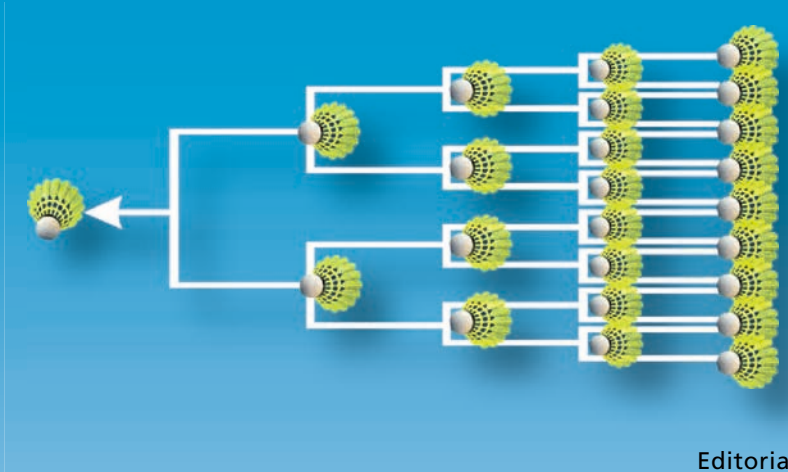


BLAISE PASCAL MAGAZINE 86

Object Pascal / Internet / JavaScript / WebAssembly / Pas2Js / Databases
CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

Blaise Pascal



Editorial

HOT OFF THE PRESS

FastMM5 changes towards commercial licensing. Be aware of the new rules

Installing Lazarus on Mac OS Catalina

Including a test for debugging By Detlef Overbeek

The new WebCore 1.4 is coming

By Bruno Fierens

MY ORM (Object-relational mapping)

By John Kuiper

Covid19 Apps Saving Lives

Using Delphi to Better the World, By Stephen Ball and Jim McKeeth

Schedule for a Badminton Knockout Tournament

Unexpected treasures in numbers, By Rik Smit

The search for a special number Factor 11

By David Dirkse

AI recognizes speech patterns coming from the brain

By Detlef Overbeek

SmartBinding with kbmMW #5

Compile Tool #1 – An easier way to compile projects

By Kim Madsen



BLAISE PASCAL MAGAZINE 86

Object Pascal / Internet / JavaScript / WebAssembly / Pas2Js / Databases
CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

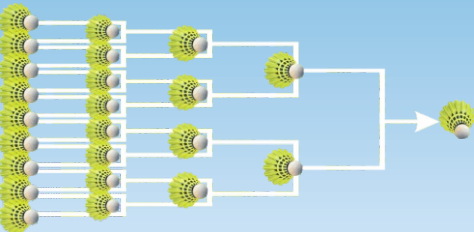


Blaise Pascal

CONTENT

ARTICLES

Editorial	Page 4
HOT OFF THE PRESS	Page 94
FastMM5 changes towards commercial licensing. Be aware of the new rules	
Installing Lazarus on Mac OS Catalina	Page 6
Including a test for debugging By Detlef Overbeek	
The new WebCore 1.4 is coming	Page 14
By Bruno Fierens	
MY ORM (Object-relational mapping)	Page 22
By John Kuiper	
Covid19 Apps Saving Lives	Page 43
Using Delphi to Better the World, By Stephen Ball and Jim McKeeth	
Schedule for a Badminton Knockout Tournament	Page 47
Unexpected treasures in numbers, By Rik Smit	
The search for a special number Factor 11	Page 88
By David Dirkse	
AI recognizes speech patterns coming from the brain	Page 91
By Detlef Overbeek	
SmartBinding with kbmMW #5	Page 92
Compile Tool #1 – An easier way to compile projects	Page 95
By Kim Madsen	



ADVERTISERS

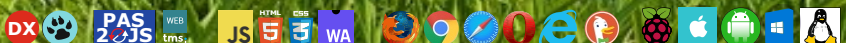
Lazarus Delphi Compatible	Page 13
TMS All Acces	Page 21
Library Blaise Pascal Magazine	Page 42
Lazarus Build once	Page 87
Barnsten	Page 90
Components4Dveleopers	Page 104

Publisher: PRO PASCAL FOUNDATION in collaboration © Stichting Ondersteuning Programmeertaal Pascal

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. A derivative known as Object Pascal designed for object-oriented programming was developed in 1985. The language name was chosen to honour the Mathematician, Inventor of the first calculator: Blaise Pascal (see top right).



Niklaus Wirth



Contributors

Stephen Ball http://delphiaball.co.uk @DelphiABall		Peter Bijlsma -Editor peter @ blaiseascal.eu
Dmitry Boyarintsev dmitry.living @ gmail.com	Michaël Van Canneyt, michael @ freepascal.org	Marco Cantù www.marcocantu.com marco.cantu @ gmail.com
David Dirkse www.davdata.nl E-mail: David @ davdata.nl	Benno Evers b.evers @ everscustomtechnology.nl	Bruno Fierens www.tmssoftware.com bruno.fierens @ tmssoftware.com
Holger Flick holger @ flixments.com		
Primož Gabrijelčič www.primoz @ gabrijelcic.org	Mattias Gärtner nc-gaertnma@netcologne.de	Peter Johnson http://delphidabbler.com delphidabbler @ gmail.com
Max Kleiner www.softwareschule.ch max @ kleiner.com	John Kuiper john_kuiper @ kpnmail.nl	Wagner R. Landgraf wagner @ tmssoftware.com
Vsevolod Leonov vsevolod.leonov@mail.ru		Andrea Magni www.andreamagni.eu andrea.magni @ gmail.com www.andreamagni.eu/wp
	Paul Nauta PLM Solution Architect CyberNautics paul.nauta @ cybernautics.nl	Kim Madsen www.component4developers
Boian Mitov mitov @ mitov.com		Jeremy North jeremy.north @ gmail.com
Detlef Overbeek - Editor in Chief www.blaiseascal.eu editor @ blaiseascal.eu	Howard Page Clark hdpc @ talktalk.net	Heiko Rempel info @ rompelsoft.de
Wim Van Ingen Schenau -Editor wisone @ xs4all.nl	Peter van der Sman sman @ prisman.nl	Rik Smit rik @ blaiseascal.eu
Bob Swart www.eBob42.com Bob @ eBob42.com	B.J. Rao contact @ intricad.com	Daniele Teti www.danieleteti.it d.teti @ bittime.it
Anton Vogelaar ajv @ vogelaar-electronics.com	Robert Welland support @ objectpascal.org	Siegfried Zuhr siegfried @ zuhr.nl

Editor - in - chief

Detlef D. Overbeek, Netherlands Tel.: Mobile: +31 (0)6 21.23.62.68

News and Press Releases email only to editor@blaiseascal.eu

Editors

Peter Bijlsma, W. (Wim) van Ingen Schenau, Rik Smit

Correctors

Howard Page-Clark, Peter Bijlsma

Trademarks All trademarks used are acknowledged as the property of their respective owners.

Caveat Whilst we endeavour to ensure that what is published in the magazine is correct, we cannot accept responsibility for any errors or omissions.

If you notice something which may be incorrect, please contact the Editor and we will publish a correction where relevant.

Subscriptions (2019 prices)

	Internat. excl. VAT	Internat. incl. 9% VAT	Shipment
--	------------------------	---------------------------	----------

Printed Issue ±60 pages	€ 250	€ 261,60	€ 85,00
Electronic Download Issue 60 pages	€ 60	€ 65,40	—
Printed Issue inside Holland (Netherlands) ±60 pages	—	€ 200,00	€ 60,00

Subscriptions can be taken out online at www.blaiseascal.eu or by written order, or by sending an email to office@blaiseascal.eu

Subscriptions can start at any date. All issues published in the calendar year of the subscription will be sent as well.

Subscriptions run 365 days. Subscriptions will not be prolonged without notice. Receipt of payment will be sent by email.

Subscriptions can be paid by sending the payment to:

ABN AMRO Bank Account no. 44 19 60 863 or by credit card or Paypal

Name: Pro Pascal Foundation-Foundation for Supporting the Pascal Programming Language (Stichting Ondersteuning Programmeertaal Pascal)

IBAN: NL82 ABNA 0441960863 BIC ABNANL2A VAT no.: 81 42 54 147 (Stichting Programmeertaal Pascal)

Subscription department

Edelstenenbaan 21 / 3402 XA IJsselstein, The Netherlands

Mobile: + 31 (0) 6 21.23.62.68 office@blaiseascal.eu



Member and donator of **WIKIPEDIA**

Copyright notice

All material published in Blaise Pascal is copyright © SOPP Stichting Ondersteuning Programmeertaal Pascal unless otherwise noted and may not be copied, distributed or republished without written permission. Authors agree that code associated with their articles will be made available to subscribers after publication by placing it on the website of the PGG for download, and that articles and code will be placed on distributable data storage media. Use of program listings by subscribers for research and study purposes is allowed, but not for commercial purposes. Commercial use of program listings and code is prohibited without the written permission of the author.



From your editor

Dear reader,

I tend to say let's not talk about Covid -19 for a while, but I hope as a community we'll get over that as soon as possible. So I wish everyone not only the best but also to stay clear of this pandemic.

Even now, besides causing accidents, this also offers new opportunities, and what is very interesting about using the computer is now being raised to an absolute peak. At least you should be able to make video calls, even if you are 90 or older.

That is excellent because it pokes up the old gray cells and that is life-extending.

In any case, this is a moment when - under pressure - the greatest possible creativity emerges. We would have liked to mention that the new version of Lazarus and Delphi were already ready for use. Lazarus 2.0.8 is out, but I'm waiting for the version that includes the latest Free Pascal version. Unfortunately, that has been postponed to September 2020, and Delphi version 10.4 is not there yet. In any case, the new season will bring a lot of news.

What I expect in the short term are two things: TMS Webcore 1.4, as soon as that appears, we will build new apps with it and I will write about it.

I want to once again announce a renewal for our website.

It has been a thorn in my side for years that the site is based on WordPress and therefore PHP.

That must be changed, I have taken steps to convert that into a whole according to Pascal - Pas2JS.

Where possible we will use TMS Webcore.

There are three basic issues that need to be solved that we did not have yet, the link to a server, the conversion of the database to one large structure and the link to our web shop.

The server issue has been resolved, the link to the database as well, now only the link of the shop needs to be done.

All code we develop for this will of course become open source.

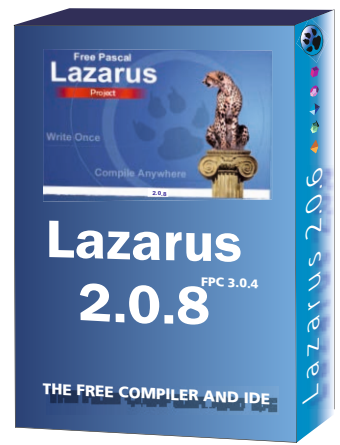
The major project of the Lazarus Handbook has almost been completed. I will present it as soon as possible. The last corrections and articles are now being added and then we can do it finally start printing.

And now happy readings and have fun in studying te projects!





The Lazarus Factory



WHAT MOST PEOPLE REALLY DO NOT KNOW ABOUT LAZARUS:

IT'S NATIVE TO



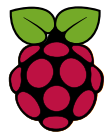
WINDOWS



LINUX



APPLE (MAC)



RASPBERRY

**BUILD ONCE
COMPILE ANYWHERE**

Because I needed to try to install Lazarus on the next version of the MacOS Catalina, I found that my Apple -machine was not any more capable of updating to this latest version Catalina.

It had nothing to do with the 64 bit version that may people think is the cause: It is simply a decision of Apple that the period ended and your machine is still working but not wit Catalina. So I turned my older Mac into a Win10 Dual-Boot machine. I found the installation of Lazarus a very simple try.

However it was good to be in touch with Mattias Gärtner and Martin Friebe. This because we wanted to make sure we would use the best debugger for our purposes. This is to be found with a small project at the end of this article.

Mattias checked the installation, Martin showed how to fine-tune the debugger.



1

Installation of Lazarus on a Mac varies according to your Mac version, but you must be running an Intel version of MacOS. Apple has been making each release of MacOS stricter in what it allows and – more importantly for installation of executables – more comprehensive in what it forbids.

To install Lazarus on any version of MacOS requires that you have already installed a recent version of XCode on your system (XCode is Apple's own MacOS developer environment). XCode contains various tools which Lazarus and Free Pascal need to create MacOS programs. You must install XCode through the MacOS App Store.

if you don't have it. XCode is a large program of several gigabytes, so the download and installation is a lengthy and disk-consuming process.

After you have installed XCode, you must install further command-line tools to accompany XCode. Type the following commands in a Terminal window to install these extra tools:

```
sudo xcode-select --install
sudo xcodebuild -license accept
```

This downloads and installs further tools that are needed to run Lazarus on OS X.

When these prerequisites are completed, you can install Free Pascal and Lazarus. The necessary installers can be found on the Lazarus download page. Just as for Linux, you need to install three packages.

For newer versions of MacOS (Mojave and later), the installation packages are located in the Lazarus macOS x86-64 folder in the linked website's download section:

```
fpc-3.0.4-macos-x86-64.pkg
```

the Free Pascal compiler

```
fpc-src-3.0.4-laz.pkg
```

the sources for Free Pascal

```
lazarusIDE-2.0.6-macosx-x86_64.pkg
```

the Lazarus IDE and sources

The installer packages are not signed, so you may get an error if you try to open the package directly from within the browser:



Figure 1: Package can't be opened

You need to let the browser save the package, and open the package from within the Mac OS Finder (it should be in the Downloads folder). Additionally, do not just double-click to open the package. You must use the context menu from within Finder, and explicitly select the Installer.app to open the package:

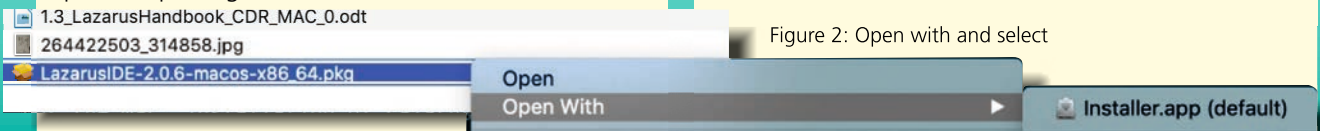


Figure 2: Open with and select

Even after doing so, you will get a warning that the package is from an unidentified developer, asking you to confirm that you really want to open it:

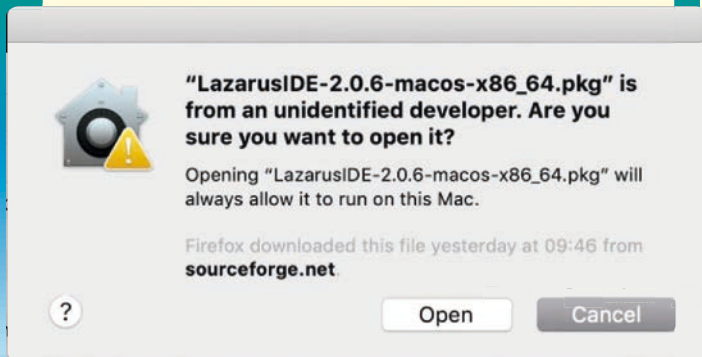


Figure 3: Open the package

You can safely ignore this warning and install the package. The packages are normal Mac OS installers, which will ask you several questions before actually installing Lazarus. It is best to accept all default locations to ensure that Lazarus will find all tools it needs. Once you have accepted all the default options, you should install Lazarus in `/Developer/lazarus`

A shortcut is created for you in the Applications location, so the application can be started from Launchpad. That's all for installing, now let's test it with a first project.





Figure 4: The background image for Catalina

DEBUGGING

After installing we can check if the right debugger has been installed and is working. This because the GDB does not work anymore on the new platform.

To find out we create a simple project with a form and a button just to test the new install.

```
TForm1.Button1Click(Sender: TObject);
begin
  Button1.caption:= 'Hello Catalina';
end;
```

Run the project.

During first time compilation of this project we be will presented with a special kind of message: **EnableDwarf 2 (-gw)?**

The project does not write debug info in Dwarf format. The "LLDB-debugger (with fpdebug) (Beta)" supports only Dwarf.

It also shows three buttons + cancel:

- ❶ Enable Dwarf 2 with sets
- ❷ Enable Dwarf 2 (-gw)
- ❸ Enable Dwarf 3(-gw3)



Figure 5: Welcome message

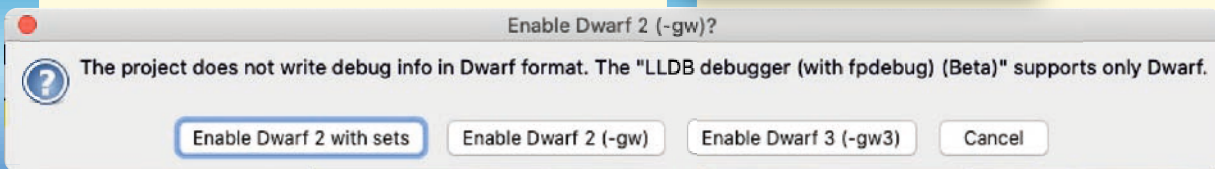


Figure 6: The message start up like this

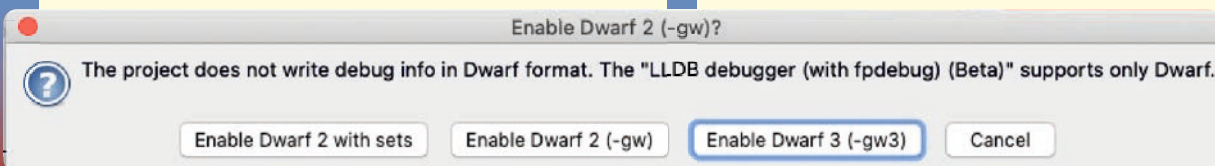


Figure 7: Choose button 3, enable dwarf 3

Info Dwarf (GDB and LLDB)

LLDB is a high-performance debugger. It is built as a set of reusable components which highly leverage existing libraries in the larger LLVM Project, such as the Clang expression parser and LLVM disassembler.

https://wiki.lazarus.freepascal.org/GDB_Debugger_Tips

Dwarf 2 with sets (-gw -godwarfsets)

This setting adds the ability to inspect sets:

"type TFoo=set of (a,b,c);". This is borrowed from the Dwarf 3 specs, but supported by most versions of GDB (any GDB from version 7 upwards should do).

Dwarf 2 (-gw)

This sets the format to Dwarf2.

This is the most basic dwarf setting.

Dwarf 3 (-gw3)

Dwarf 3 can encode additional info for some types (such as strings and arrays). It also preserves the case of identifiers in the debug info.

However there are still issues with the produced debug info. Some info may be incorrectly encoded, and other is not understood by GDB. In some cases this can lead to GDB crashing. This setting can be used, when using the FpDebug based debugger (add on package for the IDE).

According to Martin Friebe, who is the best informed about this project DWARF 3 is the best version to choose.



The choice you make is definitive for this project. As soon you will create another Project you will be confronted with that choice again. Here is how to make your selection persistent: first of all let's see if the correct debugger was installed:

choose: **Tools** → **Options** → **Debugger** as you can see in figure 7.

In the section Debugger type and path should be set: **LLDB debugger (with fpdebug) (Beta)**.

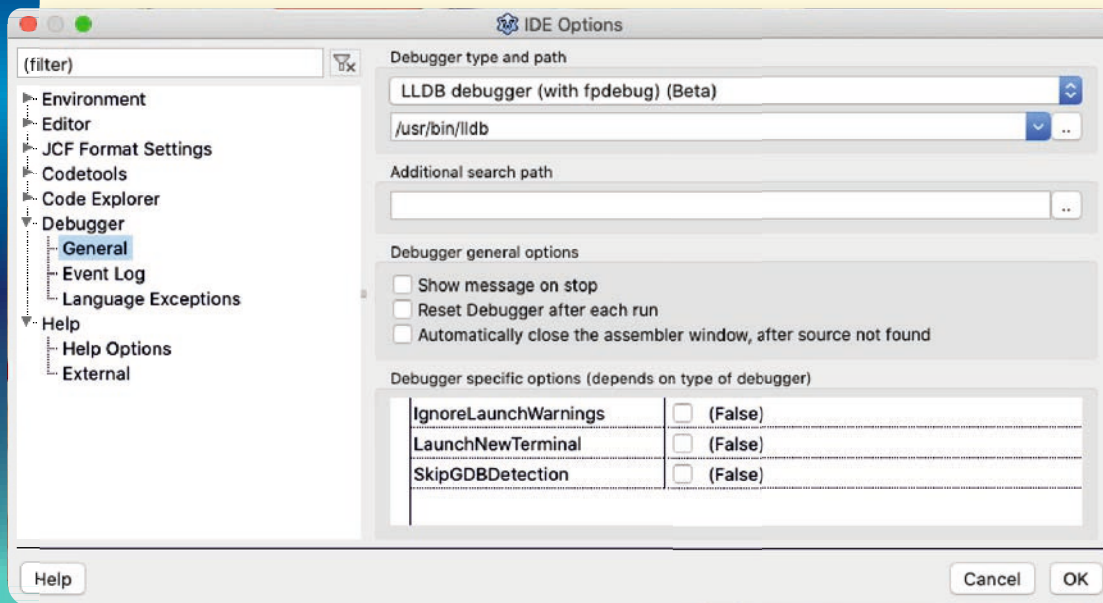


Figure 8 Checking the settings of the debugger.

In figure 8 (choose: **Project Options** from the **Menu Project**) in the left part choose: **Compiler Options** → **Debugging**.

The drop down menu is set by Default this is set to: Automatic (-g). Here you can make your choice for any type you want. Once you have chosen, you can set compiler options as default (in the left column just above Help), check the box.

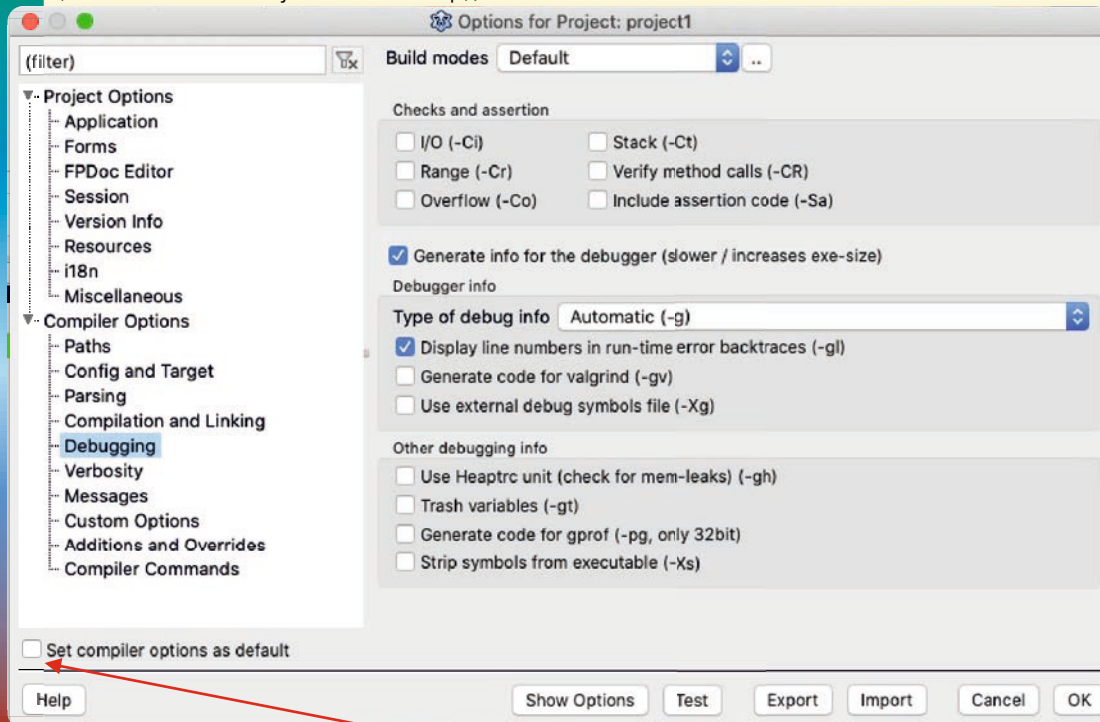


Figure 9 Here you can make settings for the project persistent, as well as default for all projects.



