner cases'; tests at the lower and upper limits of data being processed; tests to cover situations that "will never happen" – but which do seem to happen, nevertheless, on users' computers.

Fill out the `TParseWordsTestCase` class declaration to read as follows:

```
type TParseWordsTestCase= class(TTestCase)
  published
  procedure BoundToSucceed;
  procedure ParseEmptyString; //''
  procedure ParseNoSeparatorsInString; // 'ThereIsNoSeparatorInThisString'
  procedure ParseOnlySeparatorsInString; // ',,(),!'
  procedure ParseLeadingAndTrailingSeparators; // '___LeadingAndTrailing....'
  procedure ParseSeparatorsAllTogether; // 'start,,,,,end'
  procedure ParseLongText; // longText - see const above
end;
```

Add the following const statement for `longText`:

```
const longText =
'We find the need, in the course of the development of a project for a '+
'function that takes text (of arbitrary length, such as a paragraph from '+
'this book), and processes it by parsing it into its constituent words.';
```

Press [Shift][Ctrl][C] in the published procedures section of the `TParseWordsTestCase` class declaration to cause Lazarus to write skeleton implementation code for us. Then complete the test procedure implementations as follows:

Chapter 18 ALGORITHMS AND UNIT TESTS

```
procedure TParseWordsTestCase.ParseEmptyString;
begin
  AssertNull('ParseEmptyString', ParseToWords('', []));
end;

procedure TParseWordsTestCase.ParseNoSeparatorsInString;
begin
  AssertEquals('ParseNoSeparatorsInString', 'ThereIsNoSeparatorInThisString',
    ParseToWords('ThereIsNoSeparatorInThisString', []) [0]);
end;

procedure TParseWordsTestCase.ParseOnlySeparatorsInString;
begin
  AssertEquals('ParseOnlySeparatorsInString', 0,
    ParseToWords(';,.,()!', []).Count);
end;

procedure TParseWordsTestCase.ParseLeadingAndTrailingSeparators;
begin
  AssertEquals('ParseLeadingAndTrailingSeparators', 'LeadingAndTrailing',
    ParseToWords('___LeadingAndTrailing....', []) [0]);
end;

procedure TParseWordsTestCase.ParseSeparatorsAllTogether;
begin
  AssertEquals('ParseSeparatorsAllTogether', 'start',
    ParseToWords('start,,,,,,,,end', []) [0]);
end;

procedure TParseWordsTestCase.ParseLongText;
begin
  AssertEquals('ParseLongText', 39, ParseToWords(longText, []).Count);
end;
```

You see that writing tests is slightly tedious – there is little motivation to write lots of them, and lots of tests covering the widest possible variety of test cases is what is needed to give you confidence in a test suite, indeed to make it worth having at all.

Does `ParseToWords` pass all these tests on your machine, and your operating system?

How fully do you feel the tests given above exercise or **stress** this routine?

Chapter 18 ALGORITHMS AND UNIT TESTS

18.f Test-driven development

Note that, helpful as these sort of tests are (*indeed essential if we want to write high quality software that does not fall over at the first small hurdles users put in its way*), the way these tests have been introduced does not qualify as **test-driven development (TDD)**. We came up with our tests **after** we developed the parsing routine to check that it worked as we expected. The test-driven paradigm is more rigorous and more logical. It **starts** with writing tests, forcing you to consider the outputs of your routines and the output interface of your GUI controls before you code them. This whole philosophy is beyond the scope of a short introductory book like this.

However, you can probably see that developing a routine's complexity test by test is a safer way to build something complex from well-tested mini-modules and subroutines. This sort of testing tests not only each subroutine **as it develops**, but can also test the growing complexity of interactions between the various parts of more complex software as it grows.

18.g Optimising debugged routines

The more geeky you are, the more likely you may be to fall into the trap of **premature optimisation**. Once you have written a bit of Pascal code, and gained some confidence in use of the basic syntax and commonly encountered LCL classes and components you will gain ideas of how to make code run faster. However, it is nearly always a mistake to start out by writing code chosen solely because you think it will be fast. It is far better to develop and implement an algorithm that is simple, elegant, and easy to maintain and debug. The chances are it will be fast enough.

If you find some of your code proves to be sluggish when dealing with real-life data, that is the time to think about optimising it (*if you consider the time you spend on this worthwhile*). Usually the slowest portions of any program are those dealing with input and output (*or I/O as it is often abbreviated*), either to screen, file or other device. The 'slowest' parts of GUI programs tend to be the periods of inactivity resulting from lack of user input. Memory operations are generally orders of magnitude faster by comparison.

The `ParseToWords` function would probably be speeded up somewhat if instead of building up each parsed word character-by-character

```
s := s + c;
```

we instead introduced a `wordStartPosition` counter to remember the index of a word's starting position in `aString`, and then when we next met a separator (*at the *i*th character*) we copied the entire word at one go

```
Result.Add(Copy(aString, wordStartPosition, i - wordStartPosition));
```

This would mean replacing the optimised syntax "**for** `c` **in** `aString`" by the more traditional Pascal syntax using a for loop counter variable `i` that we can use both to save a value for `wordStartPosition`, and to track the current position in the string. It is likely to be faster to `Copy()` a section of text to a buffer rather than to append the same text character-by-character to a string buffer. However, that is a supposition. Unless you make the comparison by timing comparative performance under identical conditions your supposition (*however reasonable in theory*) may be incorrect in practice. The questions which follow explore this a bit more.

18.h Profiling and compiler optimisation

Linux users have an advantage in the availability of the profiler `valgrind` which is designed for making exactly the kinds of comparisons outlined above. The *Linking* page of the *Options for Project: xxxx* dialog (**Project | Project Options...**) has a specific checkbox to support this (*Generate code for valgrind (-gv)*). All platforms are able to take advantage of some built-in FPC optimisations which may (*or may not*) increase the speed of running programs.

The compiler has various optimisation techniques up its sleeve which you can invoke to see if they offer any speed advantage. The **Code Generation** page of the **Options for Project** dialog offers several such technical tweaks.

Chapter 18 ALGORITHMS AND UNIT TESTS

Make sure (if you use the *gdb debugger*) you have completed debugging your project before you activate any of these optimisations, since they remove information the debugger needs to debug your project as it runs. Don't be disappointed if none of these optimisations has much discernible effect on your program's speed. They are most effective on very processor-intensive code running in tight loops. Commonly GUI programs have I/O bottlenecks rather than CPU bottlenecks.

18.i Review Questions

1. Rewrite `ParseToWords()` to use the `Copy()` procedure so that entire words are added to the stringlist `Result` in one go, rather than being built up character-by-character before being copied. Run your altered routine through a test suite to check that it works as you expect.
2. Write a short program to call `ParseToWords` and your newly optimised `FasterParseToWords` say 100,000 times each on some test strings, timing the performance in each case, displaying the result, and determining if there is a significant difference between the two routines
(Hint: you can use the `sysutils.Now` function for simple timing tests if you don't have profiling software).
3. Is there a different, better algorithm you can devise to parse a string into individual words? Why do you think the programmer who wrote the `strutils.WordPosition()` routine chose to use the `PChar` data type, and to use the `Move()` procedure to implement `ExtractWordPos()`?



Chapter 19 DEBUGGING TECHNIQUES

If you make no mistakes, you usually don't make anything, and mistakes in software are often not apparent from simply looking at the written code. Of course the compiler will pick up typing errors such as missing semicolons, spelling typos, or forgetting to declare a variable you use (*and Lazarus' Code Completion feature can often be invoked to insert missing variables without you needing to type a declaration in full*). Often it is the case that problems only emerge when code which compiles perfectly is run and stressed with a real user and fresh data input that varies from the few cases you put into your unit tests (*you did write unit tests, didn't you?*).

How do we divine what is causing a program to give different output from the expected or intended outcome? The problem is not usually spotting the bug, but understanding what is causing it, and then working out how to eliminate it. This chapter looks at some techniques you can use to diagnose what might be going wrong when your program gives the “wrong answer”. Sometimes hard-to-pinpoint bugs will require several different approaches before you can diagnose exactly what is going wrong and where. There are also those very-hard-to-find bugs, appearing just occasionally, and with no discernible pattern whose cause may never be diagnosed or corrected. The first sections of this chapter (*a to g*) consider several **bug-avoidance** strategies, and later sections look at **bug-hunting** techniques.

19.a Preventing bugs

The simplest way to avoid a bug is not to write code yourself, but to use another programmer's well-tested routine. For example, I know that the core Lazarus/FPC developers are better and more experienced programmers than me. This is not false humility on my part, it is just a fact. Routines I write – the code offered in this book – should be treated with caution. It is not that well tested, being written to illustrate specific points about Pascal and programming in general. There are no customers depending on it or using it in critical situations, and (*at the time of writing*) no feedback from code users at all. It is folly to assume that simply because code you write “works” that it is therefore bug-free. All software requires rigorous testing to identify and eliminate bugs. Often bugs only come to light under certain (*perhaps rare*) situations, or in combinations of circumstances that might occur on another computer operated by a beta-tester which rarely if ever occurs on your development computer. So it is always better to use a RTL/FCL/LCL routine in preference to a roll-your-own routine if you can, and if you care about software reliability.

This is not because the Lazarus libraries are completely bug-free. It is because they are used, all the time, by many more people than use or test code you will be writing, and are being continuously maintained (*any parts that are not are specifically marked as deprecated*). Also people who use Lazarus tend to report bugs when they find them because they feel part of the Lazarus community and want to contribute. Reporting a bug is a valuable contribution; almost as valuable as contributing a patch to fix the bug. Whereas users of commercial software – even sometimes users of commercial development software – rarely report a bug, and perhaps do not always even realise that is what they have just stumbled across. They tend to curse computers and move on to the next task.

Chapter 19 DEBUGGING TECHNIQUES

19.b Unit tests

Writing unit tests (see Chapter 18, section g) will help you isolate simple-to-find bugs.

19.c Paying attention to compiler messages

Some bugs arise because we ignore Warnings issued by the compiler. Unless you position it elsewhere the Messages window sits below the Editor. It does not pass on all the output from the compiler (*the compiler's output is filtered to remove some messages*) but what it does display is worth noticing.

Note: If you want to see **all** compiler messages simply right-click on a compiler message and choose *Copy all Shown and Hidden Messages to Clipboard*. Paste them into any text editor to view them all.

Some of the information is not useful for tracking bugs. The message:

```
unit1.pas(8,72) Hint: Unit "math" not used in Unit1
```

(like other Hints) is unlikely to help you trace a bug. However, it reduces your code size slightly to remove unnecessary statements (*though it does no harm to leave this pointless statement in place*), and it also reduces the number of messages to review when building or compiling. In the case of this message there are several ways to remove the unused unit, and prevent the message reappearing at each build/compile. If you click on the message it jumps you immediately to the line and position (*line 8, character 72*) identified in the message. You can manually delete the `math` unit. Or if you right-click on the message and choose Quick fix: Remove unit, Lazarus will delete the unit name `math` for you.

Likewise you can right-click on Unit1 in the Editor and choose **Refactoring | Unused Units...** to open the *Unused units* dialog. This dialog may well show more than just the one unused unit since it determines what is not needed independently from the compiler's output, and does so very thoroughly, dividing the *Unused Units* treeview into two branches. These list unused units found in the **interface** separately from those found in the **implementation** section. You may notice a brief burst of disk activity while this determination proceeds.

You can either choose the [Remove all units] button to remove all the listed unused units, or (*if you know you are about to use some routines from one or more of the units which is presently unused*) you can click individual units to select them, and [Ctrl]-click to select more than one unit (*the units do not need to be adjacent*). This enables the [Remove selected units] button which removes only the unused units you have specified (see Figure 9.1) leaving the unselected units still in the **uses** clause.

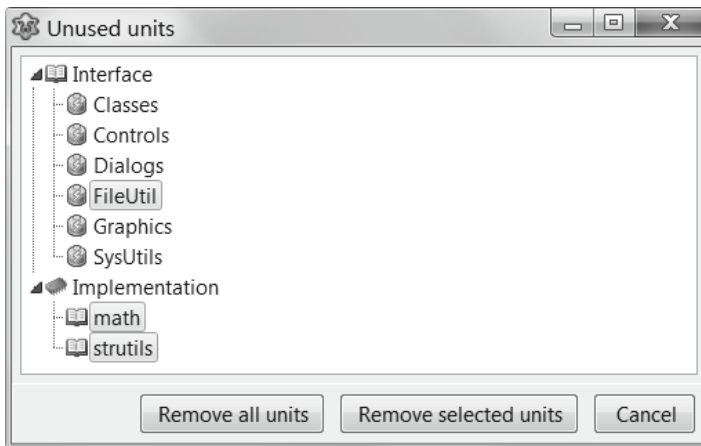


Figure 19.1 The Unused units dialog

Chapter 19 DEBUGGING TECHNIQUES

There are also compiler Messages that can be very helpful in helping prevent bugs slipping past you. Start a new Lazarus project called `fibonacci` with a main form unit called `fib_main.pas`, with the form's name set to `fibForm` and its `Caption` set to `Fibonacci example`.

This short project will calculate the first few numbers in the Fibonacci series. The Fibonacci mathematical series is related to the arrangement of various items in nature, such as leaves on a plant stem, segments of a pineapple and the whorls of a pine cone. See Figure 19.2 for typical program output.

Drop a button on the form named `BTestFib` with its `Caption` set to `Fibonacci`. Beside it drop a label named `LFibLimit` with its `Caption` set to `Number of items in Fibonacci series:`, and beside that drop a spinedit named `EFibLimit` with its `MinValue` set to 3 and its `Value` set to 30. Double-click `BTestFib` to generate an `OnClick` event handler as follows:

```
procedure TcalcForm.BTestFibClick(Sender: TObject);  
begin  
    MDisplay.Lines.Text:= FibString(EFibLimit.Value);  
end;
```

Clicking `BTestFib` will generate a Fibonacci series whose length is determined by the spinedit value. The series is stored in a string which is displayed in the memo named `MDisplay`. In the form's private section add a declaration for this Fibonacci string function:

```
private  
    function FibString(noInSeries: integer): string;
```

Use Code Completion to generate an implementation skeleton for this function, and complete it as follows (*Fibonacci series can be produced using a recursive algorithm, but here we use an iterative algorithm*):

```
function TcalcForm.FibString(noInSeries: integer): string;  
var i: integer;  
    m: integer=1;  
    n: integer=0;  
begin  
    Result:= Format('Fibonacci series (%d-%d): ', [m, noInSeries]);  
    while (i < noInSeries - 2) do  
        begin  
            Inc(i);  
            m:=m+n;  
            n:=m-n;  
            Result:= Format('%s %d', [Result, m]);  
        end;  
end;
```

Chapter 19 DEBUGGING TECHNIQUES

Compile and run this project. How come it does not display a series of 30 Fibonacci numbers? If you're lucky it will display only two numbers in the series, 0 and 1. If you're unlucky it might display something more unusual. Did you notice the compiler Warning?

```
fib_main.pas(47,10) Warning: Local variable "i" does not seem to be initialized
```

Warnings indicate that something is amiss with code. **Uninitialised variables** are a common source of bugs, and fortunately a bug source that is extremely easy to put right. All we need to do is insert

```
i := 0;
```

immediately after the first begin (or set the initial value of *i* in the **var** declaration as we did for the variables *m* and *n*). If we recompile and run the program we get a more satisfactory result (see Figure 18.1) in which each number in the series is the sum of the two previous numbers.

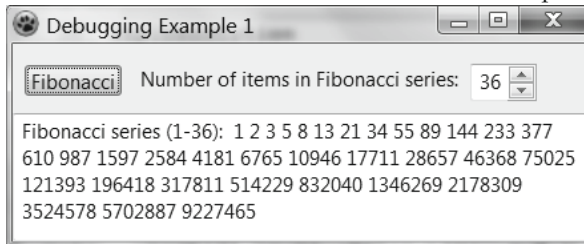


Figure 19.2 The first 36 numbers in a Fibonacci sequence

19.d Using Assertions

An **assertion** is a useful tool (*widely used in C and C++ programming where it originated*) for checking the runtime value of a variable (*checking, for instance, that it falls within a certain allowed range*). If the assertion fails the program halts with an *Assertion Failed* exception and a message identifying the offending line in the program source code. Assertions are mainly useful for catching programmer errors (*such as trying to access a class instance that has not been instantiated*), and for checking assumptions that may not always be true.

Because there is full compiler support for assertions, they can be left in code permanently, but switched off through a compiler directive (or compiler command line argument) so that no assertion code is compiled into a production executable, i.e. you can have working assertions only in a test-phase, 'debug' executable, and remove compiled assertion code routines from your 'release' version without altering the source code.

An assertion is basically a single boolean expression with an optional message. The syntax for including an assertion is

```
Assert(asserted_boolean_expression, optional_failure_message);
```

You also need to give the compiler directive `{$ASSERTIONS ON}` or `{$C+}` in one of your program's units (or specify the command line option `-Sa`). The *Options for Project...* dialog (`[Shift][Ctrl][F11]`) has a *Parsing* page on which there is an *Include Assertion code (-Sa)* checkbox to give the compiler the same instruction.

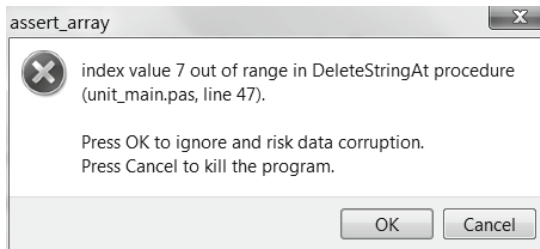


Figure 19.3 Output from a failed assertion

Chapter 19 DEBUGGING TECHNIQUES

Here is an example of how you might use an assertion both to check that an array exists, and that the index given to an array procedure has a sane value (see Figure 19.3). The procedure is one that deletes an indexed value from a string array, reducing the array size by one and shuffling every entry above the index downwards accordingly. It makes use of the powerful (and hence potentially dangerous) `Move()` procedure. Start a new Lazarus GUI project called `assert_array.lpi`, with a form unit called `unit_main.pas`. Name the form `frmTestAssert`, and set its `Caption` to `Assert` example. Add an `{ASSERTIONS ON}` directive to the unit, along with a type declaration for a dynamic string array named `TStrArray`. Drop a button on the form, generate an `OnClick` event handler for it, and add a `public` field `FArray` of type `TStrArray`, and two `public` procedures `DeleteStringAt()` and `ShowArray` to the form class. Generate method bodies for them in the `implementation` section, and complete the program as follows:

```
unit unit_main;

{$mode objfpc}{$H+}
{$ASSERTIONS ON}

interface

uses SysUtils, Forms, Dialogs, StdCtrls;

type
  TStrArray = array of string;

  TfrmTestAssert = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  public
    FArray: TStrArray;
    procedure DeleteStringAt(var anArray: TStrArray; anIndex: integer);
    procedure ShowArray;
  end;

var frmTestAssert: TfrmTestAssert;

implementation

procedure TfrmTestAssert.Button1Click(Sender: TObject);
begin
  SetLength(FArray, 7);
  FArray[0] := 'This '; FArray[1] := 'string '; FArray[2] := 'does ';
  FArray[3] := 'not '; FArray[4] := 'have ' ; FArray[5] := 'six ';
  FArray[6] := 'words';
  ShowArray;
  ShowMessage('Calling DeleteStringAt(FArray, 3)');
  DeleteStringAt(FArray, 3);
  ShowArray;
  DeleteStringAt(FArray, 7); // comment out this line to untest the assertion
end;

procedure TfrmTestAssert.DeleteStringAt(var anArray: TStrArray;
  anIndex: integer);
begin
  Assert(Assigned(anArray), 'nonexistent array given to DeleteStringAt');
  Assert((anIndex >= Low(anArray)) and (anIndex <= High(anArray)),
    Format('index value %d out of range in DeleteStringAt procedure',
      [anIndex]));
  Move(anArray[anIndex+1], anArray[anIndex], (High(anArray) -
    anIndex) * SizeOf(string));
  SetLength(anArray, Length(anArray) - 1);
end;
```

Chapter 19 DEBUGGING TECHNIQUES

```
procedure TfrmTestAssert.ShowArray;
var s: string;
    z: string = '';
begin
  for s in FArray do AppendStr(z, s);
  ShowMessage(z);
end;

{$R *.lfm}

end.
```

19.e Modularising functionality

Encapsulation of program data and code in **classes** is a further development in the direction of modular code which was one of Niklaus Wirth's goals in designing Pascal. The underlying philosophy is that nothing outside the class should influence what goes on within it, except through carefully designed data or event properties, or parameters passed in a constructor or other method. The class should be, as it were, a black box with a limited number of 'feelers' which give the class the ability to interact with the rest of the program code. But these 'feelers' are carefully controlled. Only the minimum needed for communication with classes outside the black box is exposed in the class interface.

Likewise a class should not need to know anything about the internals of other classes it relates to in order to work, and its methods and properties should be designed to communicate using the minimum possible data for interaction. For instance, in the earlier `setdemo` project (see Chapter 12, Section g) we used the form's `OnCreate` event to initialise two sets of digits (we called the sets A and Z), and to set up twenty buttons that the program needed for manipulating the values in sets A and Z.

It would have been possible to write the `OnCreate` handler so that it declared the button arrays needed, created each button, set each button's properties and events and populated the two panels with the buttons. But this would have been poor design, coupling the form's `OnCreate` event with far more detail of button set-up than is good. If we ever needed to set up button arrays elsewhere, to reuse the set-up code we would then have had to extract it from the form's `OnCreate` event, where it does not really belong.

A better design conceptualises functionality in a more modular way. To set up buttons we write a `CreateButtons` method. To initialise sets we write an `InitialiseSets` method. Then our form's `OnCreate` event becomes a matter simply of writing:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  CreateButtons('A');
  CreateButtons('Z');
  InitialiseSets;
end;
```

As well as separating functionality into logically separate 'code modules' (*i.e. methods*) this approach is also elegant and eminently readable. It is obvious from their names what `CreateButtons` and `InitialiseSets` ought to do, and it is far easier to check that `FormCreate` does what it should when its tasks are reduced to a simple list of called methods. This also makes it much easier to spot if one method is missing, or called in the wrong sequence.

Chapter 19 DEBUGGING TECHNIQUES

With the button-creation code neatly separated from the set-initialisation code there is far less chance that either method will interfere with the other in unforeseen ways, and the button-creation code can now be very simply reused and called from elsewhere in the application (*should that ever be required*) since it has been separated into its own method. Lazarus provides at least two tools to help you modularise your code in this way.

19.f Code Observer

Lazarus has a tool called the **Code Observer** (*which is off by default*) which can help in identifying over-long procedures, and other code deficiencies. Long procedures can be a sign of poorly encapsulated functionality, or over-dependence of one part of your code on another part, when less inter-dependence would lead to a cleaner design that is far easier to maintain. The longer and more spaghetti-like your code is, the harder it is to read, understand and to debug. Leopold Kohr's mantra for global economics *small is beautiful* is equally relevant as a principle behind the writing of solid code.

The Code Observer is switched on by going to the *Categories* node of the *Code Explorer* branch of the *IDE Options* dialog ([Shift][Ctrl][O]). Click on the *Categories* node of the *Code Explorer* branch and make sure *Code Observer* is checked, before clicking [OK] to save the new settings (see *Figure 19.4*). You will probably have to scroll down in the treeview at the left of the dialog to reach this branch.

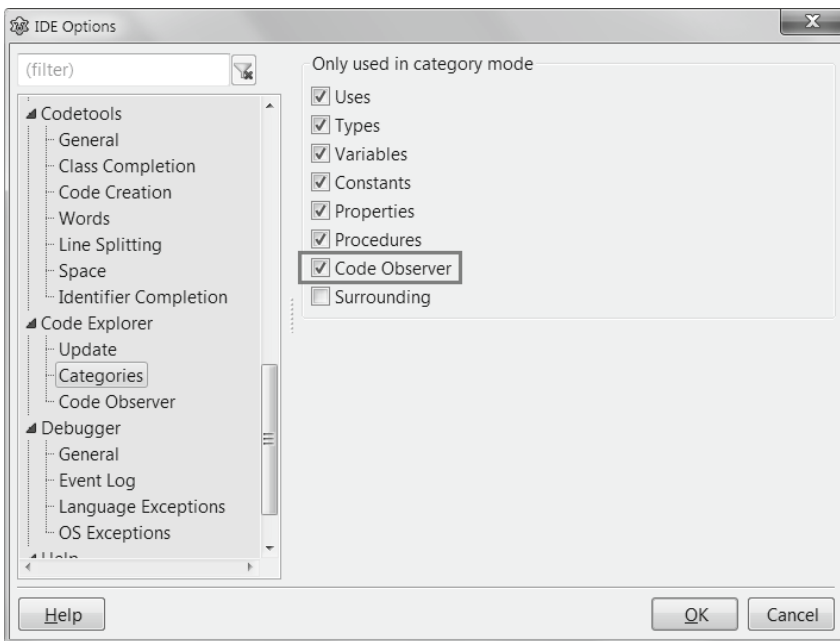


Figure 19.4 The IDE Options Categories page

You also need to click on the *Code Observer* node in the *Options* dialog to open the relevant page (see *Figure 19.5*) where it does not hurt to tick all the categories, even though you might not be bothered about some of them such as *Style*. Nevertheless, it is helpful to sort methods alphabetically, particularly in classes that have lots of methods, since it is so much easier to locate what you are looking for.

Chapter 19 DEBUGGING TECHNIQUES

The important category for this chapter's purpose is *Long procedures*, and over 25 lines is a fairly good indicator. Obviously you will customise code observations to suit your situation and coding style, and avoid being informed of 'deficiencies' that to you are not shortcomings at all.

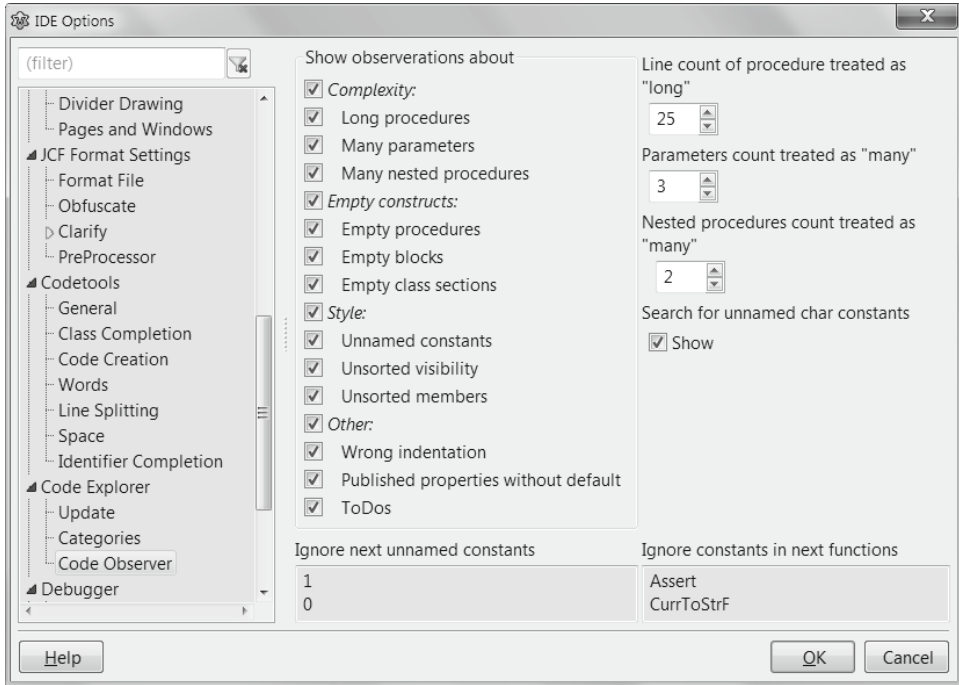


Figure 19.5 The IDE Options Code Observer page

Once you have set the Explorer and Observer options the way you want them, actual observations are viewed in the Code Explorer, accessed via **View | Code Explorer**. Make sure the *Code* tab at the top is selected, not the *Directives* tab. If you load the earlier `compbrowser.lpi` project from Chapter 11, and view the Code Explorer using the illustrated settings, and scroll to the Code Observer node in the Code Explorer's treeview you will see something like what is shown in Figure 19.6.

Three methods are highlighted as being *long*. Clicking on the method name jumps you immediately to the source, so you can examine it and see if it should be reworked, or **refactored** to use the programming term. See the next section for details of how to refactor sections of code.

There is one *unsorted method*. This is quickly rectified. To sort code alphabetically, select the unsorted lines and choose **Edit | Sort Selection...** In the resulting dialog make sure that in the Domain section the Lines radio button is selected. When you press [OK] the line sorting is performed. (Sometimes subsequent adjustment to the indentation of the sorted code are required, if the indentation was a mixture of tabs and spaces).

Chapter 19 DEBUGGING TECHNIQUES

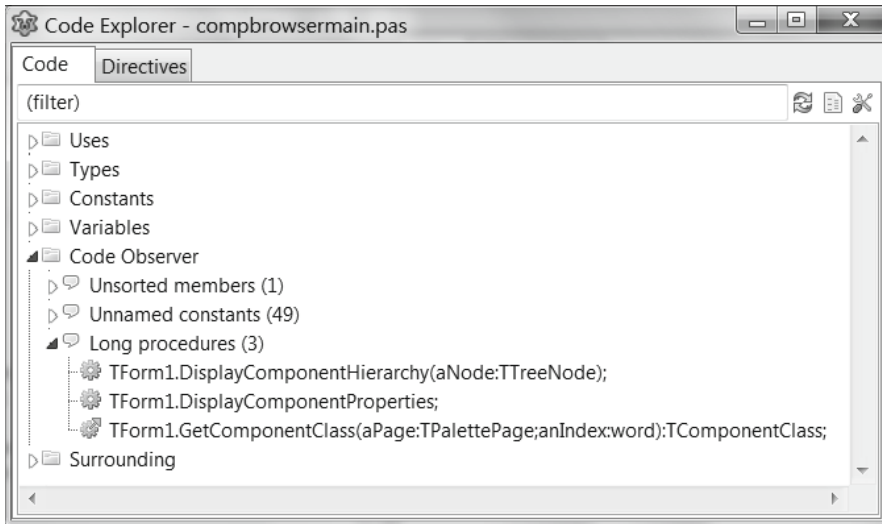


Figure 19.6 The Code Observer node reporting observations about compbrowsermain

There are also 49 *unnamed constants*. Expanding this node shows that many of them are numeric constants, which in this case are fine as they are. Again double-clicking on an entry jumps you to its source declaration or use.

The identification of unnamed constants is very helpful if you want your application to be structured in a way that makes it easy to be translated into other languages. You can use this list as the basis of converting each string literal into a named `resourcestring` suitable for internationalisation. Resourcestrings are a simple and effective way of organising literal strings into named variables that can be processed by internationalisation tools such as **poedit**.

This is best done by creating a unit which holds nothing but resourcestrings in its **interface** section. Such a unit is very simple:

```
unit r_string;
{$H+}

interface

resourcestring
  rsExample = 'This is an example string stored as a resourcestring';
  rsCompBrowserTitle = 'Component Browser';
  // other resourcestring declarations follow here ...

implementation

end.
```

To convert a string literal identified by the Code Observer into a resourcestring, all you need to do is select it in the Editor, right-click to show the context menu and choose *Refactoring* → *Make Resource String...* This shows the *Make ResourceString* dialog (see Figure 19.7).

Chapter 19 DEBUGGING TECHNIQUES

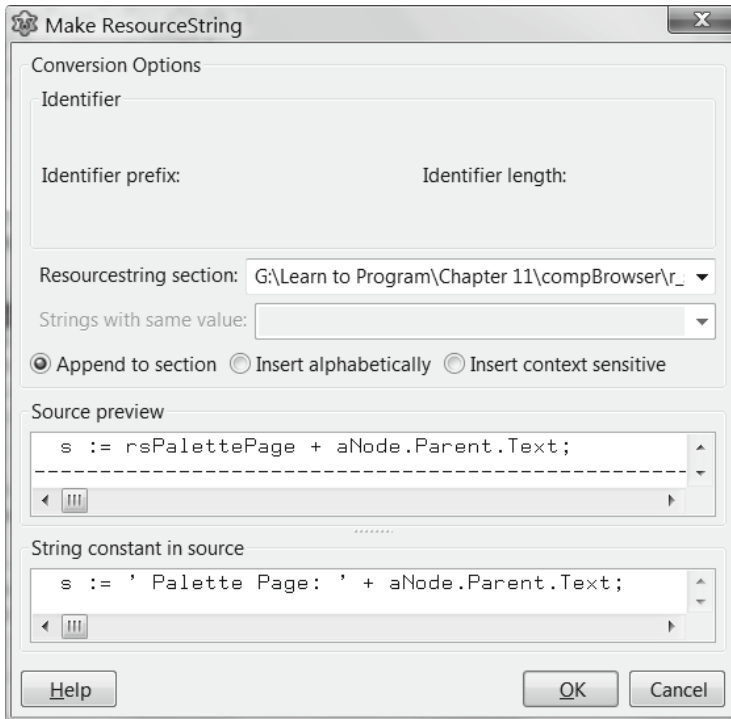


Figure 19.7 The Make ResourceString dialog

Provided you have at least one unit in your project with a `resourcestring` declaration, the IDE will find it and suggest it in the *Resourcestring section* as the location for the new resourcestring it is about to create. Usually the default name suggested for the new resourcestring (*made from rs plus the initial word(s) of the string itself*) is OK and you can just click [OK]. In the case illustrated in Figure 19.7 Lazarus inserts the name of the new resourcestring constant in place of the former string literal, and adds that constant as a new entry in the *Resourcestring section* identified in the dialog. This automated process saves a lot of tedious graft in big projects where there are many such strings that require localising and translating. When a resourcestring unit is compiled Lazarus adds an `.rst` file to the project. In this case as well as an `r_string.ppu`, Lazarus will generate an `r_string.rst` file too, corresponding to the `r_string.pas` source.

In this case Lazarus changed the source from

```
s := ' Palette Page: ' + aNode.Parent.Text;
```

to read as follows:

```
s := rsPalettePage + aNode.Parent.Text;
```

At the same time the relevant section of the `r_strings` unit was changed to the following:

```
resourcestring
rsExample = 'This is an example string stored as a resourcestring';
rsCompBrowserTitle = 'Component Browser';
rsPalettePage = ' Palette Page: ';
```

Chapter 19 DEBUGGING TECHNIQUES

If you feel this has little to do with debugging, recall that every way in which your code becomes more organised and more logically arranged, is a way in which it becomes easier to maintain. Moreover, listing string constants all together in one place enables you to see at a glance if you have duplicate constants. In a large program this is quite likely, though is otherwise not detected. Weeding out duplicates and reducing all duplicates to a single resource string used by various units reduces the overall memory footprint of your program, which is also progress.

19.g Refactoring

A further useful Lazarus tool to help with modularising code is the *Extract Procedure* refactoring tool, which you run by highlighting suitable code and choosing **Refactoring | Extract Procedure...** from the context menu which pops up on right-clicking.

For a simple example, look at the `setdemo_main` unit from the `setdemo` program from Chapter 12, which contained a method called `CreateButtons`. It is a fairly short (17-line) function, so not really a prime candidate for refactoring, but serves to illustrate the point here.

Select the lines in the `CreateButtons` procedure that are underlined in the following code.

```
function TMainForm.CreateButtons (setID: Char): TButtonArr;
const spacing = 10;
      aLeft = 40;
var i: integer;
      b: TSpeedButton;
begin
  for i := Low(TDigits) to High(TDigits) do
  begin
    b := TSpeedButton.Create(Self);
    b.Top := spacing;
    b.Left := aLeft + i * (b.Width + spacing);
    b.Caption := IntToStr(i);
    b.Tag := i;
    b.Name := setID + 'set' + b.Caption;
    case setID of
      'A' : b.Parent := pnlA;
      'Z' : b.Parent := pnlZ;
    end;
    b.OnClick := @ButtonClick;
    Result[i] := b;
  end;
end;
```

When you right-click and choose **Refactoring | Extract Procedure...** you see the *Extract Procedure* dialog shown in Figure 19.8.

Change the *Name of new procedure* field from `NewProc` to `SetButtonProperties`, leaving the *Sub Procedure* radiobutton selected. When you click the [Extract] button, Lazarus inserts a sub-procedure named `SetButtonProperties` into the code of `CreateButtons`, which now looks like this:

Chapter 19 DEBUGGING TECHNIQUES

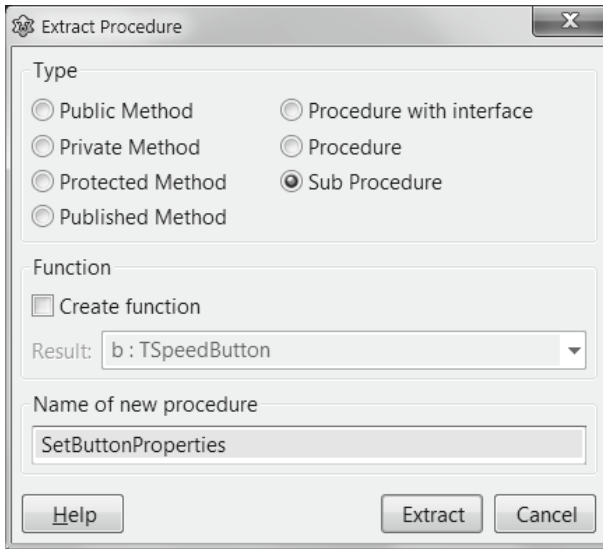


Figure 19.8 The Extract Procedure dialog

```
function TMainForm.CreateButtons(setID: Char): TButtonArr;
const spacing = 10;
        aLeft = 40;
var i: integer;
        b: TSpeedButton;

procedure SetButtonProperties;
begin
    b.Top := spacing;
    b.Left := aLeft + i * (b.Width + spacing);
    b.Caption := IntToStr(i);
    b.Tag := i;
    b.Name := setID + 'set' + b.Caption;
    case setID of
        'A' : b.Parent := pnlA;
        'Z' : b.Parent := pnlZ;
    end;
    b.OnClick := @ButtonClick;
end;

begin
    for i := Low(TDigits) to High(TDigits) do
        begin
            b := TSpeedButton.Create(Self);
            SetButtonProperties;
            Result[i] := b;
        end;
end;
```

In this case the improvement, if any, is slight. But it is one of the first code examples we have seen that is extensive enough to serve as an example of refactoring, even though here there is little to be gained. Certainly the logic in the for loop is easier to pick out immediately on reading the code. Is an improvement in readability worth the effort of refactoring? Yes, and on returning to this code some time later when you need to modify it, you will be glad you took the trouble to make the program logic as clear as you could.

Chapter 19 DEBUGGING TECHNIQUES

Of course you can go overboard and refactor code that really does not need it. However, the tendency is usually to write code that is too long and too dense, which can combine logical routines that would be better written separately and independently of each other. Programming involves careful thought about how the sections of code you design will communicate and interact with each other. It also involves knowledge of how your code should communicate with and exploit the code written by others in the support libraries (*FCL, RTL, LCL*).

Lazarus has been designed to free you from concern about communication with the OS, and you should avoid all OS calls if you want your code to be cross-platform. All the platform-specific LCL code is cleverly 'hidden' in widgetset units that keep it completely separate from the generalised LCL code interface which is all you are normally aware of (*unless you progress to the point where you are contributing patches to the LCL*).

If you come from a Delphi background, or if you port Delphi programs to Lazarus, this is often an issue, since you may be used to making Windows API calls to do something that Delphi's VCL did not specifically provide for. In Lazarus, such an approach dooms your application to be Windows-specific, which these days is not usually what users want.

19.h Watching variable values

There are several ways to watch the changing 'values inside' the routines you develop, and unit tests are one way to formalise that watching for the appearance of unexpected values. However, other strategies are often desirable as well. The following sections look at some of the possibilities for tracking the value of variables, functions and properties when you are getting the "wrong answer".

You don't want your program users to see any of this variable information, so variable watching strategies have to apply during development and yet be invisible in the release. One way to achieve this is to include variable-display code that can be switched on and off depending on the situation.

19.i The `{$DEFINE DEBUG}` compiler directive

The FPC has a `{$DEFINE ...}` directive that allows you to define an identifier (`DEBUG` is commonly used) that if present will activate some debugging code wrapped in an `{$IFDEF ...} ... {$ENDIF}` block. Lazarus provides the shortcut `[Shift][Ctrl][D]` which shows a dialog that wraps selected code in the directive(s) you select. `DEBUG` is one of the predefined choices in this dialog, also accessed via **Source | Enclose in \$IFDEF...**

Suppose we need a function that reverses a string, and do not know that the RTL already provides such a function. You might write such a function like this:

```
function StringReverse(const s: string): string;
var len, p: integer;
begin
  len:= Length(s);
  SetLength(Result, len);
  for p in [1..len] do   Result[len - p] := s[p];
end;
```

Wisely you decide to debug this function in a test program, a new GUI project called `string_reverse`. It has a label and an edit dropped on the main form, and the form has a **private** method called `StringReverse`. The edit has an event handler for its `OnChange` event. The main form unit needs to have `LazLogger` added to the **uses** clause in order to use the `DebugLn()` procedure, which writes output to the console (*or Terminal*). This means on Windows that the main program file needs to have a compiler directive instructing that this is a console app (*otherwise Windows developers will not see the debug output*).

Chapter 19 DEBUGGING TECHNIQUES

The `.lpr` file, `test_reverse` will be as follows:

```
program test_reverse;  
  
{$mode objfpc}{$H+}  
  
{$IFDEF WINDOWS}  
  {$appstype console}  
{$ENDIF}  
  
  uses Interfaces, Forms, main;  
  
{$R *.res}  
  
begin  
  RequireDerivedFormResource := True;  
  Application.Initialize;  
  Application.CreateForm(TForm1, Form1);  
  Application.Run;  
end.
```

The `main.pas` form unit will be as follows:

```
unit main;  
  
{$mode objfpc}{$H+}  
  
{$DEFINE DEBUG}  
  
interface  
  
uses SysUtils, Forms, StdCtrls, LazLogger;  
  
type  
  TForm1 = class(TForm)  
    Edit1: TEdit;  
    Label1: TLabel;  
    procedure Edit1Change(Sender: TObject);  
  private  
    function StringReverse(const s: string): string;  
  end;  
  
var  
  Form1: TForm1;  
  
implementation  
  
{$R *.lfm}  
  
procedure TForm1.Edit1Change(Sender: TObject);  
begin  
  Label1.Caption:= StringReverse(Edit1.Text);  
end;
```

Chapter 19 DEBUGGING TECHNIQUES

```
function TForm1.StringReverse(const s: string): string;
var
    len, p: integer;
begin
    len:= Length(s);
    SetLength(Result, len);
    for p in [1..len] do
        begin
            {$IFDEF DEBUG}
            DebugLn(['len: ',len,'; p: ',p,'; len - p: ',len-p]);
            {$ENDIF}
            Result[len - p] := s[p];
        end;
    end;
end.
```

When you compile and run this test program you will see console output as follows
len: 1; p: 1; len - p: 0
just before the app raises an exception, caught here in the debugger (see Figure 19.9).

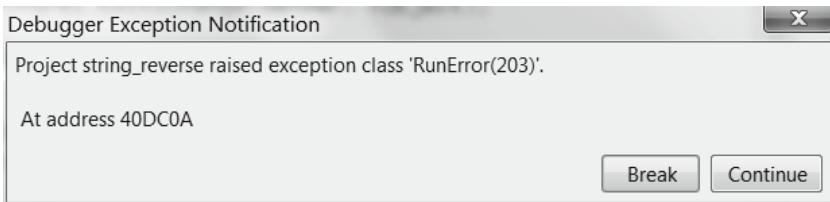


Figure 19.9 test_reverse causing an access violation

The versatile `DebugLn()` procedure has many overloaded variants. Here it shows us that `(len-p)` has the unwanted value zero; unwanted because that is not a valid index for an ansistring character. This is an example of an **off-by-one bug**, not an uncommon type of insect. Our procedure needs amending so that the line of code that assigns the indexed character in `Result` reads not as above but rather as follows:

```
Result[len + 1 - p] := s[p];
```

Having made this change, and verified that the amended function is good to go, we can undefine the debug symbol, and then the `DebugLn()` code will no longer be compiled into the final app. This is done by removing the `{$DEFINE ...}` altogether, or with either of the following amendments to the `{$DEFINE debug}` directive

```
{$ DEFINE debug} // inserting a space turns a directive into a plain comment
{$UNDEF debug} // this directive un-defines the debug symbol
```

As with all Pascal code, the capitalisation of the debug identifier is immaterial. An alternative to use of the `{$DEFINE ...}` and `{$IFDEF ...}` directives is to leave all `DebugLn` statements in place but simply comment them out manually once debugging is complete. When returning to code for later maintenance, the debug support is already written, and only requires un-commenting. The default shortcut for un-commenting code is `[Shift][Ctrl][U]`.

Chapter 19 DEBUGGING TECHNIQUES

19.j Console debug functions

In the `LazLogger` unit Lazarus provides a number of debug functions that display strings to the console, or convert numerous other types to strings for display in a console window alongside the GUI display of the main program. (As always, Windows applications will require the `{$apptype console}` directive to see this console in addition to the GUI windows).

A few of the most widely used `LazLogger` functions are given here.

All are overloaded (some quite heavily) which means that `DebugLn()`, for instance, can display between 1 and 18 strings in a comma-separated list! These functions accept either a comma-separated list of strings

```
DebugLn('one', 'two', 'three');
```

or an array of `const` such as

```
DebugLn(['varOne = ', varOne, ', ', varTwo = ', varTwo]);
```

or a format string followed by an array of arguments such as the `Format()` procedure accepts:

```
DebugLn('intVar = %d, stringVar = %s', [intVar, stringVar]);
```

You can see these functions have been designed to be as versatile as possible. They include:

- `DebugLn()` - roughly analogous to `WriteLn()`
- `DbgOut()` - roughly analogous to `Write()`
- `DebugLnEnter()` - identical to `DebugLn()`, but subsequent output is indented
- `DebugLnExit()` - identical to `DebugLn()`, but its own and subsequent output is un-indented

There are a number of string conversion functions provided for converting special types into their string representation for debug output. Three of the available functions are:

- `DbgStr()` - which returns a 'normal' string, except that if it contains unusual characters then each non-printing character is converted into `#nnn` format.
- `DbgS()` - which converts ordinal types including all integral types and booleans into strings. It also converts pointers and extended values. Additionally `TSize` and `TPoint` parameters are converted to show their `x` and `y` values and `TRect` parameters are converted to show their `Top`, `Left`, `Right` and `Bottom` values. Several other less common types are also accepted as parameters.
- `DbgSName()` - which returns the `Name` and `ClassName` of classes such as components.

If the above `StringReverse()` function is amended as follows:

```
function TForm1.StringReverse(const s: string): string;
var len, p: integer;
begin
  {$IFDEF debug}
  DebugLnEnter('StringReverse received parameter "%s"', [s]);
  {$ENDIF}
  len:= Length(s);
  SetLength(Result, len);
  for p in [1..len] do
    begin
      {$IFDEF debug}
      DebugLn(['len: ', len, '; p: ', p, '; len - p: ', len-p]);
      {$ENDIF}
      Result[len + 1 - p] := s[p];
    end;
  {$IFDEF debug}
  DebugLnExit('Exiting StringReverse, final value of p is %d', [p]);
  {$ENDIF}
end;
```


Chapter 19 DEBUGGING TECHNIQUES

then you will see output like that shown in Figure 19.10.

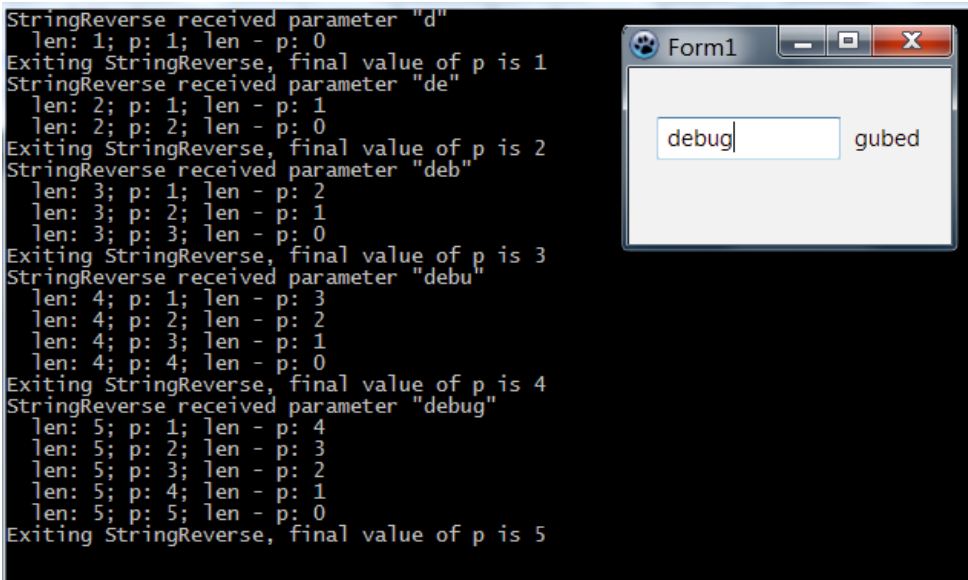


Figure 19.10 DebugLn output for the ReverseString function

19.k Program interruption

Occasionally it is helpful to interrupt a program to check output in the middle of a sequence of code statements. A modal dialog prevents further code sequences from running until the dialog is closed. One quick-and-dirty way to interrupt a running program to check value(s) at that instant is to insert a `ShowMessage()` or `ShowMessageFmt()` call at a suitable point in your code (a call you will later delete or uncomment, after debugging is finished). This requires the `dialogs` unit in your `uses` clause.

Add `dialogs` added to the `uses` clause and change the `StringReverse()` function to the following:

```
function TForm1.StringReverse(const s: string): string;
var len, p: integer;
begin
  len:= Length(s);
  SetLength(Result, len);
  for p in [1..len] do
    begin
      ShowMessageFmt('Value of (len - p) is %d',[len - p]);
      Result[len - p] := s[p];
    end;
  end;
end;
```

Chapter 19 DEBUGGING TECHNIQUES

When you run the program this time, just before an exception is raised you will be presented with a dialog such as in Figure 19.11.

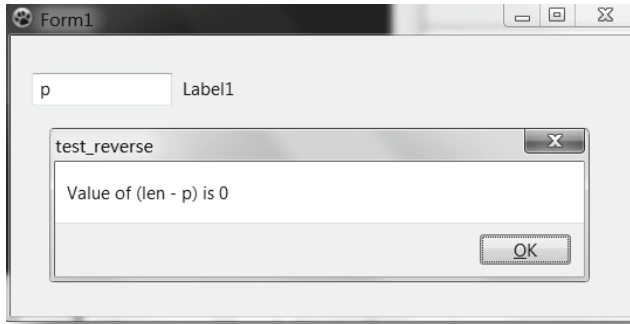


Figure 19.11 ShowMessageFmt used as a debugging aid

19.1 Logging debug output to a file

Lazarus has a `DbgAppendToFile()` routine in the `LCLProc` unit which takes two parameters, a logfile name, and a debug-string. It enables you to log a permanent, if simple file record of debug output.

Add `LCLProc` to the `uses` clause, and change the `ReverseString()` function to the following, before re-compiling and running the project.

```
function TForm1.StringReverse(const s: string): string;
var len, p: integer;
    tmp: string='';
begin
    len:= Length(s);
    SetLength(Result, len);
    for p in [1..len] do
        begin
            Result[len + 1 - p] := s[p];
            AppendStr(tmp, Format('p:%d,s[p]:"%s" ', [p,s[p]]));
        end;
    DbgAppendToFile('debug.txt', tmp);
end;
```

Typing 'rat' into `Edit1` produced the following `debug.txt` log file:

```
p:1,s[p]:"r"
p:1,s[p]:"r" p:2,s[p]:"a"
p:1,s[p]:"r" p:2,s[p]:"a" p:3,s[p]:"t"
```

Logging may be required to debug both system and Lazarus message methods and message events. More sophisticated logging is usually needed for that, such as provided by two somewhat different open source packages. The second one listed is not principally a logging library but an ORM framework for database programming. However it includes a versatile logging implementation. Details about them can be found on the following wiki pages:

<http://wiki.lazarus.freepascal.org/MultiLog>
<http://wiki.lazarus.freepascal.org/tiOPF>

Chapter 19 DEBUGGING TECHNIQUES

19.m The debugserver tool

A very useful debugging tool, found in the `Lazarus\tools\debugserver` folder on a Windows installation (and in an equivalent folder on other systems) comes as a ready-to-build `debugserver.lpi` project. It requires you to add the `dbugintf` unit to the **uses** clause of your project, which provides eleven `SendXXX` procedures which are used much like `DebugLn()`, only instead of writing text output to the console they send text to the debugserver window (which can be set to be 'Always on top'), and which has more functionality than many console windows. It is particularly suited for checking the sequence and timing of message events since it records a timestamp for every debug message received. The window contents can be saved to a log file, or the messages can be cleared. It is a good example of a focused tool that does one task very well, and you have the source code to see how it achieves this.

Load and build the `debugserver` project. Use your system file browser to run the `debugserver` executable and set its window to be *Always on top*. Reload the `string_reverse` project in Lazarus, add `dbugintf` to the **uses** clause, and change the `ReverseString()` function as follows.

```
function TForm1.StringReverse(const s: string): string;
var
    len, p: integer;
begin
    SendMethodEnter('StringReverse');
    len:= Length(s);
    SetLength(Result, len);
    for p in [1..len] do
        Result[len + 1 - p] := s[p];
    SendDebugFmt('result is "%s"', [Result]);
end;
```

When you run the project you will see the output of Figure 19.12 in the debugserver window.

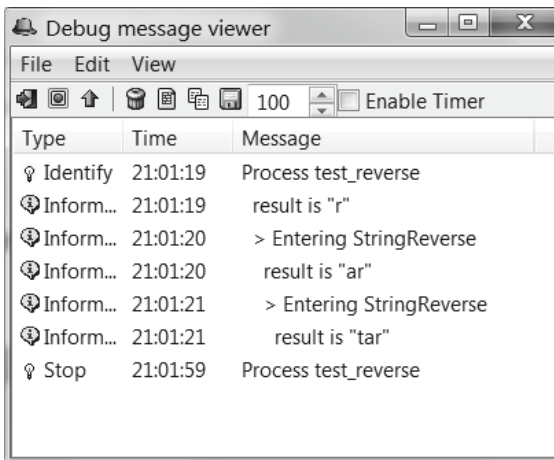


Figure 19.12 The Debug message viewer listening for reverse_string debug messages

On Linux the program performs flawlessly. A slight drawback is that on Windows if messages are sent too close together in time (say successively in a tight loop) they get lost. In Figure 19.12 you'll see that the very first `> Entering StringReverse` message was lost (presumably because it was chronologically too close to the opening `Process test_reverse` message).

Chapter 19 DEBUGGING TECHNIQUES

19.n Getting the compiler to catch bugs

Each Lazarus project gets its own project settings. Among these are options to instruct the FPC exactly how the compilation of the source should be done. You access the *Options for Project:* dialog from the **Project | Project Options ...** menu ([Shift][Ctrl][F11]). The treeview on the left of this dialog has two main branches, *Project Options* and *Compiler Options*. It is the pages under *Compiler Options* that are relevant for debugging.

The *Code Generation* node when clicked shows a page which includes a group of *Checks* (see Figure 19.13).

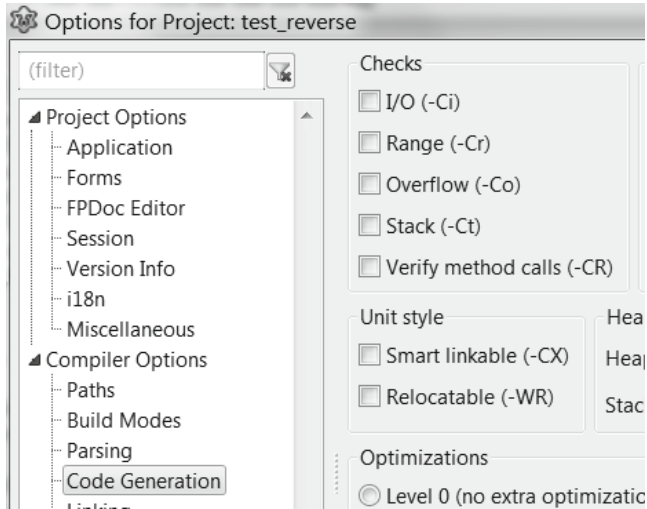


Figure 19.13 Part of the Code Generation page of the Project Options dialog

The *Checks* section offers five possible checks the compiler can perform (*extra compiler-generated code is added to your project to achieve this*).

It is a good idea to turn on Range checking during program development. This will generate extra code that checks that array and string indexes are within bounds (*similar to the `Assert` example given in Section d*). An out-of-range index in your program may cause memory corruption or an unexplained crash. If you turn on the *Range* check for the original `ReverseString()` example with the off-by-one bug, and recompile it, you will find when it runs it stops with a different runtime error (*201 rather than 203*), and if you choose the [Continue] button a new exception dialog appears as in Figure 19.14.

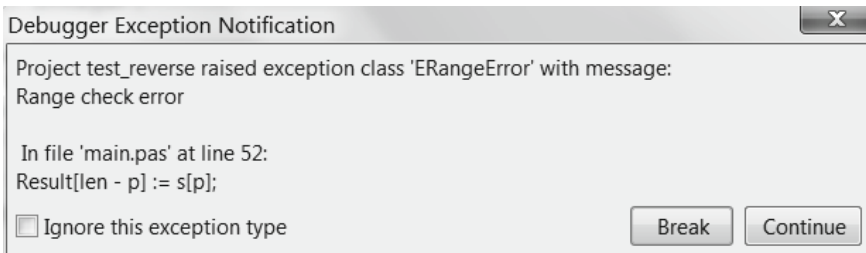


Figure 19.14 Compiler-added code showing up a range check error

Chapter 19 DEBUGGING TECHNIQUES

If you now choose [Break] the debugger will halt the program execution and take you to the exact location in the source where this range check error was produced.

Similar sorts of compiler-generated code is added to your project if any of the other Checks are enabled. Because they increase the size of the final executable, and slow the operation of the program fractionally most developers remove such checks once the debugging phase is complete, before compiling the release version.

19.o The heaptrc unit

The *Linking* page of the *Project Options* dialog includes several options related to debugging, including a checkbox in the *Debugging* section labelled *Use Heaptrc unit (check for mem-leaks) (-gh)* (see Figure 19.15).

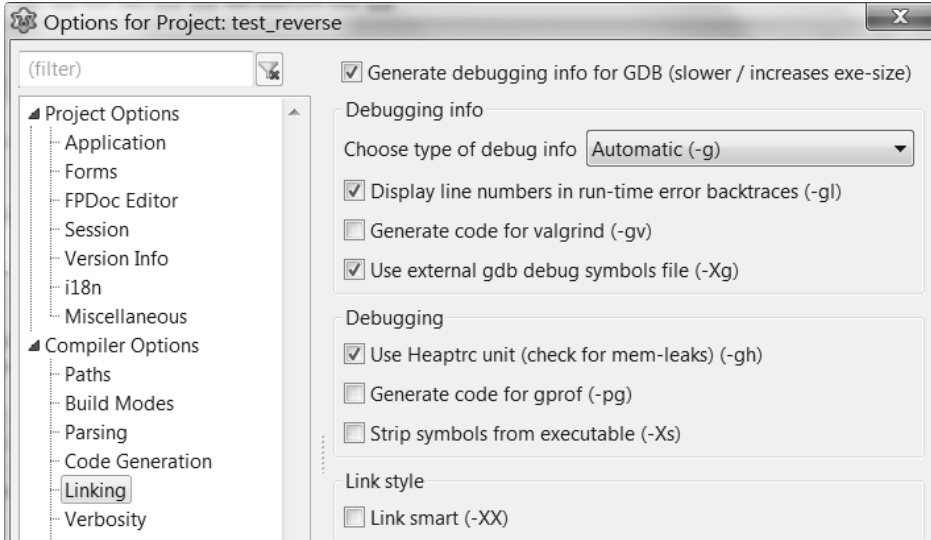


Figure 19.15 Part of the Linking page of the Project Options dialog

This is a vital tool for developers, enabling you to identify otherwise disastrous memory leaks. Such leaks not only point to faulty code (*sections of allocated memory not being properly deallocated*), but can highlight faulty program logic, poorly coded manipulation of pointers and similar shortcomings. There is really no reason not to turn on the use of the `heaptrc` unit to identify such program flaws.

The unit gives a report at program termination, as additional console output for console programs, or in a modal dialog (*or series of dialogs*) for GUI programs. If no memory problems are found the report includes the statement