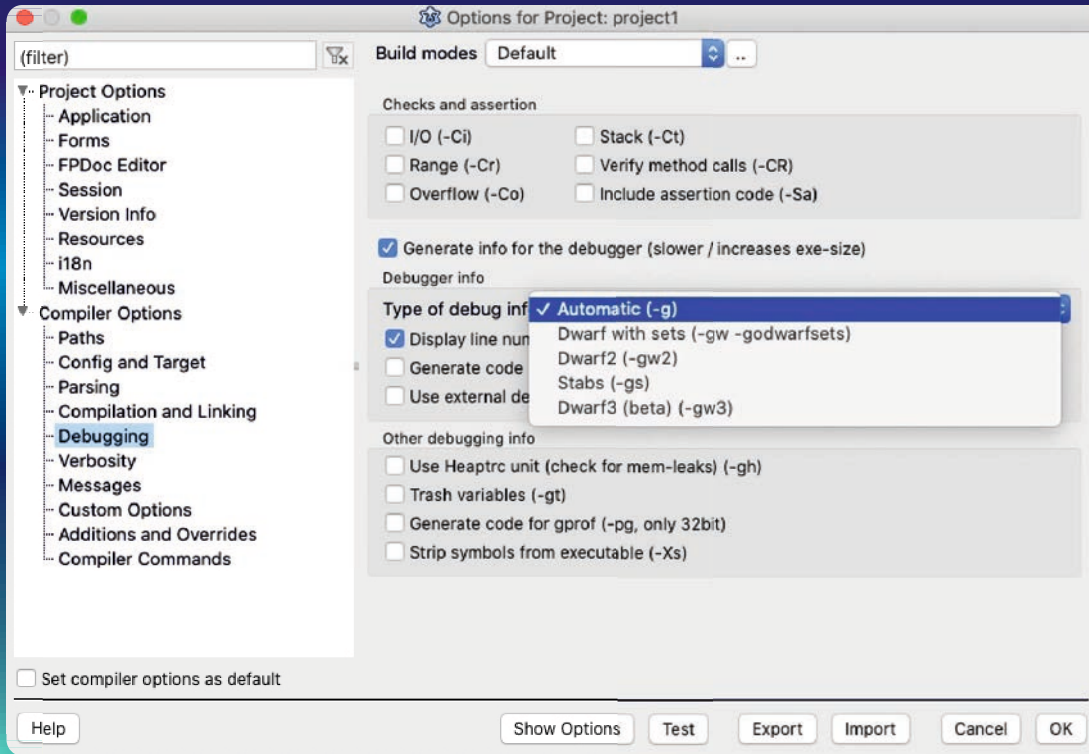


Figure 10: The drop-down shows the available options for the kind of debugger you would like to use

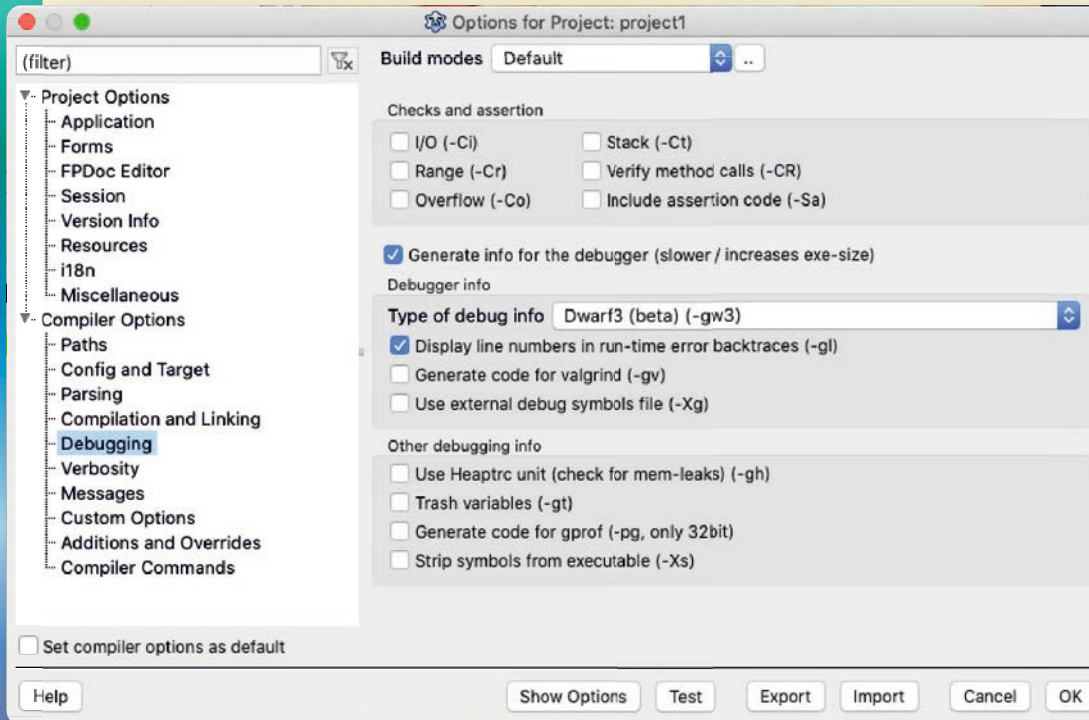


Figure 11: The choice we suggest you make...



DEBUGGING, PROGRAM

To explain this subject Martin Friebe has created a small program which shows an example of debugging.

The idea is to create an error and then test it being noticed. A simple way would be to find the next prime number of an array of prime numbers. So here is a simple example wherein you can find this.

I have illustrated the article with various pictures that show the build. The coding is shown completely in an overview and you can of course download the project.

There are two functions in the project: `FindIndexOfNextPrime` and `xFindIndexOfNextPrime`.

The first is the simple solution to create the error and the second – which you can simply replace the first by renaming it without the 'X' of the second – is a little more advanced: it is the so called Binary Search that has been used here.

The Binary Search is separately briefly explained, because it's a very simple but interesting manner to find items in a large array. There is of course a short remembrance of what a prime number really is. The result is of course the same.

PRIME NUMBERS

A prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.

A natural number greater than 1 that is not prime is called a composite number. For example, 5 is prime because the only ways of writing it as a product, 1×5 or 5×1 , involve 5 itself.

However, 6 is composite because it is the product of two numbers (2×3) that are both smaller than 6.

Primes are central in number theory because of the fundamental theorem of arithmetic: every natural number greater than 1 is either a prime itself or can be factorized as a product of primes that is unique up to their order.

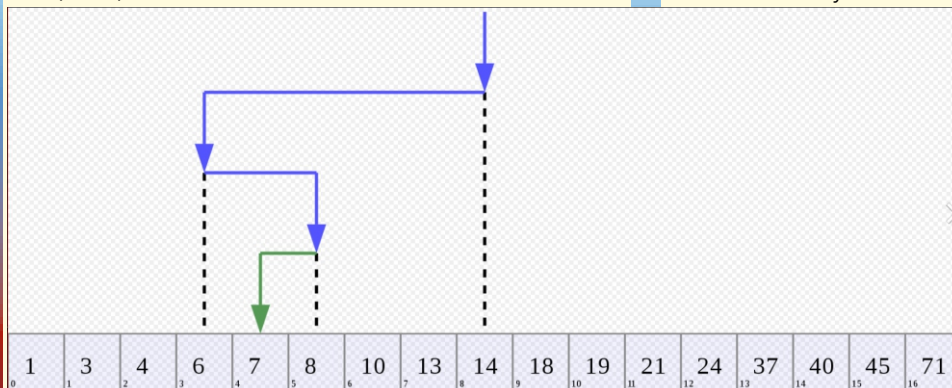


Figure 2; This is an example of a binary search steps through time.

By AlwaysAngry - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=53687795>

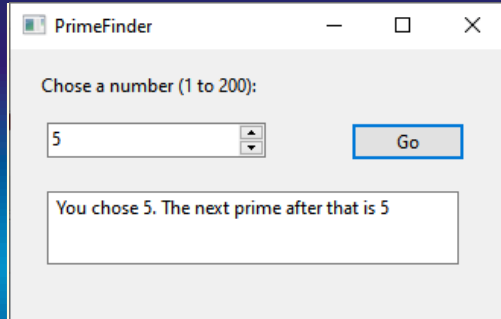


Figure 1; The program to be tested



BINARY SEARCH

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array.

Binary search compares the target value to the middle element of the array.

If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search runs in logarithmic time in the worst case, making comparisons, where n is the number of elements in the array, the O is Big O notation, Binary search is faster than linear search except for small arrays.

Important is that the array must be sorted first to be able to apply binary search. There are specialized data structures designed for fast searching, such as hash tables, that can be searched more efficiently than binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.



```

unit Unit1;
{$mode objfpc}{$H+}
interface
uses Classes, SysUtils, Forms, Controls, Graphics, Dialogs, Spin, StdCtrls;

type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Memo1: TMemo;
    SpinEdit1: TSpinEdit;
  procedure Button1Click(Sender: TObject);
  private
  public
  end;

var Form1: TForm1;

implementation

{$R *.lfm}

const
  Primes: array[1..47] of Integer = ( 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
    61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
    167, 173, 179, 181, 191, 193, 197, 199, 211 );

function FindIndexofNextPrime(APrimeToSearch: Integer): Integer;
var HighIdx: Integer;
begin
  HighIdx := High(Primes);
  Result := Low(Primes);
  while Result < HighIdx do begin
    if Primes[Result] >= APrimeToSearch then //// bug, this must be >
      break;
    Result := Result + 1;
  end;
end;

function xFindIndexofNextPrime(APrimeToSearch: Integer): Integer; //// xfunction to replace the first function above
var LowIdx, HighIdx: Integer;
begin
  LowIdx := Low(Primes);
  HighIdx := High(Primes);
  Result := (LowIdx + HighIdx) div 2;
  while LowIdx < HighIdx do begin
    if Primes[Result] < APrimeToSearch then //// bug, this must be <=
      LowIdx := Result + 1
    else
      HighIdx := Result;
    Result := (LowIdx + HighIdx) div 2;
  end;
end;

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
var UserInput, IndexofNextPrime: Integer;
begin
  UserInput := SpinEdit1.Value;
  IndexofNextPrime := FindIndexofNextPrime(UserInput);
  Memo1.Text := Format('You chose %. The next prime after that is %d', [UserInput, Primes[IndexofNextPrime]]);
end;
end.

```



The Lazarus Factory



LAZARUS IS A DELPHI COMPATIBLE CROSS-PLATFORM IDE FOR FREE PASCAL.

It includes LCL which is more or less compatible with Delphi's VCL. Free Pascal is a GPL'ed compiler that runs on Linux, Win32, OS/2, 68K RaspberryPie and more. Free Pascal is designed to be able to understand and compile Delphi syntax, which is OOP. Lazarus is the part of the missing puzzle that will allow you to develop Delphi like programs in all of the above platforms.

WHAT WIDGET SET?

You decide. Lazarus is being developed to be totally and completely API independent. Once you write your code you just link it against the API widget set of your choice. If you want to use GTK+, great! If you want it to be Gnome compliant, great! As long as the interface code for the widget set you want to use is available you can link to it. If it isn't available, well you can write it.

CAN YOU USE YOUR EXISTING DELPHI CODE?

IN GENERAL: YES. If you are using some very specific databases, OCX, or DCU then the answer would be no. THESE ITEMS ARE SPECIFIC TO WINDOWS AND WOULD ONLY WORK ON AND WITHIN WINDOWS.

CAN I CREATE COMMERCIAL PRODUCTS WITH THIS?

YES. The code for the Free Pascal compiler is licensed under the GPL.



View on Ravenna



The world of web development is evolving fast, it comes as no surprise that TMS WEB Core evolves fast.

When we embarked on this exciting adventure in 2017, we knew the road would be long. There is simply an abundance of things to do in the world of web development and we made it our mission to put Delphi developers in a front seat to apply the well known RAD approach to create web applications with a, unparalleled productivity. When we first released TMS WEB Core v1.0 on July 26, 2018, we named version 1.0 Brescia after the city where the famous car race Mille Miglia starts. And with each subsequent version, we name it after a city across the legendary Mille Miglia track of 1955. We visited meanwhile Verona with v1.1, Padua with v1.2 and Ferrara for v1.3. So, now we are heading to version v1.4 that will be named Ravenna.

The theme for TMS WEB Core v1.4 Ravenna is:

- 1 widening the UI control offerings with controls for frequently used UI patterns
- 2 enhancing the HTML-first approach
- 3 increasing easy interfacing to additional popular back-end services

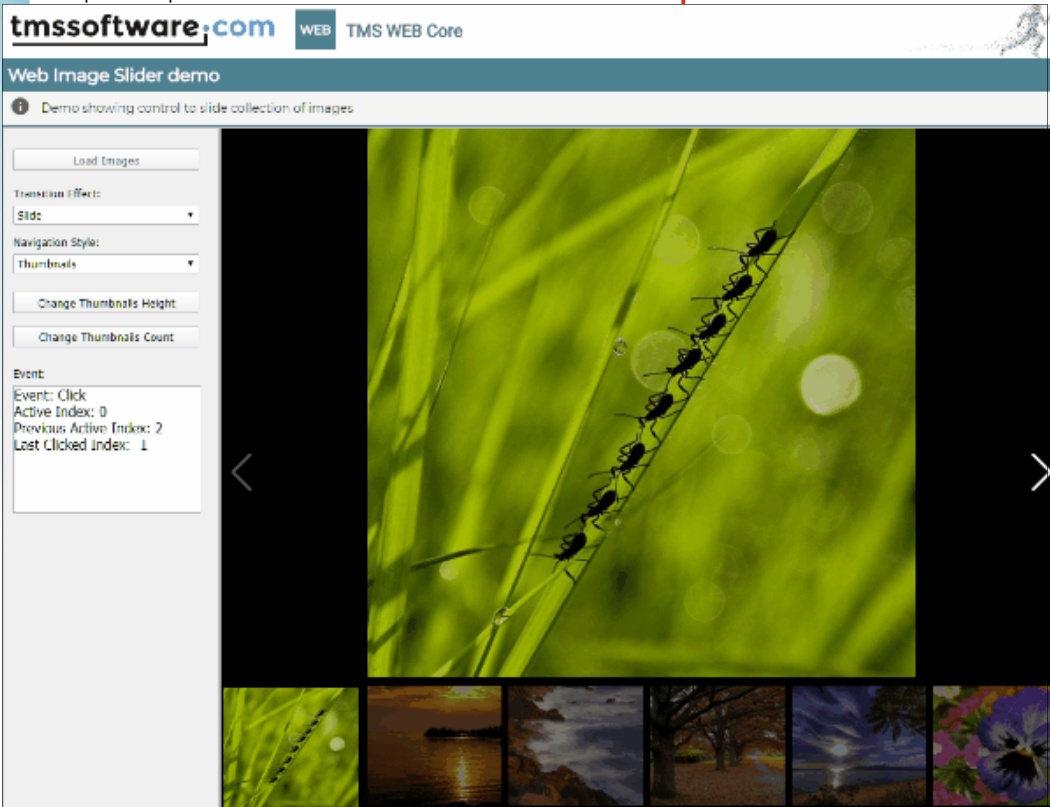


Widening the UI control offerings
 We have added two brand new UI controls in TMS WEB Core v1.4 Ravenna:

TWebImageSlider

In many scenarios, people want to show various pictures of things for specific items. Think about a product on **Amazon** that might have different pictures taken from different angles, think about an online real-estate broker presenting different houses with picture sets of the house on sale or a car dealer showing cars for sale accompanied by pictures of the car in various positions.

If you have such a use-case in your application, **TWebImageSlider** is the shortcut to achieve this. Basically this is a container control where you add the links to the images to be displayed and the control does everything else. It shows the **picture thumbnails**, a **left / right slider button** and you can click on thumbnails to **see the large version** of a specific picture.



```
var
i: Integer;
begin
for i := 1 to 8 do
ImageSlider.ImageURLs.add(Format('./images/nature-%d.jpg', [i]));
ImageSlider.RefreshImages;
ImageSlider.Appearance.TransitionEffect := tefSlide;
end;
```

Now, to integrate this kind of functionality should not take much more than a couple of minutes.



TWebContinuousScroll

Another often used pattern is to show lists of items filling the viewing area of the browser only and only load additional items when the user decides to scroll down. The reasoning behind such UI control is simple. By loading only the items in view, the initial display of the page is very fast and only when the user wants to see additional items, extra items are loaded asynchronously in the list.

The screenshot shows a web application interface with a header for 'tmssoftware.com WEB TMS WEB Core'. Below the header is a section titled 'Continuous scroll list demo' with a sub-note: 'This demo shows an infinite scrolling list.' The main content area displays a list of six items, each representing a message from a different user:

- Brady Bolton** (ID: 1): Done. Message: Meeting for my team will start 10 minutes later. Star: 1. Date: 4/24/2020 09:19:54.
- Jaydin Richmond** (ID: 2): Done. Message: Release fix for #4138. Star: 7. Date: 4/24/2020 08:38:54.
- Samara Contreras** (ID: 3): Urgent. Mark as done. Message: Handle bug report #4596. Star: 5. Date: 4/24/2020 07:43:00.
- Riley Tate** (ID: 4): Medium priority. Message: Release fix for #4157. Star: 3. Date: 4/24/2020 07:18:56.
- Samara Lutz** (ID: 5): Medium priority. Message: Handle bug report #4865. Star: 6. Date: 4/24/2020 06:59:37.
- Krystal Hendricks** (ID: 6): Low priority. Message: Meeting for my team will start 15 minutes later. Star: 0. Date: 4/24/2020 06:22:27.

TWebContinuousScroll is again a shortcut to this pattern. Drop the control on the form, add the event handler code for the event that is triggered when new items are needed and return the requested items. **TWebContinuousScroll** does the rest, it handles the rendering, it handles the UI interaction (mouse dragging / touch scrolling) and just triggers the event when new items are needed.

```
procedure TForm1.WebContinuousScroll1FetchNextPage(Sender: TObject; APageSize, APageNumber: Integer;
var AURL: string);
begin
  AURL := 'https://tmswebcore.com:8082/?page=' + IntToStr(APageNumber) + '&per_page=' + IntToStr(APageSize);
end;
```


New TWebListControl demo

The **TWebListControl** is a very versatile list control that might not be well understood enough and therefore underused by TMS WEB Core developers. **TWebListControl** uses the **Bootstrap CSS** library to do its magic. From a list of items, it can create a breadcrumb, a tab list, an item list, a list with expanding/collapsing subitems. The new demo shows the various modes of the versatile **TWebListControl**

List as breadcrumb control

Main / Products / Software / Web development / TMS WEB Core

List as pagination control

Customers Products Orders Stock

Customers
Products
Orders
Stock

List with subitems

Mercedes
Audi
Porsche
911 Turbo 4
Macan 3
Panamera 5
BMW

Electron 8 support

The fast evolving framework for creating cross platform desktop applications reached meanwhile version 8. It is being polished & enhanced all the time to allow to create responsive installable & near-native experience desktop applications for **Windows, macOS** and **Linux** with the advantage that the UI is rendered from **HTML/CSS**, meaning that in terms of graphical appeal, there are no limits.

With TMS WEB Core v1.4, we did the necessary changes to the framework and the Electron specific controls to make these work as seamless as possible with Electron 8

Enhancing the HTML-first approach

We've realized that not for all users looking at TMS WEB Core it was clear that using the **Delphi IDE** form designer for creating your web pages is by far not the only way to do it. While TMS WEB Core was developed from the ground up to facilitate this for Delphi developers familiar approach to create application forms, it was equally from the ground up built to enable the use of **HTML/CSS** based pages. This means that you can use existing **HTML/CSS** page templates which are not only created by web designers but can be obtained free or very cheap from various websites.

Sometimes you get for \$25 an extraordinary good looking web page template. Of course, we wanted to offer the capability to use such templates and from the **Delphi IDE**, you will basically just write the UI control logic and leave the page layout to **HTML** and **CSS**.



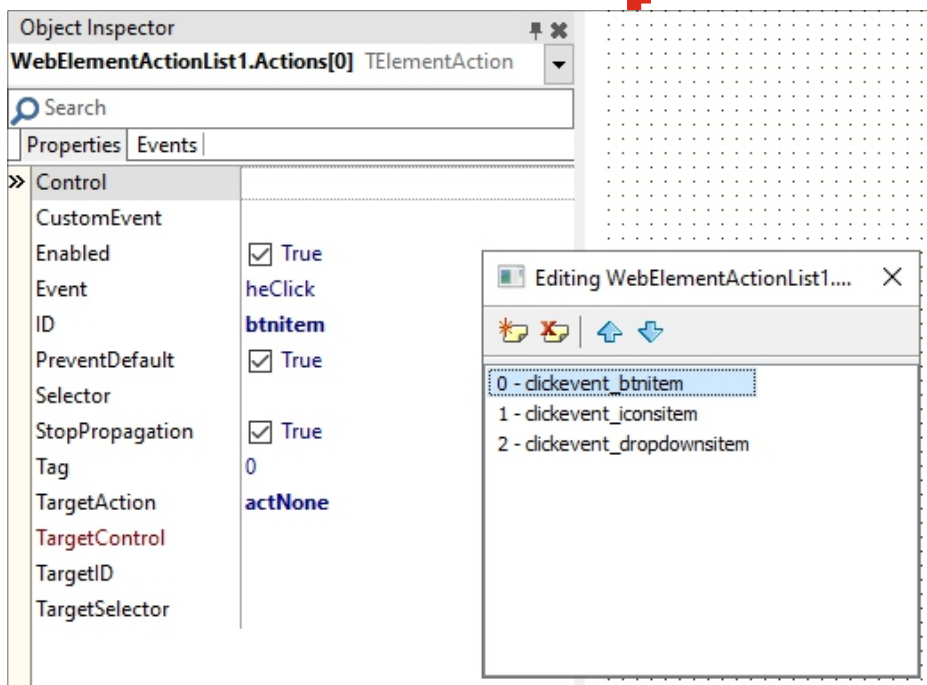
New TWebElementActionList

This new component, not to be confused with Delphi actions, facilitates easy hooking to events for all the HTML elements in page templates. It is a collection of actions that you define that happen when an event happens for a HTML element on the page. For example, the menu of your application could be a graphically very good looking **HTML/CSS** based animated menu and you can use the **TWebElementActionList** to define the actions that should happen when a specific item in this **HTML/CSS** menu is clicked. To do this, simply add the template **HTML/CSS** to your form, make sure to set a unique ID to each HTML element representing menu items and then add a **TElementAction** for each item in the menu. Define for the **TElementAction.Event** for example `heClick` and then the **TElementAction.OnExecute** event will be triggered when this menu item is clicked. In this **OnExecute** you could then for example add the UI control logic to show a DB grid with data, show a different form etc... As a **Delphi** developer, you have reused the graphical skills of a web designer and you just had to do a minimal effort to connect the logic in your application that is happening when the user interacts with the user-interface.

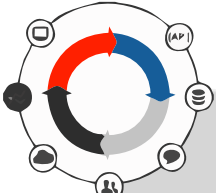
Increasing easy interfacing to additional popular back-end services

In TMS WEB Core we have already included the **TWebClientConnector**, **TWebClientDataSet**, **TWebDataSource** to bring the pattern **VCL Delphi** developers have known ever since the inception of Delphi to bind data to UI controls. This concept also exists in **TMS WEB Core**. To make the binding to the back-end easy, we have a **TWebXDataDataSet** that shields all the complexities of communicating with a TMS XData REST server. We have the **TWebmyCloudDataClientDataSet** to shield this same complexity when our **myCloudData** cloud data storage service is used (free for all TMS ALL-ACCESS users). We also have the **TWebSQLRestClientDataSet** that interfaces to the Lazarus foundation open source **SQLDBBridge** REST server. And we have **TWebFirestoreClientDataSet** for users wanting to use Google's cloud data storage solutions.

With TMS WEB Core 1.4 Ravenna, we are pleased to offer 3 more easy out of the box solutions to connect to back-end services.



New TWebRadServerClientDataset



Delphi Enterprise or **Delphi Architect SKU** users have out of the box a license to **Embarcadero Rad Server**. **Embarcadero Rad Server** offers the technology to create **REST** services and is able to create a **REST API** for performing operations on databases in the back-end. While TMS WEB Core includes a component to perform **REST** requests to work with **Embarcadero Rad Server** out of the box, the new

TWebRadServerClientDataset just makes it way easier to hook-up a UI with DB-aware controls to an Embarcadero Rad Server and perform through this dataset **CRUD** operations. Basically you set the **URL** to the data exposed as **JSON** based **REST API** from Embarcadero Rad Server and the **TWebRadServerClientDataset** middleware will perform all required **HTTP GET, PUT, POST, DELETE** requests, **JSON** handling behind the scenes and from the TMS WEB Core client you have just the DB-aware UI controls hooked up to it via a **TWebDataSource**.

We have added our todo-list demo that is using Embarcadero Rad Server just like we have this same todo-list demo. Other than the dataset, there is not much different from the demo using **Firestore, TMS XData, SQLDBBridge, myCloudData**. This shows how back-end agnostic TMS WEB Core web client applications can be.

New TWebDreamFactoryClientDataSet

From all low code back-end technologies, **Dream Factory** is without a doubt the most flexible one.

With **Dream Factory** (www.dreamfactory.com) you can create **REST APIs** for access to data on the back-end by doing all the settings and parameterizing via a web interface.

No need to do any programming, no need to dive into all technical details of **HTTP(s)** request, authentication, **JSON** packets, ... **Dream Factory** does this all for you.

We had **Dream Factory** as a very interesting back-end for TMS WEB Core already on the radar even before the inception of TMS WEB Core in 2018 as it is a very interesting technology for offering cloud data access for **VCL Windows** applications or **FMX** cross-platform applications, possibly further facilitated via a TMS Cloud Pack component.

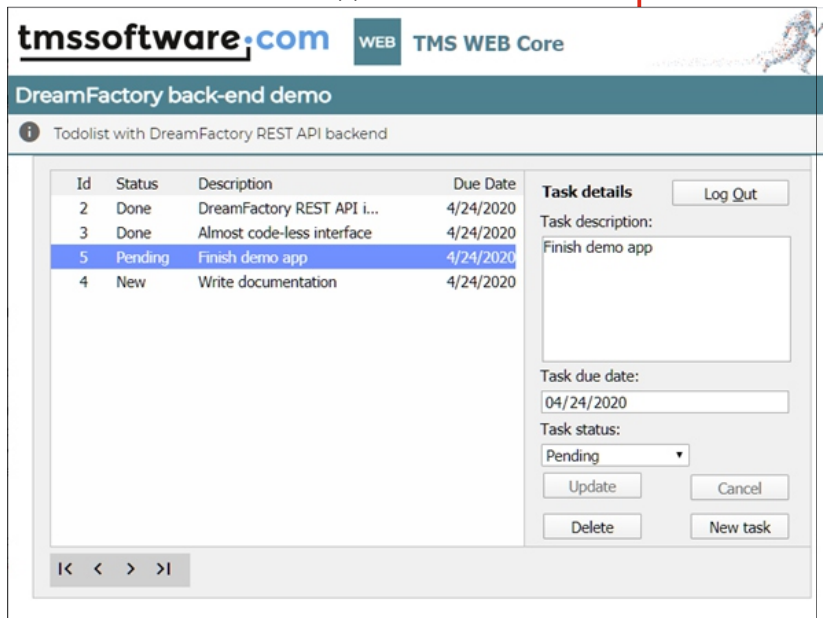
But now we embark with our first bridge component, the

TWebDreamFactoryClientDataSet that you can configure with the URL of your **Dream Factory REST API** and this bridge component does all the required communication to perform **CRUD** operations via its dataset to a database with a **Dream Factory REST API**.

This dataset is then easily hooked up via a **TWebDataSource** to the TMS WEB Core DB-aware UI controls.

Also here we have taken the same todo-list demo and with minimal effort (basically replacing the dataset) our todo-list application talks to a Dream Factory based back-end.

While **TWebDreamFactoryClientDataSet** is a first step for easy **Dream Factory REST API** back-end access from TMS WEB Core web client applications, we're eager to offer even more integration with the **Dream Factory APIs** in future TMS WEB Core versions as well as in future versions of the TMS FNC Cloud Pack that can be used in native Windows applications as well as native cross platform **Delphi FireMonkey** applications.



NEW TWEBFAUNADBCLIENTDATASET

Where **Dream Factory** offers automatic codeless REST API creation for access to a multitude of databases or services, **FaunaDB** (www.faunadb.com) is a cloud data storage service that hosts the data for you and offers as **REST API** to access it. It has similarities with our own **myCloudData.net** service and a few interesting angles. As such, to offer yet more freedom of choice, we have added the **TWebFaunaDBClientDataSet** component.

You can use the web interface on your account at **FaunaDB** to design your tables and this can automatically be consumed when setting the proper URL to the **TWebFaunaDBClientDataSet** component. There is not much more to it, go to **FaunaDB.com**, setup your tables, set the URL to **TWebFaunaDBClientDataSet** and hookup DB-aware TMS WEB Core controls to this dataset via a datasource and you are up & running to perform **CRUD** operations on these tables.

Similar as for **Embarcadero Rad Server** and the Dream Factory REST API, we have a version of the todo-list demo that works with **FaunaDB**.

GET READY

The beta for **TMS WEB Core v1.4** is around the corner. We are doing the testing, finishing the demos and writing the new documentation.

TMS ALL-ACCESS users are in the front seat and can expect this beta accessible from their account shortly and after a couple of weeks testing, we will release this new 1.4.

We hope you are as excited as we are about the new **TMS WEB Core v1.4**.

And there is more, it is this v1.4 feature set that will also be included in TMS WEB Core for Visual Studio Code.

The project to offer a TMS WEB Core version integrated in the Microsoft free and cross platform **Visual Studio Code IDE** has significantly advanced in the past couple of months. A select group of beta users is currently test-driving the newest builds. Very shortly, TMS ALL-ACCESS users will also get access to the beta and after a few more weeks of testing/feedback/updates we plan to release this version as well.



ALL

develop • faster

All-access 1 year subscription to the entire product range



VCL VCL Components

Our component products for different areas of Windows application development, including UI components, scripting, Excel/PDF reporting, instrumentation controls, workflow, ...

BIZ Business Tools

Cross platform robust framework for building REST API servers, high performance ORM, multi-tier database usage, scripting, data-replication,

WEB WEB Components

Framework for creating modern web applications

.NET .NET Components

100% managed code Excel file manipulation engine & Excel & PDF report generation for .NET



FMX FMX Components

List of our components for FireMonkey cross platform application development (Windows / macOS / iOS / Android)

IW IW Components

List of our products for creating feature-rich web apps with the IntraWeb/VCL for the Web framework, including grids, advanced edits, planner, query tools, navigational controls,...

FNC FNC Components

Framework neutral components for use with Delphi & C++Builder VCL, FMX & Lazarus LCL for cross-platform application development targeting Windows, macOS, iOS, Android, Linux & web

DEV Developer Tools

A selection of developer tools, logging framework and static Delphi code analysis tool



LCL LCL Components

Components for use with Lazarus LCL framework targeting Windows, macOS and Linux

TMS myCloudData.net

Free access to one FULL myclouddata.net account subscription is available during the active license period.

TMS ALL-ACCESS
Subscription includes:

€ 1 695

Entire product range
Previews & betas

- License for commercial use
- Free 1 year updates & new releases
- Renewal offered at 70% discount
- Free support through email & forum

**SINGLE DEVELOPER
LICENSE**

LICENSE FOR
1 DEVELOPER



INTRODUCTION: WHAT IS ORM?
BY JOHN KUIPER

starter expert



Object-Relational Mapping (ORM, sometimes O/RM) is a data-exchange technique required when attempting to relate the otherwise incompatible architecture of objects (as found in object-oriented languages) with the architecture of classic relational databases.

A typical SQL database management system (DBMS) uses normalised data containers (usually tables) whose component fields are all scalar types such as integers and strings; whereas an object-oriented approach to data management uses non-scalar object containers. For example, an object-oriented address book would consist of multiple person-objects, each person-object associated not only with fields such as name, gender and so on, but with sub-objects such as a phone-object containing lists of sub-objects each with a phone number and description; an address object containing sub-objects listing current and past addresses and descriptions, and so on. The person-object is treated as a single entity by the object-oriented programming language, which can have a single pointer variable referencing the object. The object will have various methods such as a method to return the preferred phone number, the current home address and so on. This data is retrieved from the tree-like object container. Any given person object can have zero or more phone numbers, zero or more addresses.

For the object's data to be persistent requires storing the object in some sort of database. This means disintegrating the various parts of the hierarchically structured object into simple scalar values which can be represented logically in the corresponding fields of a relational database. ORM is the attempt to do this, and has to both atomise or "explode" the object to store it, and later also to integrate the exploded parts coherently. This means correctly preserving not just the object's properties but their complex inter-relationships so that when data is subsequently retrieved from the database the objects are perfectly recreated. This is the meaning of the M in ORM: the mapping.

In effect, ORM creates a "virtual object database" that you can use from within the object-oriented programming language. Software, both free and commercial, is available to perform object-relational mapping, although some programmers opt to construct their own ORM tools, as I have done and explain in what follows.

Object Pascal has several existing ORM tools designed to work with standard GUI components, including the mORMot and tiOPF frameworks which map Pascal objects to a database.

- **mORMot** is an open source client-server ORM SOA MVC framework for Free Pascal, whose server targets Windows and Linux, and whose clients can be any platform, including mobile and AJAX. mORMot uses an internal SQLite database to store all kinds of data.
- **tiOPF** is a free, open source framework for Delphi and Free Pascal that simplifies the mapping of an object oriented business model into a relational database. The framework is mature and robust having been in use on production sites since 1999. It is available for immediate download with full source code.

There are similar frameworks which combine non-data components into a REST database without using a dataset. Both the above frameworks work well, but their documentation is problematic. One has little documentation but numerous examples. The other has an overwhelming 1000-page PDF. Exploring either of these two frameworks requires a lot of time and patience.

Although in my view both frameworks have very complex libraries, I nevertheless learned from tiOPF how to connect my standard GUI components to an object and how to work with other objects to read and write their data.

I wanted to try building a simple ORM for myself to explore what is really needed. That is how I built my own **MyMediator** framework, starting by designing the **TMyBaseMediator** class. I explain the basic functions of **TMyBaseMediator** later. This article shares the code I have developed so far, and **Blaise Pascal Magazine** provides the full sources for download.



MYMEDIATOR: THE BASICS

It is best to save components using objects. I chose to use generic code using the FGL in FPC/Lazarus (and `system.generics.collections` in Delphi). This gives you a 'static' way to read the added object without using typecasting as you need to using, say, `TObjectList` or `TList`. I also used a dictionary to add a named object. A dictionary is just like a `TStringList`, but works only with generic types. Using a dictionary gives you an easy way to search for an object with a specific name. In Lazarus I used `TFPGMap`; for Delphi I used `TDictionary`. The example below shows how the dictionary is linked to my `TMyCollection` class. `TMyCollection` has a `TControl` property called `fDisplayComponent`, `TControl` being the best ancestor for all the GUI components I want to use. To begin with, only `TEdit` will be provided to the base mediator as a display component.

```
unit clMyBaseMediator;

interface

uses classes, fgl, sysutils, controls, stdctrls;

type TMyDeclarations = (dcString, dcInteger, dcFloat, dcDate);

TMyCollection = class
  fDisplayComponent : TControl;
  fClassname       : string;
  fOldValue        : string;
  fReadOnly        : boolean;
end;

TMyBaseMediator = class
private
  fcomponentList : Tmydictionary;
  function locate(const aFieldname : string; var cIn : Tmycollection) : boolean;
protected
  procedure mdread(const aFieldname: string; const aValue: variant);
public
  constructor Create;
  destructor Destroy; override;

  procedure AddComp(aCompname : TControl; const aFieldname: string);
end;

implementation

constructor TMyBaseMediator.Create;
begin
  fcomponentList := Tmydictionary.create;
end;

destructor TMyBaseMediator.Destroy;
begin
  fcomponentList.Free;
  inherited;
end;
```




```

procedure TMyBaseMediator.mread(const aFieldname: string; const aValue: variant);
var cln : TMyCollection;
begin
  cln := nil;
  if locate(aFieldname,cln) then
  begin
    if cln.fDisplayComponent is Tedit then
      TEdit(cln.fDisplayComponent).Text := aValue;
    end;
  end;

procedure TMyBaseMediator.AddComp(aCompname : TControl; const aFieldname: string);
var cln : TMyCollection;
begin
  cln := TMyCollection.Create;
  cln.fDisplayComponent := aCompname;
  cln.fClassname := aCompName.ClassName;
  fComponentlist.Add(aFieldname,cln);
end;

function TMyBaseMediator.locate(const aFieldname : string; var cln : TMycollection) :
boolean;
begin
  result := fComponentlist.TryGetData(aFieldname,cln)
end;

```

For a demonstration let's connect a few edits to the mediator. We create a **World** application, and place several labels (**country**, **capital**, **population** and **continent**) with associated edits to receive our data on the main form, naming the form **FrmMain**.

To connect the four **TEdit** instances to the mediator I can use **TBaseMediator**, but then I have to copy the unit several times and code will be duplicated. It is better to create a new **TBaseMediator** descendant named **TMainMediator** so that in using **TMainMediator** we can inherit all needed code and type declarations from a single ancestor class:

```

type
  TMainMediator = class(TBaseMediator)
end;

```

Our main goal is to develop an application where code is separated from the GUI. The form should not know what the database is doing with the data. This separation or 'decoupling' has the advantage that we can easily modify or replace the form with its GUI while still working with data. So let's create a separate unit containing a class devoted to communication with the form. Create a new unit named **clMain** and a class named **TMain**. The form class is then declared with a public **TMain** variable named **fLink**.

clMain unit:

```

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

implementation

type
  TMain = class
  end;

end.

```



form unit:

```
uses .....; clmain;

type
  TFrmMain = class(TForm)
    .....
    procedure FormClose(Sender: TObject; var
      CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
  public
    flink: TMain;
  end;

implementation

{$R *.lfm}

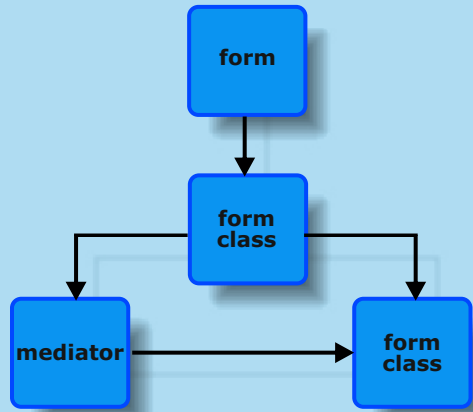
procedure TFrmMain.FormCreate(Sender: TObject);
begin
  flink := TMain.Create;
end;

procedure TFrmMain.FormClose(Sender: TObject; var
  CloseAction: TCloseAction);
begin
  flink.Free;
end;
```

We have created the **gateway**. Now we can declare **Mediator** as a property based on a **TMainMediator** field named **fMediator** (**Mediator** is of type **TMainMediator**). Using the form's **fLink** variable we can connect all the form's edits to the mediator in the form's **FormCreate** procedure:

```
procedure TFrmMain.FormCreate(Sender: TObject);
begin
  flink := TMain.create;
  flink.Mediator.AddComp(EDCountry, 'countryname');
  flink.Mediator.AddComp(ECapital, 'capital');
  flink.Mediator.AddComp(EPopulation, 'population');
  flink.Mediator.AddComp(EContinent, 'continent');
end;
```

Let me explain. **AddComp** is a method of **TBaseMediator** which takes two parameters: a component (here it is an edit) and a component name. Because **TEdit** derives from **TControl**, **TEdit** can be saved as a pointer value. The second parameter is the name identifying the component. Hence each edit can display its unique value, based on its name. For the application to know which value applies to which edit, we have to make a new class containing the variables and an object list that holds a list of values (much like **TDataset**). This class is saved in a unit named **objWorld**. The following diagram illustrates how each object is related to the others to work together:



We create a **TCountry** class with string fields for country name, capital, population and continent, and a list to contain the country records. To put data into an edit using the mediator we populate a record with data from my country of origin, add it to the list, and ask the mediator to transfer the data to the appropriate component using a procedure named **ReadToComp**, which takes as a parameter the desired country record.

```
procedure TMain.init;
var country : TCountry;
begin
  Country := TCountry.create;
  Country.countryname := 'the Netherlands';
  Country.capital := 'Amsterdam';
  Country.population := '14 million';
  country.continent := 'Europe';
  fWorld.Countrylist.Add(Country);
  fMediator.ReadToComp(fWorld.fcountrylist[0]);
end;
```

TMain.init is called in the form's **TFrmMain.FormCreate**. Once the list is filled with data we call **fMediator.ReadToComp** which is implemented as follows:

```
procedure TMainMediator.ReadToComp(aCountry : TCountry);
begin
  mhread('countryname', aCountry.name);
  mhread('capital', aCountry.capital);
  mhread('population', aCountry.population);
  mhread('continent', aCountry.continent);
end;
```

ReadToComp calls **mdRead** with three parameters: the name of the GUI object stored in dictionary, a data value declared as a variant type, and an optional value to specify the declared type of the data. In the absence of a third parameter the type defaults to string. **mdRead** is implemented like this:



```

procedure TMyBaseMediator.mdread(const aFieldName: string; const aValue: variant; aDec : TMyDeclarations);
var cln : TMycollection;
begin
  if locate(aFieldName,cln) then
    begin
      if cln.fDisplayComponent is Tedit then Edit(cln.fDisplayComponent).Text := aValue;
    end;
  end;

```

The `mdread` function first tries to locate the right component using the dictionary to search for `aFieldName`. If this succeeds it checks if the component found is an edit. If so it sets the edit's text. Initially we instructed the mediator to link to the form's edits. The reference to the edit is actually a pointer giving the edit's address, which we use to set its `Text`, the value which is displayed. Notice that the form remains unaware of what is really going on.

FURTHER DATA

We will extend the mediator to add, save and scroll data, and extend the main form with a toolbar containing four buttons, using an action list to provide each button with an appropriate action. See <https://wiki.freepascal.org/TActionList> if you are not familiar with actions. To ensure an empty country list at startup, remove the existing lines from `TMain.init`, and create a new boolean property named `NewRecord` in the `Tmain` class. In the object inspector double-click the main form's `btnAdd.OnClick` event to generate a new handler, whose implementation will call a new `TMain` method named `addrecord`.

form unit:

```

procedure TFrmMain.BtnAddClick(Sender: TObject);
begin
  flink.AddRecord;
end;

```

clMain unit:

```

procedure TMain.addrecord;
begin
  fMediator.Clear;
  fNewRecord := true;
  fWorld.IncreaseID;
end;

```

We add a `Clear` method to the base mediator to empty all connected components:

```

procedure TMyBaseMediator.Clear;
var cln : TMycollection; index : integer;
begin
  for index := 0 to fcomponentList.Count - 1 do
    begin
      cln := fComponentlist.Data[index];
      TEdit(cln.fDisplayComponent).Text := "";
    end;
  end;

```

We set the `NewRecord` property to `True` to tell the `Save` method to create a new object to be stored in the list. Generate an `OnClick` handler for the `btnSave` button and implement it as

```
flink.save;
```

which simply calls `TMain.save`.`

The variable `fNewRecord` creates a new `TCountry` record and calls `TMainMediator.CmpToWrite`, a new method we add to `TMainmediator` and `Basemediator` to accomplish this:

```

procedure TMainMediator.CmpToWrite(aCountry : TCountry);
begin
  aCountry.countryname:= mdwrite('countryname');
  aCountry.capital := mdwrite('capital');
  aCountry.population := mdwrite('population');
  aCountry.continent := mdwrite('continent');
end;

```

```

function TMyBaseMediator.mdwrite(
const aFieldName : string):variant;
var cln : TMycollection;
begin
  cln := nil;
  if locate(aFieldName,cln) then
    begin
      if cln.fDisplayComponent is TEdit then
        result := TEdit(cln.fDisplayComponent).Text;
    end;
  end;

```

After saving, the variable `fNewRecord` is set to `False`. Now we add several objects to the object list. But if the record already existed and only needs modifying, how do you know which object must be modified? Although we could search for a specific name, names can be modified. It is better to identify a record via an integer ID field. So we add an integer variable called `ID` to the `TCountry` class listed by `objcountrylist`. Each instance is saved with a unique `ID` value. In the `TWorld` class we add three further properties:




```
property LastID: integer read fLastID write fLastID;
property ID: integer read fID write fID;
property Error: boolean read fError write fError;
```

Now we can locate the right instance when saving an edit's modified **Text** value:

```
function TWorld.FindRow: TCountry;
begin
  result := nil;
  if assigned(fcountrylist) then
  begin
    if (fcountrylist.Count > 0) and fID >= 0 then
      result := fCountrylist[fID]
    end;
  end;
end;
```

The procedure **TMain.save** now looks like this:

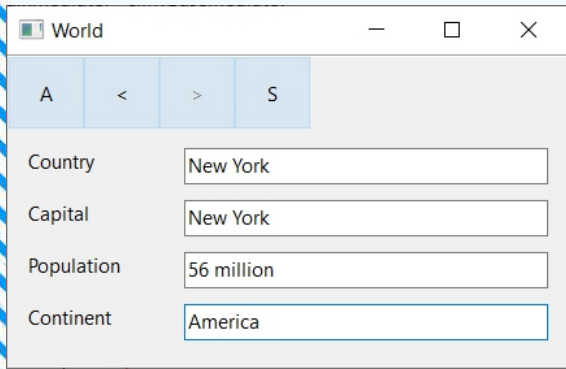
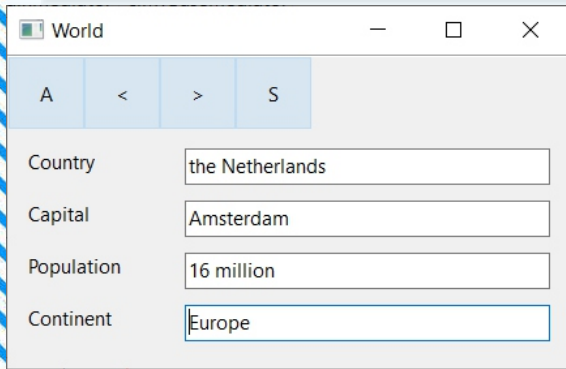
```
procedure TMain.Save;
var Country : TCountry;
begin
  if fNewRecord then
  begin
    Country := TCountry.Create;
    Country.id := fWorld.LastID;
  end else
    Country := fWorld.FindRow;
  fMediator.CmpToWrite(Country);
  if fNewRecord then
  begin
    fWorld.Countrylist.Add(Country);
    fWorld.ID := fWorld.LastID;
    fNewRecord := false
  end;
end;
```