```
0 unfreed memory blocks: 0
```

Unfortunately the modal dialog is still labelled *Error* in this case (*though there is no memory allocation error*). See Figure 19.16

Chapter 19 DEBUGGING TECHNIQUES

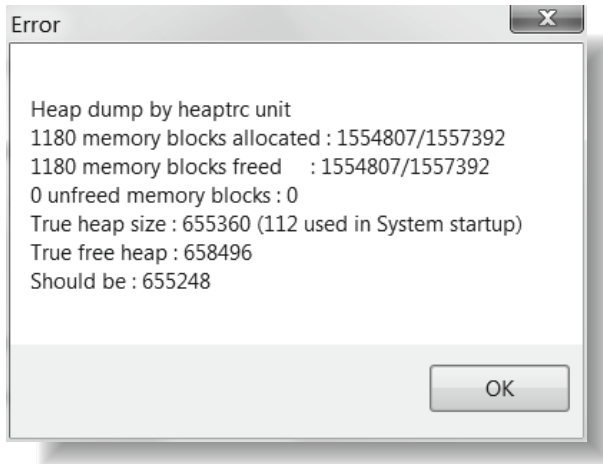


Figure 19.16 Heaptrc reporting on a program which correctly deallocates all allocated memory

You activate the heap tracing functionality either by checking the `Use heaptrc` unit option in the *Project Options* dialog as shown above, or by putting `heaptrc` as the first entry in your program's `uses` clause. As an alternative to showing `heaptrc` output in a modal dialog (*or on a console*) when your program ends, you can opt to log the information to a file. If there are many deallocation errors in a GUI program this is the preferred option, since otherwise you are forced to click away possibly dozens of modal dialogs listing reams of memory error information.

To do this simply add

```
SetHeapTraceOutput ('/path/to/heaptrace.trc');
```

to the main program file.

To give a simple example, start a new Lazarus GUI project named `trace`, with a main form unit called `mainform`. Set the *Use heaptrc* option as outlined above. Generate an `OnCreate` event for the form, and complete it so the main form unit looks like the following:

```
unit mainform;
{$mode objfpc}{$H+}

interface
uses
  Forms, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.lfm}

procedure TForm1.FormCreate(Sender: TObject);
var lbl: TLabel;
begin
  lbl:= TLabel.Create(nil);
  lbl.Caption:= 'This is a dynamically created label';
  lbl.Parent:= Self;
end;
end.
```

Chapter 19 DEBUGGING TECHNIQUES

Run this program, and after closing it you will see a series of modal dialogs, of which the first will look like Figure 19.17.

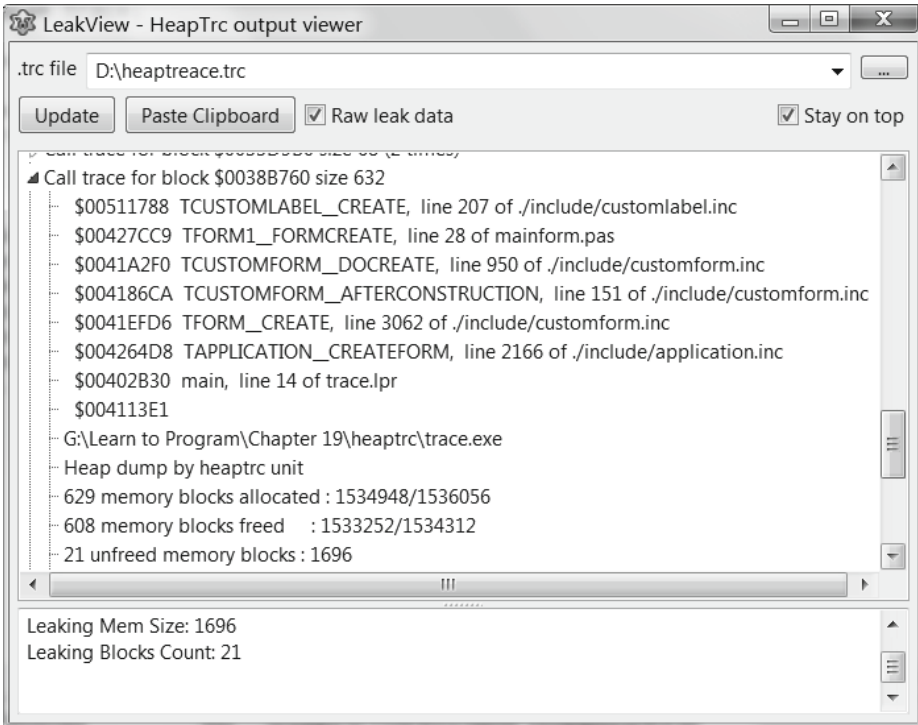


Figure 19.17 Heaptrc output resulting from not freeing a component

The `TLabel` component `lbl` was created in the form's `OnCreate` event, but never freed. This leads to 21 memory blocks (1696 bytes of memory) unallocated and orphaned at the end of the program. Try running the program again, this time adding a `SetHeapTraceOutput()` call to the main program file. For instance, the main program file might look like the following:

```
program trace;  
  
{ $mode objfpc } { $H+ }  
  
uses Interfaces, Forms, mainform;  
  
{ $R *.res }  
  
begin  
    SetHeapTraceOutput('d:\heaptrace.trc');  
    RequireDerivedFormResource := True;  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

Running the program will now generate the specified `.trc` file rather than show a modal *Error* dialog at program termination. You can examine this `.trc` file using the **Tools | Leak View** menu option. Click on the ellipsis at the top of this tool to locate and open the generated `.trc` file (see Figure 19.18).

Chapter 19 DEBUGGING TECHNIQUES

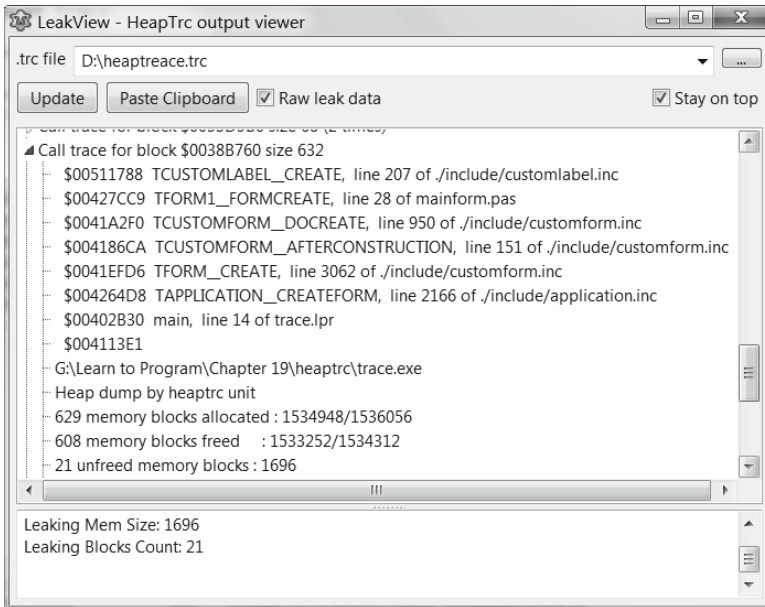


Figure 19.18 The LeakView tool showing line number information

The Leak View tool parses the `.trc` file listing each memory deallocation fault in a separate node. Expanding the nodes shows fuller information, including source code line numbers. Forgetting to free a large object such as an instantiated component causes quite a number of deallocation errors. The node expanded in Figure 19.18 mentions `customlabel.inc`, and cites line 28 of the source:

```
lbl.Parent:= Self;
```

This is a good indication that it is the label (or rather, not freeing it after use) that is the source of the leak.

19.p The gdb debugger

You may wonder why the debugger installed with Lazarus is not mentioned until the end of this chapter. One reason is that **gdb**, being a third party tool, and not designed for debugging Pascal programs (*it is much more at home in the world of C and C++ programs*) is less than an ideal companion for Lazarus, and so at times requires kludges and hacks to provide the full information you might want from it.

A cross-platform debugger has to take account of the different executable formats (*COFF, PE, ELF and variants*) and the different symbol table information formats (*STAB, DWARF*) that Lazarus/FPC programs might use. Also `gdb` is being continuously developed itself, and so is changing perhaps as fast as Lazarus changes.

Although there are drawbacks to using this C-oriented tool to debug Pascal-generated executables `gdb` is a very capable debugger, and allows Lazarus programmers to watch variables, view debug output, set breakpoints, step through code line by line and view disassembly of Pascal code, and the level of its integration with Lazarus (*considering it is an entirely independent project having no obligation to the Lazarus project, nor orientation to its needs*) is remarkable. In view of this it is not surprising to find a few shortcomings in its display of properties etc.

Chapter 19 DEBUGGING TECHNIQUES

To run a program using the debugger it must not only be installed (*which is the default for a new Lazarus installation*), but the program must be compiled with debugging information. This makes the executable much larger (*by a factor of about 10 in many cases*). For this reason you may prefer to get the compiler to generate the debugging information in a file separate from the executable.

In the *Linking* page of Project Options (see Figure 19.15) you must **check**

Generate debugging info for GDB (slower / increases exe-size)

and you must make sure the following two options are **unchecked**

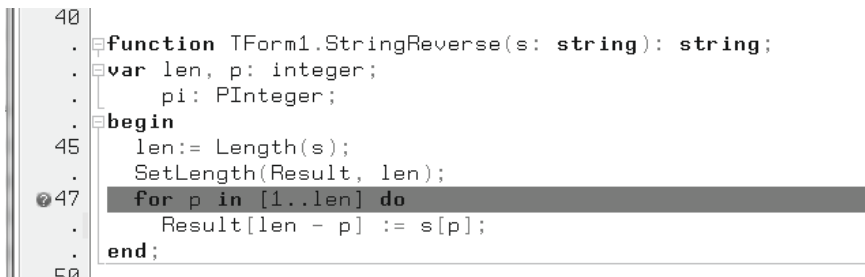
Strip symbols from executable (-Xs)

Link smart (-XX)

On the *Code Generation* page of the *Project Options* dialog the Optimizations radio group should be set to Level 0 or Level 1, but not higher.

The debugger has several windows, of which the most useful for beginners are the Watches ([Ctrl][Alt][W]) and Local Variables ([Ctrl][Alt][L]) windows, accessed via **View | Debug Windows**.

To run a project under the debugger you usually first set a breakpoint, or use **Run | Run to cursor** ([F4]), which behaves as if you had set a source breakpoint in your source where the cursor is currently located. To set a source **breakpoint** (*a source location where the debugger will pause program execution persistently, as opposed to a run-to-cursor execution pause which is attached only to the cursor position*) you click on the Editor gutter to the left of the source line where you want to break. This places a red (?) icon in the gutter and highlights the source code line in red (see Figure 19.19, the highlight being in gray, however).



```
40 . function TForm1.StringReverse(s: string): string;
.   var len, p: integer;
.       pi: PInteger;
.   begin
45     len:= Length(s);
.     SetLength(Result, len);
47     for p in [1..len] do
.       Result[len - p] := s[p];
.     end;
```

Figure 19.19 A source breakpoint set at line 47

Returning to the earlier `test_reverse` project, if you set a breakpoint as shown in Figure 19.19 in the original `StringReverse` function and **Run | Run** the program ([F9]) it will now stop at that line in the source.

Use the **View** menu to show the *Watches*, *Local Variables* and *Call Stack* windows, and add ([Ctrl][A], or use the [+]) toolbutton) to add the variables `p` and `len` to the Watch List. If you now step through the source using [F8] you see the values of the variables changing. Figure 19.20 shows typical output.

Chapter 19 DEBUGGING TECHNIQUES

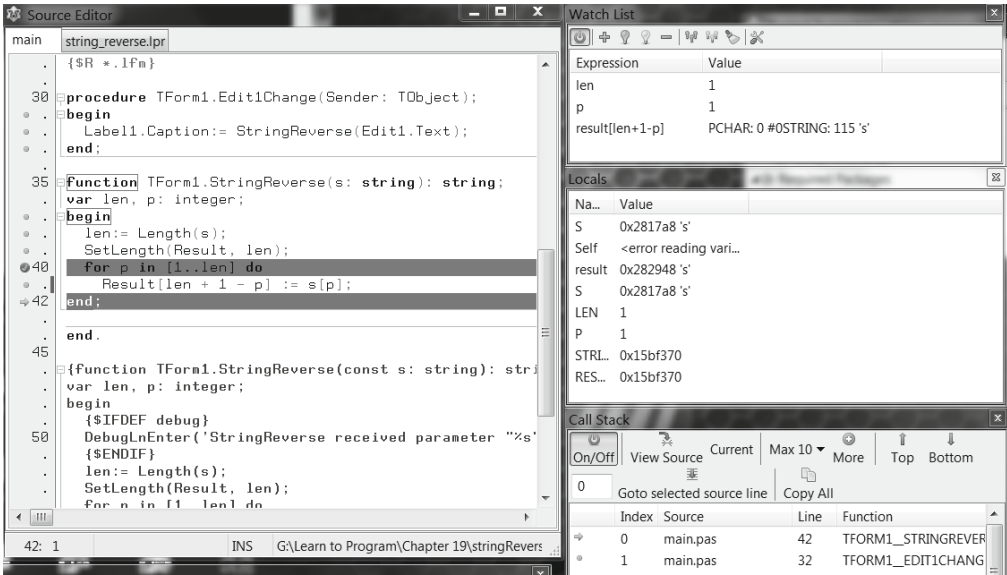


Figure 19.20 Three debug windows open while stepping past a breakpoint

Fortunately the debugger has a forum section all to itself which can be found here:
<http://www.lazarus.freepascal.org/index.php/board,12.0.html>

This, together with the wiki page:

http://wiki.lazarus.freepascal.org/GDB_Debugger_Tips

are the best resources for exploring the capabilities of the debugger, or for getting help in using it from people with more experience.

With this powerful tool, as much as with any part of Lazarus you will have fun. You will also find that hovering the mouse over variables in your source code provides an automatic popup hint window showing the current variable value, when running under the debugger.



Chapter 20 FURTHER RESOURCES

20.a Books about Pascal and Lazarus

It is impossible within the compass of an introductory book to cover even a fraction of what a beginning programmer needs to learn. Nothing of substance has been written here about databases, web access, data transfer technology and networking protocols, threads, interfaces, serial, parallel and USB ports, messaging, graphics and games, mobiles, tablets and touch/gestures, dlls and shared objects, component writing, generics, ... But many of these topics would require a book to themselves.

In addition to the internet resources indicated in Chapter 1 there are a host of books (*many older and fewer recent ones being published*) about all aspects of programming, including specific volumes devoted to Pascal or Delphi. Since Pascal is not a 'hot' language (*many regarding it disdainfully as useful only for teaching, if that*) you can often find second-hand or remaindered copies of books about Pascal and Delphi, which even if ten years old are often surprisingly relevant. Few, if any, Pascal books are available in high street book shops – an online search is the best place to begin.

Lazarus, The Complete Guide (*M Gaertner, M van Canneyt, S Heinig, F Monteiro de Carvalho, J Braun, I Ouedraogo*) is the only Lazarus reference book available at the time of writing. It is published in German and in English (*ISBN 978-94-90968-02-1*).

Essential Pascal (*M Cantù*) is a good short (*140 page*) introduction to the Pascal language of Delphi, and nearly all of the book is applicable to Lazarus/FPC.

It can be ordered at:

<http://www.lulu.com/content/2398448>

Pascal authors to look out for (*this is not by any means an exhaustive list*) writing in English include:

Charlie Calvert
Marco Cantù
Jeff Cogswell
Jeff Duntemann
Cary Jensen
Mitchell Kerman
Ray Lischner
Brian Long
Tom Swan
Bob Swart
Danny Thorpe

Several of these authors, even those not currently in print, have blogs or online resources available. Of them all, probably Marco Cantù writes most helpfully for beginners. He also contributes to Blaise Pascal Magazine, and to Embarcadero Delphi teaching events, along with Cary Jensen and Bob Swart. Several of the FPC and Lazarus core developers contribute articles to Blaise Pascal Magazine and the (*German*) Toolbox magazine, among them Michaël van Canneyt and Felipe Monteiro de Carvalho.



HISTORY OF THE COMPUTER

History of the computer

Presumably people have always counted using their fingers, which may be why our arithmetic nowadays is decimal – based on the number 10. A friend of mine amazed me with his enjoyment in using a system based on the number 14 and explained that it worked just as well. It worked.

The French mathematician Blaise Pascal enjoyed playing dice, and gambling. Naturally he wanted to know when he might win. In his efforts to unravel the probability of a specific dice roll outcome he began to develop a new science: statistics. He found that his discoveries did not help in predicting specific outcomes, only what the average outcome would be given many dice rolls. Although he failed to win a fortune through his new insights at gambling, his work was the beginning of statistical mathematics. (You can read more of his story in Blaise Magazine, issue 14).

In past eras people used pebbles or sticks, as well as their fingers, when counting. Bones were used in Stone Age times as tally-sticks. These were in fact very simple tools for making calculation easier. Many believe that later on with the invention of the abacus, the first real computing device was born.



Figure 1: The Abacus

The word abacus (plural, abaci) is derived from the Greek Abax, meaning “a table or board covered with dust” (a finger could write in the dust, or tally up addition marks).

Abacus remains dating from about 3000 years before Christ have been uncovered.

The first abaci consisted of stones or tablets in which grooves had been cut. A number of small stones could be pushed to one or other side of the groove. In this way numbers could be easily counted and remembered, and simple addition and subtraction calculations could be carried out.

In China in the first abacus appeared about the year 12 AD. It was constructed as a wooden frame containing several rows of beads on wires or rods (Figure 1). We presume that Roman traders from Asia Minor brought this idea to the Chinese as they travelled.

The modern abacus is a very convenient and simple calculator and it is still in active use in many parts of the world.

HISTORY OF THE COMPUTER

Liber Abaci (also written as Liber Abbaci, published in AD 1202) is a notable book on mathematics by Leonardo of Pisa, who was later known by his nickname Fibonacci. In this work Fibonacci introduced the Hindu-Arabic numerals into Europe (0, 1, 2 and so on), which form such an important part of our decimal system today, and which helped to wean Europe away from the Roman numerals (I, II, III, IV and so on) which had dominated arithmetic in Europe until then. Fibonacci had learned this notation while studying with north African Arabs who stayed with his father, Guglielmo Bonaccio, an Italian merchant.

Liber Abaci was not the first book to describe Arabic numerals for Western readers. The first was the Codex Vigilanus (completed in AD 976). The first French pope, Pope Sylvester II, (pope from 999 to 1003 AD) energetically promoted Arabic mathematics and astronomy, and reintroduced the abacus to Europe. Merchants and academics, and eventually the public at large became convinced of the superiority of the new numerals over the cumbersome Roman ones.

Liber Abaci means: "Calculation Book" although it is sometimes mistranslated as "The Book of the Abacus". Sigler writes that this is a misconception, since describing calculation methods without use of the abacus is the book's entire theme. Even centuries after the book's publication 'algorists' (followers of the calculation methods described in Liber Abaci) continued to be in conflict with 'abacists' (the traditionalists who stuck with use of the abacus in combination with Roman numerals).

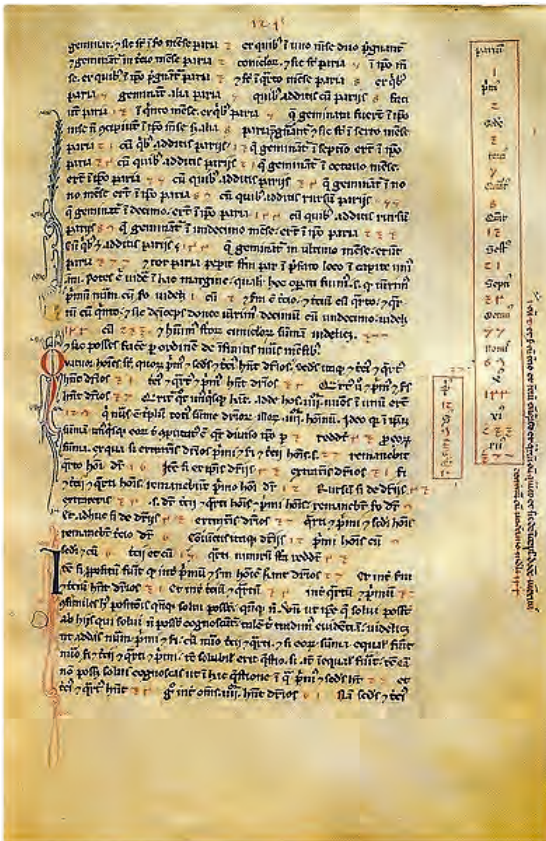


Figure 2: A page from the Liber Abbaci

HISTORY OF THE COMPUTER

Around 1600.

John Napier, inventor of the logarithm,

designed a hand-held calculation device with which you can make multiplications and divisions. This device is known as

Napier's rods (or *Napier's bones*).

The Scotsman Napier also introduced the logarithm.

The **logarithm is a mathematical function**, usually written as the abbreviated term 'log'. The logarithm of a number is a unique value, the exponent or power to which a particular base must be raised to form the original number. For instance 10 must be raised to the power 2 to make 100. Thus the log₁₀ of 100 is 2, algebraically: $100 = 10^2$

There are infinitely many logarithms possible, corresponding to all possible bases. In practice there are two useful base systems (*found on most calculators*):

- Logarithms with base 10
- Logarithms with base e
- The most commonly used logarithms are base 10 logarithms. (*These are sometimes called Briggsian logarithms after Henry Briggs who first introduced them as a variation of Napierian logarithms*). To avoid any ambiguity about the base used, logarithms can be written fully as $\log_{10}(100) = 2$, (rather than just $\log 100 = 2$).

We refer to natural logarithms, or Napierian logarithms after their inventor, John Napier. The natural logarithm is usually denoted \ln (*but you will also see log used in places where the natural, base e, logarithm is intended*). The logarithm is a third order arithmetic operation.

Definition

The logarithm q , to the base a , of a number x , is the power to which you must raise the base a to obtain x as the outcome, so:

$$q = \log_a(x) \iff a^q = x$$

Which can be written as:

$$a^{\log_a(x)} = x$$

Both the base a , and the argument x , must be greater than 0; moreover, a must be unequal to 1.

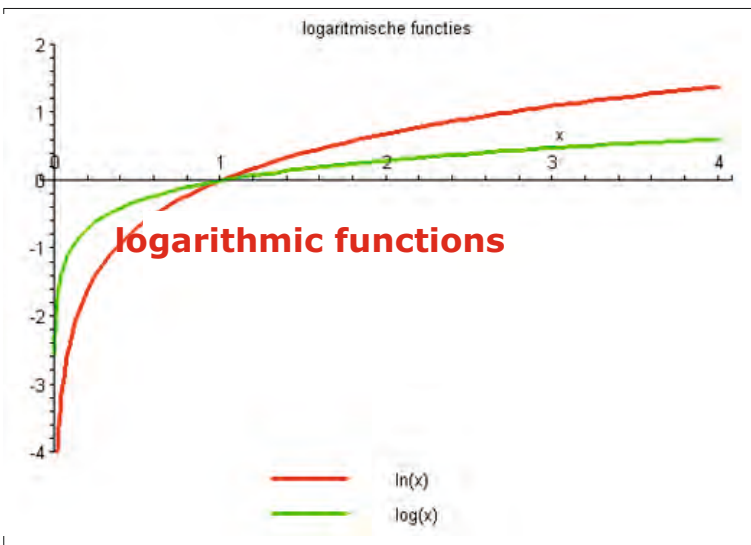


Figure 3: Natural and base 10 logarithmic functions

HISTORY OF THE COMPUTER

Instead of writing $\log(x)$ you can also write $\log_a(x)$ to make explicit the base to which the logarithm refers. However, the base (*if absent*) is assumed to be 10. For example, normally $\log(x)$ is taken to mean $\log_{10}(x)$.

Common or Briggsian logarithm

The common (or Briggsian) logarithm, \log , is the logarithm to base 10. Before the advent of electronic calculators, tables of such logs were widely used to speed up calculations on large numbers. Adding and subtracting logs from published tables, and finding the antilog of the resulting value is much faster than multiplying or dividing by long multiplication or long division. In this way multiplications were transformed to additions and powers to multiplications, because:

$$\begin{aligned}\log(xy) &= \log(x) + \log(y) \\ \log(x^y) &= y \cdot \log(x)\end{aligned}$$

Natural or Napierian logarithm

The natural (or Napierian) logarithm is usually designated with \ln , but mathematicians may also use \log . The natural logarithm has base e (where $e = 2.718281828\dots$):

$$\ln(x) = {}^e \log(x)$$

The significance of using base e is the fact that the derivative of the function $f(x) = e^x$ is again e^x .

We can represent the logarithmic function graphically. Here are graphs of the lines $y = \ln x$ and $y = \log x$

The logarithm to base a of a number x is the inverse of the exponential function with a as a base. When the graph of the logarithm for base a is mirrored with regard to the line $y=x$, the function $x = a^x$ is obtained.

HISTORY OF THE COMPUTER

1622.

By using Napier's logarithms William Oughtred invents the sliderule.

William Oughtred (5 March 1574 – 30 June 1660) was an English mathematician.



Figure 4: William Oughtred

Oughtred was born at Eton in Buckinghamshire (now part of Berkshire), and educated there and at King's College, Cambridge, of which he became fellow. Being admitted to holy orders, he left the University of Cambridge about 1603, for a living at Shalford. He was presented in 1610 to the rectory of Albury, near Guildford in Surrey, where he settled.

About 1628 he was appointed by the Earl of Arundel to instruct his son in mathematics. He also offered free mathematical tuition to pupils, who included Richard Delamain, and Jonas Moore, making him an influential teacher of a generation of mathematicians. He corresponded with some of the most eminent scholars of his time.

It's being said that he died of joy when he heard that Charles II was restored as king of England.

After John Napier invented logarithms, and Edmund Gunter created the logarithmic scales (lines, or rules) upon which slide rules are based, it was Oughtred who first used two such scales sliding by one another to perform direct multiplication and division; and he is credited as the inventor of the slide rule in 1622.

His original design of some time in the 1620s was for a circular slide rule; but he was not the first into print with this idea, which was published by Delamain in 1630. Oughtred published a book in 1632 about his sliderule and he became involved in a priority dispute with his former student Delamain.

In 1631 he introduced the symbol „×“ for multiplications and „÷“ for divisions as well as the abbreviations "sin" and "cos" for the sine and cosine functions. Concerning the abbreviations "sin" and "cos" Albert Girard probably was a few years earlier.

HISTORY OF THE COMPUTER

1623.

Wilhelm Schickard designs the first mechanical calculator.



Figure 5: Wilhelm Schickard

Wilhelm Schickard (22 April 1592 – 24 October 1635) was a German polymath who drew a calculating machine in 1623, twenty years before Pascal's calculator was invented. He called it a Speeding Clock or Calculating Clock (*a misleading name, since it did not tell the time*) on two letters that he wrote to Johannes Kepler and explained that the machine could be used for calculating astronomical tables. The machine could add and subtract six-digit numbers, and indicated an overflow of this capacity by ringing a bell; to add more complex calculations, a set of Napier's bones were mounted on it. Schickard's letters mention that the original machine was destroyed in a fire while still incomplete. A working replica was eventually constructed in 1960. Schickard's work, however, had no impact on the development of mechanical calculators

HISTORY OF THE COMPUTER

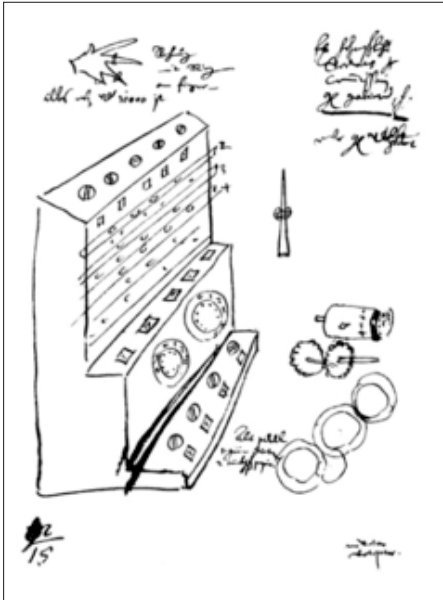


Figure 7: The working replica

Figure 6: Original drawing of Schickard's „Rechenmaschine“

1642.