We have to be able to scroll up and down the country list in order to show the correct edit values, so we write **TMain.recordprior** and **TMain.recordnext** methods:

```
procedure TMain.recordprior;
var Country : TCountry;
begin
  if fWorld.ID > 0 then
  begin
    fWorld.ID := fWorld.ID - 1;
    Country := fWorld.FindRow;
    fMediator.ReadToComp(Country);
  end;
end;

procedure TMain.recordnext;
var Country : TCountry;
begin
  if fWorld.ID < fWorld.LastID then
  begin
    fWorld.ID := fWorld.ID + 1;
    Country := fWorld.FindRow;
    fMediator.ReadToComp(Country);
  end;
end;
```

We can now view and edit existing data as the next screenshots show:

How about deleting an item? We need a further **acDelete** action associated with a **btnDelete** button added to the toolbar, along with a **DeleteRecord** method and an implementation of the **Execute** method of **acDelete**.

```
procedure TfrmMain.acDeleteExecute(Sender: TObject);
begin
  flink.DeleteRecord;
  if flink.ID = -1 then
  begin
    flink.AddRecord;
    ECountry.SetFocus;
  end;
end;

procedure TMain.DeleteRecord;
var country : TCountry;
    emptylist : boolean;
begin
  emptylist := fWorld.RowDelete;
  if not fWorld.Error then
  begin
    if emptylist then
    begin
      country := fWorld.FindRow;
      fMediator.ReadToComp(country);
    end else
      AddRecord;
    end else
      showmessage('cannot delete item')
  end;
```



The `TMain.DeleteRecord` method calls `TWorld.RowDelete` to delete an `fcountrylist` item and re-index the `ID` in the `TCountryObject`:

```
function TWorld.RowDelete: boolean;
var myID : integer;
begin
  try
    fError := false;
    myID := fID;
    fcountrylist.Delete(fID);
    fcountrylist.Pack;
    if fcountrylist.Count = 0 then
      begin
        result := false;
        fID := -1;
      end else
      begin
        if myID = fLastID then
          begin
            myID := myID - 1;
            if myID = -1 then
              begin
                fID := -1;
                result := false;
              end else
              begin
                fID := fID - 1;
                result := true;
              end;
            end else if myID = 0 then
              begin
                fID := -1;
                result := false;
              end else if myID < fLastID then
              begin
                result := true;
              end;
            end;
            Reindex;
          except
            fError := true
          end;
        end;
      end;
end;
```

Adding a non-TEdit component

The `continent` field value is identical for European countries such as the United Kingdom, Belgium, the Netherlands and Germany. The user could repeatedly type `Europe` in the appropriate field for each record, but sometimes you mistype (Europe, europe, Europa). It is better to select a correctly spelled continent name from a provided list. We use a `TComboBox` for this functionality. So replace the `Econtinent` edit with a `CBContinent` combobox. We then have to tell the mediator a new component is being used (`fLink.Mediator.AddComp(CBContinent, 'continent')`). We also need to make modifications to `TBaseMediator` to ensure we are reading from and writing to the correct component:

```
procedure TMyBaseMediator.mdread(const aFieldname: string;
const aValue: variant; aDec : TMyDeclarations);
var cln : TMycollection;
begin
  if locate(aFieldname,cln) then
    begin
      if cln.fDisplayComponent is Tedit then
        Tedit(cln.fDisplayComponent).Text := aValue
      else if cln.fDisplayComponent is Tcombobox then
        Tcombobox(cln.fDisplayComponent).Text := aValue;
    end;
  end;

function TMyBaseMediator.mdwrite(const aFieldname : string)
:variant;
var cln : TMycollection;
begin
  if locate(aFieldname,cln) then
    begin
      if cln.fDisplayComponent is Tedit then
        result := Tedit(cln.fDisplayComponent).Text
      else if cln.fDisplayComponent is Tcombobox then
        result := Tcombobox(cln.fDisplayComponent).Text;
    end;
  end;
end;
```

Now we have to populate the `CBcontinent` combo box with items for display and selection. Choosing from a fixed list always gives the user consistent values for the continent field. However, more often the list is dynamic, and the items are retrieved from values in a database, often via a key which corresponds to a database value, a lookup field. For this example we will create `objContinents` just as we created `objWorld`. We create a new `objbase.pas` unit with the following `TObjectBase` class declaration:




```

type
TObjectBase = class
  fError : boolean;
  fID : integer;
  fLastID : integer;
public
  property LastID : integer read fLastID write fLastID;
  property ID : integer read fID write fID;
  property Error : boolean read fError write fError default false;
end;
    
```

The **TWorld** and **TContinentTable** classes can use **TObjectBase** as a base class.

```

TWorld = class(TObjectBase)
end;
    
```

This enables us to save the same properties with less code, simply by putting

```
uses objbase;
```

in the interface sections of the appropriate units.

Again we create the **init** procedure to call the **TContinentTable** class. We declare the object locally, since we only need the continent values to populate **TComboBox.Items**, they are not needed in the rest of the application.

```

procedure TMain.init(aContinent: TStrings);
var fContinentTable : TContinentTable;
    Continent : TContinent;
begin
  fContinentTable := TContinentTable.create;
  try
  for continent in fContinentTable.Continentlist do
    aContinent.Add(continent.name);
  finally
    fContinentTable.free;
  end;
end;
    
```

form unit:

```

procedure TFrmMain.FormCreate(Sender: TObject);
begin
  <some code>
  flink.init(CBContinent.items);
end
    
```

Now when we run the application we can select a continent, without making mistakes when typing its name.

Data Persistence

This is all very well, but when you close the application, all the data is lost. We need a small database to make our data persistent. I chose **SQLite** for this since it is small, easy to use, and has drivers for almost every OS. For Lazarus I need one DDL to connect the SQLite database.

The library can be found at

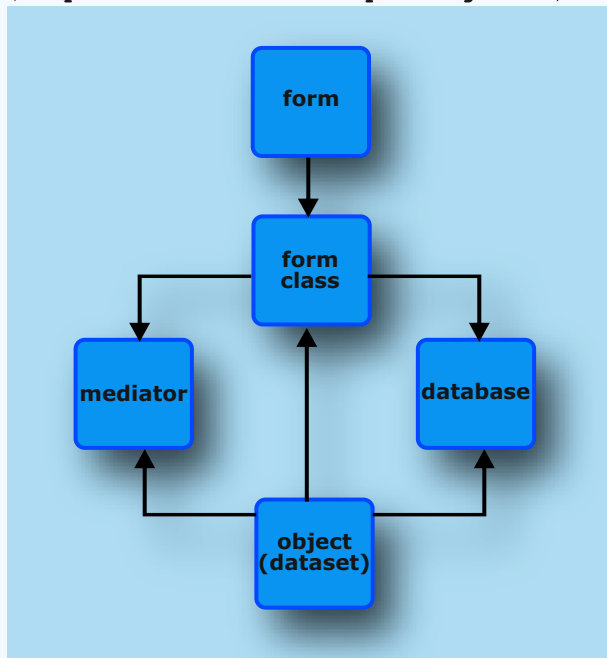
<https://www.sqlite.org/download.html>.

For our application we need the pre-compiled Windows driver.

Download the correct driver for the bitness of your Lazarus (either 32-bit or 64-bit). Unzipping the file gives two files : **sqlite3.dll** and **sqlite3.def**. I place these files in my development directory.

You can also put them in **Windows\system32**. Of course under Unix you require different client libraries than these, which apply to Windows. For working with **SQLdb** I refer you to **my SQLdb** article in **Blaise Magazine Issue 80 page 16** or the tutorial at https://wiki.freepascal.org/SQLdb_Tutorial1.

The diagram above explained how the form class is related to the mediator. Now we have added a database the diagram is expanded with **dbtools** (<https://online.visual-paradigm.com>):



The database works with both formclass and object, and the form continues to be ignorant of any database. Object is the communication layer for the data. We need to create a new unit called **clDBWorld**. This unit inherits from the **TMyDatabase** base object, whose values and properties are used to communicate with the database unit:

clDatabase unit:

```
TMyDatabase = class
  fSQLQuery      : TSQLQuery;
  fSQLTransaction : TSQLTransaction;
private
  fdberror: boolean;
  fRecordCount: integer;
public
  constructor create;
  destructor destroy; override;

  procedure GetTable(const aQuery : string);

  function GetLastID: int64;
  function dataset : tdataset;

  property recordcount :
    integer read fRecordCount write fRecordCount;
  property dberror :
    boolean read fdberror write fdberror;
end;
```

clDBWorld unit:

```
type TDBWorld = class(TMyDatabase)
strict private
  fworld : TWorld;
  procedure ReadValuesToObject;
public
  procedure GetValues(
    const aTablename : string; aWorld : TWorld); overload;
  procedure GetValues(aQuery : TStrings; aWorld : TWorld);
overload;
  procedure SaveToDatabase(const aNewRecord : boolean);
  procedure RowDelete(const aID : integer);
end;
```

fWorld is a reference to the **TWorld** object in **objWorld**. We need this to load values from a database table. In the **clmain** unit the **formclass** creates a new object **fdatabase: TDBWorld**. In the **TMain.init** method we insert a new line of code to load the records into the **TWorld** object:

```
fdatabase.GetValues('countries', fWorld);
```

The first parameter specifies which table has to be called, and the second parameter links the **TWorld** object to the **TDBWorld** object **fdatabase**:



```

procedure TDBWorld.GetValues(const aTablename: string; aWorld : TWorld);
begin
  GetTable(format('select * from %s',[aTablename]));
  fWorld := aWorld;
  ReadValuesToObject;
end;

procedure TDBWorld.ReadValuesToObject;
var Country : TCountry;
begin
  if not dataset.Active then
    dataset.Active := true;
  RecordCount := dataset.recordcount;
  if RecordCount > 0 then
    begin
      fWorld.fID := 0;
      dataset.first;
      while not dataset.eof do
        begin
          Country := TCountry.create;
          Country.id := dataset.FieldName('id').AsInteger;
          Country.countryname := dataset.FieldName('countryname').AsString;
          Country.capital := dataset.FieldName('capital').AsString;
          Country.population := dataset.FieldName('population').AsString;
          Country.continent_id := dataset.FieldName('continent_id').AsInteger;
          fWorld.Countrylist.Add(Country);
          dataset.next;
        end;
      end;
      fWorld.fLastID := fWorld.Countrylist.Count - 1;
      dataset.Active := false;
    end;
end;

```

We only have to open the required table in my database to read all records. After loading, the table can be closed. All data is held in **TWorld**. If we run the application and no database is found, it creates a database with a **countries** table:

```

procedure TDMConnection.CreateTables;
var MyTables : TStrings;
begin
  MyTables := TStringlist.create;
  try
    SQLConnector1.GetTableNames(MyTables);
    if MyTables.IndexOf('countries') = -1 then
      begin
        SQLConnector1.ExecuteDirect(
          'CREATE TABLE "countries" (' +
          ' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +
          ' "countryname" VARCHAR (30) NOT NULL,' +
          ' "capital" VARCHAR (40) NOT NULL,' +
          ' "population" VARCHAR (20) NOT NULL,' +
          ' "continent_id" INTEGER NOY NULL DEFAULT 0);');
        SQLConnector1.ExecuteDirect('CREATE UNIQUE INDEX ,,
          countries_id" ON "countries" ("id");');
        SQLTransaction1.commit;
      end;
    finally
      MyTables.free;
    end;
  end;

```



In the previous example we put continent name strings directly into the **CBContinent** combobox. The new design means we cannot now save text in the table, only an integer as **continent_id**. It is always better to save the ID rather than the string because it uses less memory, and the description connected to the ID can easily change without changing the ID in the main table. To let the application find the right ID for a text item in **CBContinent**, create a new unit called **clLookup**. This gives us only one lookup table, but the object is created for multiple tables. Therefore we use this construct:

```
Type TLookupItem = class
  id : integer;
  value : string;
end;

TLookupItemList =
  specialize TFPGObjectlist<TLookupItem>;
TLookupList =
  specialize TFPGMap<string,TLookupItemList>;
```

TLookupList: **TFPGMap** (for Delphi use **TDirectory**) holds the name of the lookup table with its items found in **TLookupItemList**. The **TMain** object gets a new object **fLookup**: **TLookup**. In **TMain.init** we call the procedure

```
fLookup.LoadContinent(fContinentTable.Continentlist, aContinent);
```

The first parameter has all the continent records placed in the **TContinentTable** object, and the second parameter holds the items of **CBContinent**. We still have to fill the items to select one of them. We also have to tell the **TWorld** object that an extra value will be used to perform a lookup to populate **CBContinent**:

```
type
  TCountry = class
    id : integer;
    countryname : string;
    capital : string;
    population : string;
    continent_id : integer; //for database/lookup purpose
    continent : string;
  end;
```

At start-up the application finds no records, so it immediately adopts an **append** mode. We make one record for the country **England** (in the continent **Europe**). When saving the record we also have to call a procedure in the **fDatabase** object to physically save the values in the database:

clmain unit:

```
procedure TMain.Save;
var Country : TCountry;

begin
  if fNewRecord then
  begin
    fWorld.IncreaseID;
    Country := TCountry.Create;
  end else
    Country := fWorld.FindRow;
  fMediator.CmpToWrite(Country);
  Country.continent_id :=
    fLookup.GetKey('continent',Country.continent);
  if fNewRecord then
  begin
    fWorld.Countrylist.Add(Country);
    fWorld.ID := fWorld.LastID;
    fDatabase.SaveToDatabase(true);
    fNewRecord := false
  end else
    fDatabase.SaveToDatabase(false);
  if fdatabase.dberror then
    showmessage('record not saved');
end;
```



cIDBWorld unit:

```

procedure TDBWorld.SaveToDatabase(const aNewRecord: boolean);
var Country : TCountry;
    objJID : integer;
    MyQuery : TStrings;
begin
    Myquery := TStringlist.create;
    dberror := true;
    try
    if aNewRecord then
    begin
    MyQuery.AddText('INSERT INTO countries (countryname, capital, population, continent_id)');
    MyQuery.AddText('VALUES (:name, :capital, :population, :continent_id)');
    if fWorld.Countrylist.Count = 0 then
    objID := 0
    else
    objID := fWorld.Countrylist.Count - 1;
    end else
    begin
    MyQuery.AddText('UPDATE countries SET');
    MyQuery.AddText('countryname = :name, capital = :capital,');
    MyQuery.AddText('population = :population, continent_id = :continent_id');
    MyQuery.AddText('WHERE id = :id');
    objID := fWorld.fID;
    end;
    Country := fWorld.Countrylist[objID];
    fSQLQuery.SQL.Text := MyQuery.Text;
    fSQLQuery.Params[0].AsString := Country.countryname;
    fSQLQuery.Params[1].AsString := Country.capital;
    fSQLQuery.Params[2].AsString := Country.population;
    fSQLQuery.Params[3].AsInteger := Country.continent_id;
    if not aNewRecord then
    fSQLQuery.Params[4].AsInteger := Country.id;
    fSQLQuery.ExecSQL;
    dberror := fSQLQuery.RowsAffected = -1;
    if DMConnection.SQLConnector1.ConnectorType <> 'SQLite3' then
    fSQLTransaction.Commit
    else
    dmConnection.SQLTransaction1.Commit;

    if aNewRecord then
    Country.id := GetLastID;
    finally
    MyQuery.free;
    end;
end

```

If we close the application and look with the SQLiteStudio editor <https://sqlitestudio.pl> we see the record is saved:

	id	countryname	capital	population	continent_id
1	1	England	London	56 million	5

Now running the application again shows we have data. We now need a routine to display the continent's name rather than its ID. We do this with the procedure:



```

procedure TMain.init(aContinent: TStrings);
begin
    .....
    if fWorld.Countrylist.Count > 0 then Display;
    .....
end;

procedure TMain.Display;
var Country: TCountry;
begin
    Country := fWorld.FindRow;
    //lookup
    Country.continent := fLookup.GetValueFromIndex(
        'continent',Country.continent_id);
    fMediator.ReadToComp(Country);
end;
    
```

When you scroll the **Display** method searches in the **fLookup** object to display the right continent on screen.

A screenshot of a Windows application window titled "World". At the top, there are navigation buttons labeled "A", "S", "D", "<", and ">". Below these are four input fields: "Country" with the text "Republic South Africa", "Capital" with "Pretoria/Capetown/bloemfontein", "Population" with "52 million", and "Continent" with a dropdown menu showing "Africa".

A screenshot of a Windows application window titled "World". At the top, there are navigation buttons labeled "A", "S", "D", "<", and ">". Below these are four input fields: "Country" with the text "Engeland", "Capital" with "London", "Population" with "56 million", and "Continent" with a dropdown menu showing "Europe".

A screenshot of a Windows application window titled "World". At the top, there are navigation buttons labeled "A", "S", "D", "<", and ">". Below these are four input fields: "Country" with the text "Japan", "Capital" with "Tokyo", "Population" with "127 million", and "Continent" with a dropdown menu showing "Asia".

All that remains is to add record deletion functionality to the application. If you know how to save it, you can use almost identical code for deletion:

```

procedure TMain.DeleteRecord;
var country : TCountry;
    emptylist : boolean;
begin
    country := fWorld.FindRow;
    fDatabase.RowDelete(country.id);
    if fdatabase.dberror then
        showmessage('record not deleted')
    else begin
        .....
    end;
end;

procedure TDBWorld.RowDelete(const aID: integer);
begin
    dberror := true;
    fSQLQuery.SQL.Text :=
        'DELETE FROM countries WHERE id = :id';
    fSQLQuery.Params[0].AsInteger := aID;
    fSQLQuery.ExecSQL;
    dberror := fSQLQuery.RowsAffected = -1;
    if DMConnection.SQLConnector1.ConnectorType <>
        'SQLite3' then
        fSQLTransaction.Commit
    else
        dmConnection.SQLTransaction1.Commit;
end;
    
```



ADVANCED LOOKUP

While it is possible to load the lookup items from a hard-coded object, it gives more flexibility to load the items from a database. We amend the **TLookup** class with a **TMyDatabase** base class so we can load data from a table, adding a **TSQLQuery** to load data into the items:

```
TLookup = class(TMyDatabase)
  fLookupList : TLookupList;
private
  procedure GetItems(const aItems: TStrings);
public
  .....
  procedure LoadContinent(aItems : TStrings);
  .....
end;

implementation
{ TLookup }

procedure TLookup.LoadContinent(aItems : TStrings);
begin
  try
    GetTable('SELECT id, continent FROM continents');
    GetItems(aItems);
    TableCommit;
  finally
    fSQLQuery.Active := false;
  end;
end;

procedure TLookup.GetItems(const aItems: TStrings);
var Item: TLookupItem;
begin
  while not fSQLQuery.eof do
  begin
    Item := TLookupItem.Create;
    Item.id := fSQLQuery.Fields[0].AsInteger;
    Item.value := fSQLQuery.Fields[1].AsString;
    Add('continent', Item);
    aItems.Add(Item.value);
    fSQLQuery.Next;
  end;
end;
```

We call the **TableCommit()** method of **TMydatabase** to free the transactions after reading and closing the dataset (table). Otherwise we work with **TLookup** exactly as before. We no longer need **TContinentTable**. Later on we do use it to read the data for **adding / modifying / deleting** lookup table items. The only thing left to do is creating a table with records in the database unit:

```
if MyTables.IndexOf('continents') = -1 then
begin
  SQLConnector1.ExecuteDirect(
  'CREATE TABLE "continents" (' + ' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +
  ' "continent" VARCHAR (30) NOT NULL);');
  SQLTransaction1.commit;
  SQLConnector1.ExecuteDirect(
  'INSERT INTO continents (continent) VALUES ' + ('Asia'),('North America'),('South America'),('Autralia'),' +
  ('Europe'),('Africa'),('Antartica');');
  SQLTransaction1.commit;
```

en



ADDING APPLICATION FORMS

To see if it really works let's expand the application with a new form. A Countries form will show that country's cities with information about each city. A new form has its own units (**formclass**, **objectclass**, **dbobjectclass**). The database is expanded with a table view (the text within brackets is already put in code):

```

Procedure TDMConnection.CreateTables;
var MyTables : TStrings;
begin
  MyTables := TStringlist.create;
  try
    SQLConnector1.GetTableNames(MyTables);
    if MyTables.IndexOf('countries') = -1 then
      begin
        {countrysettings}
      end;
    if MyTables.IndexOf('cities') = -1 then
      begin
        SQLConnector1.ExecuteDirect(
          'CREATE TABLE "cities" (' + ' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +
          ' "country_id" INTEGER NOT NULL,' + ' "cityname" VARCHAR (30) NOT NULL,' +
          ' "major" VARCHAR (30) NOT NULL,' + ' "capital" BOOLEAN NOT NULL DEFAULT FALSE,' +
          ' "sightseeing" VARCHAR (100),' + ' "square" INTEGER NOT NULL DEFAULT 0,' +
          ' "poprange_id" INTEGER NOT NULL,' + ' "township_id" INTEGER NOY NULL DEFAULT 0););
        SQLConnector1.ExecuteDirect('CREATE UNIQUE INDEX "city_id" ON "cities" ("id"););
        SQLTransaction1.commit;
      end;
    if MyTables.IndexOf('continents') = -1 then
      begin
        {continent settings}
      end;
    if MyTables.IndexOf('popranges') = -1 then
      begin
        SQLConnector1.ExecuteDirect( 'CREATE TABLE "popranges" (' +
          ' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' + ' "rangename" VARCHAR (30) NOT NULL););
        SQLTransaction1.commit;
        SQLConnector1.ExecuteDirect( 'INSERT INTO popranges (rangename) VALUES ' +
          '("1 - 1.000"),("1.001 - 10.000"),("10.001 - 25.000"),("25.001 - 50.000"),' +
          '("50.001 - 75.000"),("75.001 - 100.000"),("100.001 - 250.000"),("250.001 - 500.000"),' +
          '("500.001 - 1.000.000"),("1.000.001 - 2.500.000"),("2.500.001 - 5.000.000"),("5.000.000 and more"););
        SQLTransaction1.commit;
      end;
    if MyTables.IndexOf('townships') = -1 then
      begin
        SQLConnector1.ExecuteDirect( 'CREATE TABLE "townships" (' + ' "id" INTEGER NOT NULL PRIMARY KEY
          AUTOINCREMENT,' + ' "townshipname" VARCHAR (30) NOT NULL););
        SQLTransaction1.commit;
        SQLConnector1.ExecuteDirect(
          'INSERT INTO townships (townshipname) VALUES ' + '("Village"),("Town"),("City"););
        SQLTransaction1.commit;
      end;
  finally
    MyTables.free
  end;
end;

```

We create `objcities` as a new form object. Most of the code will be copied from `objworld`. The only code that has really changed is this:

```

type
  TCity = class
    id : integer;
    country_id : integer; //for database purpose
    cityname : string;
    major : string;
    capital : boolean;
    sightseeing : string;
    square : integer;
    poprange_id : integer; //for database purpose
    population : string;
    township_id : integer; //for database purpose
    township : string;
  end;

TCitylist = specialize TFPGObjectList<TCity>;

```



We rename the class to **TCities** changing **TWorld** in all procedures, changing **@countrylist** to **@Citylist** and **@Country** to **@City** (@ means all characters in the word).