Blaise Pascal builds a mechanical calculator (Pascaline).

Blaise Pascal (19 June 1623 – 19 August 1662), was a French mathematician, physicist, inventor, writer and Catholic philosopher. Pascal had many contributions in mathematics (*Pascal's Triangle*), hydrolics (*Pascal's law on pressure*), etc.

He invented a mechanical calculator in 1642. He conceived the idea while trying to help his father who had been assigned the task of reorganizing the tax revenues of the French province of Haute-Normandie. Pascal went through 50 prototypes before presenting his first machine to the public in 1645. It was first called **Arithmetic Machine**, **Pascal's Calculator** and later **Pascaline**, it could add and subtract directly and multiply and divide by repetition.

Its introduction launched the development of mechanical calculators in Europe first and then all over the world, development which culminated, three centuries later, in the invention of the microprocessor in 1971. Although Blaise Pascal was not the inventor of a programming language, „our“ language Pascal was named after him in his honour.



Figure 8: Pascal's mechanical calculator, the Pascaline

HISTORY OF THE COMPUTER

1673.

Gottfried Leibniz builds a digital mechanical calculator.

Gottfried Wilhelm Leibniz (or von Leibniz, July 1, 1646 – November 14, 1716) was a German mathematician and philosopher. Early in life, he documented the binary numeral system (base 2), then revisited that system throughout his career. In 1671, Leibniz began to invent a machine that could execute all four arithmetical operations, gradually improving it over a number of years. This "Stepped Reckoner" attracted fair attention and was the basis of his election to the Royal Society in 1673. The operating mechanism, invented by Leibniz, called the stepped cylinder, drum or Leibniz wheel, was used in many calculating machines for 200 years. Also, Leibniz is the most important logician between Aristotle and 1847, when George Boole and Augustus De Morgan each published books that began modern formal logic.



Figure 9: Gottfried Leibniz

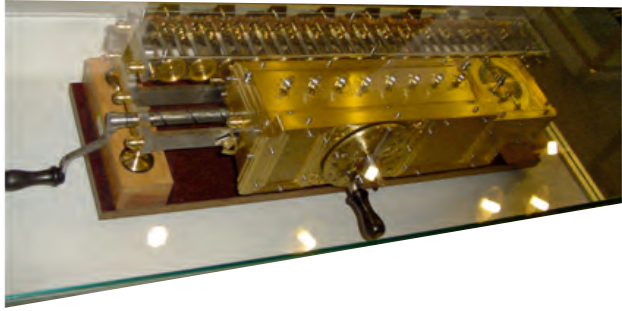


Figure 10: The replica of Gottfried Leibniz's Stepped Reckoner

1773.

Phillip Mathieus Hahn developes an improved calculator.

Pascal's and Leibniz's designs were the basis for most of the mechanical calculators built during the 18th Century. Special consideration deserves the German glergyman **Phillip Mathieus Hahn** (1730-1790) who developed in 1773 the first functional calculator based on Leibniz's Stepped Drum. Hahn's calculator had a set of 12 drums in a circular arrangement actuated by a crank located in the axis of the arrangement.

1820.

Thomas de Colmar invents the arithmometer.

An **Arithmometer** or **Arithmomètre** was a mechanical calculator that could add and subtract directly and could perform long multiplications and divisions effectively by using a movable accumulator for the result. Patented in France by **Thomas de Colmar** in 1820 and manufactured from 1851 to 1915, it became the first commercially successful mechanical calculator. Its sturdy design gave it a strong reputation of reliability and accuracy and made it a key player in the move from human computers to calculating machines that took place during the second half of the 19th century.

Its production debut of 1851 launched the mechanical calculator industry which ultimately built millions of machines well into the 1970s. For almost forty years, from 1851 to 1887, the Arithmometer was the only type of mechanical calculator in commercial production and it was sold all over the world. During the later part of that period two companies started manufacturing clones of the Arithmometer, they were: Burkhardt from Germany which started in 1878 and Layton from the UK which started in 1883. Eventually about twenty European companies built clones of the arithmometer until the beginning of WWII.



Figure 11: Thomas de Colmar

HISTORY OF THE COMPUTER



Figure 12: The Arithmometer

1822.

Charles Babbage designs the 'difference engine'.

Charles Babbage, (26 December 1791 - 18 October 1871) was an English mathematician, philosopher, inventor and mechanical engineer who originated the concept of a programmable computer. Considered a "father of the computer", Babbage is credited with inventing the first mechanical computer that eventually led to more complex designs. His Difference Engine was designed to compute values of polynomial functions and would weigh 13,600 kg! It was never completed during Babbage's life.



Figure 13: The difference engine of Babbage

1833.

Babbage designs the 'analytical engine'.

Soon after the attempt at making the difference engine crumbled, Babbage started designing a different, more complex machine called the Analytical Engine. The engine is not a single physical machine but a succession of designs that he tinkered with until his death in 1871. The Analytical Engine could be programmed using punched cards. He realised that programs could be put on these cards so the person had only to create the program initially, and then put the cards in the machine and let it run. This machine was also intended to employ several features subsequently used in modern computers, including sequential control, branching, and looping. Charles Babbage is therefore known as the "father of the computer".

HISTORY OF THE COMPUTER

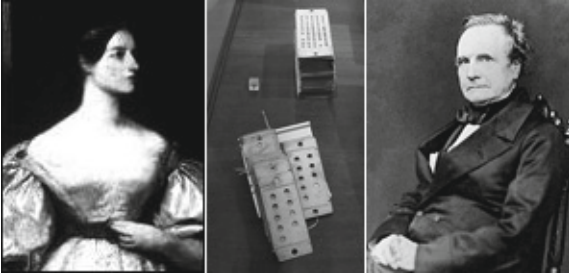


Figure 14: Ada Lovelace, Charles Babbage and a set of punched cards to program the Analytical Engine

1842. Ada Lovelace, the first programmer

Augusta Ada King, Countess of Lovelace (10 December 1815 – 27 November 1852), born **Augusta Ada Byron**, was an English writer chiefly known for her work on Charles Babbage's early mechanical general-purpose computer, the analytical engine. Her notes on the engine include what is recognised as the first algorithm intended to be processed by a machine; thanks to this, she is sometimes considered the "World's First Computer Programmer". The computer language Ada, created on behalf of the United States Department of Defense, was named after Lovelace.

1854. George Boole developed a logical calculus of truth values

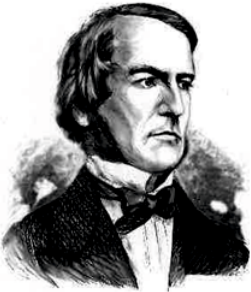


Figure 15: George Boole

George Boole (2 november 1815 - 8 december 1864) was an English mathematician and philosopher. In 1849 he was appointed as the first professor of mathematics of the university in Cork, Ireland. He invented a system for symbolic and logical reasoning, now known as Boolean algebra. This algebra is used in mathematics, for designing computer circuitry and programming. Boolean algebra contains the algebraic structures with the logical operators AND, OR and NOT, based on the "truth values" 0 and 1. The operators are directly related to be conjunction, disjunction, and negation concepts of the set theory. The operators and variables (*called Booleans after George Boole*) are widely used in search engines to specify the search logic.

Examples of logic expressions:

- 1 **and** 0 = 0 (result only true when both variables are true)
- 1 **or** 0 = 1 (result true when at least one variable is true)
- not** 1 = 0 (result is the inverse of the variable)

HISTORY OF THE COMPUTER

Carlson is born in Seattle on February 8, 1906 and moved with his family to San Bernardino in California. His father was a barber who developed arthritis of the spine. When he was 14 years, both his mother and his father got tuberculosis, so that Chester became the main source of income.

Despite all these setbacks Chester succeeded in enrolling in a high school in 1930 and later took a degree in Physics at the California Institute of Technology. After several jobs Carlson attended night school to become a patent attorney. He always needed more copies of patents and in that time there were only two possibilities: either to photograph the patents elsewhere or to laboriously copy them by hand. Because Carlson was increasingly frustrated by the slow mimeograph machine and the high cost of photography, this led him to think about a new way of copying. He invented an electrostatic process that could reproduce words on a page in a few minutes.

Experiments and many years of labor led to Carlson's first patent, issued in October 1937 for "electrophotography". In subsequent years IBM, Kodak, General Electric and many others rejected Carlson's idea. Two companies, however, snapped, Battelle and Haloid which were inclined to invest money and manpower into the project. **In 1948 the electrophotography was demonstrated to the world.**

Because the name electrophotography was not easy on the ear, a professor from Ohio came up with "xerography" from the Greek words xeros (dry) and graphics (writing). Therefore Haloid called the first copying machine Xerox Model A, the last X was added to the name to be comparable to Kodak. Haloid changed its name officially in 1958 into Haloid Xerox and finally, in 1961, in only Xerox. Xerox Corporation has placed the name "Xerox" solid as a trademark and protected the name carefully.

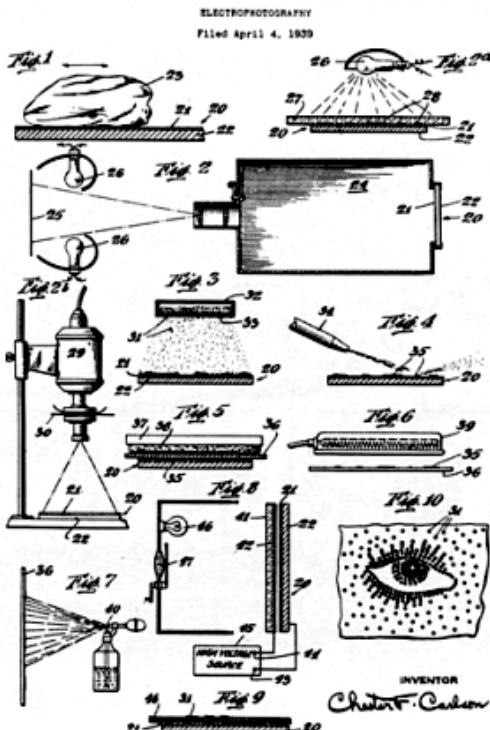


Figure 18 Carlson's original patent application material

HISTORY OF THE COMPUTER

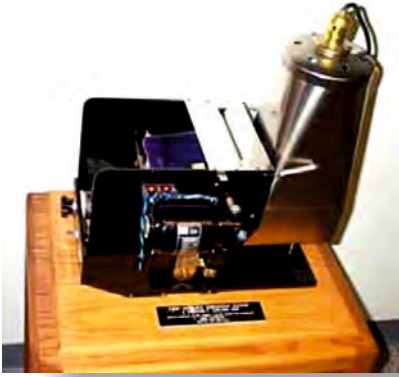


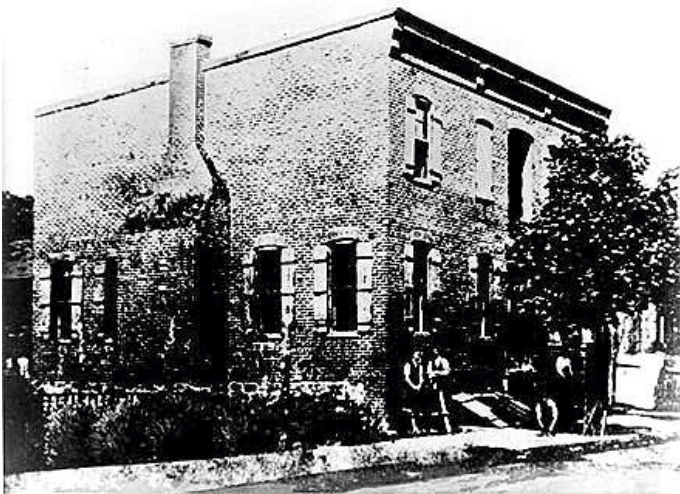
Figure : 19
A working replica of Chester Carlson's first electro-static copying machine

The Palo Alto Research Center PARC (owned by Xerox), was started in 1970. A remarkable number of ideas and inventions were developed here which have later been adopted by the ICT industry.

1911.

The Tabulating Machine Company is founded.

In 1896 Herman Hollerith, inventor of the punch card, founded the Tabulating Machine Company together with the Dutchman **Jan Broeksma**. After several mergers the Computing Tabulating Recording Company was incorporated in 1911 in the U.S. state of New York. CTR manufactured employee time keeping systems and punched card equipment.



HISTORY OF THE COMPUTER

In 1924 CTR Company is renamed International Business Machines (IBM).

Citing the need to align its name with the "growth and extension of its activities" CTR was renamed the International Business Machines. The part of the organization operating under the name Hollerith was responsible for the punched card systems used in the concentration camps in Nazi Germany. IBM was best known for the manufacture of typewriters, punch card machines, copiers and the big mainframe computers. In 1957, IBM developed the FORTRAN (FORmula TRANslation) scientific programming language. Nowadays everyone knows IBM (nickname Big Blue) for its Personal Computers. IBM has many patents on its name such as the hard disk and the floppy disk. The latter is hardly in use anymore.

An IBM invention for the electric typewriter was the typeball, introduced in the Selectric typewriter in 1961. It was a spherical element with 88 characters on its surface and was able to turn around in all directions to put those characters on paper, with a maximum speed of 15 characters per second. The ball replaced the traditional typebars (letter hammers), which could become entangled when typing fast. The ball could be replaced easily to change to a different font, which was not possible for traditional typewrites. Another difference was that the typeball moved laterally in front of the paper as opposed to the traditional moving carriage. The Selectric typewriter was during the seventies also produced in Amsterdam.

Since the sixties IBM mainframes builds large computer servers. Programs at that time had to be fed to the computer via punched cards. Since the second half of the seventies IBM expanded its market leadership also with midrange servers for midsize businesses. In this category IBM is still the market leader with its System i (i5/OS or Linux), System p (AIX or Linux) and System x (Windows and Linux) servers.

IBM presented in August 1981 the Personal Computer, a small computer for consumers with a 16-bit microprocessor from Intel, and PC-DOS 1.0, the first operating system from Microsoft. This type of computer would be the standard in daily use. IBM would later introduce an own operating system for Intel processors, OS / 2. This was (commercial) not very successful, although in a number of companies still use it.



The actual plant of CTRC

In 1991, the production and sale of printers placed in a separate division, Lexmark. It was sold in 1995. The consulting division of PricewaterhouseCoopers was acquired in 2002. They had 30,000 employees worldwide, in the Netherlands 1700.

In December 2004 IBM sold the PC division to Chinese computer manufacturer Lenovo. Lenovo paid \$ 650 million cash and \$ 600 million in shares for the division. Initially there was uncertainty whether the U.S. government would approve the acquisition, the government thought that Lenovo might be consigned to too sensitive U.S. technology. Proponents of the takeover, however, said that any business would have computer technology at its disposal and that Lenovo itself already possessed the technology. Eventually, on March 9, 2005 the deal was approved by the Committee on Foreign Investment in the U.S. (CFIUS) and this acquisition was a fact.

On June 6, 2005, IBM, in collaboration with the Ecole Polytechnique Fédérale de Lausanne (EPFL), launched the Blue Brain Project. The aim of this study is to make a detailed model of the neocortex (*the part of the brain thought to be responsible for higher functions such as conscious thought*), running an artificial neural network on supercomputers.

IBM's motto was 'Think.' Apple's slogan was 'Think differently.'

HISTORY OF THE COMPUTER

1927. Vannevar Bush constructs a large-scale differential analyser.



Figure 20: Vannevar Bush

Vannevar Bush was, without a doubt, one of the most influential scientists of the twentieth century. Starting in 1927, Bush constructed a Differential Analyser, an analog computer that could solve differential equations with as many as 18 independent variables. He had a political role in the development of the atomic bomb as a primary organizer of the Manhattan Project.

But that is not the reason why Bush today is still known. His on-going reputation is due to an article which was published in *The Atlantic Monthly*: '**As We May Think**'. It described the concept of what he called the **memex**, which he imagined as a microfilm-based "device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory." In the article, Bush predicted that "wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified".

This article inspired many pioneers in the computer technology and the article is often cited as a conceptual forerunner of the World Wide Web. Bush is now regarded as one of the founders of the information technology.

1935. Konrad Zuse builds the Z1-computer.

Konrad Zuse (1910–1995) was a German civil engineer and computer pioneer. He builds the Z1, world's first program-controlled computer. Despite certain mechanical engineering problems it had all the basic ingredients of modern machines, using the binary system and today's standard separation of storage and control. The instructions were read from a perforated 35 mm film.

1936. John Vincent Atanasoff and John Barry invent the first electronic digital computing device for special purposes.

The Atanasoff–Berry Computer (ABC), conceived in 1937, was not programmable, being designed only to solve linear equation systems. The ABC included binary math and Boolean logic to solve up to 29 simultaneous linear equations, but it lacked a changeable, stored program.

For digital computing the machine used vacuum tubes.

HISTORY OF THE COMPUTER

1937. Alan Turing, a British mathematician, develops the theoretical 'Turing Machine'.

A Turing machine is a device that manipulates symbols on a strip of tape according to a table of rules. The Turing machine is not intended as a practical computing technology, but rather as a hypothetical device representing a computing machine. Turing machines help computer scientists understand the limits of mechanical computation. He laid the foundation for developing programmable computers for general purposes.

Turing is also famous because his success of breaking the „Enigma Code“ during Worldwar II, enabling the allies to decipher secret messages exchanged by the German military.

1939. Bell Labs develops a Complex Number Calculator (CNC).

The Bell Telephone Laboratories or Bell Labs originally was the research and development subsidiary of Western Electric and the American Telephone & Telegraph Company (AT&T), the American national telephone company, also called Bell System. Its principal work was to design and support the equipment that Western Electric built for Bell System operating companies. Bell Laboratories was the premier facility of its type, developing a wide range of revolutionary technologies from telephone exchanges to special coverings for telephone cables, the transistor and more general telecommunications (*including radio astronomy*) and information theory.

Over 40,000 inventions were made since the foundation in 1925. Many Bell-developers were awarded: seven Nobel Prizes in Physics for eleven developers and 28 IEEE Medals of Honor, almost a third of the total awarded since 1917.

Bell Labs was founded in 1925 by Walter Gifford (the chairman of AT & T) and then took over the work of the research department of Western Electric with Frank Jewett as first president. Bell Labs was 50% owned by AT & T and 50% of Western Electric.

The Complex Number Calculator (CNC) is completed. In 1939, Bell Telephone Laboratories completed this calculator, designed by researcher George Stibitz. In 1940, Stibitz demonstrated the CNC at an American Mathematical Society conference held at Dartmouth College. Stibitz stunned the group by performing calculations remotely on the CNC (located in New York City) using a Teletype connected via special telephone lines. This is considered to be the first demonstration of remote access computing.



Figure 21:
TheBell Labs Complex

HISTORY OF THE COMPUTER

In **1984** the President Ronald Reagan administration, split the monopolist AT & T to achieve more competition in the telecom market. This separation yielded apart from 'Ma Bell' (the reduced AT & T), a total of six "Baby Bells", which offered local telephone services in their region. Bellcore was split off from Bell Labs to serve as research and development branche for the local phone companies.

In 1996 AT & T Bell Labs, together with most of the equipment production facilities, merged into a new company named Lucent Technologies. AT & T retained a small number of researchers that became AT & T Laboratories.

In januari **2002** Jan Hendrik Schön, a German physicist, was dismissed by Bell Labs, because his publications were found to contain fraudulent data. It was the first known case of fraud in the history of the laboratory. More than a dozen of Schön's publications were based on fabricated or altered data, including a document on very small transistors, which was seen as a breakthrough.

At its peak, Bell Labs had research and development facilities all over the United States, the most important ones in New Jersey. Nowadays Bell Labs has its headquarters in Murray Hill, New Jersey. Since 2006 Bell Labs has been owned by the French group Alcatel-Lucent.

From the point of view of computer development, some of the most significant research spin-offs to emerge from Bell Labs have been the invention of the transistor, the laser, the development of information theory, the UNIX operating system, the C programming language, and the C++ programming language.

1943. De Colossus Mark I deciphering computer was

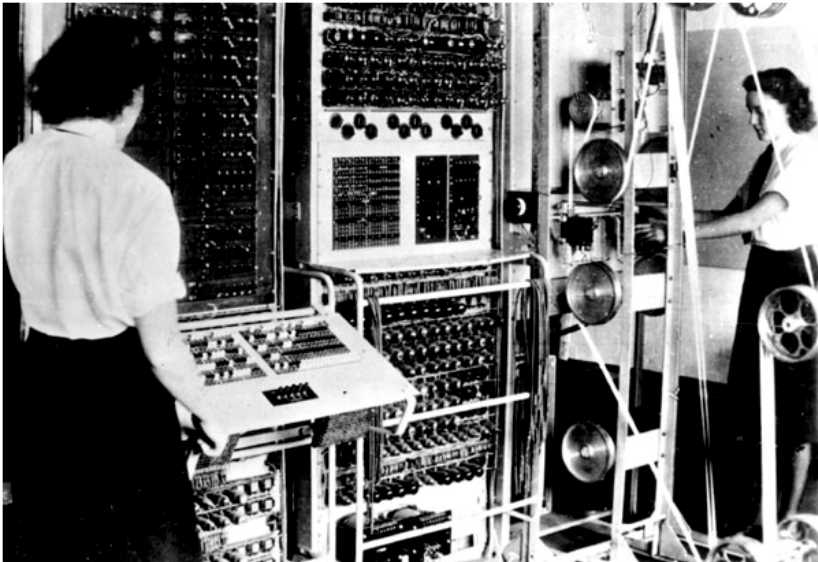


Figure 22: Colossus Mark II was used for cryptanalysis of high-level German communications (1944)

HISTORY OF THE COMPUTER

1944. The Harvard Mark I computer is developed (US).

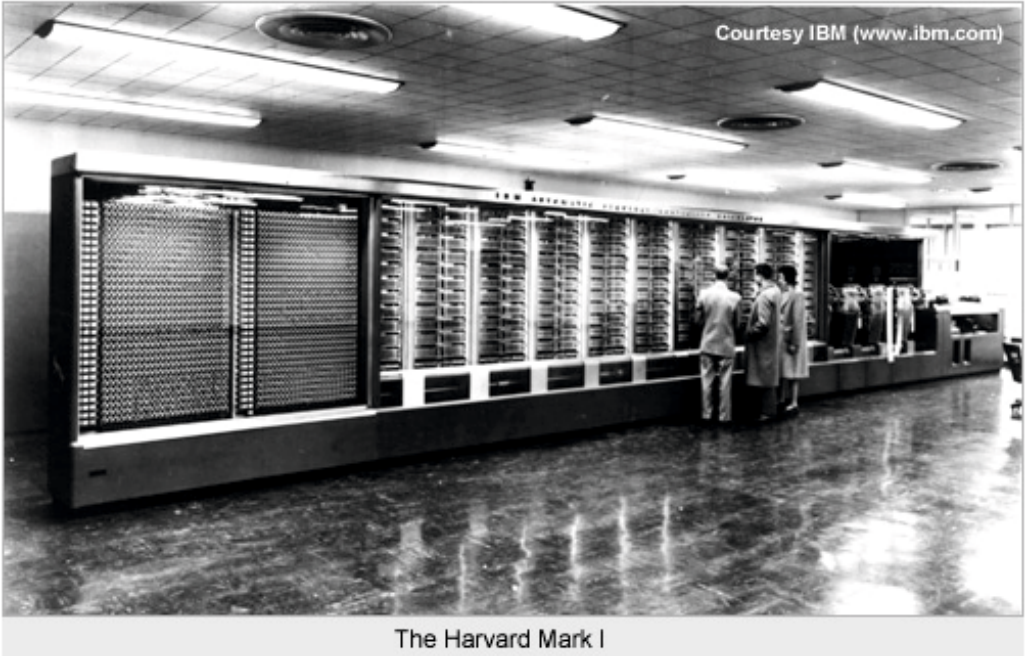


Figure 23: The Harvard Mark I – an enormous electro-mechanical computing machine, built at IBM and installed at Harvard in 1944

1947. Grace M. Hopper finds the first real bug.

Rear Admiral Grace Murray Hopper (December 9, 1906 – January 1, 1992) was an American computer scientist and United States Navy officer. A pioneer in the field, she was one of the first programmers of the Harvard Mark I computer, and developed the first compiler for a computer programming language. While she was working on a Mark II Computer at Harvard University in 1947, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. She conceptualized the idea of machine-independent programming languages, which led to the development of COBOL, one of the first modern programming languages.



Figure 24: The first actual case of a bug being found

HISTORY OF THE COMPUTER

Hopper was born Grace Brewster Murray and married in 1930 professor Vincent Foster Hopper. In 1934, she earned a Ph.D. in mathematics from Yale and was promoted to associate professor in 1941. In 1943, Hopper was sworn in to the United States Navy Reserve. She served on the Mark I computer programming staff headed by Howard H. Aiken, at Harvard's Bureau of Ordnance Computation Project, where she became an expert at the heavy mathematics then required for programming, and literally wrote the operating manual. Hopper and Aiken coauthored three papers on the Mark I and II also known as the Automatic Sequence Controlled Calculator.

In 1949, Hopper became an employee of the Eckert-Mauchly Computer Corporation as a senior mathematician and joined the team developing the UNIVAC I. Her conviction that computer programs should be written in English was invariably answered that "computers don't understand English". In the early 1950s the company was taken over by the Remington Rand corporation and it was while she was working for them that her original compiler work was done. The compiler was known as the A compiler and its first version was A-0. Three years later she and her team surprised the computer world at that time with the compiler B-0 that could translate English program instructions to machine language.

She is best known for her 1952 invention of **Flow-Matic**, the first compiler-software that translates instructions written in English into machine language for the target computer. Compilers have allowed the development of computers that seem to "understand" English, a breakthrough which lets computers be programmed by people who might lack an advanced degree in mathematics.

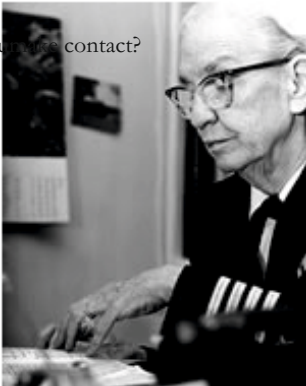


Figure 25: Grace Hopper, a gifted mathematician

In the spring of 1959 Hopper served as the technical consultant to the committee that defined the new language, COBOL. The new language extended Hopper's FLOW-MATIC language with some ideas from the IBM equivalent, the COMTRAN. Hopper's belief that programs should be written in a language that was close to English rather than in machine code or languages close to machine code (such as assembly language) was captured in the new business language.

During her career Grace Hopper worked successively for university, industry and the military. She was one of the first software engineers, and was known for her inspiring personality and great perseverance. She predicted that software will be more expensive than hardware, which was unimaginable at that time.

HISTORY OF THE COMPUTER

For her contribution to computer science Grace Hopper received numerous awards:

- * In **1950** she received the title "senior programmer", one of the first people ever to receive this title.
- * In **1969** she won the inaugural "computer sciences man of the year" award from the Data Processing Management Association. and in 1971 a new annual award for young computer scientists was subsequently named after her.
- * In **1983** the White House promoted her to Commodore status and then two years later this was merged with Rear Admiral and she became Admiral Grace Hopper.
- * In **1991**, one year before her death, Grace Hopper received the National Medal of Technology "for the success of her pioneering work in developing the programming language, that significantly simplified information technology, and opened the door to a new universe of computer users."

Grace Hopper received nearly 50 honorary degrees from universities worldwide during her lifetime. But the most striking honorary titles may be the nick names given to her: Grand Lady of Software, Grandma COBOL and Amazing Grace.

1946.

Two electrical engineers at the University of Pennsylvania, John Mauchley and J. Presper Eckert, built ENIAC

ENIAC (Electrical Numerical Integrator And Calculator) was the first large-scale, fully electronic, general purpose digital computer. The ENIAC was designed to calculate artillery firing tables for the U.S. Army's Ballistic Research Laboratory, and completed in 1946. It used nearly 18,000 valves (vacuum tubes) for storage and computation, weighed 30 tons and occupied an area of 167 square metres.

ENIAC could perform 357 10-digit by 10-digit multiplications per second, or 35 divisions or square roots per second. It cost nearly \$500,000, and needed a 150 kW power supply (mainly for the vacuum tubes). Only much later were computers able to be smaller, lighter and less power-hungry, when vacuum tubes were replaced by transistors and still later by microchips.

Programming the ENIAC was a difficult task because the valves had to be connected differently for each program. Only in future would fully programmable hardware be introduced. The ENIAC was designed to quickly calculate trajectories of missiles and grenades for wartime use. The ENIAC was the supercomputer of its time. Using an analog Differential Analyser an able mathematician could compute the trajectory of a missile in about fifteen minutes. The ENIAC completed this task in thirty seconds, which was a revolutionary development at the time. The ENIAC filled a large room. Today the same work can be performed by a chip a few millimetres square. The ENIAC was irreparably damaged by lightning in 1955.