Create a new **c1DBCity** unit and copy all code from **c1DBWorld** changing the name of the class from **TDBWorld** to **TDBCity** and all related objects and variables related to the **objCit*** file.

As you can see in the **TCity** class, the **square** variable is of integer type. String variables have unlimited characters, but database fields are always limited in storage. When a query is posed to a database, much extra metadata is also extracted from the table definition such as length and datatype (**(var) char** , **integer** , **boolean** , **float**). We want the GUI components to know about these attributes, so we expand the **c1database** file (related to **c1dbcity**) to load these attributes:

```
interface
uses Classes, SysUtils, sqlDB, dConnection, sqlite3conn, DB, fgl, typinfo;

type
TobjFielddef = class
    fieldname : string;
    varlength : integer;
    datatype : string;
end;

TObjFielddeflist = specialize TFPGObjectList<TobjFielddef>;
{ TMyDatabase }
TMyDatabase = class
    {previous code}
    Private
    {previous code}
    fObjFielddeflist : TObjFielddeflist;
    protected
    procedure LoadAttributes;
    public
    {previous code}
    property ObjFielddeflist : TObjFielddeflist read fObjFielddeflist write fObjFielddeflist;
end;

implementation
{ TDatabase }
constructor TMyDatabase.create;
begin
    fObjFielddeflist := TObjFielddeflist.create;
    {previous code}
end;

destructor TMyDatabase.destroy;
begin
    fObjFielddeflist.free;
    fSQLQuery.free;
    inherited destroy;
end;

procedure TMyDatabase.LoadAttributes;
var index : integer; Fielddef : TobjFielddef;
begin
    fSQLQuery.First;
    for index := 0 to fSQLQuery.Fields.Count -1 do
    begin
        Fielddef := TobjFielddef.Create;
        Fielddef.fieldname := fSQLQuery.Fields[index].FieldName;
        Fielddef.varlength := fSQLQuery.Fields[index].DataSize;
        Fielddef.datatype := GetEnumName(TypeInfo(TFieldType), Ord(fSQLQuery.Fields[index].DataType));
        ObjFielddeflist.Add(Fielddef);
    end;
end;
```



```

procedure TDBCity.ReadValuesToObject;
var City : TCity; Fielddef : TobjFielddef;
begin
  if not dataset.Active then
    dataset.Active := true;
  {previous code}
  //this piece of code must be after reading the records.
  //If there are no records the definitions can still read
  if dataset.Active then
    LoadAttributes;
  {previous code}
end;

```

We also change this in the `clDBWorld` unit. The next file to create is `clCity.pas`. As with `clDbCity` we copy all the `clMain` code, changing the class name to `TFCity`, and changing all related variables and objects to the objects of `objcity.pas` and `clDbcity.pas`. There is one extra `country_id` property in `clCity.pas`. This is because all records created must be related to a country. Otherwise all records would be shown for every country. To give extra functionality to the GUI-components we will read the field definitions. So we add this code to the formclass `init()` procedure:

```

if fdatabase.ObjFielddeflist.Count > 0 then
begin
  for Fielddef in fdatabase.ObjFielddeflist do
    fMediator.SetAttributes(Fielddef.fieldname, Fielddef.datatype, Fielddef.varlength);
end;

```

As you can see, we call a procedure of `Mediator`. This can be in the baseobject of the mediator. Now every time a new `TMediator` is created the components are given corresponding attributes:

```

procedure TMyBaseMediator.SetAttributes(const aFieldName, aDatatype:
string;
  const aVarLength: integer);
var cln : TMycollection;
begin
  cln := nil;
  if locate(aFieldName, cln) then
    begin
      if cln.fDisplayComponent is TEdit then
        begin
          TEdit(cln.fDisplayComponent).MaxLength := aVarLength;
          if aDatatype = 'ftInteger' then
            begin
              TEdit(cln.fDisplayComponent).NumbersOnly := true;
              if aDatatype = 'ftInteger' then
                cln.fDatatype := dcInteger;
            end;
          end;
        end;
      end;
    end;

```

This change gives the program the opportunity to use the right data type for a variable in object and database without having to specify to `mdread` or `mdwrite` which data type must be used. Now we can create the form, dropping labels, edits, comboboxes, toolbar and actionlist on the form. We copy the code in the implementation section of `fmain.pas` to `fcity.pas` and link the action events to an action.

In `fmain.pas` we create a new button `spCities` and in its `OnClick` handler in `clmain.pas` create a new `LoadCityForm` procedure:

```

procedure TMain.LoadCityForm;
var frm : TFrmCity;
    Country : TCountry;
begin
  frm := TFrmCity.Create(nil);
  try
    Country := fWorld.FindRow;
    frm.Link.country_id := Country.id;
    frm.ShowModal;
  finally
    frm.free;
  end;
end;

```



MASTER/DETAIL RELATIONS

It is nice to see the countries of the world, but I want to see the cities as well. So we create a view in the first form with all related cities. It can be done with a grid or listview. Listview is the easiest way. In report mode it can hold multiple text and as reference I can save a pointer to each node.

Let's put a listview on the main form. All GUI components are controlled by the mediator. So we tell the mediator to link the listview:

```
fLink.Mediator.AddComp(LVDetail,'list1');
```

We add a new unit to the **uses** clause so

TBaseMediator can find the listview:

```
uses classes, fgl, sysutils, controls, stdctrls, comctrls;
```

The listview is blank, and to see which fields are used I created a new procedure in **TBaseMediator** to show the given fields. This procedure can be used in any form linked to the mediator.

```
procedure TMyBaseMediator.LVHeader(const aFieldname: string; aHeader: array of string);
```

```
var cln : TMycollection;
    LV : TListview;
    Column : TListColumn;
    index : integer;
    fielddef : TStringArray;
begin
    cln := nil;
    if locate(aFieldname,cln) then
    begin
        LV := TListview(cln.fDisplayComponent);
        LV.ViewStyle := vsReport;
        for index := 0 to length(aHeader) - 1 do
        begin
            fielddef := aHeader[index].split(';');
            Column := LV.Columns.Add;
            Column.Caption := fielddef[0];
            if length(fielddef) > 1 then
                Column.Width := StrToIntDef(fielddef[1],0);
        end;
    end;
end;
```

We call the

```
fMediator.LVHeader ('list1', ['cityname;200', 'population;150','township;150'])
```

procedure in the **init()** procedure in the **clmain** unit. The first parameter specifies which component is called. The second parameter is an open array.

This covers two functions: a field name and a field length separated by `;`.

If the field length is not given, the column's default length is used.

Now we need data. As for all data I use an object to hold the data from a query. The **TWorld** object in the **objWorld** unit was used for the main form's data. It is also the place for the new listview's data:

The 'World' form is a data entry window with a title bar and standard window controls. It features a toolbar with 'A', 'S', 'D', '<', and '>' buttons. The form contains four input fields: 'Country' with the value 'England', 'Capital' with 'London', 'Population' with '56 million', and 'Continent' with a dropdown menu set to 'Europe'.

The 'Cities' form is a data entry window with a title bar and standard window controls. It features a toolbar with 'A', 'S', 'D', '<', and '>' buttons. The form contains several input fields: 'Name' with 'London', a checked checkbox for 'Capital of state / province', 'Major' (empty), 'Sightseeing' with 'Big Ben, Westminster Abbey, Tower of London', 'Square km2' with '1577', 'Range of population' with a dropdown set to '5,000,000 and more', and 'Township' with a dropdown set to 'City'.

The 'Cities' form is a data entry window with a title bar and standard window controls. It features a toolbar with 'A', 'S', 'D', '<', and '>' buttons. The form contains several input fields: 'Name' with 'Adlington', an unchecked checkbox for 'Capital of state / province', 'Major' (empty), 'Sightseeing' with 'St Paul's Parish Church', 'Square km2' with '0', 'Range of population' with a dropdown set to '1,001 - 10,000', and 'Township' with a dropdown set to 'Town'.




```

TCityDetail = class
  country_id : integer; //for database purpose only
  cityname   : string;
  population : string;
  townrange  : string;
end;

TCityDetaillist = specialize TFPGObjectList<TCityDetail>;

{ TWorld }

TWorld = class(TObjectBase)
  fcountrylist : TCountrylist;
  fCityDetaillist : TCityDetaillist;
private
  procedure Reindex;
public
  constructor create;
  destructor destroy; override;

  function FindRow : TCountry;
  function RowDelete : boolean;

  procedure IncreaseID;

  property Countrylist : TCountrylist read fCountrylist;
  property CityDetaillist : TCityDetaillist read fCityDetaillist;
end;

```

Now we can call `fdatabase.GeDetaillist(fWorld)` to query and retrieve all detail data:

```

procedure TDBWorld.GeDetaillist(aWorld: TWorld);
var CityDetail : TCityDetail;
begin
  try
    if fSQLQuery.Active then fSQLQuery.Active := false;
    fSQLQuery.SQL.clear;
    with fSQLQuery.SQL do
      begin
        Add('SELECT c.id, c.country_id, c.cityname,');
        Add('      r.rangename, t.townshipname FROM cities c');
        Add(' LEFT OUTER JOIN popranges r on r.id = c.poprange_id');
        Add(' LEFT OUTER JOIN townships t on t.id = c.township_id');
        Add(' ORDER BY c.country_id, c.cityname');
      end;
    fSQLQuery.Active := true;
    aWorld.CityDetaillist.Clear;
    while not fSQLQuery.EOF do
      begin
        CityDetail := TCityDetail.create;
        CityDetail.country_id := fSQLQuery.Fields[1].AsInteger;
        CityDetail.cityname   := fSQLQuery.Fields[2].AsString;
        CityDetail.population := fSQLQuery.Fields[3].AsString;
        CityDetail.townrange  := fSQLQuery.Fields[4].AsString;
        aWorld.CityDetaillist.Add(CityDetail);
        fSQLQuery.Next;
      end;
    finally
      fSQLQuery.Active := false;
    end;
  end;
end;

```



The last thing I have to do is to show detail data in the listview for each country using the procedure

```
fMediator.ReadDetail('list1', Country.id, fWorld.CityDetaillist).
```

The first parameter specifies which component to use. The second parameter specifies which country's data is to be retrieved. The last parameter holds all the data the query retrieves.

cityname	population	township
Liverpool	250.001 - 500.000	City
London	5.000.000 and more	City
Worksop	25.001 - 50.000	Town

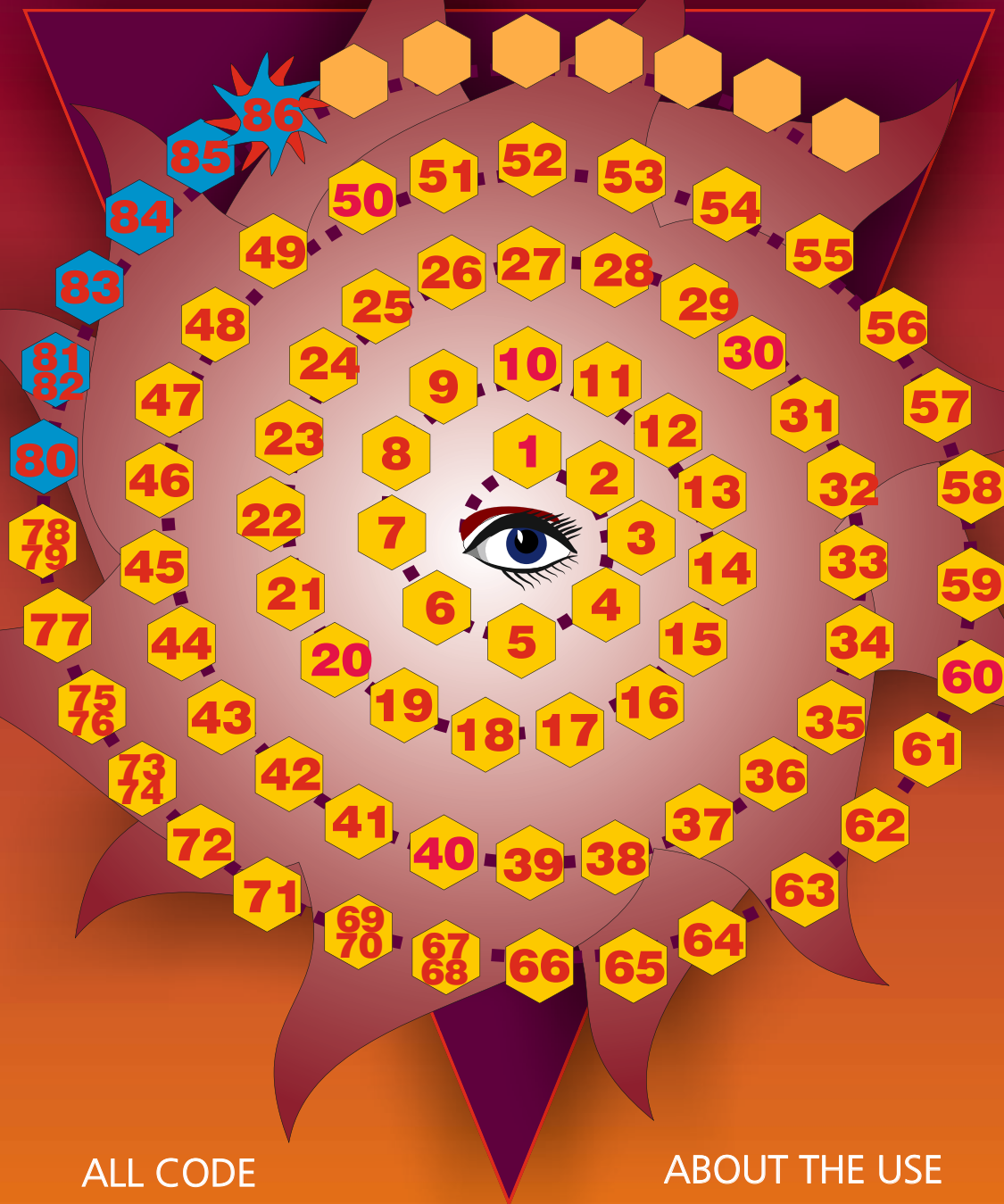
cityname	population	township
Bahia Blanca	250.001 - 500.000	City
Buenos Aires	2.500.001 - 5.000.000	City

CONCLUSION

It is a lot of work but these objects work well and faster than the traditional way of using a `TDataSet`. As far as I know I have managed to separate all data from the form. This works on every OS which Lazarus supports. In future I want to add creating a database connection. This way of working can also be used by SOAP / Rest servers. When data is called from these servers, it can be put in the objects and your application still works. I managed to read a spreadsheet into an object to see the result. Once a spreadsheet is open, it is read-only for other users, but I created an extra object that holds extra data outside the



LIBRARY 2020



BLAISE PASCAL MAGAZINE

ALL ISSUES IN ONE FILE





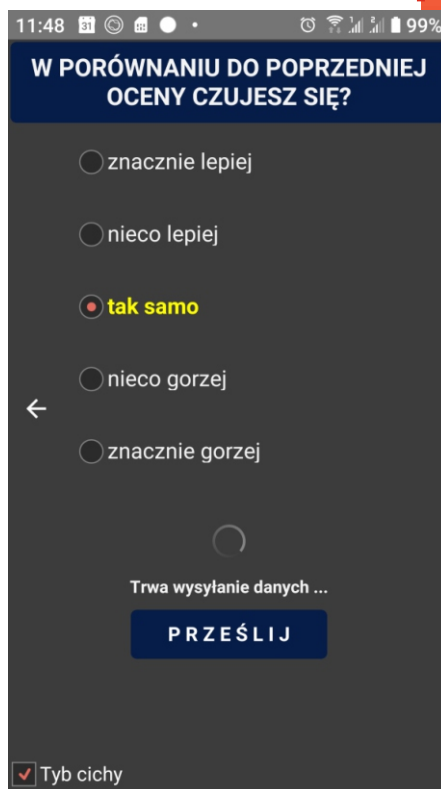
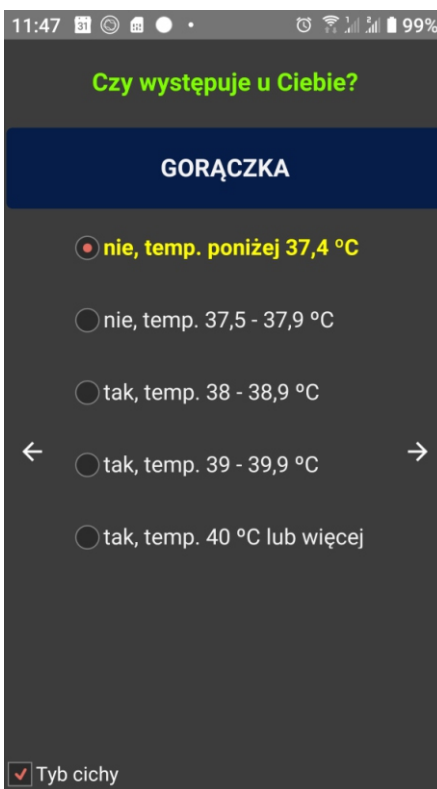
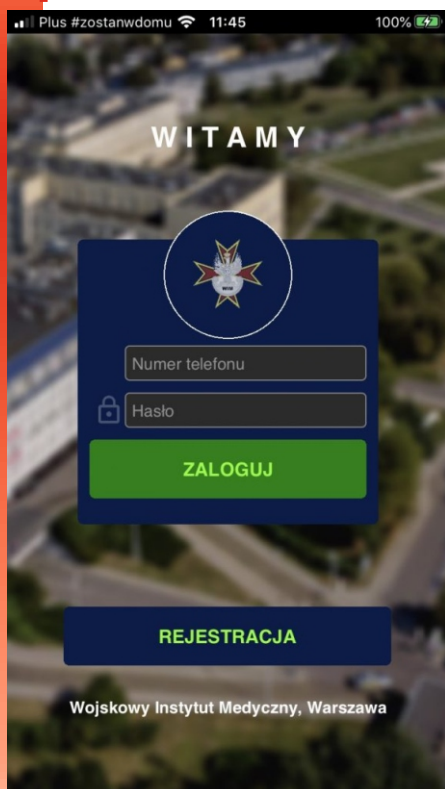
By Stephen Ball

The Military Institute of Medicine in Poland has been making the news recently with its app that is helping fight the spread of the global SARS-CoV-2 pandemic.

Piotr Murawski Ph.D, (Head of ICT, Military Institute of Medicine, Warsaw, Poland) shared recently with me how back on the 13th March, 2020, that after the SARS-CoV-2 epidemic reached Europe, the experience of other countries alerted his team that it will not be easy to control the infection, and testing will take a longer time than normal due to the numbers. This intern would have a negative impact on controlling the spread of the disease.

The solution

With an awareness that home quarantine was a likely step, the team started building an extension to their hospital-based software used by Epidemiologists to monitor the health of people at home. Initially tested by the Military Institutes Employees, it has since spread further detecting potential epidemic outbreaks and enabling rapid response.



For more details go to this address:

<https://www.medexpress.pl/mks-covid-19-ta-aplikacja-pomoze-chronic-zdrowie-i-zycie/77113>



After registration onto the system, the app gathers responses to 8 multi-choice questions, completed twice a day, and takes seconds to complete. Users are reminded via notifications and SMS to complete the survey.

Results are securely sent immediately to experts from the Epidemiology section of the Military Medical Institute in Warsaw where they are analyzed using artificial intelligence, to prioritize review by Epidemiologists. This mix ensures are Epidemiologists able to focus on those with the most need, rather than sifting through large volumes of data.

If the Epidemiologists makes contact with the patient, they are then able to supplement the data on the system with additional case notes.

As soon as answers raise any doubts, the relevant services make contact to inform the user what the next steps should look like, saving critical time and helping prevent transmission.

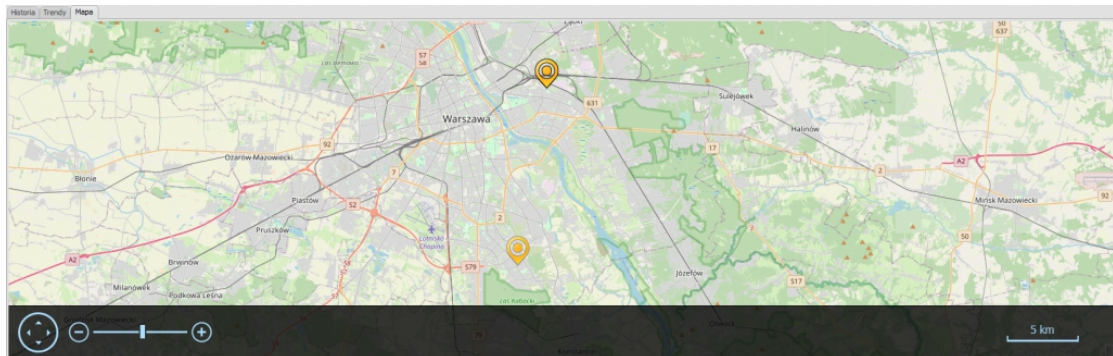
The app has three states:

Self Control

Quarantine

COVID-19 Positive

In the case of the last two states, after the user's consent, the system additionally sends location information, ready in case help is needed. This information is then available to staff



The development

The work for the project was undertaken by a team of four, including two medical scientists (Grzegorz Gielerak Prof, Paweł Krześciński Ph.D) and two members of the IT team. (Piotr Murawski Ph.D, Agnieszka Opłocka Msc) and even though it was their first mobile app, it was completed in under two weeks, including deployment to corporate app stores.

The solution is built using a single code base, build with Delphi 10.3.3, to deliver the life-saving remote app to Windows, iOS, and Android. The mobile app architecture includes components from TMS (for transmission of data securely).

<https://www.medexpress.pl/mks-covid-19-ta-aplikacja-pomoze-chronic-zdrowie-i-zycie/77113>



Jim McKeeth

<https://bs-sd.de/unsere-softwareprodukte/>

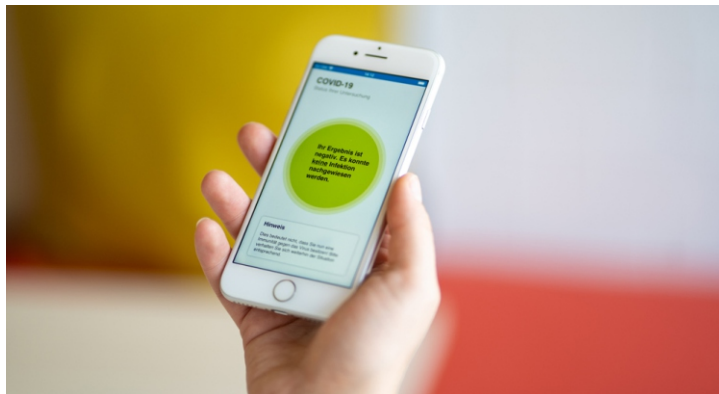
One of our customers, **BS Software Development**, is a leading manufacturer in networking in the medical field. Laboratories, clinics and resident doctors, and specialists are networked via the software solutions from BS Software (document exchange, laboratory results, etc.)

A current app for **iOS** and **Android** has been developed in record time with **Delphi** in the past few days that enables direct communication between the laboratories and the patients in Germany regarding test results for the Corona Virus SARS-CoV-2. This saves the patients time, because the app enables faster communication processes. After scanning a QR code, which is individually assigned to each patient, the test result is communicated in the app by a traffic light display. The app is free of charge and practical for patients. The result is communicated via a push notification, thus saving telephone inquiries from doctors or the health authorities.

Get more information about app including links to download the app:
<https://www.embarcadero.com/case-study/bs-software-development-case-study>

ERGEBNIS DES CORONA-TESTS IN ECHTZEIT

Angesichts der aktuellen Situation bezüglich des neuartigen Corona-Virus SARS-CoV-2 geht es vor allem darum, Zeit zu gewinnen. Um schnellere Abläufe in der Diagnostik zu gewährleisten, haben wir die COVID-19 App entwickelt, mit der auf Corona getestete Patienten schnellstmöglich über ihre Testergebnisse informiert werden. Durch die Corona App entfällt eine zeitintensive telefonische Abfrage, so bleiben die Leitungen frei und im Falle eines positiven Testergebnisses können alle erforderlichen Maßnahmen direkt eingeleitet werden.



Wie läuft das Ganze ab?

Zuordnung über QR-Codes. Grundlage für die Zuordnung der Proben sind Etiketten mit jeweils zwei zusammengehörenden QR-Codes. Wird bei einem Patienten ein Abstrich genommen, erhält der Patient eines der beiden Etiketten. Das zweite Etikett wird zusammen mit der Probe an das Labor gesendet.



Download und Login

Der Patient kann die COVID-19 App im jeweiligen App-Store herunterladen, installieren und sich mithilfe des QR-Codes oder der ID-Nummer auf dem Etikett einloggen.

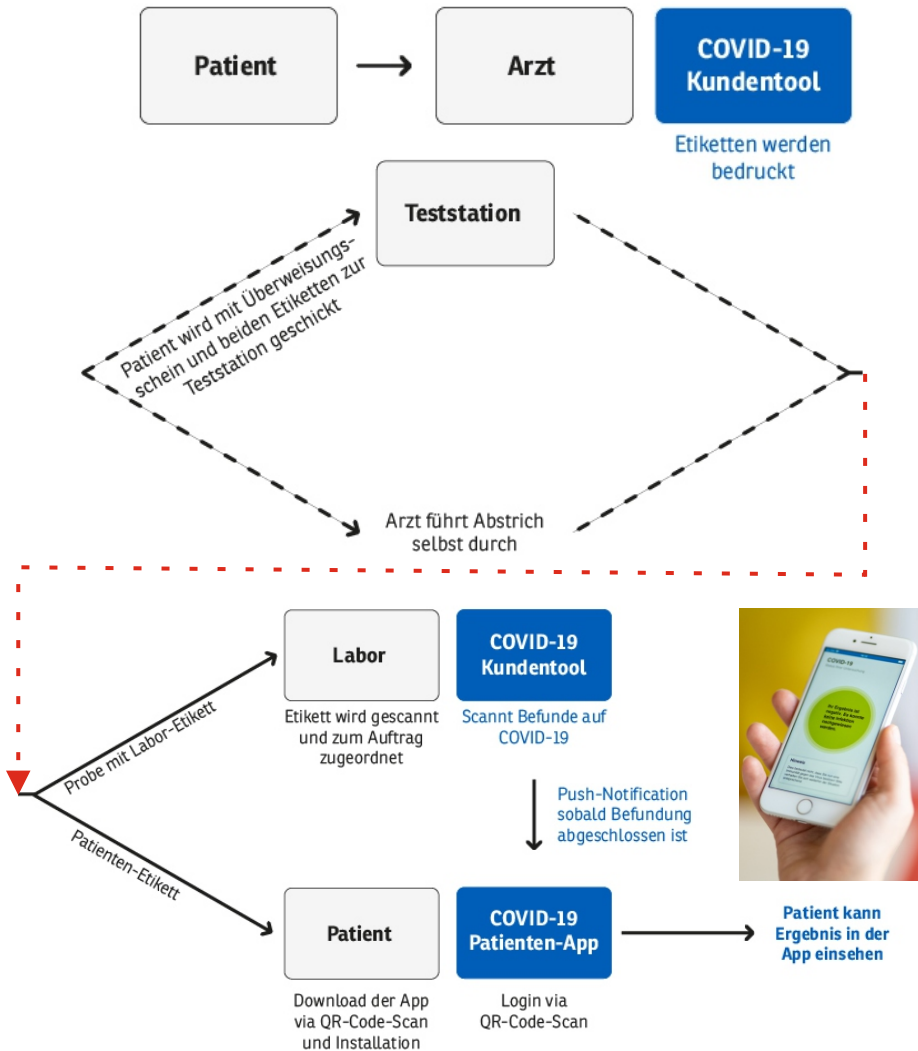
Benachrichtigung in Echtzeit

Sobald der Befund verfügbar ist, erhält der Patient eine Push-Notification. So wird er in Echtzeit informiert und kann das Ergebnis sofort in der App einsehen. Im Falle eines positiven Ergebnisses können sofort Maßnahmen ergriffen werden.

Zuordnung im Labor

Im Labor wird der QR-Code eingescannt und zum entsprechenden Auftrag zugeordnet. Anschließend wird die Befundung durchgeführt.

<https://bs-sd.de/unsere-softwareprodukte/>





INTRODUCTION

Writing many large game schedules by hand on paper is very time consuming. The manual process was therefore converted step by step via pseudo code to Pascal code, while looking for the essence in the original paper design. Due to a "happy" choice early in that process of converting the paper idea into code, it later surprisingly turned out that all the information needed to correctly print the schedule lines was stored in a simple number of 8 bits.

Recursivity was used as a programming technique for making, printing and cleaning.

The first version of this program was written in DOS Pascal in the 90s of the last century and as a demonstration of this article, it was tracked down to its modern versions: Lazarus 2.0.6 and Delphi 10.3.3. whereby with the code shown and the extensive explanation about it, the beginning programmer was especially kept in mind.

The new code uses Unicode chars to process the graphic line elements of the game schedules.

Under DOS you could write directly to the computer screen with Pascal. This is no longer permitted under Windows. If you want to do this directly, you will have to (re) write the original code into a console application.

In the Ftree program, a memo component was used to display the output lines directly on the screen. Another possibility to still display a knock out schedule within the program on the screen is the Canvas. As a demonstration, the print routine has been rewritten to send the output to the Main Form canvas instead of to a text file (see fig 33 a / b / c) and placed under a Button.

The explanation thereof falls outside the context of this article.

ABOUT THE GAME:

Badminton is a racquet sport played using racquets to hit a shuttlecock across a net.

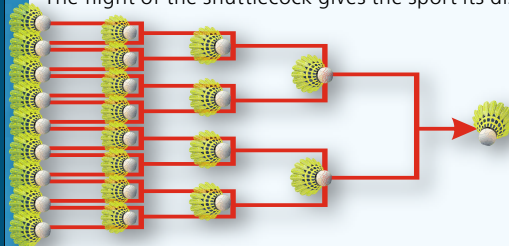
Although it may be played with larger teams, the most common forms of the game are **singles** (with one player per side) and **doubles** (with two players per side). **Badminton is often played as a casual outdoor activity in a yard or on a beach;**

formal games are played on a rectangular indoor court. Points are scored by striking the shuttlecock with the racquet and landing it within the opposing side's half of the court. Each side may only strike the shuttlecock once before it passes over the net. Play ends once the shuttlecock has struck the floor or if a fault has been called by the umpire, service judge, or (in their absence) the opposing side.

The shuttlecock is a feathered or (in informal matches) plastic projectile which flies differently from the balls used in many other sports.

In particular, the feathers create much higher drag, causing the shuttlecock to decelerate more rapidly. Shuttlecocks also have a high top speed compared to the balls in other racquet sports.

The flight of the shuttlecock gives the sport its distinctive nature



PURPOSE OF THE APPLICATION

There are two main goals to write this article:

First: Updating the program.

show how to update a program from a Turbo Pascal version up to the latest Delphi and Pascal programs and solving the possible errors.

Second: Update the printing of schemas

Adjust printing of schemas to modern pascal in as well Delphi and Lazarus, to create a possible better way of printing on a normal printer instead on canvas and documentation.

ITERATION STEPS

- 1 Starting the Tournament schedule, basic sketching.
- 2 From pseudocode to DOS Pascalcode
- 3 Binary numbers
- 4 Tree Index record definition
- 5 Pointers
- 6 Tree Object definition
- 7 Code testing solving problems
- 8 Code highlighths.
- 9 TTrNd Class definition
- 10 Binary bit manipulation
- 11 Difference of Value- and Variable parameter.
- 12 Exception Class Error
- 13 MakeTree
- 14 Recursion
- 15 Logfile
- 16 Tree print procedures
- 17 Ranking of participants
- 18 Printing with graphic unicode chars

Figure 1: Battledore and Shuttlecock: The popular and amusing game as at present played in the Principal Thoroughfares.

Source: john-leech-archive.org.uk (photo of copy taken by w:user: BozMo, who owns the site).





1 STARTING THE TOURNAMENT SCHEDULE, BASIC SKETCHING.

A long time ago - still in the DOS era - a club tournament was organized every year within my sports association (badminton). Over the years, the number of members grew so strongly that the tournament had to be divided over several evenings and the days during the two weekends in one week. In addition to the size, there was also a large level difference between youth, adults and veterans. It was therefore possible to register in 5 different game levels for single, mix and double, in which in principle the games were played in a knockout schedule.

For the higher levels the number of participants increased by the day and the registrations were numerous so that it could no longer be done in the normal pool-system.

In the initial period we wrote these schedules by hand. There was clearly a need for a form of automation here.

This article is about the designing process of converting a handmade paper reality into a software-based solution.

It is also about the surprises that showed hidden in a simple number and the challenges of transforming "old" software into modern.

It was written for the beginning programmer, so the code will be carefully explained.

In that at that time I had the early version (V3, another console version) of Borlands Turbo Pascal, that became my first choice for a development environment.

From version 4 onwards, it became a RAD-like development environment. We are still talking about **DOS Pascal** in the early 90s of the last century. Delphi 1.0 was not born yet. Three decades now: code starting early 1993 was used in the early development process.

And surprisingly apart from the print routines, still runs smoothly under Free Pascal 2019!

This code is then converted to a modern Free Pascal Class object whereby the old code (ideas) are reused as much as possible.

The new code was written in Lazarus 2.0.6 at the end of 2019 and Delphi Rio 10.3.3 executed. More than 25 years later!

KNOCKOUT TOURNAMENT SCHEDULE

We're going to talk about designing, building and displaying knockout tournament schedules. For those of you who don't really know what a knockout tournament schedule is, listed below is an example of such a schedule for up to 4 participants, schedule4:

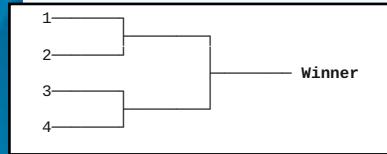


fig 1: waste tournament schedule4 for max 4 participants

Here participant 1 and 2 play against each other (a) and the winner then plays (c) against the winner of 3 and 4 (b) and that is where the finalist of the schedule comes from:

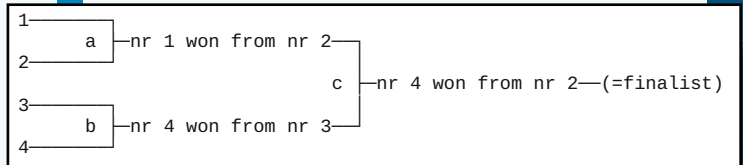


fig 2: finished schedule4

Now this is an example of a **scheme** with only four participants. Imagine that in the same section, say a women's single competition, 64 participants can participate. Then there must be 127 lines with player names filled in before this **scheme64** is completely played out and a winner is known.

There is a very simple calculation rule for calculating the total number of competition places in this **scheme64**:

Take the maximum number of starting places, which is now 64.

Multiply that by 2 (= 128) and reduce that by 1, giving $127 = (64 * 2) - 1$. You can check that with the **scheme4** above: $7 = (4 * 2) - 1$.

Now assume that they are doubles instead of singles. That gives 254 lines of paperwork. Then think of five playing strength levels, each with three different parts.

This all together creates an enormous amount of writing lines and then you need to remove writing errors and cancellations.

It is clear that at that time, as the "administrator" of the club tournament, I was in serious need of a solution, preferably generally applicable, for all schemes and that it should be possible to write it out via a printer instead of by hand.





The fundamental sketching can start. At that time I had no programming experience, so I had no idea where to start. After reading a book about fractals, I noticed, looking at the diagrams, that it looked like a kind of fractal: An ever-repeating pattern that in our case essentially looks like this:

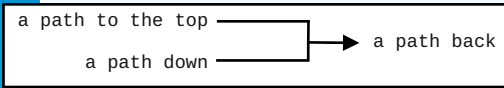


fig 3: the central three-way diagram

Whether you look at the entire **scheme** or just a single part of it: it keeps the same shape.

While reading, I came across the concept of tree-structures and the diagram was somewhat like such a tree structure.

It immediately became clear to me that the branches of this tree would be at the basis of my design. The **scheme** was given the appropriate name **Tree** and the branching in it the logical name **TreeNode**. I have an irresistible preference for very short abbreviations when writing my code so that these two names in the code were abbreviated to **Tr** and **TrNd**.

Tree structures consist of parts that are always connected in the same way. There are endless numbers of different tree structures. When designing code, our paperized play-schedules are leading in terms of structure in which the branches (the actual competitions) play a central role. It is therefore obvious to design the central basic idea **TreeNode** (**TrNd**) as first sketch.

TreeNode

It follows from the previous three-way diagram (Fig. 3) that the **TreeNode** has three connections: an Up, Down, and a Backward direction that has been named Root. All three can either point to one other **TreeNode** (branche) or have No branch at all.

The definition of **TreeNode** (**TrNd**) then gets an up (**Up**), down (**Dn**) and root (**Rt**) field that can each refer to a different **TreeNode**. In pseudo code, for example, it looks like this:

```

Type
TreeNode
Begin
    field Up ? TreeNode;
    field Down ? TreeNode;
    field Root ? TreeNode;
End
    
```

Code fragment 1: In Pseudo Code the three Tree directions

POINTERS:

How the above pseudo code is translated to Borland's TurboPascal is discussed here later, but first explain a construct that will be used for this: **Pascal Pointers** can be used for references to data structures.

Definition Pointers: pointers are data type structures that refer to the starting point of the object somewhere in the computer memory. We write that as: ^ pointer-name. (NOTE: note the circumflex ^ as a prefix).

The used **Pascal Pointer** definition to a **TreeNode**, with abbreviation **TrNd**, will look like this in code:

```

type
    PTrNd = ^TTrNd;
    
```

Why here the use of the capital P or T respectively as a prefix for the variable name **TrNd**?

INFO

You can write your code in many ways. Sometimes easy to read for others, but often that is not the case.

Think of my tendency to abbreviate names, something that is essentially a bad feature in this perspective. Everyone ultimately develops their own writing style. Quite soon, for better readability, so-called writing instructions, called convention rules, were created for others. According to such a "writing style" you put the capital T as a prefix for the name of a type definition:

type name = T name.

With pointers, the usual prefix is the uppercase letter P.

The code above reads as follows:

Pointer **type PTrNd** is a pointer (^) of (to) type **TTrNd**.

Another example of write convention rules is to indent the code on a new line under, for example, the reserved word **begin** and to jump back the same distance at the corresponding **end** (see fig 3).

It is then clear at a glance that the code between this start / end block belongs together. This write convention rule is also frequently followed here when writing the code. This indentation is also applied to other code that belongs together.

Such as the **If-then-else**, **While-do**, **Repeat-Until**, **Try-Finally-End**, etc. blocks.





But, by me, also with the const, type and var declarations. For clarity, there are several write convention rules about how you (for others and perhaps yourself) can write "readable" code. Back to our code.

Because these three fields belong together, they were put in a Pascal Record type structure. The basic **scheme** element **TreeNode** (**TrNd**) of record type **TTrNd** was created like this:

```

type
  PTrNd = ^TTrNd;

  TTrNd = Record
    fUp : PTrNd;
    fDn : PTrNd;
    fRt : PTrNd;
  End;
    
```

Code 2: the TreeNode record definition in 1993 pascal code

This provides a generally useful description of the **TreeNode** data structure. The **TreeNodes** as a whole then form the **scheme**.

If we want to be able to approach these nodes individually, for example to link a player name, match number or result, they must be given a sort of address so that we know which node is where in the **scheme**.

The next step in the **paper-to-code-development** process is therefore to record the unique position of each node in a **scheme**.

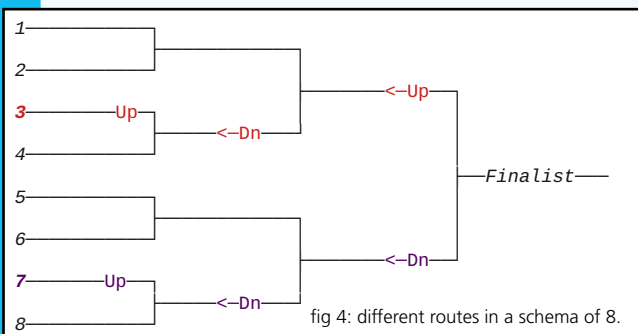
POSITION DETERMINATION

The only logical place to start the route to a specific node is at the finalist, at the far right of the **scheme**: because it is easiest to get anywhere in the schedule starting there.

If you start reading from the finalist (right) to the left you can only move in two directions: up or down / one step back (higher / deeper) into the schedule.

This also applies to every subsequent node.

If you put these "turns" one after another, you will get a sketch of route-description: eg **Up → Dn → Up** or **Dn → Dn → Up →** etc. In the (larger) **scheme8** (see fig. 4) it looks like this:



My first thought was to put hooks around this route description so that it could be put in an Array type.

You then get something like: type: **routearray = array [1 .. ?] Of type TRoute**, where **TRoute = Set of (Up, Dn, Rt)**.

But in the end I thought that was too cumbersome. In addition, the question then arose whether the construction would not cost too much memory space?

Something that was very important at the time. My computer had a "Large Memory" for that time: it had 1MB of memory (650k for the system and the rest defined as extended memory).

So it had to become simpler.

After looking at some other options I decided to simplify the words Up and Dn to the numbers 0 and 1 where the 0 symbolizes the up and the 1 the down.

If we now put the two series in the example above, one to the next in a larger **scheme64**, then that becomes the route to node no. 23: (See Figure 5 on the next page).

In my search for a suitable way to accommodate this series of ones and zeros, my eye fell on the binary representation of a number: a series of ones and zeros.

That is the way numbers are stored in the computer.

This is called the **Binary Number System**.

DEFINITION OF BINARY NUMBER

A binary numbers composed of two digits, 0 and 1. This base-2 system is the basis for digital systems. Smallest binary item, called a bit (binary digit). The binary number can be specified by preceding it with a percent sign (%).

Binary	Hexadecimal	Decimal
%0000000000000000	\$0000	0
%0000000000000001	\$0001	1
%0000000000000010	\$0002	2
%0000000000000011	\$0003	3
%0000000000000100	\$0004	4
%0000000000000101	\$0005	5
%0000000000000110	\$0006	6
%0000000000000111	\$0007	7
%0000000000001000	\$0008	8
%0000000000001001	\$0009	9
%0000000000001010	\$000a	10
%0000000000001011	\$000b	11

Table 1: Example of the first 11 binary numbers and their representatives



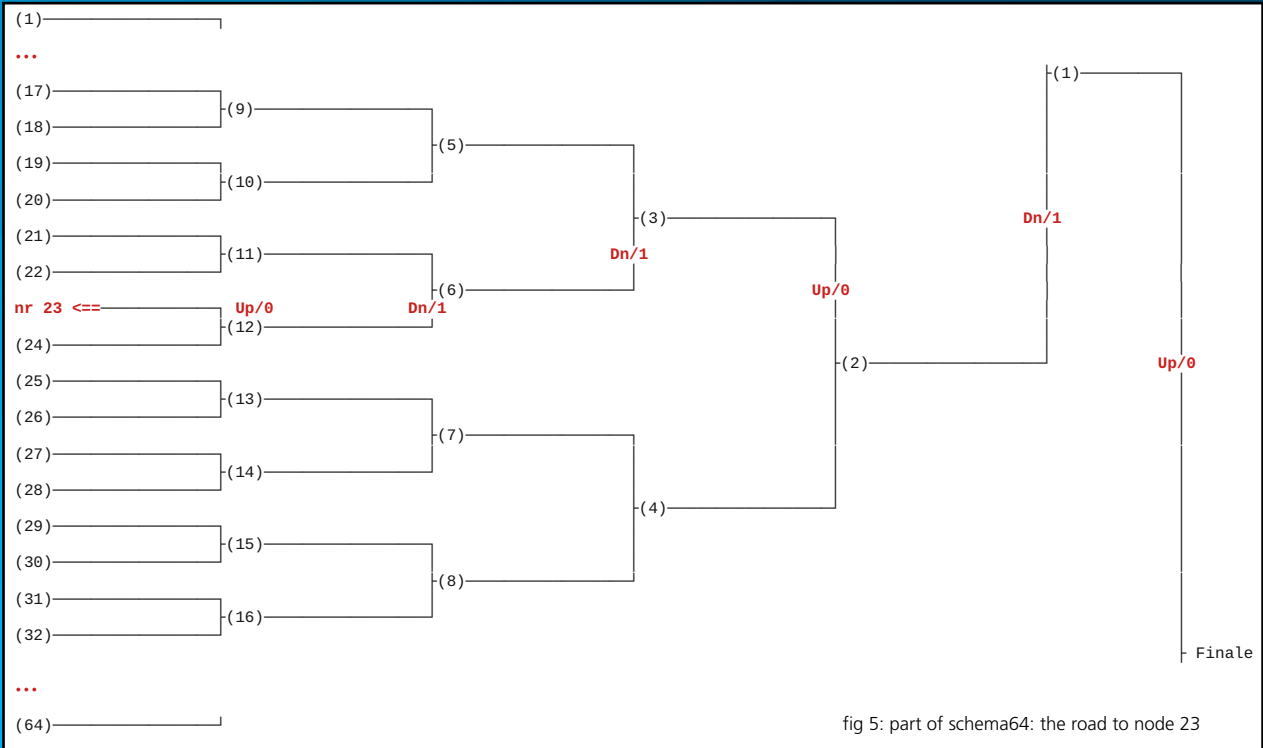


fig 5: part of schema64: the road to node 23

Our range of 0 . 1 . 0 . 1 . 1 . 0 . if it's a binary number, it looks like this: 010110 .
 If you look at the binary numbers from of table1, you will see that they consist of 16 "ones" and / or "zeros" there.

Eg % 0000000000001011 .

These 16 ones and zeros are the **word type** bits, we say that a word is 16 bits in size.

You can also see in the table that such a series of ones and zeros represents a decimal value.

For this series that is 11 .

Before we get started, we first need some extra knowledge: "how do you turn a binary number to a decimal number?" .

The positions of the bits in the series of a binary number by definition have the following fixed meaning.

1st bit stands for	20	= 1
2nd bit stands for	21	= 2
3rd bit stands for	22	= 4
4th bit stands for	23	= 8
5th bit stands for	24	= 16
6th bit stands for	25	= 32
7th bit stands for	26	= 64
8th bit stands for	27	= 128
9th bit stands for	28	= 256
10th bit stands for	29	= 512
11th bit stands for	210	= 1024
12th bit stands for	211	= 2048
13th bit stands for	212	= 4096
14th bit stands for	213	= 8192
15th bit stands for	214	= 16384
16th bit stands for	215	= 32768

Table 2: the value calculation of the individual bits in a binary number of type word

Added together is that 65,023, a known number, right, for a 16-bit system?
 Furthermore, it is very important here that you know that the series is read from the right ← left by default.

So from the example % 0000000 . 00001011 the zero next to the % char bit number is 16 and the rightmost is 1 bit number 1!

During reading of the bits, the number of the 2 power associated with that position is multiplied by the value of the bit at that position, a 0 or a 1. Then all is added.