HISTORY OF THE COMPUTER

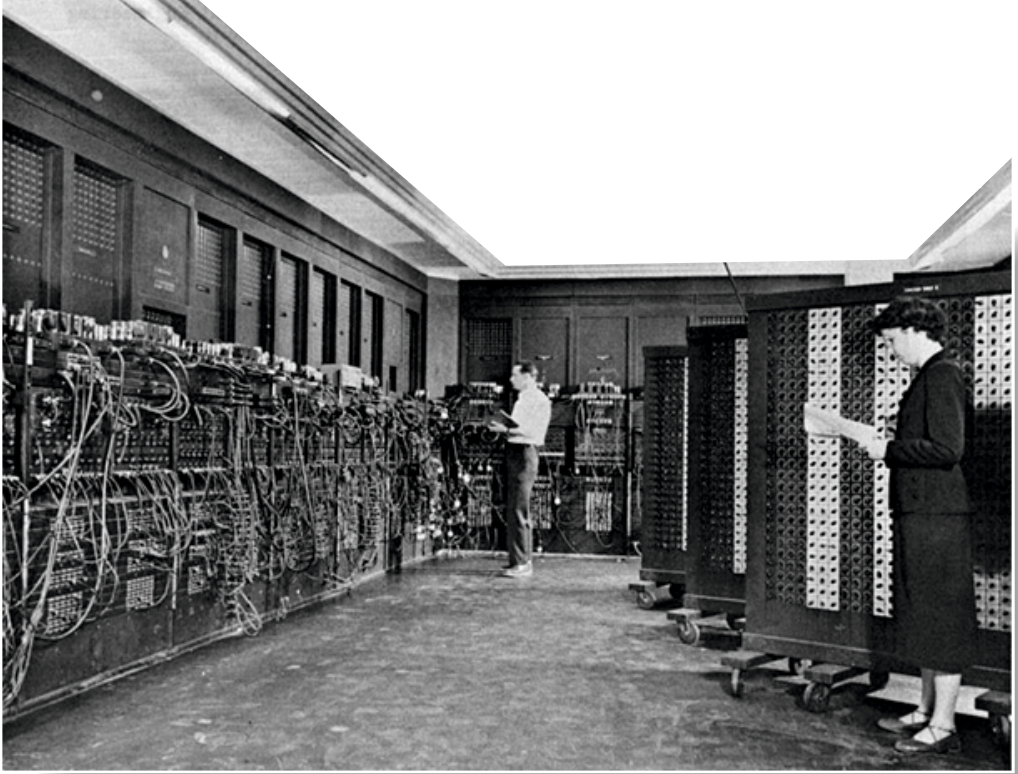


Figure 26: The ENIAC Computer

For many years the ENIAC was wrongly credited as the world's first programmable electronic computer. However, two years earlier towards the end of the Second World War, at Bletchley Park in England the Colossus was being used to decrypt secret German telexes. These messages, intercepted by the Allies had been encrypted with the Lorenz machine. Colossus was used successfully for decryption from January 1944 onwards. Because British intelligence classified all details of this code-breaking operation, the existence of Colossus was suppressed, and so all the credit went to the ENIAC. After the war, all trace of what occurred at Bletchley Park was destroyed. The 10 Colossus machines were dismantled, and all technical drawings were burned. Such was the culture of secrecy at Bletchley Park that no word of what happened began to emerge until the mid 1970s. Surprisingly it was not until 1996 when the U.S. National Security Agency declassified some wartime documents describing the Colossus (sent originally to Washington by US liaison officers stationed at Bletchley Park), that incontrovertible evidence of some of its capabilities was first published. Following these revelations the previous UK Official Secrets Act restrictions on talking about the Colossus were gradually lifted.

HISTORY OF THE COMPUTER

1947. Three engineers at Bell Labs: John Bardeen, Walter Brattain and William Shockley, invented the transistor.



Figure 27: The three inventors pictured together

The transistor has become the most important component used in semiconductor electronic devices, serving to amplify electronic signals. The transistor is the fundamental circuit component in all modern computers. Shockley, Bardeen, and Brattain were jointly awarded the 1956 Nobel Prize in Physics for this accomplishment.

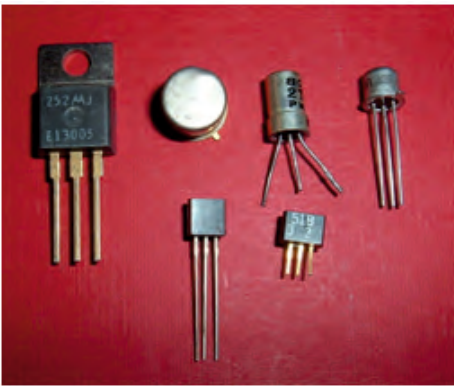


Figure 28: Transistors can be used singly as individual components, but usually many transistors are combined together into an integrated circuit.



Figure 29: A replica of the original transistor at the Bell Labs museum

HISTORY OF THE COMPUTER

John Bardeen (1908-1991).

Bardeen was an American physicist and electrical engineer, the only person to have received the Nobel Prize in Physics twice: first in 1956 with William Shockley and Walter Brattain for the invention of the transistor, and again in 1972 (with Leon Cooper and John Robert Schrieffer) for a fundamental theory of superconductivity known as the BCS theory.

Bardeen was born in Madison (Wisconsin), the son of Charles Russell Bardeen (a professor of anatomy) and a teacher named Althea Harmer.

Aged only fifteen he went to the University of Wisconsin, graduating in 1928 and completing his master's thesis in electrical engineering in 1929. After working as a geophysicist in Pittsburgh he began to study for a doctorate at Princeton University, where he received his Ph. D. in mathematical physics in 1936.

After World War II, he took up a lucrative appointment in solid-state physics at Bell Labs where his collaboration with Shockley and Brattain led to the development of the point-contact transistor. Shockley later excluded him from work on developing the junction transistor, and he left Bell Labs to become professor of electrical engineering and physics at the University of Illinois where he spent the rest of his working life.

Walter Houser Brattain (1902-1987).

Brattain was an American physicist who is known as co-inventor (with John Bardeen) of the point-contact transistor. He devoted much of his life to research on surface states, mainly carried out at Bell Labs. After the invention of the transistor he became unhappy working under Shockley, and transferred to another Bell Labs department where he remained until his retirement in 1967.

William Bradford Shockley (1910-1989).

Shockley, an American physicist and inventor, trained as a scientist and received his doctorate in solid state physics from the Massachusetts Institute of Technology in 1936. He worked as part of a research group at Bell Labs in New Jersey, publishing a number of papers on solid state physics. At the end of the war in 1945 Bell Labs formed a solid state physics group led by Shockley and chemist Stanley Morgan, which included John Bardeen and Walter Brattain. Their attempts to make a solid-state alternative to radio valves as amplifiers based on Shockley's ideas of using an external electric field to influence semiconductor conductivity failed until Bardeen suggested a theory involving surface states. Brattain and Bardeen's success in producing the first amplifying point-contact transistor (without Shockley) led Shockley to work on ideas for a junction transistor without Brattain and Bardeen. He obtained a patent for this invention in 1951. He left Bell Labs and became Director of Shockley Semiconductor Laboratory which led to the development of Silicon Valley as a focal area for microelectronics development as many of his former employees, distressed by his abrasive style, left to form their own successful companies.

Shockley will be remembered by many for his racist statements, and his outspoken support for eugenics, such as his suggestion that all people with an IQ of less than 100 should be paid to undergo voluntary sterilisation. In later life, Shockley became a professor at Stanford University, and Time Magazine named him as one of the 100 most influential people of the twentieth century. He died in 1989 from prostate cancer.

HISTORY OF THE COMPUTER

1951.

The UNIVAC (Universal Automatic Computer) is delivered to the U.S. Census Bureau.

The UNIVAC computer was a brand of American business computers built by a division of **Remington Rand** which had been founded in 1950. This company specialised in large mainframe computers. A UNIVAC was used in 1951 to forecast the presidential election results. Today the company is called **Unisys**.

The Computer Centre of the Netherlands in Heerlen used Univac 1100/42 and 1100/60 mainframes in the 1970s and 1980s to automate psychiatric and health insurance records. To cache data the UNIVAC 1100 series used a magnetic drum mass storage system (also called drum memory), a six foot long metal cylinder weighing several tons, spinning at high speed. At the time these Fastrand drums exceeded the capacity of any other random access mass storage disk or drum.



Figure 30: A UNIVAC 1100 Series computer

1952.

IBM develops its first computer, the IBM 701 EDPM.

The system used electrostatic storage, consisting of 72 Williams tubes (each of 1024 bit capacity), giving a total memory of 2048 words of 36 bits each. Each Williams tube was 3 inches in diameter. The memory could be doubled by adding another set of 72 Williams tubes, or by replacing the entire memory with magnetic core memory. The Williams tube memory and the later magnetic core memory each had a memory cycle time of 12 microseconds. The Williams tube memory had to be regularly refreshed, so refresh cycles had to be inserted into the timer of the 701. An addition operation required five 12 microsecond cycles of which two were refresh cycles.

A multiplication or division required 38 cycles (456 microseconds). The instructions were 18 bits long including the opcode and a single 12 bit address..

Known as the Defense Calculator while in development, the new machine emerged from the IBM Poughkeepsie Laboratory later that year and was formally unveiled to the public on April 7, 1953 as the IBM 701 Electronic Data Processing Machines (*the plural „Machines“ was used because the 701 was comprised of 11 compact and connected units*).

HISTORY OF THE COMPUTER

1957.

John Backus and his team develop the programming language FORTRAN

(FORmula TRANSlation), a compiled language that was useful for the scientific and academic community.

John Warner Backus (1924-2007)

John Backus was a leading American computer scientist who pioneered the field of programming language. He led the team that developed FORTRAN, implementing the first widely used high-level language. He invented the **Backus-Naur-formalisme (BNF)**, now used universally as a way to describe the syntax of computer languages (as well as the syntax of document formats and communication protocols) where precision and lack of ambiguity in the definition is of paramount importance. He also carried out research in the field of function-level programming, designing the canonical function-level programme language FP, and helping to popularize it.



Figuur 31: John Backus

The first computers were programmed using virtually unreadable machine code instructions which had to be entered one at a time, which was a tedious and error-prone procedure. When John Backus started working at IBM their computers were generally programmed in assembly language which was somewhat more readable than machine code, but still preserved a one-to-one correspondence with machine code instructions. High-level programming languages were not yet in use.

Backus was dissatisfied with the difficulties involved in computer programming. In 1953 IBM authorised him to lead a research team charged with developing better methods of programming. The result of these efforts was a FORTRAN compiler for the IBM 704 computer, which was released commercially in 1957. In some cases a single Fortran instruction gave rise to twenty lines of raw machine code. Abstractions of this sort enabled system programmers using FORTRAN to produce working programs much more quickly than using assembly language. FORTRAN was also the first computer language which non-specialists could master.

Although FORTRAN was not the first high-level language, it was the first for which a compiler was implemented and as a result it became widely used. For the first time there was a language non-computer specialists could use to get computers to solve problems outside the world of computing, and FORTRAN was taken up by both scientists and engineers. Backus and his team established a standard for the language in 1958 and introduced the term "software" for the first time.

HISTORY OF THE COMPUTER

1960.

The programming language ALGOL (ALGOritmic Language) is created.

The name Algol covers a set of very powerful programming languages, designed between 1958 and 1973. The name is an acronym for **Algorithmic Language**. Algol was born from the desire for a universally usable, machine-independent programming language which would be easy for people to understand, by combining normal mathematical notation with expressions taken from natural languages such as English or Dutch. Part of this endeavour was unsuccessful: a universal programming language has not been accepted. But the many programming languages that have been developed since Algol's appearance all build on ideas introduced in one of the Algol versions.

There are several programming languages named Algol, among them Algol-58, Algol-60, W-Algol and Algol-68. Algol-58 (originally called IAL) was developed at the same time, and with similar goals to Cobol and Fortran. Algol has never become a completely finished product, serving rather as an inspiration for many other languages.

Algol-60 was developed as a successor to Fortran, partly because one of the designers of Fortran, John Backus, participated in its development along with Dutchmen Edsger W. Dijkstra and Aad van Wijngaarden.

The lack of support for Algol from computer manufacturers means its use has been limited largely to academia. A very influential innovation in this language was support for randomly placed block structures in which variables can have a local scope.

The U.S. trade association ACM decided that this would be the default language for representing algorithms (pseudo code) in the pages of its journal Communications of the ACM.

Algol-W was developed by Tony Hoare and Niklaus Wirth as a simplified Algol-60.

Wirth developed a compiler for the IBM 360 to popularize the language.

Algol-68 was designed as a successor to Algol-60, with the goal of making it a universal programming language, able to run on any computer for any programmable purpose. The final report contained an elaborate formal specification, the Van Wijngaarden grammar, which described the language in a machine- and compiler-independent way. This grammar provides formal criteria for reasoning about the correctness of programs and compilers written in Algol. Several members of the design team, including Dijkstra, Hoare and Wirth, thought the language and specification to be too large and complex for a practical compiler to be designed for it, or for programmers to be able to use it easily.

Despite a revised 1973 specification which removed some difficult constructs and which was much more understandable, the general view tended to be that a universal programming language may be too high a goal, and that a commission designing it on paper (without producing any practical implementation of it) might not be the best development method. Programming languages developed since then have nearly all been limited to certain types of computers, certain types of applications, or certain programming paradigms. Ada is an exception to this trend.

Algol-68 has never had a fully implemented compiler or interpreter. However, there are some compilers which implement subsets of Algol-68 that are much richer than, say Pascal. These were popular in the 1970s and 1980s in academia. Today the language is sometimes used for pseudocode. Some Algol-68 language constructs have been newly introduced into existing languages (such as combining static typing with higher-order functions and lambda expressions).

HISTORY OF THE COMPUTER

1963.

Douglas Engelbart develops the mouse.

At the Stanford Research Institute (SRI) Engelbart gradually obtained over a dozen patents. He and his team developed computer-interface elements such as bit-mapped screens, the mouse, hypertext, collaborative tools, and precursors to the graphical user interface. He conceived and developed many of his user interface ideas back in the mid-1960s, long before the personal computer revolution, at a time when most individuals were kept away from computers. Engelbart applied for a patent in 1967 and received it in 1970, for the wooden shell with two metal wheels (computer mouse). He never received any royalties for his mouse invention. During an interview, he says "SRI patented the mouse, but they really had no idea of its value. Some years later it was learned that they had licensed it to Apple for something like \$40,000."

1964.

John G. Kemeny and Thomas E. Kurtz, both mathematics professors at Dartmouth College, develop the BASIC language.

That same year also saw the first computer mouse in production. In addition, IBM introduced System 360, its first computer 'family'.

The BASIC (*Beginners All-purpose Symbolic Instruction Code*) programming language is usually implemented as an interpreted language, originally designed to help people to learn programming quickly. BASIC was based on FORTRAN II and Algol-60. There are numerous different implementations still in use, most versions producing interpreted code (depending on an installed library) rather than producing compiled executables.

John George Kemeny (1926-1992)

(Hungarian: Kemény János György). Kemeny was a Hungarian-American mathematician, computer scientist and teacher, known best for co-development of the BASIC language in 1964 with Thomas Kurtz. Kemeny pioneered the use of computers in education, serving as the 13th president of Dartmouth College from 1970 to 1981. He chaired the presidential commission that examined the Three Mile Island incident in 1979.

Thomas Eugene Kurtz (born 1928)

Dr. Kurtz's first experience with computing in 1951 at a summer session of the Institute for Numerical Analysis at the University of California, Los Angeles. His interests included since numerical analysis, statistics and computer science.

Kurtz graduated from Knox College in 1950 and he was awarded a Ph. D. from Princeton University in 1956, after which he went to Dartmouth College to work in their Mathematics Department. Kurtz, working with Kemeny from 1963, developed the first version of the Dartmouth Time-Sharing System (a system for simultaneous computing) and the BASIC language. Kurtz was director of the Kiewit Computation Center at Dartmouth from 1966 to 1975, and of the Office of Academic Computing from 1975 to 1978. From 1980 to 1988, Kurtz directed the Computer and Information Systems Program at Dartmouth, a pioneering multidisciplinary graduate program for IT professionals which helped the industry to train new talent. He then returned to teaching as a full-time mathematics professor, with a strong emphasis on statistics and computer science.

In 1983, Kurtz and Kemeny co-founded a company called True BASIC, Inc. to market True BASIC, an updated version of the BASIC language. Dr Kurtz was also board chairman and director of Educom and NERComP and was a member of the Pierce Panel of the Presidential Science Advisory Committee.

HISTORY OF THE COMPUTER

1967.

The first computers using integrated circuits are built.

The integrated circuit (or IC, or chip, or microchip) represented a major improvement over traditional discrete circuits assembled manually from many components soldered to a circuit board. Integrated circuits are monolithic, meaning that numerous components are formed simultaneously by photolithography (through patterned diffusion of trace elements into a thin semiconductor substrate). The microscopic size of the components and their interconnections means that many thousands of components can be concentrated in a small region (up to a million transistors per square millimetre). Cost is low because the chips, with all their components, are printed as a unit by photolithography rather than being constructed one transistor at a time. Furthermore, far less material is used to construct a packaged IC. Performance is high because the components switch quickly and (as a result of the small size and close proximity of the components) consume little power.

1968.

The Intel company is founded.

Intel is an American company specialising in the manufacture of chips, motherboards, software and other components needed for computers, computer networks and communication systems. Intel is mainly known for the microprocessors used in many of today's personal computers including the 8086, 286, 386 486 and the Pentiums.

The company was founded in 1968 by **Robert Noyce** and **Gordon Moore**.



Figure 32 and 33: Robert Noyce (left), Gordon Moore (right)

“Moore's Law” states that continuing technological advance allows the number of transistors in an integrated circuit to double every two years.

Gordon Moore, a founder of chip maker Intel, first made this prediction in 1965. Moore's Law has applied to the present (2011), but experts say that these improvements in transistor packing will slow soon before they cease, since ever-diminishing miniaturization depends not only on technological progress, but also on limits imposed by fundamental physical (atomic) barriers. When Moore made the prediction in 1965, he suggested a doubling every 12 months. In 1975, he altered the prediction by assuming that the rate of growth would slow to a doubling every two years.

The original prediction was about the density of transistors, but later he also adjusted at this point to refer to the density of transistors that could inexpensively be placed on a chip. Intel did not possess a good copy of the Electronics magazine in which Moore first made this prediction in April 1965, and so placed an eBay ad in the hope of receiving a copy in good condition. A British engineer, David Clark, was the first to respond to Intel's advert and received \$10,000 in April 2005 (40 years later) from Intel for his well-preserved copy!

HISTORY OF THE COMPUTER

Recently the growth in developing ever-faster maximum chip clock speeds has slowed. It is no longer possible to go much higher than 3.8 GHz economically. The chip manufacturers' solution to this 'ceiling' has been to place multiple processors (also called cores) on a chip. Multicore processors are now standard in most newly built computer equipment. Dual or quad cores are widespread in consumer computers, though it is only possible to take advantage of this capability if software is rewritten to run in parallel threads.

When Gordon Moore retired in 2006 he announced that his law would not always apply. "There are now limits which we will reach and not exceed." Although alternatives such as nanotechnology may replace conventional electronics, Moore sees a number of limitations including the size of miniaturized circuits all being on the same scale.

Intel's third employee, **Andrew Grove**, led the company as CEO from early 1960s to the late 1990s and is still chairman of the board.

Moore and Noyce first wanted to name their company "Moore Noyce" but that name did not sound as good as calling it Intel (abbreviated from *INTEgrated Electronics*). Intel was then the name of a hotel group, from whom they had to buy the name.

Intel began making memory for computers before they switched to the manufacture of microprocessors. Andrew Grove described this shift in his book "**Only the paranoid Survive**".

Intel developed their first microprocessor (the Intel 4004) in the 1970s.

IBM produced the first personal computer using Intel 8086/8088 processors in 1981 (See IBM Personal Computer).

During the 1990s Intel Architecture Labs (IAL) was responsible for many hardware developments such as the PCI bus, Universal Serial Bus (USB) and the architecture for multi-processor servers.

Intel licensed manufacture of its 8086 line to competitor **Advanced Micro Devices (AMD)** in 1982, but cancelled the contract in 1986. The two corporations have been fiercely competitive with ongoing lawsuits between them subsequently.

Officials of the European Commission visited the German branch of Intel in 2008 following a complaint from AMD accusing Intel of abusing its monopoly position.

On May 13, 2009 the then European Commissioner **Neelie Kroes** fined Intel **€1.06 billion**, the largest fine ever imposed by the EU for antitrust practices.

In 2003 Intel had 78,000 employees and operations in more than forty countries.

HISTORY OF THE COMPUTER

1969.

ARPANet, the forerunner of today's Internet begins.

The ARPANET (ARPA Network) was the first operational packet switching computer network, the predecessor of the Internet, created in the late 1960s by the U.S. Department of Defense (DoD) **Advanced Research Projects Agency (ARPA)**.

The reason for ARPANET's creation was economic: computers were expensive to buy, so it was beneficial if laboratories working on projects with a military purpose could share each other's equipment, even though spread out. Because in the late 1960s neither equipment nor communication lines were very reliable, it was a requirement for ARPANET that in the event of inevitable local technical problems as many operations could continue as possible.

It is a popular misconception that the ARPANET was intended as a communications network in times of nuclear war. There was the idea of a network that could survive a nuclear war (*suggested by Paul Baran, who proposed a similar packet switching architecture*) but this suggestion did not play a role in ARPA's decision to set up a network.

The first concrete plans for the ARPANET were drafted in 1968, and in 1969 ARPA decided to outsource the implementation to the firm Bolt, Beranek and Newman.

A major problem in the designing ARPANET was that computers of very different sorts needed to be inter-connected. The designers of ARPANET solved this potential incompatibility by adding a minicomputer at each network location to work as an **interface message processor (IMP)**.

This IMP interfaced the computer and the network connect. These IMPs were 16-bit DDC-516 computers from Honeywell.



Figure 34 The ARPA Net

HISTORY OF THE COMPUTER

The IMP can be considered the forerunners of today's routers, although they were the size of a refrigerator (small compared to most computers of the time).

There were four of these MIPs active on the West Coast of the USA by 1969. The first sites were the University of California at Los Angeles (SDS Sigma 7), the University of California at Santa Barbara (IBM 360/75), Stanford University (SDS 940) and the University of Utah (DEC PDP-10).

In 1970 the number of sites increased to fifteen and by in 1971 23 computers were connected.

Gradually, the ARPANET was connected to other networks.

The ARPANET became the backbone of the ARPA Internet.

ARPANET was originally developed for logging on another computer (Telnet) and sending and receiving files between computers (FTP). After commissioning a third application, e-mail, developed very quickly. Initially this was achieved via text files uploaded by one computer, but fully interactive e-mail programs were quickly developed.

In the 1980s ARPANET lost its military function as a network. US defence had developed a private network, MILNET, which is not connected directly to the internet.

The ARPANET was finally decommissioned in 1988.

HISTORY OF THE COMPUTER

1971.

Intel develops the first microprocessor.

That same year IBM introduced the floppy disk.



1972 Niklaus Wirth

**develops the programming language Pascal,
named after the French mathematician Blaise Pascal.**

Figure 35: Niklaus Wirth in 1969

Niklaus Wirth (born in Winterthur, Switzerland in 1934) has developed several programming languages.

Wirth studied electronics at the Eidgenössische Technische Hochschule (ETH) in Zürich where he graduated in 1959. He then obtained his doctorate at the University of Laval (*Québec-Canada*) and he studied at the University of Berkeley in the United States.

He taught at Stanford University and the ETH in Zürich, where from 1968 until his retirement in 1999 he was Professor of Computer Science.

Wirth earned his Masters title with a fundamental study of the various dialects of the programming language Algol, from which he developed a new, formalized dialect: Algol-W (1968).

In 1970 Wirth introduced **Pascal**, a formally defined, statically-typed programming language. The language was primarily intended for teaching programming, but on account of its simplicity and the fast, free compilers that became available, it quickly attracted a large number of users beyond those beginners it was first designed for.

In 1980 Wirth began a new computer project, since he needed a computer language that, unlike Pascal, was also suitable for writing system-level operating systems. This language became Modula (later Modula-2). Like Pascal, Modula is a formalized, functional language with strong type-checking. Unlike Pascal, Modula solved a number of practical problems faced by software developers. Wirth introduced in Modula the concept of modular programming, in which functions and variables are neatly arranged in modules facilitating their reuse via explicit export and import.

Under pressure from the rapidly emerging concepts of object oriented programming (OOP) Wirth developed another language, **Oberon**, in the 1990s. Oberon distinguishes itself from all other languages because it is both a programming language and an operating system.

In the introduction to this language Wirth argues strongly against the usual tenor of object oriented languages, which in his view usually offer the wrong solutions to the problems raised. Wirth made Oberon a simpler and more compact language than competing languages are. Oberon was the first language to introduce software components, and is therefore sold commercially under the name **Component Pascal**.

Pascal is in some respects a competitor to the slightly older C language. The two languages have clear differences in their assumptions, which Brian Kernighan (one of the designers of C) once summarized in his 1981 paper „**Why Pascal is not my favorite programming language**“. Incidentally, Wirth had already dealt with many of the issues identified by Kernighan in Modula 2, while other aspects were later added to ANSI C, because they were shown to clarify its main structure.

As a simplification, you might say that “Wirth-languages” are based on the idea that the compiler must make the programmer think clearly about what he is doing, and force him to structure his code accordingly, whereas C-like languages leave more responsibility in the hands of the programmer.

HISTORY OF THE COMPUTER

Compilers for Wirth-languages generally produce code of similar performance to C compilers, and sometimes produce executables that are smaller and faster than that produced by equivalent C code.

Many programming courses are based on course material developed by Wirth, and if you compare the simple examples of these programming courses as pseudocode, the pseudocode for Pascal is almost indistinguishable from that for Modula 2.

Since his retirement in 1999, Wirth has advised an ETH Zürich spin-off company, Oberon Microsystems, that markets his latest creation, Oberon (more precisely the Component Pascal dialect). In 2005 they released BlackBox Component Pascal as an open source version.



Figure 36: Niklaus Wirth in 1984

A selection of Niklaus Wirth's comments on programming:

- * *A good designer must rely on experience, on precise, logic thinking; and on pedantic exactness. No magic will do.*
- * *But active programming consists of the design of new programs, rather than contemplation of old programs.*
- * *But quality of work can be expected only through personal satisfaction, dedication and enjoyment. In our profession, precision and perfection are not a dispensible luxury, but a simple necessity.*
- * *Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.*
- * *Experience shows that the success of a programming course critically depends on the choice of these examples.*
- * *I have never designed a language for its own sake.*
- * *In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever.*
- * *Indeed, the woes of Software Engineering are not due to lack of tools, or proper management, but largely due to lack of sufficient technical competence.*
- * *It is evidently necessary to generate and test candidates for solutions in some systematic manner.*
- * *Many people tend to look at programming styles and languages like religions: if you belong to one, you cannot belong to others. But this analogy is another fallacy.*
- * *My being a teacher had a decisive influence on making language and systems as simple as possible so that in my teaching, I could concentrate on the essential issues of programming rather than on details of language and notation.*
- * *My duty as a teacher is to train, educate future programmers.*
- * *Nevertheless, I consider OOP as an aspect of programming in the large; that is, as an aspect that logically follows programming in the small and requires sound knowledge of procedural programming.*

HISTORY OF THE COMPUTER

- * *Our ultimate goal is extensible programming (EP). By this, we mean the construction of hierarchies of modules, each module adding new functionality to the system.*
- * *Program construction consists of a sequence of refinement steps.*
- * *Programming is usually taught by examples.*
- * *Software development is technical activity conducted by human beings.*
- * *The idea that one might derive satisfaction from his or her successful work, because that work is ingenious, beautiful, or just pleasing, has become ridiculed.*
- * *The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision.*
- * *Usually its users discover sooner or later that their program does not deliver all the desired results, or worse, that the results requested were not the ones really needed.*



Figure 37: Niklaus Wirth

HISTORY OF THE COMPUTER

1970

Xerox PARC (Palo Alto Research Center)



Figure 38: Palo Alto Research Center, established under the leadership of Dr. George Pake (centre) who puts the first spade into the ground.

1970

Xerox PARC

Xerox Corporation assembles a world-class team of experts in information and physical sciences to become

The Architects of Information,

establishing the company's **Palo Alto Research Center** under the leadership of Dr. George Pake (*in the centre of the picture above*). The charter for Xerox PARC is to create *The Office of the Future*".

1972

object-oriented programming

Xerox PARC designs the first object-oriented programming language, Smalltalk, creating an integrated development environment that enables programs to be improved without having to write them all over again from scratch. Smalltalk pioneered code reuse. This innovation was a true revolution in the software industry and influenced all later programming systems.

1971

laser printing

demonstrates a brand new way to print documents. demonstrates a completely new way to print documents. Xerox PARC researcher Gary Starkweather modulated a laser beam so that each part of a bit-mapped electronic figure could be stored on a photosensitive xerographic copy cylinder. The electrostatically charged cylinder then picks up charged toner particles according to the bit-mapped pattern of charge, which are then transferred to the paper, and fused permanently into ink by a heating element. The invention of laser printing allowed scalable prints to be made from digital documents in a variety of fonts, which created a multi-million dollar printing industry for Xerox. Hewlett-Packard, IBM, Canon, Brother and other companies quickly followed suit with their own variations on the laser printing process, and colour laser printers began to appear in the mid-1990s.

HISTORY OF THE COMPUTER

1973.

Ethernet, the first local area network (LAN) is developed.

Ethernet distributed computing.

An internal Xerox memo proposes a system of interacting workstations, files, and printers, linked via one coaxial cable within a local area network, where individual components can join or leave without disturbing the data traffic. The memo's author coins the term "Ethernet" to describe the network. Ethernet grows into a global standard.



Figure 39: An Ethernet cable and connector

Ethernet (IEEE 802.3) is a network protocol that governs communication between computers on a Local Area Network (LAN). Ethernet is now widespread and has been released in several variants. Other protocols can run above the Ethernet layer, of which TCP/IP is the best known and most widely used.

Ethernet was one of the many pioneering projects developed at Xerox PARC (Palo Alto Research Center). The generally accepted story is that Ethernet was invented in 1973, when Robert Metcalfe wrote a memo to his bosses at PARC about its potential. Metcalfe says that Ethernet matured over subsequent years. Robert Metcalfe and David Boggs (Metcalfe's assistant) published a document entitled Ethernet: Distributed Packet Switching For Local Computer Networks in 1976.

Metcalfe left Xerox in 1979 wanting to promote networks (LANs) for use with personal computers, and founded the company 3Com. He was successful in convincing DEC, Intel, and Xerox to work together in agreeing a standard for Ethernet. The standard was published in September, 1980. The two main alternatives were Token ring (which was developed by IBM), and ARCNET. Both have fared well with many Ethernet products coming to the market. 3Com experienced enormous growth during this period in the 1980s and 1990s.

1973.

Superpaint frame buffer

Xerox PARC computer scientists record the first video image on the first computer paint system – a graphics program and frame buffer computer – paving the way for the earliest computer animations, and later earning its inventors Emmy and Academy Awards.

1973.

The Alto personal workstation

The Xerox Alto personal workstation with its client-server architecture moves computing beyond the hierarchical world of large, centralised mainframes. This evolving PC will subsequently employ the world's first bit-mapped display, sport a graphical user interface (GUI) with windows and icons, a what-you-see-is-what-you-get WYSIWYG editor, local area network connections and file storage, and commercial mouse.

HISTORY OF THE COMPUTER

1974.

WYSIWYG in the Bravo word processor

Introducing cut-and-paste bitmapped editing, Xerox PARC coins the catchy phrase describing its benefits: what-you-see-is-what-you-get (pronounced “wizzy-wig”). Xerox PARC also demonstrates the seminal Bravo word processing program (a precursor of Microsoft Word) and device-independent imaging (which leads to the development of Page Description Languages and influences the subsequent design of Postscript).

1974.

Solid-state lasers

Xerox PARC demonstrates the first gallium-arsenide (GaAs), distributed feedback, solid-state laser. In 1982, Xerox PARC demonstrates the world's first high power solid-state semiconductor diode laser. This was later brought to market by Spectra Diode Labs (acquired subsequently by JDS Uniphase in 2001).

1974.

The first 'personal computers', the Scelbi and the Mark-8, are introduced.

The Mark-8 was a microcomputer designed in 1974, based on the Intel 8008 CPU (which was the world's first 8-bit microprocessor). The Mark-8 was designed by the graduate student Jonathan Titus and billed as “loose kit” in the July 1974 issue of the magazine Radio-Electronics. The cover article introduced the Mark-8 as a Do It Yourself construction project, offering a \$5 booklet containing circuit board layouts, with Titus himself arranging for \$50 circuit board sets to be made by a New Jersey company for delivery to hobbyists. Prospective Mark-8 builders had to gather the various electronics parts themselves from a number of different sources. A couple of thousand booklets and some hundred circuit board sets were eventually sold. The phrase “Your Personal Minicomputer” first appeared here.

As the microcomputer revolution had yet to happen; the word 'microcomputer' was still far from being common fare. Thus, in their announcement of their computer kit, the editors quite naturally placed the Mark-8 in the same category as the era's other 'minisize' computers

Scelbi (Scientific Electronic Biologicals, pronounced “sell-bee”) Computer Consulting was a personal computer hardware and software manufacturer in Milford, Connecticut, founded in 1973 by Nat Wadsworth and Bob Findley. Initially they sold hardware (called the Scelbi-8H), based on the first 8-bit microprocessor from Intel, the 8008. The 8H came with 1K random-access memory, and you could either buy the Scelbi-8H fully assembled or in the form of a kit of parts for self-assembly. Some sources consider Scelbi as the first (March 1974) personal computer that could be bought as a kit, with advertisements in QST, Radio Electronics and later in BYTE magazine.

Scelbi soon had several competitors such as the Mark-8 mentioned above which was also based on the 8008 processor. Firms like PROVIDED began selling systems based on more powerful microprocessors, such as the 8080 used in the MITS Altair 8800. The Scelbi-8B was then introduced, with 16K memory (the limit of 8008). Initially, no programming help was available for the Scelbi-8H. Wadsworth wrote a book, Machine Language Programming for the 8008 and Similar microcomputers that taught the assembly language and machine language programming techniques needed to program the 8H. The book contained a code example for a floating point package, making it one of the earliest examples of what would later come to be called “open source”. Because of the similarities of the machine language used by both the 8008 and the 8080, the book was also purchased by users of other computers.

HISTORY OF THE COMPUTER

The Scelbi company found that they made more money selling software books than hardware, so by the late 1970s the company had discontinued making hardware and switched to well documented software published in book form, including many games, a monitor, an editor, an assembler, and a high-level language dubbed SCELBAL (*a dialect of BASIC that incorporated Wadsworth's floating-point package*) to compete with Altair BASIC.

1975.



Figure 41: Bill Gates and Paul Allen in 1983



Figure 40: The Scelbi-8H

Paul Allen and Bill Gates found Microsoft.

Microsoft (*originally Micro Soft*) was started in Albuquerque, New Mexico in 1975 by Allen and Gates who had been buddies and fellow Basic programmers at Lakeside School in Seattle. Their first product was a **Basic interpreter**.

Paul Allen later bought a CP/M clone operating system for \$50,000 called QDOS (Quick and Dirty Operating System). This became Microsoft Disk Operating System, MS-DOS, introduced in 1981. Microsoft licensed their system to multiple computer companies requiring the use of the MS-DOS name, with the exception of the IBM variant (PC-DOS), which was installed on all its new PCs. Through this 1981 deal Microsoft's income soared.

Paul Gardner Allen (*Born in Seattle, 1953*)

According to Forbes magazine, he is one of the richest people in the world with an estimated fortune of \$13 billion. In high school he became friends with **Bill Gates** who was two years his junior. Both teenagers were computer geeks. After high school, Paul Allen went to Washington State University. He left after two year's study to develop software for personal computers. In 1983, Paul Allen with diagnosed with Hodgkin's disease, following which he resigned from Microsoft in 2000. Radiation treatment effected a complete cure. In November 2009, he was diagnosed with a non-Hodgkin's lymphoma.



Figure 42: Bill Gates

Bill Gates (*born Seattle, 1955*)

Bill Gates was born the son of a lawyer and a schoolteacher. He has two sisters. He was sent by his parents to the Lakeside Prep School in Seattle where aged 13 he first started to program a computer (a DEC PDP-10 on which the school rented time from General Electric).

HISTORY OF THE COMPUTER

Gates with Paul Allen, a friend and later a business partner, were hardly able to tear themselves away from this computer terminal. The pair even hacked into the system that recorded the amount of computer time used, and altered it.

The Computer Centre Corporation in Seattle struck a deal with Lakeside programming group (*Gates, Allen and two other students*) offering them unlimited computer time on the Lakeside terminal in return for their help in identifying bugs and weaknesses in the system. The Computer Centre Corporation went out of business in 1970, but the Lakeside programmers were hired by Information Sciences Inc. to create a payroll program in exchange for free computer time and royalties on the software. Gates and Allen then worked together on Traf-O-Data, a program which measured traffic flow which earned them about \$20,000. Gates then went to college, but dropped out in his first year, 1975, to form Micro-soft, since his heart was not in his studies.

During a visit to Mozambique in 2003, Gates pledged \$168 million from the **Bill & Melinda Gates Foundation** to fund research into malaria, to develop new vaccines and drugs effective against strains of the malaria parasite which are resistant to currently available treatments.

Together with the Rockefeller Foundation, several companies (including Dow Chemical) and the Norwegian government, Gates' Foundation has invested tens of millions of dollars in the Svalbard Global Seed Vault. Here millions of seeds such as wheat, rice and corn are stored in a huge bunker on a Spitsbergen mountain for the protection of future crop diversity.

In 2005, Gates and his wife received the TIME Magazine Persons of the Year award. Gates has said that after his death his three children will each inherit \$10 million, and his remaining billions will be given to charity.

On January 6, 2008 the 52-year-old billionaire gave his last major speech in Las Vegas before retiring from Microsoft saying his farewell at the Consumer Electronics Show (CES). CES is the biggest consumer electronics trade show held annually in Las Vegas. Gates had opened this IT fair eleven times presenting his vision of new Microsoft Plans.



Figure 43: A police photograph of Bill Gates taken following a traffic violation.

HISTORY OF THE COMPUTER

1975.

IBM begins mass production of its first personal computer the IBM 5100.



Figure 44: The Model 5100, IBM's first non-mainframe computer, believed to be the world's first portable computer.

The Model 5100 weighed nearly 12 kilos, so is better described as a “luggable” rather than a portable computer. There were few personal computers available in 1975, and nothing that even came close to the capabilities of the 5100. It was a very complete system - with built-in monitor, keyboard and disk storage. It was a very expensive system costing up to €15,000, but was designed for professional and academic researchers, not for commercial or hobbyist use.

The MITS Altair 8800 personal computer is introduced.



Figure 45: The MITS Altair 8800

The MITS Altair 8800 was a microcomputer based on the Intel 8080A processor. Ed Roberts' company PROVIDED manufactured the Altair 8800 as a kit, and sold it through the American Popular Electronics magazine for \$397 dollars. Its designers expected to sell a few hundred kits to hobbyists, but to their surprise, they sold ten times that number within the first few months of advertising it. Today the Altair is widely recognized as marking the beginning of personal computer development which mushroomed in subsequent years. The internal computer bus (S-100 bus) and the first programming language for this unit (Altair BASIC) were sourced from Microsoft

HISTORY OF THE COMPUTER

1975.

Stephen Wozniak and Stephen Jobs establish Apple Computer Corporation to establish and sell the Apple I in kit form.



Figure 46: Steve Wozniak

Steve Wozniak, nicknamed '**Woz**' (*born in Sunnyvale, 1950*) is an American engineer who founded the computer company **Apple** in 1977 with **Steve Jobs** and **Ronald Wayne**.

Wozniak and Jobs knew each other at high school where they were considered nerds because they were both interested in electronics.

Steve Wozniak made a so-called "blue box" which enabled its owner to make free phone calls from an ordinary telephone.

After high school they both worked at computer companies in Silicon Valley, Wozniak at **Hewlett-Packard** and Jobs at **Atari**. They remained in touch when Wozniak received his computer engineering degree from the University of Berkeley.

Wozniak was Apple's technical expert in the early days of the company, responsible for the first versions of the operating system and hardware of the Apple I and the more famous Apple II, which was the most advanced PC produced until it was eclipsed by the success of the Commodore 64.



Figure 47: The Apple II with two disk drives

1975.

Graphical user interface (GUI)

Xerox PARC debuts the first GUI, a user interface which used icons, pop-up menus, overlapping windows and could be controlled easily using a point-and-click technique. That GUI famously (or infamously) influenced the development of all subsequent personal computer interfaces.

1977.

Apple Computer introduceert de Apple II. 1977.

Steven Paul Jobs (born in 1955) is an American business magnate and inventor. He is best known for being the co-founder and chief executive officer of Apple. Jobs also served as chief executive of Pixar Animation Studios, and became a member of the board of the Walt Disney company in 2006, following the acquisition of Pixar by Disney.

In the late 1970s, Jobs, with Apple co-founder Steve Wozniak, Mike Markkula and others designed, developed, and marketed one of the first commercially successful lines of personal computers, the Apple II series.

In the early 1980s, Jobs was among the first to see the commercial potential of the mouse-driven graphical user interface, which led to the creation of the Apple Macintosh computer. After losing a power struggle with the board of directors in 1985 Jobs resigned from Apple and founded NeXT, a computer platform development company specializing in the higher education and business markets. Apple's subsequent 1996 buyout of NeXT brought Jobs back to the company he cofounded, and he has served as its CEO since 1997.

In 1986, he acquired the computer graphics division of Lucasfilm Ltd which was spun off as Pixar Animation Studios. He remained CEO and majority shareholder until its acquisition by the Walt Disney company in 2006. Jobs is currently a member of Disney's board of directors. Jobs' business history has added to the image of the idiosyncratic, individualistic Silicon Valley entrepreneur, emphasizing the importance of design and understanding the crucial role aesthetics play in public appeal. His work driving forward the development of products that are both functional and elegant has earned him a devoted following.

Jobs is listed as either primary inventor or co-inventor of over 230 patents (some pending) relating to a wide range of computer applications including portable devices, user interfaces (including touch-based interfaces), speakers, keyboards, power adapters, clasps and lanyards.



Figure 48 Steven Jobs

HISTORY OF THE COMPUTER

1977.

Radio Shack introduces the TRS-80



Figure 49: TRS-80 Model 4P

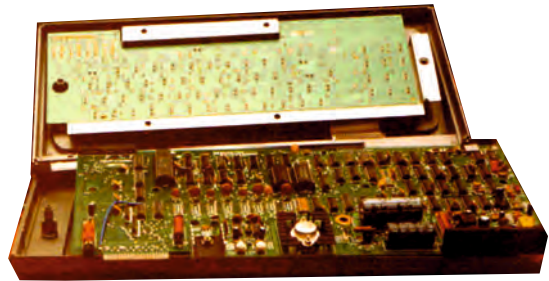


Figure 50: A disassembled TRS-80 Model 4P keyboard

In the late 1970s and early 1980s the then newly merged companies Tandy and Radio Shack named several series of their computers TRS-80. TRS is an acronym made from the names Tandy Radio Shack (formed from the initials of these two companies), and the number 80 referred to the Z80 microprocessor, on which the first model was based.

There were different series:

- The Z80-based home models. (Model I, III and IV)
- The Z80-based business models. (Model II and 16)
- The laptop style TRS-80 Models 100 and 200
- The TRS-80 Color Computer. ('CoCo' 1, 2, and 3)
- The TRS-80 MC-10.
- TRS-80 Pocket Computer (PC1, PC2, PC3 and PC4).
- PC-compatible computers.

The models based on the Zilog Z80 processor were extremely popular in the late 1970s. The Color Computers were based on the Motorola 6809 processor. Their Motorola 6847 video display controller provided a high resolution color display which was a great improvement over the blocky, monochrome displays of the Z80 models. The MC-10, or Micro Color Computer, was a miniaturised version of the Color Computer, equipped with a 6803 processor. The TRS-80 brand of Pocket Computers were also released as American models made by Sharp and Casio. Later on, the PC-compatible computers modeled after the IBM line of 80x86-based personal computers were released. The Tandy 2000 had better graphics, a faster 80186 processor and larger disk drive capacity than the original IBM PC.

The TRS-80 Model 1 was the first of the series. It had a thick keyboard connected to a separate monochrome monitor. The computer equipped with ROM which included the Basic language and 4 KB of RAM. Software and data could be stored on compact cassette recorder connected to the main unit. The first simple version of BASIC was soon replaced by a more complete version licensed from Microsoft (level 2). The Model I could also be extended with a so-called Expansion Interface (EI) which provided more memory (up to 48KB), a parallel printer connection, up to four disk drives (single sided, single-density 5¼ inch floppy drives), and a RS232 serial connection. In addition to BASIC, you could also program in assembly or machine language. Many hobbyists extended the Model I with hardware modifications such as increasing the processor speed (from 1.7 to 2.5 MHz). A TRS-80 Users Association was started in the Netherlands in October 1978 which published a bi-monthly magazine *Remarks*.

HISTORY OF THE COMPUTER

Initially, the TRS-80 was bought only by computer enthusiasts, but the appearance of spreadsheet and word processor applications (Visicalc and Scripsit) led to its widespread adoption by small businesses, and this formed the start of the PC era.

The Model III when it appeared had either one or two integrated floppy disk drives, and a more advanced Basic. Unlike the Model 1, the keyboard, floppy drive(s) and monitor were all contained in a single enclosure.

The Model II was a different type of computer that was aimed at business users, and had larger 8 inch floppy drives.

The Model 4 (April 1983, with "4" written as an Arabic numeral), was the successor of the Model III, and included the capability to run CP/M.

VisiCalc was the first spreadsheet which became available for personal computers. The spreadsheet is widely regarded as the “killer application” which took the personal computer from being a hobbyists' toy to being an essential and widely used business tool.

1978.

Dan Bricklin and Bob Frankston write the first spreadsheet application: Visicalc and found Software Arts Inc. in 1977.



ITEM	NO.	UNIT	COST
MUCK	1	1	5
BUNZ	4	1	6
CUT	2	4	12
TOE	2	6	12
TONE	2	6	12
SNUFF	2	6	12
SUBTOTAL			131
9.75% TAX			12.8
TOTAL			144.38

Figure 51: Dan Bricklin Figure 52 A small Visicalc spreadsheet.

Daniel (Dan) Bricklin (born in 1951) is the American co-creator, with Bob Frankston, of the VisiCalc spreadsheet program, which ran first on the Apple II. Bricklin soon produced versions for the Tandy TRS-80, Commodore PET, Atari 800 and IBM PC. Soon after its launch Visicalc became a best seller at \$100.

The program was created by Dan Bricklin, refined by Bob Frankston, developed by their company Software Arts Inc. and distributed by Personal Software (later renamed VisiCorp) for the Apple II computer, transforming the Apple II from a hobbyist's toy into a much desired, useful financial tool in the business world.

Bricklin had noticed his Harvard professor's frustration when drawing a financial model on the blackboard.

If he found an error, or wanted to change a parameter, the corresponding data needed a lot of annoying delting and rewriting. Bricklin could foresee the value of computerising these sequences of calculations, and realised at an electronic spreadsheet would enable almost instant recalculation of underlying formulas.

Bricklin also founded Software Garden, Inc., of which he is currently president, and Trellix Corporation.

HISTORY OF THE COMPUTER



Figure 53: Dan Fylstra

Dan Fylstra is a pioneer of the computer industry. In 1975 he was a founding associate editor of **BYTE Magazine**. In 1978 he co-founded Personal Software which became the distributor of a new program called VisiCalc, the first widely used computer spreadsheet. In his marketing efforts Fylstra ran teaser ads in Byte that asked (oddly enough for an entirely new product) "How did you ever do without it?"

Once VisiCalc caught on, people came into computer stores asking for VisiCalc and then also the computer (the Apple II) they would need to run the program. VisiCalc sales exceeded 700,000 units by 1983, and Fylstra's software products company, later called VisiCorp, was the leading personal computer software publisher in 1981 with revenue of \$20 million, as well as in 1982 with \$35 million (exceeding Microsoft which became the largest such firm in 1983). Fylstra is the former president of Sierra Sciences, and is currently president of software vendor Frontline Systems. In 1998 he joined the Libertarian Party.

The successors to Visicalc

Despite the fact that an electronic spreadsheet was a revolutionary idea, Bricklin was advised that it was unlikely that he would be assigned a patent, so he missed the chance to gain a lot to do with his invention.

It is said that at that time the law on patent did not apply to software, so the product could only be copyrighted, and after some time the copyright would fail to protect the product (though simple look-and-feel tests for copyright infringement were developed later).

More powerful clones of VisiCalc soon came along: **SuperCalc**, Microsoft **Multiplan**, Borland's **Quattro Pro**, **Lotus 1-2-3**, Microsoft **Excel**, OpenOffice.org **Calc**, and AppleWorks' spreadsheet module **gnumeric**.

Lotus 1-2-3 was the first successful VisiCalc clone for the IBM PC.

Because of Visicalc's lack of a patent protection, none of the companies that developed competing software needed to pay royalties to VisiCorp.

1977.

VLSI circuit design

Xerox PARC (with Caltech) defines a new type of Very Large Scale Integration circuit design, which provides greater computing power in more compact machines, reduces design time, and leads to a new generation of computer-aided design tools in which many thousands of transistors and other components can be compressed into a single integrated circuit to make microprocessors.

1978.

Worm programs

While experimenting with distributing computations across machine boundaries, PARC scientists invent a "worm" program that searches out other computer hosts and replicates itself in idle machines.

HISTORY OF THE COMPUTER

1979.

Corporate ethnography

Initiating collaboration among computer scientists, engineers, anthropologists, sociologists, psychologists, and other social scientists, PARC pioneers the use of ethnography for human-centred technology design, work practice redesign, and more. This approach leads to improvements in many workspaces, office products, and processes.

1979.

Natural language processing

To enable computerised spell-checking, dictionaries, and other computer text tools, Xerox PARC invents computational linguistic technologies based on understanding the structure of language. These lead to computer-automated visual recall, intelligent retrieval, and linguistic compression... and later enable deep meaning-based language parsing systems for search, text analytics, and more.

1980.

Optical storage

Non-erasable, magneto-optical storage device technologies, developed at Xerox PARC initially to enable high-speed data access on the Alto, are commercialized through Optimem (which evolves into Cipher Data Products).

1980.

Programming language development

Xerox files the software copyright for Smalltalk-80 – one of three software copyrights in existence. Meanwhile, Interlisp, the Mesa programming environment, and its successor, Cedar, are implemented in Xerox systems, enhancing reliability and supporting rapid development.

1981.

IBM introduces the IBM PC, a personal computer

IBM introduced their IBM PC in August 1981. It was a 16-bit computer with a 20-bit address bus, based on the 8086/8088-microprocessors running at 4.77 MHz. This PC was designed and built by IBM, but built from standard components that were not specific to the IBM PC. Only the BIOS firmware was specifically developed for the IBM PC and contained a BASIC interpreter. The first computers stored their data and software on floppy disks. IBM wanted other companies to produce ISA bus expansion cards for the for the PC and therefore published the specifications in a \$49 book.

They also released a PC version with a 10 MB hard drive, called the **IBM PC XT** (for eXtended Technology). In 1984 they released the **IBM PC AT** (Advanced Technology), based on the Intel 80286 processor. These ran at 6 MHz and had a 20 MB hard disk as standard.

The IBM PC was the basis for the popular IBM-PC compatible computers or clones. Since 1984 the IBM personal computer division has released several series of personal computers for home and business use, such as the PCjr, PS/1, PS/2, PS/ ValuePoint, ThinkPad, Aptiva, PC Series, NetVista and ThinkCentre. Ultimately, IBM sold their PC division to Lenovo in 2005. IBM introduced x86-based servers starting with certain PS/2 server models, which were succeeded by the IBM PC Servers, Netfinity, eServer xSeries and today's current IBM System x.



HISTORY OF THE COMPUTER

1981.

Microsoft brings out MS-DOS version 1.0.

MS-DOS stands for Microsoft Disk Operating System. It was one of the first operating systems for personal computers.

When IBM started development of the IBM PC it had intended to use its own operating system, and use only Microsoft's Basic interpreter.

When the proprietary operating system negotiations proved problematic, IBM turned to Microsoft who contracted to write an operating system. Bill Gates decided not to start writing a new system, but rather look to license or buy another company's product.

The operating system that Paul Allen found was QDOS (a CP/M derivative, the Quick and Dirty Operating System), which Microsoft bought for \$50,000 and then modified to meet IBM's needs. IBM called it PC-DOS.

Under Microsoft's contract with IBM, Microsoft could also sell PC-DOS independently to others. Microsoft did this under the name MS-DOS. When IBM PC clones hit the market, MS-DOS was sold with almost all of them, and thus Microsoft started its dominance in the world of computer operating systems.

```
CP/M-86 Bootstrap Loader 1.0

CP/M-86 for the IBM Personal Computer.
Version 1.0
Copyright 1982, Digital Research Inc.

Hardware Supported :
      Diskette(s) : 2
      Printer(s)   : 1
      Serial Port(s) : 1
      Memory (Kb) : 256

A>dir
A:  ASHG6  CHD : TOD      CHD : COPYDISK  CHD : DDT06  CHD
A:  ASSIGN  CHD : NEWDISK  CHD : FUNCTION  CHD : PROTOCOL  CHD
A:  ED      CHD : SPEED   CHD : DU-075A  DOC : SUBMIT   CHD
A:  HELP   CHD : HELP    HLP : UF17     CHD : GENCMD   CHD
A:  DU     CHD : UNERRA31  CHD : PIP     CHD : Z80     CHD
A:  DU-75E CHD : STAT    CHD : UP      DOC : INSTL0  DOC
X>dir86
DDT86 1.1
-
|U=00|02-10-82|00:37:18|
```

Figure 55: The dir command in MS-DOS version 1.0

MS-DOS in Windows

Windows 3.0, Windows 3.1, Windows 95, Windows 98 and Windows ME all used MS-DOS to start, after which the graphical interface takes over, running on top of MS-DOS. MS-DOS is not a multitasking operating system, meaning that only one program can run at a time. The use of virtual mode (in practice, the V86-mode of the 80386 processor and higher) made it possible for different MS-DOS programs to each run in their own virtual machine. This method was problematic, however, because many MS-DOS programs then tried to directly control the hardware. This was possible in advanced systems, but at the expense of stability (since setting up the hardware wrongly, caused the computer to crash).

Windows 3.x and 9x operating systems allow direct communication with the hardware, (unless a driver virtualizes access). Windows NT operating systems do not allow direct hardware access. There must always be drivers that enable hardware access for MS-DOS programs running under NT. The Windows NT operating systems, however, cannot control how third-party hardware drivers implement this.

HISTORY OF THE COMPUTER

MS-DOS versions:

Date	Version	Features
August 1981	1.0	Introduction of autoexec.bat
March 1982	1.1	Date and time change from the command line Support for double-sided disks
March 1982	1.25	First OEM-version New command: VERIFY
March 1982	2.0	Support for hard drives and subdirectories Open multiple files at the same time Printer buffer introduced ANSI-driver introduced
Octob. 1983	2.1	
Aug 1984	3.0	Support for networks
March 1985	3.1	
Decer 1985	3.2	XCOPY command introduced
April 1987	3.3	NLSFUNC and FASTOPEN introduced Support for hard drives larger than 32 MB
June 1988	4.0	DOSSHELL Introduced
June 1991	5.0	Introduces QBasic, MIRROR, UNDELETE, EDIT, UNFORMAT and memory optimization
August 1993	6.0	DoubleSpace (disk compression) introduced with Defrag (Disk Defragmenter) and antivirus
1993	6.1	Version number only for IBM PC-DOS users
1993	6.2	
1993	6.21	Version released without DoubleSpace due to Stac Electronics suit for patent infringement
1994	6.22	DriveSpace replaces DoubleSpace Scandisk introduced
1995	7.0	included with Windows 95 The computer usually starts with MS-DOS "hidden" in Windows Long file name support for Windows 95 (not MS-DOS mode)
1997	7.1	included in Windows 95 OSR2 and in Windows 98 FAT32 introduced
2000	8.0	Included in Windows Me SYS-command removed Command-line boot capability removed



Figure 56: The last standalone DOS – MS-DOS 6.22

The development of MS-DOS as an independent operating system was discontinued after version 6.22, the last standalone version. From version 7.0 MS-DOS was no longer a separate operating system, but integrated directly into Windows. DOS was still quite visible in Windows 95 and 98. By Windows ME however, most traces of DOS had been erased.

A Chinese DOS fan (Wengier) has released the most comprehensive standalone version of DOS. This is MS-DOS 7.1. Microsoft introduced Windows 95 OSR2, which was free for anyone to download, but currently this is not possible anymore.

The acronym DOS stands for Disk Operating System, or an operating system for devices with disk drives. With the advent of computers with floppy disks an operating system was needed that after the bootstrap opening cycle could load a command processor and other software utilities.

HISTORY OF THE COMPUTER

DOS (Disk Operating System)

The History of DOS

In 1973 Gary Kildall of Digital Research wrote one of the first disk operating systems in the programming language PL/M. He called it **CP/M** (Control Program for Microcomputers). Six years later, in 1979, Apple had developed its own operating system with **Apple DOS 3.2**.

In 1980 Tim Paterson of Seattle Computer Products (SCP) developed a DOS for the 8086 based on CP/M, because Digital Research had delays in releasing the CP/M-86 operating system. This SCP operating system was named **QDOS** which stands for Quick and Dirty Operating System because it was built in only two man-months. Despite its rapid development, it operated very well.

In October 1980 Paul Allen from Microsoft contacted SCP with the request that Microsoft sell SCP's QDOS to an unnamed client (which later turned out to be IBM). Microsoft would pay \$50,000 for the rights to SCP. Two months later SCP renamed QDOS to 86-SCP and brought it out as version 0.3. Microsoft bought the (*non-exclusive*) rights to SCP's DOS.

In February 1981 a prototype of the IBM personal computer was running "MS-DOS" for the first time and in July 1981 Microsoft bought all rights to SCP's DOS and officially renamed it the MS-DOS operating system.

EDLIN is a line editor included in Microsoft operating systems and was the first program that allowed text files to be edited under MS-DOS. The program works like the MS-DOS command line (making it difficult to use by modern standards). It is not a full word processor, but intended for editing configuration files, etc.

The program was written by Tim Paterson in two weeks in 1980 on the assumption that it would have a life of about six months. Those who use Windows XP, Windows Vista, Windows 7 or Windows 2003 Server, can see that EDLIN is still included, as its successor the EDIT program.