The range % 0000000 . 00001011 then becomes: $1 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 + 0 \times 16$ divided by $0 \times 32768 = 1 + 2 + 8 = 11$.

If you now remove the last 8 zeros (the left series from the dot) from the series, this will remain:

% 00001011 . And still the decimal number 11!

If you then fill the first 8 bits with a 1: % 11111111, this represents decimal 255. The numbers 0 to 255 can therefore all be made with the first 8 bits and this has been given its own name.

This smaller member of the type is called a **byte** .

A byte is therefore 8 bits in size.





NOTE: the size / length of the **binary** number depends on the type: a **byte** as a **binary** number has **8 bits**, a **word** or **integer** has **16** and a **double** **32**, etc.

The smallest type in **Free Pascal** that fits a decimal value is the byte type. **8 bits** are required for one byte. If all eight bit locations for a route are used by a schedule, then up to a schedule **256** can be contained in one byte. That is more than sufficient. For the attentive reader: that is **256** while the binary number **11111111** stands for **decimal 255**. Yes, the **scheme** fits just fine but creates a small challenge elsewhere in the code. More about that later.

The binary number becomes the requested container for storing the 'ones' and 'zeros' of a route designation to a node somewhere in a **scheme**.

Back to the last route example:
Up, Dn, Up, Dn, Dn, Up.

This route is therefore converted to a binary number by placing it from the right ← towards left in the series: **011010**.

This equals the reading of the route in the diagram.

This allows the route of this schedule to be recorded in a variable of type **Byte (8 places)** or **Word (16 places)**.

Now suppose that we increase this **scheme64** two steps larger / deeper (64 → schema 256) and only left up, with the last 2 steps.

Then the new route looks like this: **00011010**.

And now we have a challenge, because from **011010** one Up shows **0011010** as a binary number and two Up shows **00011010** as a series.

But these three binary numbers all have the same decimal value 22.

The same decimal number for three different positions in a **scheme** is of course unworkable. So, which position in the diagram do we indicate with the number 22?

In fact, if we only move up from the finalist to the top left, all these 8 nodes have the decimal value 0! Because the route for all these 8 nodes is 00000000. That's rubbish, right?

The solution for this problem comes with the introduction of an additional parameter. In this parameter a number is added that indicates how far we should follow the specified route path from the **scheme**.

In other words:

how many bits must be read from the binary number to reach the desired **TreeNode**?

With the series **00011010**

lets say level **6 (011010 = 22)** and for level **7 (0011010 = 22)** the same decimal number **22** is used but then read out at **2 different levels** and therefore each with a **different binary series of bits, 6 to 7 bits**.

The unambiguous position of a treenode can now be determined by combining these two parameters.

In the first node the route is stored as a binary number of type **Byte** and the second is a parameter of type **Byte** which indicates how far you should go that "route" from the first parameter.

Below the examples of four different treenodes with the same decimal number 22

```
[5-00010110 = 22]
[6-00010110 = 22]
[7-00010110 = 22]
[8-00010110 = 22]
```

wherein the first number indicates the level in the game schedule.

The 2nd, the binary number, shows the route in the game schedule, while the decimal number always remains the same.

You can also express the length of the path in the number of times that you go to the next node or **how deep or far** you have to move in the Schema and has therefore been given the name **Depth (dpth: byte)** as a **variable**.

The path to be followed has been given the name "Position Code" (pscd: byte) as a variable.

The record type in which these two variables are included has been given the name **TreeIndex (Trndx)**. As well a pointer variable has been defined.

Finally, an extra number field (**nmcd: byte**) has been added to be able to assign a competition number. In a record code structure the index will look like this:

```
type
  PTrndx = ^TTrndx; { Pointer to tree-index }

  TTrndx = Record
    dpth: Byte;      { tree depth }
    pscd: Byte;     { tree position code, binary bitwise used }
    nmcd: Byte;     { node number code }
  End;
```

Code 3: The 1993 Tree Index record definition





The **TTrndx** record was then replaced by defining its three components as the Fields of the Object **TTrndx**. The new Index Object was housed in its own **unit STDNODE.pas**:

```
// Standard TreeNode-object definition
// developed on 20 februari 1993.
// last updates: 21 mei 1994 , 3 jan 1998

Unit STDNODE;

interface

type
PTrndx = ^TTrndx;
TTrndx = OBJECT { tree-index object type }
  dpth, { tree depth }
  pscd, { tree position code, bit wise oriented }
  nmcd : Byte; { node number code }

  constructor Init(Adpth, Apscd, Anmcd : Byte);
  destructor Done; Virtual;

  function Givedpth : Byte;
  function Givepscd : Byte;
  function Givenmcd : Byte;
  ... { ... = more code here }
END; { Objec }
... { ... = more code here }
```

Code 4: The 1993 Tree Index Object definition

The connection

To link the pointer of the new node-index object (**PTrNdx**) to a node of the type **TTrNd**, an additional field with the name **index (Ndx)** of type **PTrNdx** has been added to the **TTrNd** record. With this expansion, the definition of a **TreeNode** schema will look like this:

```
type
PTrNd = ^TTrNd;

TTrNd = Record
  Up : PTrNd; { pointer to another Up node }
  Dn : PTrNd; { pointer to another Dn node }
  Rt : PTrNd; { pointer to another Rt node }
  { New }
  Ndx : PTrndx { pointer to the corresponding index }
  { object, see code 4 }

End;
```

Code 5: The 1993 Tree Node record definition

A **TTrNd** record has now been created with only pointers and therefore as compact as possible.

THE TREE SCHEDULE

Now that the two basic elements, the Node record and its Index object, are ready, it's time to convert the paper playing schedule to code.

The **scheme** was given the name **Tree (Tr)** in code. A separate unit with the name **STDTREE.pas** was also created for this.

Here the **TrNd** record got its place next to the new tree Object **TTr** definition and via its pointer **PTrNd** was linked to the Field **Nds** of the new **TTr** Object. The **STDTREE.pas** unit:

```
// Standard Tree-object definition
// developed on 20 februari 1993.
// last update 20 mei 1994.
Unit STDTREE;

interface
uses
  STD_CTV, { standard Const, Type and Var defintions }
  StdNode; { node/index definitions }

type
PTrnd = ^TTrnd; { tree-node pointer }
TTrnd = RECORD { tree-node record type }
  rt, { Root ptr }
  up, { Up ptr }
  dn : PTrnd; { Down ptr }
  ndx : PTrndx { treeindex ptr, unit stdnode.pas }
END;

PTr = ^ TTr; { tree Object pointer }
TTr = OBJECT { tree Object type }
  Nds :PTrnd; { nodes field }

  constructor Init;
  destructor Done; Virtual;

  procedure Add(nwndx: PTrndx);
  procedure MakeTree(ATrdpth: Byte);
  procedure PrintTree;
  ... { ... = more code }
END; { Objec }
```

Code 6: The 1993 Tree Object definition

OLD CODE TESTING

To test whether this 1993 code still works, these two original **DOS Pascal** units were inserted in a small Lazarus 2.0.6 program (**Ftree.exe**).

By means of the **Tree.MakeTree (4)** procedure, this old code still proved to produce a play schedule with depth 4 (= max. 16 participants) without any problems.

The **scheme** is displayed on screen in a memo component and written to a text file. The number of created nodes is displayed in a label. Exactly the same program is also executed under **Delphi Rio 10.3.3**.

Now you can drop the components **TOpenDialog** and **TSaveDialog** on your form and always use it to retrieve the required output file from a directory somewhere (**TOpenDialog**) and save it again (**TSaveDialog**),





but you have to repeat that process manually each time.

That is a lot of "unnecessary" work if, during development, you still want to save the same file to the same dir. That is why the opening and saving of the output file is included in the code, so-called "hard coded".

Here the `btnMake4` code from the **Lazarus Ftree** program that controls the print output:

```

unit FTree;

{$mode objfpc}{$SH+}

interface

uses Windows, Classes, SysUtils, Forms, Controls,
    Graphics, Dialogs, StdCtrls, lazUTF8;
...
var Form1: TForm1;

implementation

uses STDNODE, STDTREE; { the old 1993 DOS pascal units }

{$R *.lfm}

{ TForm1 }
...
procedure TForm1.btnMake4Click(Sender: TObject);
var tree: Ttr; MyFile: String;
begin
...
MyFile := FappPath+'Output\Print-Objecttree.txt';
AssignFile(LST, MyFile);
with tree do
begin
try
    Rewrite(LST);
    { tree. }init;
    { tree. }maketree(4);
    { claim extra memory for the pointers }
    form1.label2.Caption:= IntToStr(CntNds);
    { tree. }PrintTree;
    { write to textfile LST and to Memo.lines.Append }
finally
    { tree. } done;
    { free the extra memory pointers }
    CloseFile(LST);
end; { try }
end; { with }
end; { procedure }
...
end.
    
```

Code 7: The make4 code of the Lazarus Ftree program

In the **Lazarus** and **Delphi** program the output lines are written both to the memo component on the main form1 itself (see fig 6a / 7a) and to an external text file (see fig 6b / 7b).

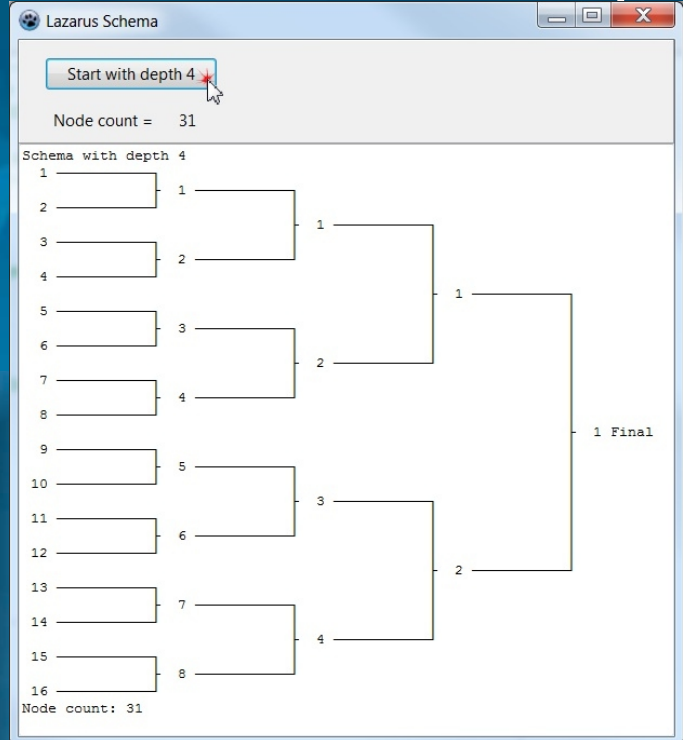


fig 6a: The Lazarus 2.0.6 Ftree program with memo



fig 6b: schema 4 textfile output: 1993 code running in 2020 Lazarus Free Pascal





The same program running in Delphi Rio 10.3.3

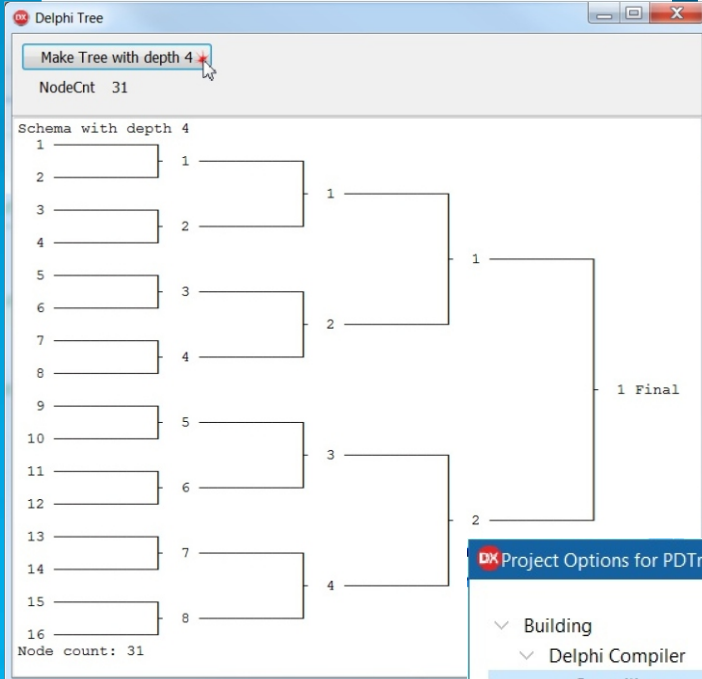


fig 7a: The Delphi Rio 10.3.3 Free program with memo

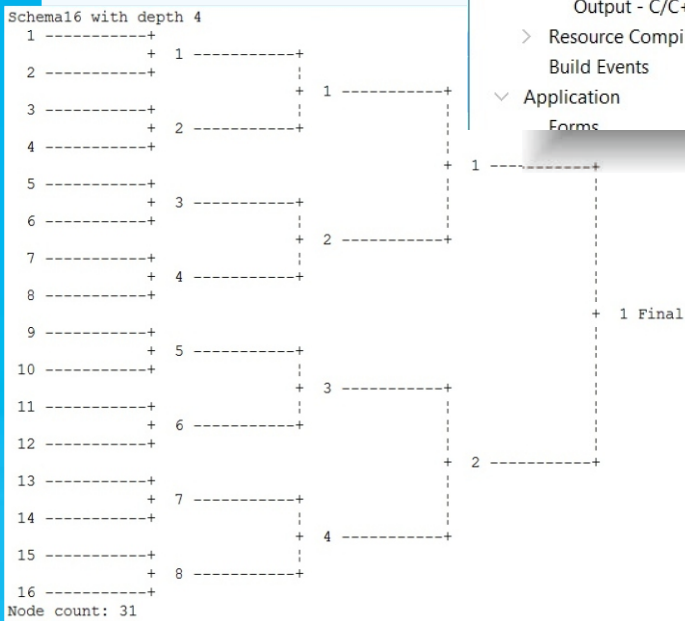


fig 7b: schema 4 textfile output: 1993 code running in 2020 Delphi

If you now look at figures 6b and 7b you will see a clear difference in output between Lazarus and Delphi. Lazarus prints the Unicode chars directly to the text file while Delphi only shows half of our unicode widechars in the text file.

Again: the exact same code runs here in both Lazarus and Delphi. And yet the output result to a text file is surprisingly different.

After some research on the internet, the solution in specifying code page 65001 was found in the **Project Options → Building → Delphi Compiler → Compiling: Code page** (see fig. 8).

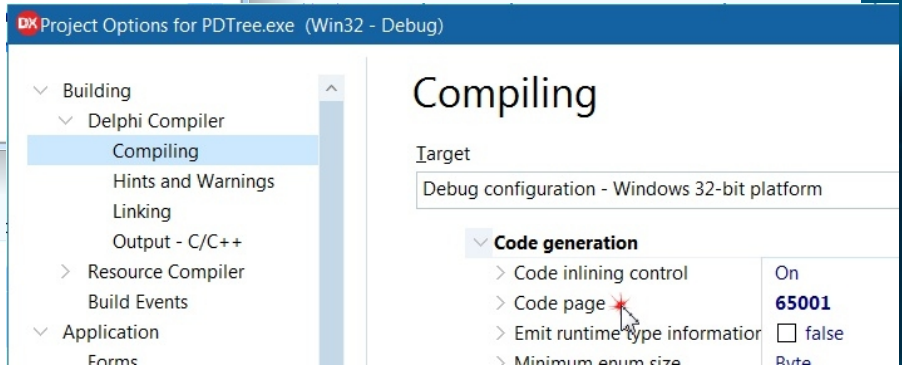


fig 8: Delphi options add codepage 65001

Unfortunately that turned out not to work. Finally, after much searching, the solution was to add, in **Delphi** and not in **Lazarus**, the same **codepage 65001** as a parameter to the **AssignFile()** command:

```
assignFile (LST, MyFile, 65001) ;
```

The **Assignfile()** function has an optional parameter, which can be seen from the use of the square brackets around one or more parameters [**Aparameter; ...**] in the function definition.

The **Delphi Help** provides the following definition:

```
Def System.AssignFile:
function AssignFile(var F: file; Filename: String; [Codepage: word ]): Integer; Overload;
```





When in **Delphi** the output is now sent to the text file again, this produced the desired result (See fig. 9).

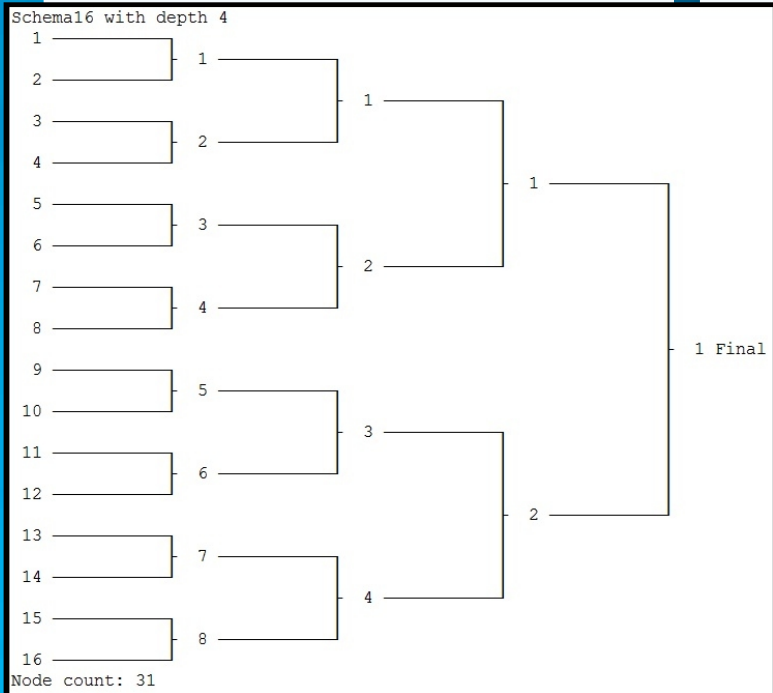


fig 9: The correct Delphi schema 4 textfile output with the use of codetable 65001

With **Lazarus** you need the unit `lazUTF8` to work in code with unicode chars, but since 2007 **Delphi** is already internally completely ready for the use of unicode and therefore does not need an extra unit.

Why the above difference between **Delphi** and **Lazarus** still exists I have not found anywhere in any documentation. Maybe one of our readers knows the answer to this? But you can understand that I was just unpleasantly surprised.

CODE HIGHLIGHTS

We will take a closer look at the code of the **Ftree program** (see Code 7). The Main Form1 of Ftree has a public field `FAppPath: string`

```
public
  FAppPath : String;
```

where, in the `TForm1.FormCreate` procedure the path to the executing program is stored. In 1993 the path of the executable was assigned to the variable `exepath` of type `String` in the following way:

```
exepath := GiveEXEPath; // Path by Dos Pascal
```

where `GiveEXEPath` is a function that retrieves the path to the executable:

```
FUNCTION GiveEXEPath: string;
var chrps, { char position }
    hldchrps : byte; { hold char position }
    Exepad : string; { path to executable program }
BEGIN
  //Exepad gets the full path name (.exe inclusive)
  // to the running program
  Exepad := paramstr(0);
  // search for the location of the last \ in the path string
  chrps := Pos('\',Exepad);
  hldchrps := 0;
  Repeat
    hldchrps := hldchrps+chrps;
  { keeps track of \ in the path string }
  Exepad := copy(Exepad,(chrps+1),255);
  { copy the path from the last found \ }
  chrps := Pos('\',Exepad);
  { locat next \ }
  Until chrps = 0;
  // Give the Exepad the string including the last
  // found \. So excluding the executable name.exe *)
  if hldchrps <> 0
  then Exepad :=
    Copy(paramstr(0),1,hldchrps)
  else Exepad := "";
  GiveEXEPath := Exepad;
END;{ GiveEXEPath }
```

Code 8: The old GiveEXEPath code





Lazarus and Delphi make it a lot easier for you. The following code returns the same path to the `Form1.FAppPath` field:

```
FAppPath := ExtractFilePath(Application.ExeName);
// Path by intern Lazarus/Delphi functions
```

It's easy to guess how that works. Code 8 shows what it takes. And because the path is already known when creating a program, let's get it directly in the procedure `TForm1.FormCreate(Sender: TObject)`.

INFO:

You create this procedure by selecting the form in the IDE, in the Object Inspector on the Events tab and then double-clicking on the entry field after the `OnCreate` event.

Lazarus will then create the `FormCreate()` procedures for you:

both the procedure heading in the Class object of the form and the procedure in the Implementation section and immediately places the cursor there after the first start, so that you can immediately enter the desired code. (See Fig. 10).

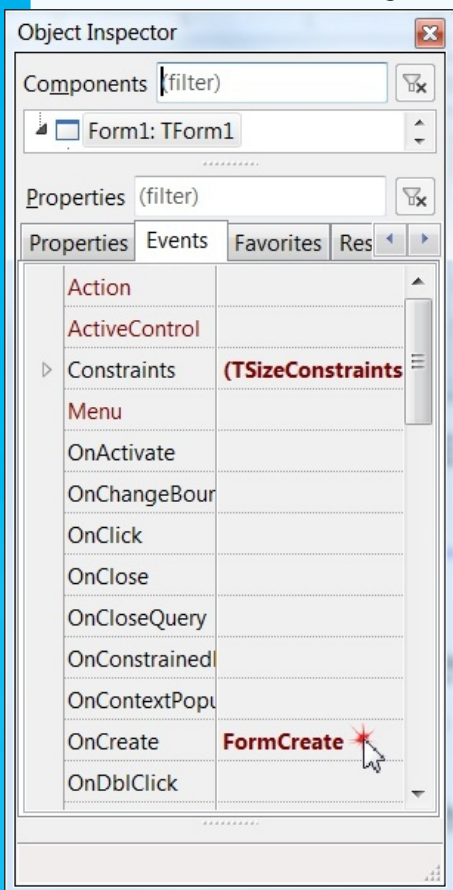


fig 10: Double click in the form1 Object Inspector near the event OnCreate.

The code in the generated `OnCreate` event: `FormCreate()`

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    FAppPath := ExtractFilePath(Application.ExeName);
end;
```

stores the requested path in the variable `FAppPath`. This path is stored together with the name of the output file in the variable `MyFile`: `MyFile := FAppPath+'Output\Print-Objecttree.txt'`;

To link the Textfile variable `LST` of type Text, to the output file (`MyFile`), the internal routine `AssignFile()` is used: `AssignFile(LST, MyFile)`. Or `AssignFile(LST, MyFile, 65001)` at Delphi. The output file will then be placed in the directory Output of the dir `FAppPath`.

Because you want to write text here to a clean empty file, first set the text file to receive text by calling the internal `Rewrite(LST)` procedure before you can send text to it.

It does not matter for `Rewrite(LST)` whether or not the specified file already exists. If it does not exist, it first creates it before placing the file cursor at the beginning of the file. If the file does exist, it will first be cleaned before the cursor is placed at the beginning of the file. After this you can write text (lines) to the file with `Write(LST, 'text')` or `Writeln(LST, text line)`.

The code of `TForm1.btnMake4Click(Sender: TObject)` contains a special code Block (see code 7). You will see the following code construction around the code discussed above:

```
try
...
finally
...
end; { try }
```

This is a widely used construction that ensures when executing the code between the reserved words `try` and `finally` something would go wrong, causing that (wrong) code to cause the program to crash, the code between `finally` and `end` is still executed before the execution of the program is forced to stop. In short: the code between `finally` and `end` is always executed if this try construction is invoked.





In our procedure that is important for two reasons:

- Firstly, it must be ensured that the extra memory requested by us (for the node pointers) is always released before the execution of the program is stopped or interrupted. Otherwise that memory would always be occupied for other programs. The latter is also called **memory leaking**.
- Second, the text output file must be closed so that all information is actually written to it before the program closes. This prevents any information loss with the text file.

If you want to read its contents instead of writing to the file, you must first use the internal **Reset (var file: textfile)** routine that sets the file to readonly.

With the **readln (LST, ALine: string)** you can then read a file line.

However, for **Reset (LST)** it now matters whether the file exists.

If you are following an **AssignFile (LST, '...')**; causes a **Reset (LST)** while the requested file does not exist you will be presented with a **runtime I / O Error 103: "File not open"**.



fig 11: I/O error 103: File not open error

INFO:

Reported by **CloseFile, Read Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, or BlockWrite** if the file is not open. (Input-Output Errors From RAD Studio).