How DOS operates

After the POST (Power On, Self Test) process, the PC starts with a number of DOS programs including the AUTOEXEC.BAT file. This file specifies the processes to be run as DOS starts. After running AUTOEXEC.BAT, the user is placed at the command line, where commands can be typed.

CP/M successors include:

- . **MS-DOS**
- . **DR-DOS (successor to CP/M-86)**
- . **FreeDOS**
- . **OpenDOS**
- . **PC-DOS (5.0, 6.0, 6.1, 6.2, 6.22, 6.3, 7.0, 8.0)**
- . **PTS-DOS**

1982.

Fibre Optics

The first optical-cable-based local area network became operational in 1982. Eventually fibre optic media came to enhance all commercial communications, particularly over large distances, for computer networks, phone lines, internet communications and cable TV networks. Optical fibre suffers less signal attenuation and less interference compared to copper cables, and has an inherently high data-carrying capacity. General Telephone and Electronics had already sent the first live telephone traffic through a fibre optic cable (at 6 Mbit/s) in Long Beach, California in 1977.

HISTORY OF THE COMPUTER

1983.

a-Si for printing

Amorphous silicon can be applied in much thinner layers and at lower temperatures than the crystalline silicon used in the 1970s. This led to new (or cheaper) applications. Xerox used a-Si thin-film transistors to drive a small Corjet ionographic printhead. The technology enabled Xerox to offer lower-cost multifunction machines and, in 1988, the first wide-format engineering laser plotter. Amorphous silicon has become the material of choice for the active layer in thin-film transistors (TFTs), which are most widely used in large-area electronics applications, such as the liquid-crystal displays (LCDs) used in large-format computer monitors and flat-screen TVs.

1984.

Macintosh

Mac or Macintosh is the name of a series of computers developed by the American company Apple.

The first Macintosh was introduced in 1984 primarily as a cheaper sequel to the **Apple Lisa**, the computer which introduced a visual, **mouse-driven** user interface which, though ahead of its time, was too expensive to be a commercial success.

The name was coined by Jef Raskin, who named his favorite Apple McIntosh.

To avoid problems with the US audio brand McIntosh, the name was spelled Mac (with an 'a'). The first Apple Macintosh (later branded the Macintosh 128K) appeared in 1984, with a full graphical user interface, called Mac OS.

The development of the Mac was begun in 1979 under the leadership of Jef Raskin at Apple, who introduced several ideas from Xerox PARC. Steve Jobs, Bill Atkinson and other Apple staff visited the Xerox PARC lab, after which Apple adapted several different ideas from Xerox PARC's mouse-driven Alto workstation.

However, the Apple team developed the overlapping window interface, the ability to move or remove icons, the "cut-and-paste" metaphor, and a menu bar that looked the same in each program.

Douglas Engelbart's 1963 introduction of the mouse was taken up fully in the Macintosh GUI. User did not need to type cryptic commands at the command-line. After turning on the Macintosh computer, you saw (on a nine inch black-and-white display) a symbolic desktop, with icons for a cabinet, sheets of paper, folders for storing them, and a waste bin where they could be thrown away.

Apple kept the operating system out of sight as much as possible.

The Mac was based on a **Motorola 68000** series processor, and consisted of a system (called System) that loaded as much as possible into RAM, a file management program (called the Finder) and a growing number of extensions. Modifications to the system could be implemented by control panels which set various parameters, the Control Panel.

The Macintosh was introduced on January 22, 1984 with a one-minute commercial during the U.S. Super Bowl game. Approximately half the population of the United States saw this. Two days later it was officially introduced by Apple Computer founder Steve Jobs. The first Mac cost between \$1,995 and \$2,495.



Figure 57: The first 1984 Macintosh

HISTORY OF THE COMPUTER

In many ways the Macintosh was the forerunner of what is now called the PC. Computers without mouse control and overlapping windows were hardly on sale anymore around 2002.



Figure 58: Power Mac G3 B&W from 1999

The first Macintosh ran at a clock speed of 7.83 MHz. By 2003 that speed had been increased by a factor of 500.

The original Mac operating system (MacOS) was phased out around 2002 and replaced by the BSD Unix-based Mac OS X. The first version of this derived from the mouse-based Unix variant NeXTStep and debuted in 2001 (Mac OS X 10.0). More recent versions of Mac OS X have lost emulation of the Classic.

On June 5, 2005 Steve Jobs announced that Apple would begin to switch from PowerPC to **Intel microprocessors** on account of the high power consumption and high heat produce by the G5 processors from IBM. Apple introduced the first Intel Macs with Core Duo processors in January 2006, the last Power Mac being produced in August 2006. The switch to Intel was also indicated by a name change: the MacBook becoming the iBook, the PowerBook becoming the MacBook Pro and Power Mac becoming the Mac Pro.



Figure 59: The iMac (2009)



Figure 60: Douglas Engelbart, inventor of the mouse

Douglas Engelbart was born in Portland, Oregon in 1925 to Carl and Gladys Engelbart. He grew up in Oregon, graduating from Franklin High School in Portland in 1942 and going on to study electrical engineering at Oregon State University.

Shortly before the end of World War II, however, he was conscripted into the US Navy where he served for two years as a radar technician in the Philippines.

He then completed his studies at Oregon State University, graduating in 1948. In 1953 he obtained an MS degree from the University of California, Berkeley, and in 1955 he obtained a Ph. D. at the same university.

He formed a start-up, Digital Techniques, to commercialise some of his doctoral research on storage devices, but after a year decided instead to pursue the research he had dreamt of since 1951, when he had first been inspired by reading Vannevar Bush's seminal article "As We May Think", taking a position at Stanford Research Institute (SRI) in Menlo Park in 1957, where he received funding from ARPA to found the Augmentation Research Center at SRI, where he recruited a new research team.

HISTORY OF THE COMPUTER

He was awarded US a patent in 1970 for a wooden box with two metal wheels, which was described as an “**X-Y position indicator for a display system**”. He nicknamed the device a “**mouse**” because the cord (tail) emerged at the back of the device. The on-screen cursor was termed a bug, but that name has not stuck.

It was first developed in 1963 by Douglas Engelbart and William English at (SRI). “It was Bill English,” says Engelbart, “who screwed together the first working prototype single-handedly, because English was one of those rare people whose hands can make anything that his eyes have visualised.”

The first mouse made by Engelbart and English had only one button fitted. “But,” Engelbart explains, “We quite quickly moved to fitting three buttons on our mice. We used simple switches such as those that were available commercially at the time. No one had heard of miniaturisation, so compared to today’s switches those first mouse buttons were quite crude. Three buttons was simply the maximum possible because of this physical limitation. Had the switches been smaller I would definitely have mounted more buttons on the mouse. For more buttons automatically mean more opportunities.”

This conception of the inventor is in stark contrast to the appearance of the first mouse that came standard with a computer: the Xerox Star was marketed with a single-button mouse. Computers, including those from Apple, had single-button mice for a long time. Only with the appearance of the IBM PC and compatible personal computers were mice given more buttons. The mouse as an input device, for managing graphical user interfaces had led to the subsequent booming development of operating systems such as the Mac OS X and Windows.

Engelbart invented not only the mouse, but was also the driving force behind hypertext, network software, teleconferencing, and the windowed graphical user interface - as it was first commercially applied by Xerox, and later by Apple and Microsoft. Many of these techniques were applied in the 1960s to the then revolutionary NLS (online system). Engelbart showcased the mouse, **chord keyboard**, **video conferencing**, **email**, **hypertext**, and other ARC inventions at “The Mother of All Demos” on December 8, 1968 where now commonplace technologies were first introduced as experimental ideas to about 1,000 computer professionals. Engelbart was also involved in the development of **Arpanet**, the Internet precursor that came into use in 1969.

Engelbart himself earned nothing for his inventions. For him, they were just footnotes in a much larger project: **to help humanity solve increasingly complex problems**. Engelbart slipped into relative obscurity after 1976. Fewer funds were available for the Augmentation Research Center, and many of his employees moved to Xerox PARC, disagreeing with his vision of a collaborative, networked future for computing.

Engelbart now leads a somewhat reclusive life, but is still active. With his daughter Christina he founded the Bootstrap Institute, now housed in a modest office with computer mouse manufacturer Logitech. He received several honors, including the Lemelson-MIT Prize, the Turing Award (both 1997) and from the hands of former President Bill Clinton, the National Medal of Technology (2000). In 2001 the British Computer Society awarded him the Lovelace Medal. Engelbart holds 20 patents.

HISTORY OF THE COMPUTER

The development of the mouse

A mouse was initially connected to the computer via the serial port. Later a PS/2 interface was used in IBM PCs. Today mice typically connect using a USB cable connector, or they work wirelessly via radio, infrared or Bluetooth.

The mouse works best if moved over a specially manufactured small mouse mat (or mouse pad). This is usually made from a layer of hard plastic with a foam backing underneath.

Displacement unit

A mouse works digitally and the movement is measured using a unit called the mickey (named after the famous Walt Disney mouse). The mouse driver might determine, for example, that the mouse has been moved left three mickeys.

A mechanical mouse has a mouse ball made of dense, hard material which rolls as the mouse moves over a flat surface. This rolling movement is transmitted by friction to two wheels mounted on axes perpendicular to each other. Two light beams monitor the movement of each axle, converting the movement into electronic pulses. This information is processed by the mouse driver and eventually turned into a corresponding cursor movement on the monitor. The ball picks up dirt in use which gets deposited on the axles, impeding the movement and accuracy of the mouse. The ball and the axles therefore need periodic cleaning. To this end, mechanical mice are designed to be easily disassembled.

Optical mouse

An alternative to the mechanical mouse that is less sensitive to dirt and wear and works even on surfaces that are not perfectly smooth is the optical mouse. This contains an LED (or laser in the case of a laser mouse) and a mini CCD camera which measures the mouse's movement relative to the stationary layer below the mouse optics. This provides more reliable and more manageable on-screen cursor movements.

The first optical mouse (invented by Steve Kirsch of Mouse Systems Corporation) depended on a specially patterned mouse pad, showing a grid of lines which were scanned by a dual photo sensor as the mouse moved over the pad. This type of specialised mouse pad is no longer needed, since the modern optical mouse detects tiny irregularities present in the subsurface. However, most optical mice still have problems with highly reflective, transparent or absorbing surfaces.

Trackball

Another variant is the trackball or rollerball mouse in which the ball is placed on top of the mouse. This has several advantages:

- Less space is required, because the mouse unit does not move.
- The ball is not contaminated by rolling on a dirty surface.
- The movement can be accurately controlled with a finger rather than the whole hand.
- The user is less prone to repetitive strain injury or RSI.

There are also disadvantages:

- Dragging (moving the mouse while a button is held down) is more difficult.
- Large distance movements are a little more time consuming.

HISTORY OF THE COMPUTER

Pen tablet

Another type of 'mouse' is the pen-tablet combination, where the tablet provides a surface that can be written on by the pen (or icons that can be selected with the pen). The pen tablet is primarily intended for artistic drawing or in medical and chemical laboratories where a panel of icons is used to give an overview say of particular cell types or chemical compounds. The pen tablet provides an alternative to the mouse to control the cursor and clicking. The pen can do more than just indicate the cursor position. The pressure of the pen, the position (angle) of the pen or the rotation of the pen can all be monitored, provided suitable software is installed. From 2007 the most popular drawing software used pen pressure variations to enhance its drawing capabilities.

The Penmouse

There are different types of pen mice:

- A pen-tablet combination (see above).
- A pen with a tiny normal mouse as its nib.
- A pen that passes all movements wirelessly to the computer. Unlike the pen-tablet, this type of mouse pen works on all surfaces. It requires an internal battery.

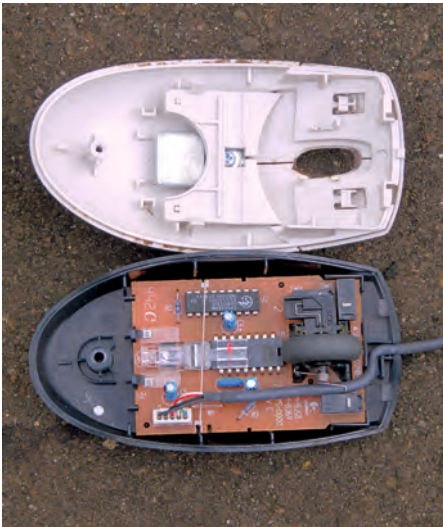


Figure 61: Inside a mouse

Head, eye, foot and hand devices

There are also devices that can be put on the head, so that your head movements will control the cursor.

There are several systems in which a camera looks at the pupil of the eye, and moves the cursor on-screen based on the position of the eye, though these systems do not yet work satisfactorily.

Then there is the foot mouse, where you control the cursor with your feet.

It is also possible to use a webcam that scans hand movements as a mouse. Currently a few games consoles use this system.

Touchpads are used in many laptops to replace the conventional mouse.

HISTORY OF THE COMPUTER

Multi-button mice

The first computer that provided a mouse as standard, the Xerox Star, only had a single button, and this style of mouse remained the norm for a long time with the Apple Macintosh series.

Apple has only recently succumbed to the temptation of a multi-button mouse.

Most mice produced in the 1980s for the original IBM PC and its clones had one button, except the Microsoft mouse. This had two buttons, one **for point-and-click selection**, and the other for invoking a so-called **context-sensitive menu**.

Today, three buttons are standard, but variations occur in specific applications such as Computer Aided Design and Computer Aided Manufacturing.

Scroll wheel mice

Incorporating a scroll wheel into mouse functionality is a relatively recent addition, where it usually replaces the centre button, and is then programmed to scroll the contents of a window horizontally or vertically, (or resize it). This is particularly useful when you view lengthy blocks of text or browse Web pages and don't want the cursor location to change.

The original middle button function is still available simply by depressing the central scroll wheel. This scroll wheel dual function avoids the need for a separate third button. Some earlier two-button DOS programs used simultaneous pressing of both buttons to invoke a third function, and under the X Window System this functionality is preserved. If a two-button mouse is connected, pressing both buttons simultaneously functions identically to pressing the middle button on a three button mouse.



Figure 62: The internals of a mechanical mouse

1. Mouse movement rolls the ball
2. Rollers record X and Y movements
3. Optical encoding disks with light holes
4. LED light source shines through the disk holes
5. Light-sensitive sensors transmit movement data to the computer

Other enhancements to functions are possible by combining a mouse click with a keypress such as [Ctrl] or [Alt] using the keyboard. This is common on Mac OS with a one-button mouse, where control-click (clicking while the [Ctrl] key is pressed) is equivalent to clicking with the right button on a multi-button mouse.

HISTORY OF THE COMPUTER

1985.

Microsoft markets its first 16-bit Windows version, followed by a 32-bit version in 1993 and a 64-bit version in 2003.

Windows is the name of Microsoft's line of personal computer operating systems. It was launched in 1985 and has dominated the personal computer market since the launch of Windows 95 in 1995. In 2004, Microsoft had 90% of the personal computer market. Microsoft's original design for Windows was probably influenced by previous initiatives by Xerox and Apple, which had foreseen the advantages of a graphical user interface for computer users.

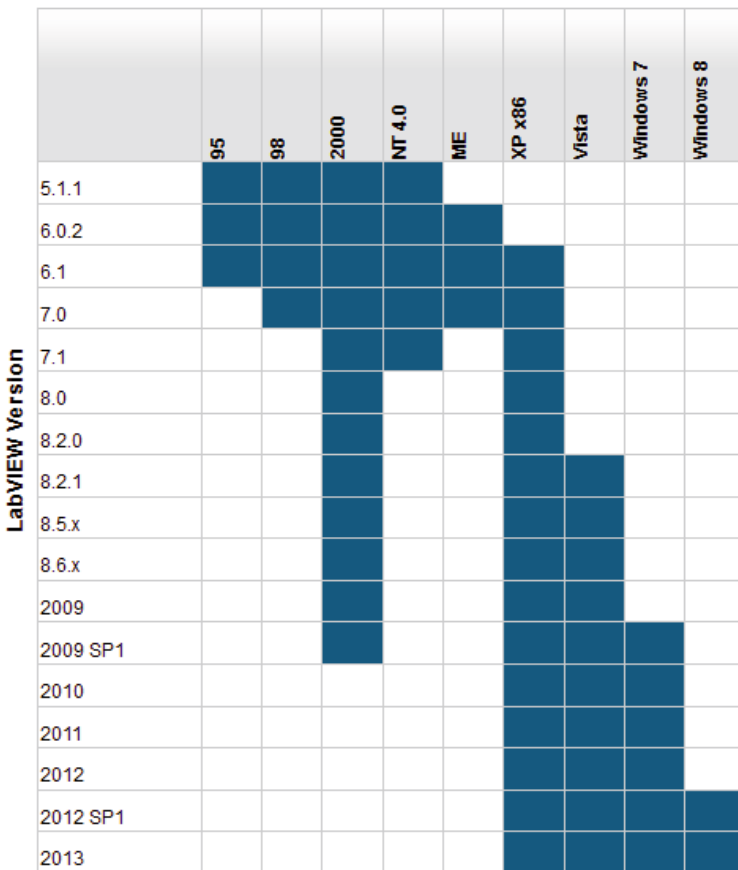
1985.

Intel makes the 80386 microprocessor.

The Intel 80386, also known as the i386, or just 386, was a 32-bit microprocessor introduced by Intel in 1985. The first versions had 275,000 transistors and were used as the central processing unit (CPU) of many workstations and high-end personal computers of the time. As the original implementation of the 32-bit extension of the 8086 architecture, the 80386 instruction set, programming model, and binary encodings are still the common denominator for all 32-bit x86 processors, this is termed x86, IA-32, or i386-architecture, depending on context.

The 80486 and P5 Pentium line of processors were descendants of the 80386 design.

Microsoft Windows OS Version



Compatible Version

HISTORY OF THE COMPUTER

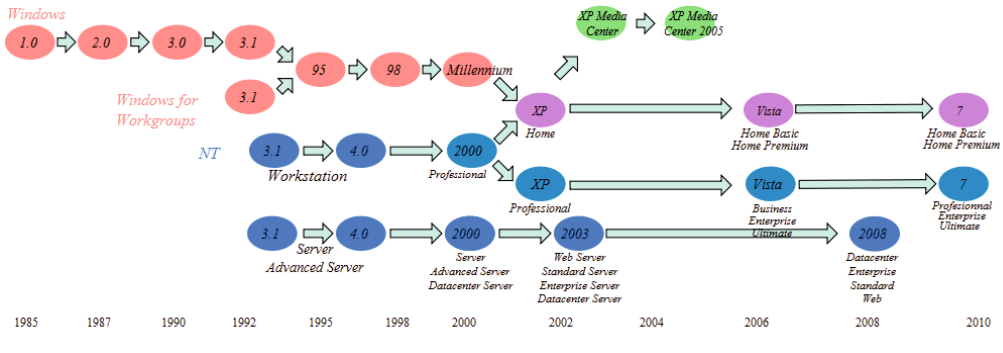


Figure 65: An overview of Microsoft Windows development up to 2010

1986. Multi-beam lasers

Xerox PARC is home to the world's first dual-beam laser printer, which prints twice as fast as single-beam models. Multi-beam lasers become a key factor in the development of high-speed, high-resolution production printing systems

1987. Collaborative workspaces and tools

A novel conference room at Xerox PARC, "COLAB", help inspire the development of document-based collaboration products for local and remote team members. for computational The room provided support for shared and private consultations and calculations through a 36 inch touch screen, and a million-pixel interactive personal "live board" which displayed multimedia images.

1987. Unicode and multilingual computing

Xerox PARC designs a 16-bit encoding system to represent any script from any language in the world, to be used in documents, user and file names, and in network services, in all combinations. This lead to the ISO/IEC 10646 and the corresponding Unicode industry standard which allowed computers to represent text consistently from country to country.

1988. Ubiquitous computing

The phrase "ubiquitous computing" was coined at Xerox PARC to describe a vision in which anyone anywhere, using a mobile device, could access resources and control digital environments seamlessly. Xerox PARC invents and builds Fundamental Enabling Devices, such as the palm-sized PARCTab, notebook-sized PARCPad, (a lightweight portable document reader), and a flexible computational infrastructure to enable fully interoperable wireless communication between devices.

1989. Embedded data glyphs

After inventing data glyphs, Xerox PARC pioneers the development of embedded data schemes that transforms paper, and other surfaces, into computer-readable interfaces. Applications evolve which include check verification, smart paper, and tracking tables

HISTORY OF THE COMPUTER

1989.

Intel introduces the 80486 microprocessor

The 80486 is a clock-tripled i486 microprocessor with 16 kB L1 cache. The product was officially named the IntelDX4, but OEMs continued using the i486 naming convention.

1989.

Encryption Systems

Encryption research at Xerox PARC enables the release of an NSA-endorsed electronic device that encoded computer signals mathematically so that they would travel safely over ordinary local area networks. Performing hardware-level encryption, these encryption systems (later commercialized through spinout Semaphore Communications) work much faster than most software-based products.

1989.

Information visualization

Taking a unique approach to the visualization of information, Xerox PARC invents techniques that use human cognitive perception capacities to help people make sense of large amounts of diverse information. The approach results in 3-D Rooms, a hyperbolic browser, and other "Focus+Context" visualization techniques that present three-dimensional views of database information.

1990.

Multi-user virtual world

Xerox PARC creates LambdaMOO – one of the longest continuously operating, real-time multi-user "dungeons" or online environments. LambdaMOO provides a foundation for the U.S. Department of Defense's collaborative computing systems. It later resulted in a company that provides live, Web-based meeting and presentation solutions (which ultimately became Microsoft Live Meeting).

1990.

X-ray imaging

Using amorphous silicon displays and digital x-ray imaging, Xerox PARC builds the first x-ray imager. Continuing research will result in the formation of the dpiX company, which will market the world's highest resolution active-matrix, liquid-crystal, flat-panel displays and a digital x-ray system for medical imaging that eliminates traditional film.



Figure 66: The Intel DX4 chip

HISTORY OF THE COMPUTER

1992.

Internet standards

The MBone multicast backbone is co-founded and first implemented at Xerox PARC to deliver real-time multimedia over the Internet. Xerox PARC scientists will also play a key role in co-designing the IPv6 protocols that govern and define how the Internet works, and help develop the HTTP-NG protocol based on Inter-language Unification (ILU) from Xerox PARC.

1992

Collaborative filtering

Collaborative filtering is implemented at Xerox PARC, inspired by the idea to involve human input (such as past user preferences and collaborators' feedback) in helping information systems auto-filter content. Today, this approach enables recommender systems.

1993.

Intel introduces the Pentium microprocessor

Following Intel's previous series of 8086, 80186, 80286, 80386, and 80486 microprocessors, the company's first P5-based processor was released as the original Intel Pentium on March 22, 1993.

1993.

Live artistic performance on Internet

Performing at Xerox PARC, the band "Severe Tire Damage" is the first musical group to broadcast live video and audio on the Internet, using the MBone.

1995.

Unistrokes

Xerox PARC's input technology for palm-sized devices, which enables single-stroke touch-screen input, is patented.

1996.

Intel introduces the Pentium Pro microprocessor.

Pentium is a registered trademark that is included in the brand names of many of Intel's x86-compatible microprocessors, both single- and multi-core.

The name Pentium was derived from the Greek pente (πέντε), meaning 'five', and the Latin ending -ium, a name selected after courts had disallowed trademarking of number-based names like "i586" or "80586" (model numbers cannot always be trademarked). The P5 was first released under the Pentium brand in 1993. In 1995, Intel started to employ the registered Pentium trademark also for x86 microprocessors with radically different microarchitectures (e.g., Pentium Pro, II, III, 4, D, M, etc.). In

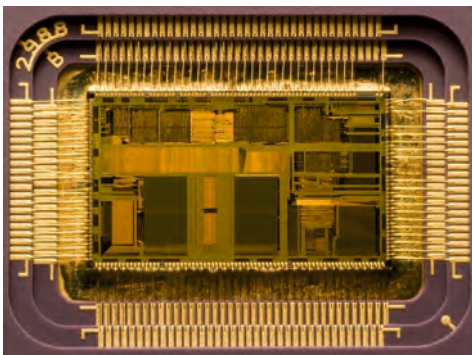


Figure 68: A CPU viewed from inside

HISTORY OF THE COMPUTER

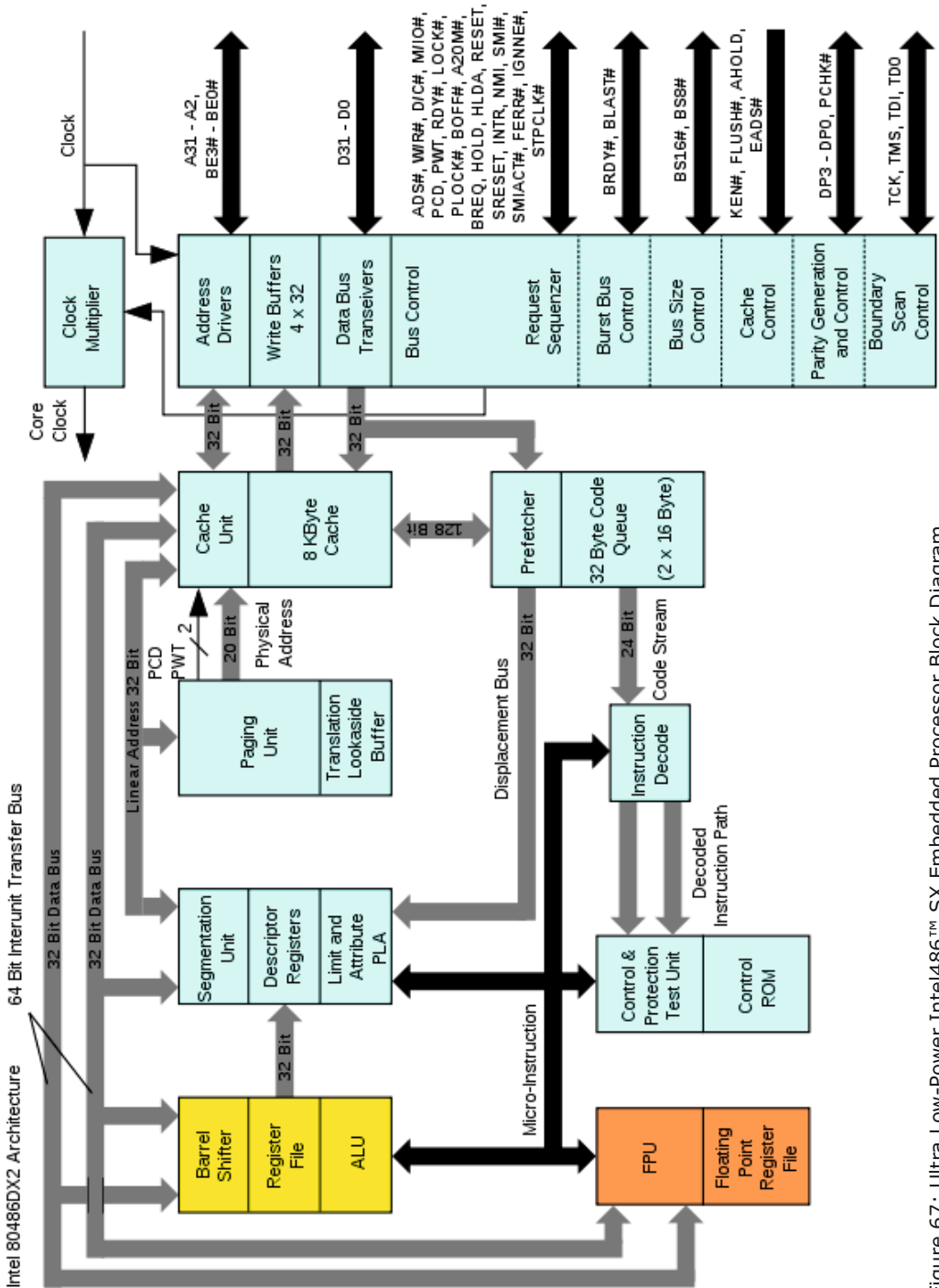


Figure 67: Ultra Low-Power Intel486™ SX Embedded Processor Block Diagram

HISTORY OF THE COMPUTER

1997.

Blue laser

Xerox becomes the first printing company to create a blue laser. The reduced wavelength of a blue laser may ultimately allow much higher-resolution printing than is possible with standard red and infrared lasers.

1997.

Intel introduces the Pentium II microprocessor.

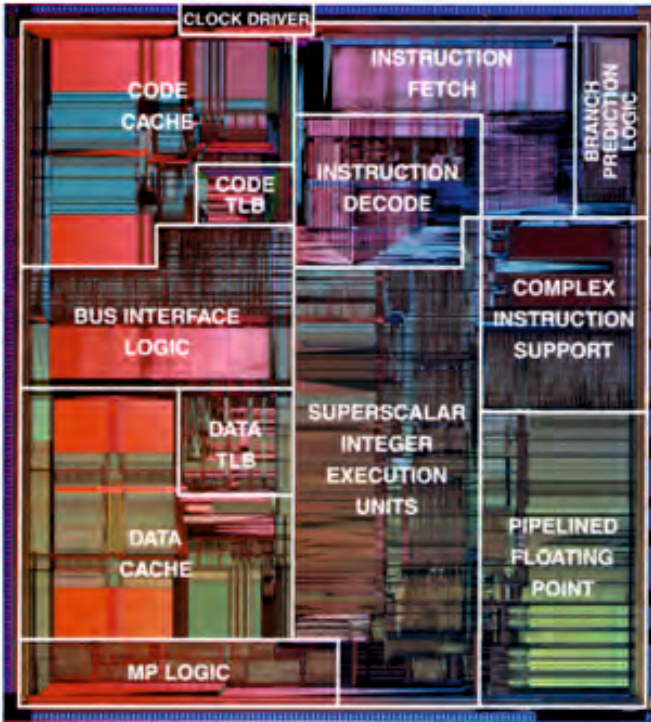


Figure 69: Schematic representation of a CPU

1998.

Intel introduces the Celeron.

The Celeron brand marks a new line of low-priced Intel microprocessors. With the 2006 introduction of the upper Core 2 brand, there idea was to discontinue use of the Pentium trademark, but Intel developed a line of mid-range dual-core microprocessors under the Pentium Dual-Core name at the request of laptop manufacturers. The Pentium brand thus lost its upper position and was repositioned between the Core 2 and Celeron Dual-Core lines as of 2007. In 2009, the Dual-Core suffix was dropped, and newer x86 microprocessors began to carrying the plain Pentium name again.

1999.

Intel introduces the Pentium III microprocessor.

2000.

Electronic reusable paper

“Electronic reusable paper” is a document display technology invented at Xerox PARC. It is thin, flexible, and portable, like paper, but can display different text and graphics when an electric charge is applied to it.

HISTORY OF THE COMPUTER

2000.

Digital rights management (DRM)

Xerox PARC's concepts for trusted systems and digital property rights lead to ContentGuard, a joint venture, which develops DRM software and offers content owners greater control and flexibility over the distribution of their material. To enable authorization of content access in a universal language, Xerox PARC develops eXtensible rights Markup Language (XrML).

2001.

Biomedical systems

Establishing a biomedical initiative, Xerox PARC partners with The Scripps Research Institute to develop instrumentation and information systems for accelerating discovery in the life sciences. In 2002, PARC demonstrates an operational prototype of the Fiber Array Scanning Technology Cytometer for screening blood samples about one thousand times faster than automated digital microscopy.

2002.

To broaden PARC's ability to innovate, build breakthrough technology platforms, and develop business concepts for many different organizations, PARC is established as an independent company.

2003.

The space shuttle Columbia explodes fifteen minutes before it is scheduled to land on February 1, 2003, resulting in the death of all seven crew members.

Apple opens the iTunes store April 28, 2003.

The Safari Internet browser is released June 30, 2003.

The Internet VoIP service Skype goes public August 29, 2003.

President George W. Bush signs CAN-SPAM into law December 16,

2003, establishing the first United States' standards for sending commercial e-mail.

2004.

Google jumps into the Social Networking with the release of Orkut in January 2004.

Mark Zuckerberg launches **Thefacebook** February 4, 2004, which later becomes **Facebook**.

Google announces **Gmail** on April 1, 2004. Many people take it as an April Fools joke.

Apple introduces Mac OS X 10.4 code named Tiger at the WWDC on June 28, 2004.

Microsoft Windows XP Media Center Edition 2005 is released on October 12, 2004.

Firefox 1.0 is first introduced on November 9, 2004.

Blizzard's **World of Warcraft** game, the most popular and successful MMORPG is released November 23, 2004.

IBM sells its computing division to Lenovo

Microsoft purchases the software developer GIANT Company Software,

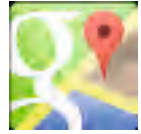
Inc. on December 16, 2004. The companies software would later become **Windows Defender.Group** for \$1.75 billion on December 08, 2004.



HISTORY OF THE COMPUTER

2005.

Google Maps is launched February 8, 2005.



YouTube is founded and comes online February 15, 2005.



Microsoft Windows XP Professional x64 Edition is released on April 24, 2005.

Star Wars Episode III: Revenge of the Sith is released May 19, 2005.

Apple announces it plans on switching its computer to the Intel processors June 6, 2005.

Microsoft announces its next operating system, codenamed "Longhorn" will be named **Windows Vista** on July 23, 2005.

IBM officially announces on July 14, 2005 that all sales of OS/2 will end on December 23, 2005 and that all support from IBM for OS/2 will end on December 16, 2005.

On September 12, 2005 **eBay acquired Skype** for approximately \$2.6 billion.

AMD and Intel both release their first versions of a dual-core processor.



2006.

The blu-ray is first announced and introduced at the 2006 CES on January 4, 2006.

Apple announces Boot Camp, which will allow users to run Windows XP on their computers April 5, 2006.

Intel releases the Core2 Duo Processor E6320 (4M Cache, 1.86 GHz, 1066 MHz FSB)

Toshiba releases the first **HD DVD** player in a computer computer with the introduction of the Toshiba Qosmio 35 on May 16, 2006.

Twtr, now known as **Twitter** is officially launched July 15, 2006.

U.S. President George W. Bush signs the USA Patriot Act into law October 26, 2006, giving law enforcement reduced restrictions on searching telephone, e-mail, and other forms of communication and records. Sony releases the PlayStation 3 November 11, 2006.

Nintendo releases the **Wii** November 19, 2006.

Google introduces Patent search December 13, 2006, which searches over 7 million patents.

2007.

Apple introduces the **iPhone** to the public the January 9, 2007
Macworld Conference & Expo.



Dropbox is founded.



2007 (Continuation 1)

Steve Jobs is inducted into the California **Hall of Fame** on December 5, 2007.



2008.

Acer officially acquires Packard Bell January 31, 2008.

Microsoft releases Windows Server 2008 February 27, 2008.

AOL ends support for the Netscape Internet browser March 1, 2008.

Intel announces the Intel Atom family of processors March 2, 2008.

Arthur C. Clark passes away March 19, 2008 (age 91)

Apple introduces Mac OS X 10.6 code named **Snow Leopard** and MobileMe

T-Mobile's G1 phone (**HTC Dream**) is the first phone to be released with **Google Android**



HTC Dream is the first phone to be released with Google Android

The first **Intel i7** is released to the public in November of 2008.

Google releases the first public version of **Chrome** December 11, 2008.

2009.

A person under the fake name of Satoshi Nakamoto introduces the Internet **currency Bitcoin**.



Microsoft Internet Explorer 8 is introduced March 19, 2009.
Apple removes support for AppleTalk in August 28, 2009 with its introduction of Mac OS X v10.6 that also is the first version of the Mac OS that no longer supports PowerPC processors.
Microsoft launches the **Bing** search engine June 3, 2009.



The analog TV signal begins to be phased out as broadcasts moved to high-definition
Google announces the **Google Chrome OS** July 7, 2009.
Microsoft releases **Windows 7** October 22, 2009.
Steve Jobs is named CEO of the decade by Fortune Magazine November 5, 2009.
Palm introduces WebOS.
USB 3.0 begins being released in November of 2009.
Google releases the **Chrome OS** as open source with the name Chromium OS

2010.

Oracle announces it has completed the acquisition of Sun Microsystems January 27, 2010.
Apple introduces the **iPad** on January 27, 2010.
Apple surpasses Microsoft as the most valuable technology company May 26, 2010.
Apple introduces the **iPhone 4** on June 24, 2010.
Microsoft announces plans to release **Windows Phone 7** October 11, 2010.
Microsoft first releases the **Kinect for the Xbox 360** in November 4, 2010.
Mark Zuckerberg is named TIME Person of the Year 2010.



2011.

On January 6, 2011 Aaron Swartz is arrested by federal authorities in connection with systematic downloading of academic journal articles from JSTOR.
Hitachi sells its hard drive division to Western Digital in March of 2011.
Microsoft releases Internet **Explorer 9** March 14, 2011.
The United States government even holds a hackathon in 2011 to improve city transit systems.
Microsoft announces plans on May 10, 2011 to acquire **Skype** for \$8.5 billion in cash.
The first **Chromebooks** with **Chrome OS** begin shipping on June 15, 2011.
Microsoft introduces **Office 365** June 28, 2011.
Steve Jobs resigns as Apple's CEO due to health reasons on August 24, 2011.
Steve Jobs passes away on October 5, 2011 (Age: 56)

2012.

Boing Boing, Computer Hope, Craigslist, Google, Reddit, Tumblr, Twitter, Wikipedia, and more than 115,000 other websites go dark in protest of the Stop Online Privacy Act (SOPA) on January 18, 2012.
Google and several other companies migrate to **IPv6** on June 6, 2012.

The space craft Curiosity lands on Mars August 5, 2012.

Apple **iPhone 5** goes on sale September 21, 2012.

YouTube breaks an Internet record as over 8 million concurrent live viewers watch Felix Baumgartner break his own record by jumping from the edge of space (128,100 feet) on October 15, 2012.

Apple introduces the **iPad mini** October 23, 2012.

Microsoft **Windows 8 and Microsoft Surface** is released October 26, 2012.

Microsoft unveils the **Xbox One** on May 21, 2013, a new gaming console to replace the Xbox 360.

2013.

The Furusawa group at the University of Tokyo succeeds in demonstrating complete quantum teleportation of photonic quantum bits on September 11, 2013, bringing quantum computer even closer to reality.

Akira Furusawa and Peter van Loock

WILEY-VCH

Quantum Teleportation and Entanglement

A Hybrid Approach to
Optical Quantum Information Processing



Apple introduces iOS7 on September 18, 2013.

Leap Motion is introduced.

Blaise pascal Magazine orders to create software for Leap Motion for Pascal



LEAP MOTION



INDEX of selected words

Symbols

+ , 49
- , 49
/ , 49
* , 49
** , 49
= , 50
< , 50
> , 50
<= , 50
>= , 50
<> , 50
<< , 51
>> , 51
:= , 107

File types

.csv, 113
.dbf, 113
.inc, 16
.lfm, 96,99
.lpi, 12
.lpk, 101
.lpr, 10
.lps, 12,99
.po file, 100
.rtf, 130
.txt, 195

Compiler directives

{\$apptype console}, 47,191
{\$ASSERTIONS ON}, 226
{\$DEFINE DEBUG}, 234
{\$H+}, 39
{\$IFDEF ...}, 15
{\$M+}, 81
{\$mode ...}, 11
{\$mode, 69,73
{\$R *.res}, 100
{\$TYPEINFO}, 81
{\$UNDEF debug}, 236
 Comment directive
{ToDo ...}, 16, 142

A

abstract (class), 87,173,176
Add unit to Uses Section dialog, 93
Add method, 172
Adding external tools to the IDE, 138
Additional Palette page, 119,150
address, 26,33,35
AfterPost method, 85
algorithm, 180,210,211,212
alias type, 20
aliases, 34
Align property, 115,194
Alignment property, 115,125
ambiguity, 43
anagrams, 204
analogue data, 3
ancestor, 87,88
Anchors property, 108,113
and, 51
anonymous types, 37
ANSI characters, 29
ansichar, 20,40
AnsiProperCase function, 40
ansistring, 18,25,34,38,39
apostrophe, 28
Append, 46
application, 6,82,100
Application.OnException, 59
Application.ProcessMessages, 210
Application Settings, 101
arguments, 63
arithmetic on pointers, 35
array constants, 31,37
array property, 75,76,79
array, 34,40,59,117
ASCII, 197
assembler, 3
assertion, 225
AssertNull, 216
Assign method, 172
Assigned function, 35,136
AssignFile, 46,47,48,59,82
assignment operator :=, 27,107
assignment, 27,28
asterisk, 99
automatic destruction, 92,103
AutoSize property, 106,109

B

- backup folder, 14
- backup, 14
- BeforeInsert, 85
- begin, 14,53
- beginning programmer, 7
- BevelInner, 164
- BevelOuter, 164
- binary arithmetic, 2
- binary files, 45
- binary, 2,28
- bit pattern, 3
- bit, 2
- BlockRead, 196
- Blowfish algorithm, 180
- books about Pascal and Lazarus, 248
- boolean expression, 21,50
- boolean type, 20,21,27
- BorderIcons, 109
- BorderStyle, 120,164
- brackets (parentheses), 23
- breakpoint, 246
- Brook framework, 8
- bug-avoidance, 222
- bug-hunting, 222
- bugs, 6,17,171,222
- business rules, 167
- byte, 3,20
- byteBool, 22

C

- C and C++, 34,37
- cache, 208
- Call Stack window, 246
- calling a routine, 62
- camel-casing, 17
- CanClose parameter, 158
- Canvas property, 89,92
- Capacity, 172
- Caption property, 114,125
- cardinal, 20,169
- caret, 33
- carriage return, 29
- case-sensitive operating systems, 17
- cast, 118
- Char, 20
- character array, 34
- CharCase, 125
- checkbox component, 115,119
- Chr function, 24
- cipher, 180
- circuit, 2
- circumflex symbol, 33
- class function, 72
- class method, 72,123
- class, 4,18,23,35,43,69,70,180
- ClassName property, 122

C

- Clear method, 172
- closed source, 6
- CloseFile, 46,47,48,82
- Code Completion, 91,103,105
- Code Explorer, 229
- Code Observer, 79,228
- COFF format, 245
- Color property, 110,115
- Columns property, 121,169
- combinations, 49
- command-line, 10
- CommaText property, 175
- comments in code, 14,16
- Common Controls Palette page, 121
- comparison, 27
- compilation, 4,5
- compiler checks, 25
- compiler directives, 16,17,234
- compiler messages, 223
- component browser tool, 130
- Component Palette, 10,111
- Components property, 111
- ComponentStyle property, 82
- computer languages, 3
- console program, 10,11,96
- console window, 9,10,47
- const, 19,27,28,31,44,64,212
- constant declaration, 28
- constructor, 71,79,92
- control characters, 29
- Controls, 91
- CopyFrom, 176
- core developer of Lazarus, 7
- Count property, 172
- CPU, 2,4,5,19
- crash, 19,192
- Create, 71
- CreateForm, 103
- cross-platform, 6,11
- CustomDrawn library, 86

D

- data access, 167
- Data Controls Palette page, 112
- data hiding, 69,85
- data structures, 34
- data, 3,19
- DataField, 112
- datamodule, 167
- DataSource, 112
- date dialog, 163
- DateToStr function, 24
- dBase, 113
- DbgAppendToFile, 239
- DbgOut, 237
- DbgS, 237
- DbgSName, 237
- DbgStr, 237

D

- debugging windows in IDE, 246
- debug, 17
- debugger (gdb), 60,245
- DebugLn, 236,237
- DebugLnEnter, 237
- DebugLnExit, 237
- debugserver, 240
- Dec, 21,35
- decimal, 28
- declaration, 20
- default array property, 172
- default parameter, 65,95
- DefaultExt, 193
- Delete, 172
- Delimiter, 175
- Delphi, 5,8,18,95
- dereference a pointer, 33
- descendent, 88
- Designer (Form Designer), 10,96,103,106
- Destroy, 71
- destructor, 71,79
- Dialogs Palette page, 193
- Dialogs unit, 105
- digital code, 3
- digital information, 2
- digitisation, 3
- DirectoryExists, 188
- display controls, 114
- DLLs, 34
- docking, 10
- documentation, 1,7,54
- dot notation, 41
- double quotes, 28
- double (type), 19,20
- DupeString, 42,66
- duplicate constants, 232
- DWARF format, 245
- dynamic array, 35,39,79,211

E

- EchoMode property, 124
- Editor, 10
- EDivByZero, 60
- EIntOverflow, 202
- ELF format, 245
- Enabled property, 124
- encapsulation, 85,147,227
- encoding conversion, 192
- encoding, 40
- end, 14,53,55
- endless loop, 200
- engineering notation, 28
- enumerated type, 21,22,23,44,57
- Eof, 46,48,58
- EOLn, 47
- error message, 27

E

- escape key, 29
- event handler, 84
- event properties, 75,84
- event-based paradigm, 11
- event-driven model, 82
- Events (OI page), 85
- events, 75,82,83
- example projects, 8
- except, 60
- exception, 60,188,191,203,204
- Exception.CreateFmt, 59
- Exclude, 119
- executable program file, 5
- executable size, 11
- Execute, 193
- Exit, 66,195
- expressions, 20,27
- extended (type), 19,20,202
- Extract Procedure dialog, 232

F

- factorial, 200
- Fail, 216
- Favorites (OI page), 109
- FCL, 5,19,62
- feedback, 206
- Fibonacci, 224
- fields, 43,70
- file access, 188
- file copying example, 177
- file extension, 12
- file, 41,46
- FileExists, 46
- FileMode, 46,196
- FileName, 193
- FileOpen, 188
- FilePos, 46
- FileSize, 46
- FillChar, 36,98
- FillRect, 92
- Filter, 193
- finalization, 97
- finally, 137
- Find in Files dialog, 212
- Firebird database, 113
- flag, 21
- floating point types, 19,126
- flow of control, 189
- fmOpenRead, 177
- fmOpenReadWrite, 177
- fmOpenWrite, 177
- focus rectangle, 124
- focus, 124
- FocusControl property, 114
- Font property, 108
- Font.Style property, 115

F

for loop, 213
form definition file (.lfm), 96
Form Designer, 10
Form Editor, 10
format specifiers (%d etc.), 48
Format, 48,106,203
FormatDateTime, 106
formatting, 5
forum for discussion, 211
FPC (Free Pascal Compiler), 4,5,6
FPCUnit, 215
fpGUI, 86
frame, 166
Free Pascal Compiler (FPC), 4,5,6
FreeBSD, 86
freeze, 19,205
fsBold, 108
function, 62,65

G

gdb debugger, 221,245
GetMem, 34
Git, 142
global variable, 26,80,103
goal, 141
grammar, 14
grouping several components, 106
GUI (graphical user interface), 9
GUI program, 10,11

H

hardware, 2,4,6
heap memory, 71,79
Heaptrc unit, 242
help, 7
hexadecimal, 28
hierarchy, 69
high level language, 4,5
High, 21,154
highlighter, 135
Hints, 223
hourglass cursor, 205

I

icon, 101
IDE (Integrated Development Environment), 1,5,12
Identifier Completion, 105
identifier, 20
image file, 100
implementation, 75,97
in operator, 44
Inc, 21,35

I

Include, 119
indentation, 15
indexed properties, 86
IndexOf, 175
infinite loop, 200
information theory, 2
information, 3
Inheritance (OI page), 112
inheritance, 69,85
inherited, 71,79,92
initialised variable, 31,37
initialization, 97
Insert, 172
installing Lazarus, 7
instance, 72,79
InstanceSize, 74
instructions in code, 2,3,19
Int, 24
int64, 20
integer, 20,126
Integrated Development Environment (IDE), 1,5,12
interface, 4,10,43,75,96,97,98
Interfaces unit, 86
internationalisation, 5,230
IntToStr, 24
IOResult, 46
ItemIndex, 90,93,118
Items property, 126

K

keyboard, 9
keyword, 4,20
KOL, 86
Kylix, 8

L

layout, 115
Lazarus forum, 7
Lazarus project, 9
LazLogger unit, 235,237
lazutf8 unit, 40
LCL, 5,19,23,62,101
Leak View tool, 245
Left property, 93,109
Length, 59
lib folder, 14
library, 19,101
limits of data, 218
LineEnding, 47,98,186
linker, 5
Linking, 242
Linux Terminal, 10
Linux, 6,18,86,110
literal text, 16
LoadFromStream, 176

L

- local declaration, 62
- local variables, 246
- logical operators, 21
- longBool, 22
- longint, 20
- longword, 20
- loop, 155
- low level access, 33
- low level instructions, 4
- Low, 21,154
- LowerCase, 40

M

- Mac OS X, 6,203
- Mac, 18,86,110
- machine code, 3
- Macpas, 69
- mailing list for Lazarus/FPC, 7
- main form file, 102
- main form, 82
- main program file, 10,99
- malware, 188
- math unit, 127
- MaxLength, 125
- memory footprint, 74
- memory leak, 72,137
- memory management, 71
- memory, 19,20,26,33,71
- Mercurial, 142
- message loop, 82
- MessageDlg, 203
- Messages (Window), 10,37
- messages, 83
- metadata, 101
- method pointer, 84
- methods, 62,69,73
- Misc Palette page, 126
- modal dialog, 65
- Modified, 99
- modular design, 62
- Modularising functionality, 227
- mouse, 9
- MS SQLServer, 113
- MultiLog, 239
- MultiSelect, 121
- MySQL, 113

N

- Name property, 82,122
- names in Pascal, 16,107,143
- natural language, 4
- NDA, 171
- New, 34
- newline, 29
- newsgroup, 211
- Niklaus Wirth, 4,11,227

N

- nil, 35,85
- not, 47,49,50
- notification, 82
- Now, 106
- null-terminated, zero-indexed character array, 34

O

- Object Inspector (OI), 10,107
- Object oriented file access, 188
- object oriented programming (OOP), 4,69,85,111
- Object Pascal, 4,18
- objects, 4
- objfpc, 18,69,73
- octal, 28
- Odd, 98
- OffsetRect, 64
- OI (Object Inspector), 10,107
- OnAcceptFilename, 178,194
- OnActivate, 129
- OnChange, 85,135,186
- OnClick, 82,84,118,155,178
- OnCloseQuery, 158
- OnCreate, 82,117,167,171
- OnDestroy, 167,171,186
- one-line comment, 16
- OnKeyUp, 85
- OnMouseDown, 85
- OnMouseMove, 122
- OnResize, 85
- open source development, 6,7,171
- operand, 49
- operating system, 4,6,9
- operators, 27,49
- optimisation, 207,212,220
- Options for Project dialog, 101
- Oracle, 113
- or, 51
- Ord, 21,24
- ordinal type, 20,21
- ordinality, 21
- OS calls, 234
- out parameter, 64
- overflow, 202,204
- overload(ing), 24,94,95
- override, 79,85,88,89,91,94
- Owner, 82,92,103,160

P

package, 101
Paint, 89,91,94
Palette (Component Palette), 10
Panels Editor, 121
parameter, 42,62,63,94
Parent property, 93,118,160
parentheses, 25
parse, 211
Pascal expression, 27,49
Pascal language, 1,4,5,19
Pascal types, 19
PasswordChar property, 124
patches, 6
patterns of bits, 3
PChar type, 25,33,34
PE format, 245
permutations, 204
persistent medium, 188
personality, 156
Pi, 63
PInteger, 33
platforms, 4,15
poedit, 230
pointer, 26,33,35
polymorphism, 69,79,85,86,94
popup hint window, 247
Position, 176,181
Postgres, 113
Pred, 21
predeclared types, 19
premature optimisation, 220
private, 62,80
procedural paradigm, 11
procedural type, 75,83
procedure, 62,65
ProcessMessages, 82,83,205
processor, 2,4
profiler, 220
program file, 96
program icon, 100,101
program interruption, 238
programming environment, 12
programming interface, 176
programming, 1,3
progress bar, 205
project directory structure, 143
project group, 12
Project Inspector, 26,100,101
project name, 10
Project options, 173,202
Properties (OI page), 108
properties, 23,43,75
property declaration, 75
protected, 80
pseudo-code, 211
public domain, 6
public, 80
published, 80

Q

QuestionDlg, 179,195
quote character, 28,29
qword, 202

R

RAD (Rapid Application Development), 141
radio button, 90
raise, 59
RAM, 26,188
Random function, 181
Randomize, 181
RandomRange, 181
range check, 241
Read, 46,47,176
readability, 15,16,54
ReadComponent, 176
ReadLn, 15,47,48
ReadOnly, 124,126
real types, 19
record constant, 41
recursion, 62,200,210
refactoring, 232
reference, 64
registers, 4
release, 6,17,242
required package, 101
RequireDerivedFormResource, 100
reserved word (keyword), 4,14,15
Reset, 46,47,48,59
resize message, 82
resource file, 96
resourcestring, 230,231
Restricted page, 110
Result (predefined variable), 63
return, 29
reusability, 111
ReverseString, 40,66
Rewrite, 46
root node, 132
Round, 24
routines, 62
RTL, 5,19,62
RTTI component, 80,112,125
RTTI, 80,172
Run Time Type Information (RTTI), 80,172
Run, 82
run-time error, 188,189,191

S

Save Project, 10
SaveToStream, 176
scope, 96,105
Seek, 46,176
SeekEOF, 47
SeekEOLn, 47
Self (predefined variable), 74
semicolon, 14,55,62
set intersection, 147
set property, 108

S

set type, 44
set union, 146
SetHeapTraceOutput, 244
SetLength, 39
sets, 44,144
Shape, 120
shl, 51
short term memory, 26
shortint, 20
shortstring, 34,38
Show Compile dialog, 13
Show method, 159
ShowAccelChar, 114
ShowMessage, 65,104
ShowModal, 159
shr, 51
signal, 2
simple type, 19,36
SimplePanel, 121
SimpleText, 121
single quote, 28
single type, 19,20
Size, 176
SizeOf, 72,181
sizing grip, 116
skeleton code (template), 12,14,31
sLineBreak constant, 30
smallint, 20
software, 2,6
sort code alphabetically, 229
Sort method, 175
Sorted property, 121,175
Sorting lines, 173
source code, 8
Source Editor, 10
splitter component, 166
SQLdb, 113
SQLite, 113
Sqrt, 63
square brackets, 44
square root, 63
ssAutoBoth, 194
STAB (debug format), 245
stack (memory), 31,37,58,71,188
standard events, 85
Standard Palette page, 114
state, 214
statements in Pascal, 20
static array, 36
static variable, 31,48,71
statusbar in Editor, 11,13
stdctrls unit, 116
strict, 80
StrictDelimiter, 175
string constants, 28
String Editor dialog, 90
string, 29,35

S

strong typing, 19,24
StrPas, 24
structured types, 19,36,41
strutils unit, 40,50,194
Style property, 44,108,120,126
subrange type, 20,23
Succ, 21
support, 7
SVN version control software, 142
system codepage, 40
system event, 82,85
system unit, 15,98
SysUtils, 40,50,106,180,188

T

tab, 29
TabOrder, 114,119
TabStop, 114,119
Tag property, 82
TAlignment, 118
TApplication, 82
TApplicationProperties, 189
task list, 142
TBevel, 120
TBorderStyle, 23
TButton, 108,110
TChart, 111
TCheckGroup, 117
TCollection, 113,168,169
TCollectionItem, 168,169
TComboBox, 126
TComponent, 81,157
TComponentClass, 131
TCompressionStream, 187
TControl, 157
TCustomXXX classes, 81
TDatamodule, 167
TDataSet, 112
TDataSource, 112
TDateTime, 20,106,180
TDBEdit, 129
TDBGrid, 112
TDividerBevel, 120
TEdit, 124,129
template, 12,17
test-driven development (TDD), 142,220
text files (*.txt), 45,47
Text property, 124
TextFile, 47
TFileNameEdit, 177,194
TFileStream, 176,177,188
TFloatSpinEdit, 126
TFont, 44
TFontStyle, 119
TFontStyles, 44
TForm, 23,157

T

TFormBorderStyle, 23
TFrame, 166
TGraphicControl, 89,91
TGroupBox, 163
theme, 6
TImage, 101
TIObjct, 125
tiOPF library, 239
title bar, 10
TLabel, 114
TLabeledEdit, 124,125,151
TLCLComponent, 157
TList, 113,168
TListBox, 121,194,211
TListView, 123
TMemo, 124,130,173,194,211
TMemoryStream, 180,188
TModalResult, 195
TNotifyEvent, 84,85
TObject, 69,71,118
ToDo file, 100
ToDo functionality, 16,142
Top property, 93,109
TOpenDialog, 193
TOpenPictureDialog, 193
TPanel, 23,152,163
TPersistent, 81,157,168,172
TRadioGroup, 90,117,118,173
translation, 5
TRect, 64
TResourceStream, 188
Trunc, 24
try...except...end, 192
try, 59,137
try...finally...end, 175
TSaveDialog, 193
TSavePictureDialog, 193
TSpeedButton, 153
TSpinEdit, 126
TSQLQuery, 113
TSQLTransaction, 113
TStaticText, 119,120
TStatusBar, 121
TStream, 113,168,176
TStringGrid, 111
TStringList, 113,137,168,172
TStrings, 113,121,173
TStringStream, 180,188
TSynEdit, 124,130
TSynMemo, 124,130
TTestCase, 215
TTextLayout, 119
TTIGrid, 170,172
TTIPropertyGrid, 125
TToggleBox, 125
TTrackBar, 101,180
TTreeNode, 132

T

TTreeView, 123,130
TUpDown, 180
Turbo Pascal notation, 29
tutorials, 7
TWinControl, 157,160
type conversion, 24,188
type declaration, 20
typecast, 25
typed constant, 31,37,48
typed pointer, 33,34
types, 19
typinfo unit, 137,149

U

Ubuntu, 6
UI (user interface), 5
unassigned, 35
underscore (underline character), 16,17
Unicode, 29,192
uninitialised-variable bug, 26,225
unit file, 96
unit tests, 223
unit, 14,16,96,97
Unix, 15,18,40
unknown identifier, 96
untyped files, 46
UpperCase, 40
user interface (UI), 5
uses clause, 14,96
utf8, 40,192
UTF8ToSys, 192

V

Val, 24
value parameter, 64
Value property, 126
var parameter, 63
var (variable declaration), 19,26,64
variable, 17,19,21,26
VCL, 234
version control system (VCS), 142
versions, 6
vertical divider, 121
View menu, 10
virtual method,
1,3,8,9,12,13,14,31,76,79,87,88,89
virus, 188

W

- Warnings, 27,37,38,223
- Watches, 246
- Watching variable values, 234
- web programming, 8
- while, 47,58
- Width property, 109
- window handle, 82
- Window menu, 10
- windowed control, 82,119
- Windows API, 22,34,234
- Windows OS, 6,15,18,40,86
- Windows, 86
- Wirth (Niklaus), 4,11,227
- with...do, 42,70
- word, 20
- wordBool, 22
- WordWrap, 119
- Write, 46,48,176
- WriteComponent, 176
- WriteLn, 15,47,48

X

- XML, 12
- xor, 51

Z

- zero-based array of Char, 38
- zero-based array, 37