```

procedure TForm1.btnMake4Click(Sender: TObject);
var tree : Ttr; MyFile : String;
    // extra variable
    MyHandle : THandle;
begin
    MyFile := FappPath+'Output\Print-Objecttree.txt';
    // New code
    if NOT FileExists(MyFile)
    then
    Begin
        MyHandle := FileCreate(MyFile);
    ...
    End;

    assignFile(LST, MyFile, 65001);
    with tree do
    begin
        try
        ...
    End;
    
```

However, if you now run the **FTree** program, as shown above, you will receive the following error message from the **Lazarus Debugger**:

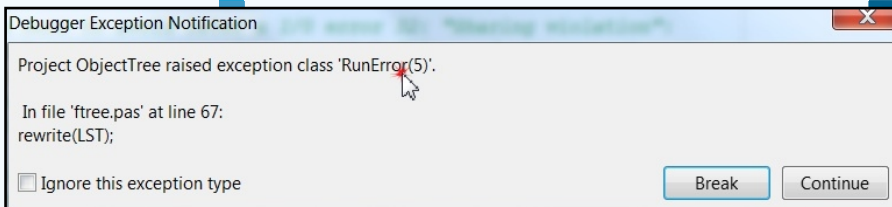


fig 12: Lazarus I/O Error 5 : ERROR_ACCESS_DENIED

INFO

Lazarus Run-time error 5: "permission to access the file is denied". This error might be caused by one of several reasons:

- Trying to open for writing a file which is read-only, or which is actually a directory.
- File is currently locked or used by another process.
- Trying to create a new file, or directory while a file or directory of the same name already exists.
- Trying to read from a file which was opened in write-only mode.
- Trying to write from a file which was opened in read-only mode.
- Trying to remove a directory or file while it is not possible.
- No permission to access the file or directory.





And the Delphi 10.3.3 Debugger raises this exception:

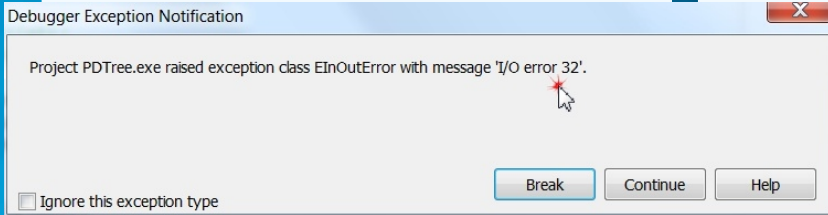


fig 13: Delphi I/O Error 32: ERROR_SHARING_VIOLATION

This means that the process cannot access the file because it is being used by another process.

Both **Lazarus** and **Delphi** indicate that you are accessing a file that is being used by another process. You know that: if you want to open a text document that is still being used by another word processor, you will be denied access to it. In our code above, exactly the same thing happens and the error message follows.

But why is it causing a problem here?

The **FileCreate** function (**MyFile**) creates the MyFile file and returns the file handle to the MyHandle variable before closing. If we then try to write text to this file with, for example, a **writeln** ('text'), we get the error message: "MyFile is being used by another process".

you must manually release the file Handle before another (sub) process can gain access to this file.

I expected that the newly created file would be immediately available but, despite the **FileCreate** () function being closed and the file-Handle being offered as a function result. The run-time errors show that the **FileCreate** () function in this file-handle has not been released! I do not know whether this was done deliberately or if it is a Bug, but it does mean that **you must manually release the file Handle before another (sub) process can gain access to this file**. In addition, it is now very handy that you have already received the correct file Handle: MyHandle. The file-Handle is released with the internal procedure **CloseHandle** (var Ahandle: THandle). In code:

```
// New
if NOT FileExists(MyFile) then
Begin MyHandle := FileCreate(MyFile);
(* NOTE:
Close now MyHandle here to avoid later a I/O error 32:
"Sharing violation": "that means that another process is using
that file, and you can not save your changes to the same file
until that process is done." So free the handle with this file: *)

// New
CloseHandle(MyHandle);
End;
...
```

Or shorter but less legible, if you want to avoid the extra variable MyHandle:

```
CloseHandle (FileCreate (MyFile));
```

Now you can use the following routines with this text file without any problems:

```
var
  ALine : String;
  // For reading lines from this text file:
  reset(LST);
  // opens de file for reading from
  readln(lst,ALine);
  // read a line and put it in Aline

  // For writing to this text file:
  rewrite(LST); // opens de file for writing
  write(LST,'piece of text');
  // writing text and stay on the line
  writeln(LST,'text line');
  // writing a line and go to the next line
  writeln(LST);
  // go straight away to the next line
  // when you are done close the file (this frees
  // the file handle automaticly);
  CloseFile(LST);
```

But most of the above code dates from the early 90s of the last century.

PASCAL IN THE YEAR 2020

The Pascal programming language has undergone a major transformation in the last 25 years. From Dos Borland Pascal via Windows Borland Pascal to Delphi or Lazarus IDE. In addition, the computers and memories have become much more powerful and larger.

In the above code from 1993 the tree is defined as **TTr = OBJECT**. This OBJECT is something very different from the current **TObject**.

It was a record type extended with functions and procedures that was stored on the stack. Nowadays the stack can be very large, but at the time it was quite limited, while a very large **scheme** requires many of these objects and the stack quickly got filled up. As a result, a so-called stack overflow error was always lurking.

In contemporary Pascal we use a Class type instead. This **class** is of type **TObject** and is not stored in the stack or in the heap but only once in the executable code block of the program.





And from there again and again when needed. Converting the **DOS Pascal Object** type to a **Class (TObject)** type was slightly more difficult than I initially thought.

In the 1993 code, a new node pointer was created in the **Add()** procedure using the **New (nwnd)** procedure. Then the object fields were immediately filled with a value: (See code right →)

Running this code as part of an **OBJECT** instance in Lazarus did not cause any problems as shown above (See figs 6/7). But executing this code as part of a **Class (TObject)** produced the following message in the message window of the IDE:

```

type
  PTrNd = ^TTrNd; { TTrNd }

PROCEDURE TTr.Add(nwndx: PTrNd);
  var nwnd: PTrNd;

BEGIN
  New(nwnd); create new Tree Node pointer
  with nwnd^ do { with that new Tree Node do}
  BEGIN
    ndx := nwndx;
    up := NIL;
    dn := NIL;
    rt := Nds;
    Nds := nwnd;
  END;
END; { TTr.Add }
    
```

Warning: use extended syntax of NEW and DISPOSE for instances of objects

Note: Local variable "nwnd" is assigned but never used

Warning: use extended syntax of NEW and DISPOSE for instances of objects

If you have a pointer a to an object type, then the statement new(a) will not initialize the object (i.e. the constructor isn't called), although space will be allocated. You should issue the new(a,init) statement. This will allocate space, and call the constructor of the object.

fig 14: Compiler warning.

The constructor of the new class is called **Create** instead of **init**. My code changed and re-executed,

```

BEGIN
  New(nwnd, Create(Adpth, Apscd, Anmcd));
    
```

again posted a message in the message window, this time with fatal consequences:

Code converting

The old code was therefore adjusted when transferring to a modern Class type. And because we were put to work, the **TTrndx** and **TTrNd** records were both merged as fields in the new Class definition **TTrNd**. As a result, the old **Ndx** field has been canceled. The more modern 2020 looks (next page:→)

...Compile Project, Target: clasrtd.exe; Exit code 1, Errors: 2, Warnings: 8, Hints: 1

Warning: Implicit string type conversion from "AnsiString" to "WideString"

Warning: Implicit string type conversion from "AnsiString" to "WideString"

Error: The extended syntax of new or dispose isn't allowed for a class

Fatal: Syntax error ")" expected but "" found

Error: The extended syntax of new or dispose isn't allowed for a class

You cannot generate an instance of a class with the extended syntax of new. The constructor must be used for that. For the same reason, you cannot call dispose to de-allocate an instance of a class, the destructor must be used for that.

fig 15: Fatal Error

The program cannot be compiled like this! The texts in both message windows are crystal clear: You cannot use the procedure **new()** / **dispose()** when working with Classes. The Class **Constructor Create** and Class **Destructor Destroy** are designed for this.





```
TTrNd = class (TObject)
private
  Fdpth: Byte; { tree depth tree branches}
  Fpscd: Byte; { tree position code, bit wise oriented }
  Fnmcd: Byte; { tree number code }
  Fup : TTrNd; { forwards tree-up link}
  Fdn : TTrNd; { forwards tree-dn link}
  Frt : TTrNd; { backwards tree-root link}

public
  constructor Create(Adpth, Apscd, Anmcd : Byte);
  destructor Destroy; virtual;
...
end; { TTrNd = Class() }
```

Code 9: The new TTrNd Class definition

What catches the eye here is that the pointer prefixes (^) and the pointer variables have disappeared.

Note that it now seems as if there are no pointers in use, but in reality these are used extensively by the IDE in the background.

Figure 16 below shows a screenshot of the include file in which the basic definitions of the TObject and TClass types are given.

Pay special attention to the pointer variable Pclass:

```
70 type
71   TextFile = Text;
72
73   { now the let's declare the base classes for the class object
74   model. The compiler expects TObject and IUnknown to be defined
75   first as forward classes }
76   TObject = class;
77   IUnknown = interface;
78
79   TClass = class of tobject;
80   PClass = ^tclass;
```

fig 16: Free Pascal Object and Class definitions

Hence the up / dn / rt fields in the new code are of the Type TTrNd instead of the Pointer PTrNd type. To indicate that the data fields of the Class are, again according to a naming convention rule, the F of Field is said: Fup, Fdn and Frt. Similarly with: Fdpth, Fpscd and Fnmcd.

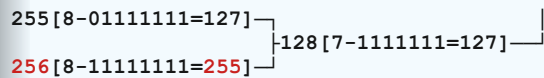
When converting the old DOS code to the new Class, another bug came over you. For that we must return to the relationship of schema256 with node (255) that was mentioned at the beginning of this article. (page 6).

As so often, the boundaries of our ideas throw a spanner in the works during the development process and so do my ideas about the code being developed here.

Because if a **scheme** is made in which the entire binary number series of a byte is used, then the

scheme must have a depth of 8.

If you now look at the lower part of such a very large **scheme256** in which a maximum of 256 players can participate, the lower treenode has a binary position code of 11111111. That stands for decimal 255:



In the program the number of nodes is recorded with the variable ndcnt (node counter) of type Byte. No problem until **scheme256** is going to be printed. A byte type is not sufficient here: it lacks one place number.

That is, not for the schedule itself, but for displaying the number (ndcnt) of the last node.

Because that is node nr 256.

If we take the second-last node of **scheme256**, then that **ndcnt** has 255 as the place value.

The last node must have the place number 256 but will receive the value 0 at **Inc (ndcnt)** instead!

	MyWord	MyByte
250 + 1 =	251	251
250 + 2 =	252	252
250 + 3 =	253	253
250 + 4 =	254	254
250 + 5 =	255	255
250 + 6 =	256	0
250 + 7 =	257	1
250 + 8 =	258	2
250 + 9 =	259	3
250 + 10 =	260	4

fig 17: Byte overloading





Info Inc

Inc (Ordinal) is an internal procedure that increases the ordinal entered by 1 value. You can also increase the ordinal by more than one value at once by using the lookalike **Inc (Aordinal, Acnt)** for this.

The variable in which the number of nodes is stored must therefore be of type **Word** to be able to display the number 256, while in the old code it was of type **Byte**.

Oops a borderline case.

In short, it is always wise to look at the possible limit values of your variables before you 'just' choose a type, as I apparently did at the time.

As a small demonstration of this the following program Bite overloading (See fig 17).

The variables **MyByte: Byte** and **MyWord: Word** are both assigned the number 250.

For both variables, a value of 1 is added 10 times and the result for both variables is shown behind it. The value 250 + 6 for the **Word** 256 type but for the **Byte** type a 0!

So a byte type for the variable **pscd** is just sufficiently large here to record the route in, but insufficiently large to display the decimal number of participant 256.

In order to still be able to display the value 256, the variable from type byte has become a type: 16 bits instead of 8.

For the 1993 code, this boundary problem naturally also applies. Only it was not relevant there because a schedule of 256 participants never occurs and this limit value was therefore never achieved.

The current program is limited to a depth of 8. **scheme256** gives 256 participants a place and that is very extreme and actually nonsense. However, if you want to print even larger schedules, you must change the type of **pscd** and **nmcd** from byte to word. This does not matter for the speed or storage on the computer.

Just as with the "old" Object type, corresponding functions and procedures can be included as methods in the class. Again a Class type is of a different order than the old **DOS Pascal Object** type, so not to be confused with **Tobject**, which is really something very different.

The Class (TObject)

In a Class type definition there are different zones, called sections, that determine the access of (external) code to the code defined there. E.g. private, public, published section.

The fields of **TTrNd** are in the Private section. This means that they are only accessible in the Interface section of this Class definition and not outside of it.

The **Public** section is accessible for every code in the final program. In order to gain access to the fields in the **Private** section, we use functions and procedures that have to be defined in this Public section in order to be accessible anywhere in the program.

A Class definition also requires the use of two special methods, each with its own fixed name: a constructor and a destructor.

According to the convention rules, they have been named **Create** and **Destroy**, you can give them any name, although that does not improve readability for others.

In short: the **Create** constructor ensures, among other things, that **all memory required for an instance of this class is reserved** and the **Destroy destructor automatically ensures that all the reserved memory is also released afterwards.**

In the constructor, however, you can also directly reserve memory for all your own instances.

But then you must also neatly release all this extra memory in the destructor yourself.

The rule is: everything that you request here (constructor) must be cleaned up here (destructor) yourself.

The fields of the Class **TTrNd** are therefore placed in the **Private** section so that they are unreachable outside of this class definition. To provide them with a value (**Set... .**) Or to get the value out (**Get... .**).

Methods have been placed in the **Public** section. Here the prefixes "Set" and "Get" follow convention rules, just like the prefix "A" for the parameters in the methods below declarations. The following methods are needed to read the fields in the private section or to provide them with a new value: (next page)





```
TTrNd = class (TObject)
...
public
function GetFdpth : Byte;
function GetFpscd : Byte;
function GetFnmcd : Byte;
function GetFup : TTrNd;
function GetFdn : TTrNd;
function GetFrt : TTrNd;

procedure SetFdpth(Adpth: Byte);
procedure SetFpscd(Apscd: Byte);
procedure SetFnmcd(Anmcd: Byte);
procedure SetFup (ANode: TTrNd);
procedure SetFdn (ANode: TTrNd);
procedure SetFrt (ANode: TTrNd);

function NdInfoStr:String;

end; { TTrNd = Class(TObject)}
```

Here is an example of the simple Get and Set code for the Fdpth field:

```
(**-- Get -----**)
FUNCTION TTrNd.GetFdpth: Byte;
BEGIN
    result := Fdpth;
END;
(**-- Set -----**)
PROCEDURE TTrNd.SetFdpth(Adpth: Byte);
BEGIN
    Fdpth := Adpth;
END;
(**-----**)
```

Finally, in the TTrNd code example, you will see the NdInfoStr function. The name stands for Node-Information-String, which for now speaks for itself. With this we have all methods to provide the t T codes with the correct information. How we generate that information is discussed in the next paragraph.

GATHERING INFORMATION

BINARY BIT MANIPULATION

Back to the schedule route information processing and storage thereof in a binary number. In order to be able to store the route to a specific node in a binary number, the individual bits thereof must be accessible. To manipulate the individual bits in a binary number, Free Pascal provides the following Operators bit:

DEFINITION

The Bitwise operators supported by Pascal are listed in the following table. Assume variable A holds 60 (60 = 0011 1100) and variable B holds 13 (13 = 0000 1101), then :

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61, which is 0011 1101
!	Binary OR Operator copies a bit if it exists in either operand. Its same as operator.	(A ! B) will give 61, which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111





Please note that different implementations of Pascal differ in bitwise operators. Free Pascal, the compiler we used here, however, supports the following bitwise operators:

Operators	Operations
<code>not</code>	Bitwise NOT
<code>and</code>	Bitwise AND
<code>or</code>	Bitwise OR
<code>xor</code>	Bitwise exclusive OR
<code>shl</code>	Bitwise shift left
<code>shr</code>	Bitwise shift right
<code><<</code>	Bitwise shift left
<code>>></code>	Bitwise shift right

Table 3: Free Pascal - Bit Operators

Just like with the binary numbers, we will look at the diagram from right to left. The starting point is the finalist of, for example, the **scheme4** with 4 participants. This node gets a zero as position code (`pscd`) value, zero as a depth (`dpth`) value and `nr1` as a count value. From the position of the finalist you can go one level to the left, to the semi-finals: one up (`0000 0000`) `nr 2` and one down (`0000 0001`) `nr 3` in the schedule. From the up treenode (no. 2) you can also have one up (`0000 0000`) no. 4 and one down (`0000 0010`) no. 6 and one down (`0000 0011`) `nr 7`. In the `schem4` it looks like this:

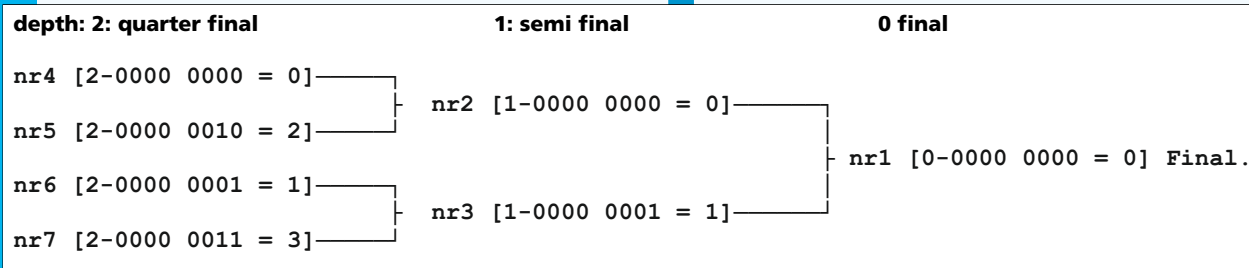


fig 18: Tree schema with nr, depth, binary number and its decimale value

Because we move from right to left in our **scheme**, we also want to be able to manipulate the individual bits of the binary number from right to left. The `Shl` (pronounce Shift left) operator from the operator list above is suitable for manipulating individual bits of a binary number from right to left, assuming 0 (`0000 0000`) as the starting value. What does `Shl` do?

So with the `shl` function you can reach any bit position in a binary number. Only here does the whole series of bits in the binary number shift the number of places specified to the left.

As a result, the route through the schedule, which is stored in the binary number, is also immediately invalid.

That is of course not the intention: we only want to manipulate one bit in one specific place in the binary number.

To make that possible, we are going to make a 2-stage "missile" that uses an auxiliary byte and yet another bit operator.

For the auxiliary byte, we take `bt: = $ 01`. That has a binary byte sequence of `00000001`.

We define the following calculation rule: `bt: = bt SHL (Pos-1)`

Note: Pos minus 1 because position 1 of the auxiliary byte `bt` already has a 1, then we can put the 1 in the rightmost position using the value in `Pos` on each of the other 7 bit positions. That's staircase.

The second stage is: How do we get that 1 in position (`Pos-1`) at that same position in the relevant position code (`pscd`) without changing the position of the other bits?

For this we use another **Bit Operator**: the `OR` operator.

DEFINITION:

`ShlShift left (shl)` performs a **leftbit-shift** operation, shifting the value `byte` the amount of `bits` specified as an argument (opposite of `shr`: Shift Right).

E.g

Command is: `00000100 shl 2` (shift left 2 bits)

Action is: `00000100 <- 00` (00 gets added to the right of the value; left 00 "disappears")

Result is: `00010000`





Definition: Or

The reserved word `or` is a binary operator. Originally it stood for the logical disjunction of two boolean values only, but with the advent of operator overloading FPC allows everything else, too. FPC also defines the `or` operator accepting two ordinal types while performing calculations on their internal binary representation. bitwise Or operation
 Since virtually all instruction sets have an `or` instruction, it is no surprise some high-level languages, especially those which aim to be suitable for hardware programming, provide some comparable functionality by itself. In FPC the `or` operator is defined appropriately. Such an expression, also known as bitwise `or`, requires two ordinal operands. The operation virtually performs a logical `or` taking each corresponding bit from both operands.

```

0101'1010
or 0000'1011
= 0101'1011
    
```

So the bitwise `or` places all bits that are 1 of the ordinal to his right in the same bit position as the ordinal to his left. It is therefore important to keep the correct order: "left of or gets from right of or".

The following function puts a bit at position `pos` in the number `item` by combining these two operands:

```

// SetBit returns the bit 1 at the specified location (pos)
// in the specified variable (Item).                               1993

FUNCTION SetBit(pos, item: byte): Byte;
  var bt : Byte;
BEGIN
  bt := $01; { bt ==> 0000 0001 }
  bt := bt SHL (Pos-1);
  { bt ==> 0001 0000 as pos = 5: ==> put 4 zeros in front of it }
  Result := item or bt;
  { Result ==> ???1 ???? depending on the content of item }
END;
    
```

In this function, the result is returned via the **Result identifier** instead of the function name `SetBit`. What is the internal identifier `Result`?

DEFINITION: RESULT
14.3 Function results

The result of a function can be set by setting the result variable: this can be the function identifier or, (only in ObjFPC or Delphi mode) the special `Result` identifier:

```

Function MyFunction : Integer;
begin
  MyFunction:=12; // Return 12
end;
    
```

In Delphi or ObjFPC mode, the above can also be coded as:

```

Function MyFunction : Integer;
begin
  Result:=12; // Return 12
end;
    
```

As an extension to Delphi syntax, the ObjFPC mode also supports a special extension of the `Exit` procedure:

```

Function MyFunction : Integer;
begin
  Exit(12);
end;
    
```

The `Exit` call sets the result of the function and jumps to the final `End` of the function declaration block. It can be seen as the equivalent of the `C` `return` instruction.

REMARK: Function results are treated as pass-by-reference parameters. That is especially important for managed types: The function result may be non-nil on entry, and set to a valid instance of the type.

In a similar way you can also remove a bit (i.e. give the value 0).

```

// ClearBit returns the bit 0 at the specified location
// (pos) in the specified variable (Item).                       1993
    
```

```

FUNCTION ClearBit(pos, item: byte): Byte;
  var bt : Byte;
BEGIN
  bt := $01;
  if Pos > 1 then
    { protection agains zero or negative number }
    bt := bt SHL (Pos-1);
    { move the first bit (pos-1) positions to the left }
  if (item and bt) > 0 then
    Result := (item xor bt)
    // the bit is here 1, so returns the bit 0
  else Result := item;
END;
    
```

To understand what's going on here, this is clear, readable code, right? On the internet I found a much more compact code. Now perhaps a challenge for you to decipher this "misty" code below:

```

function ClearBit(const aValue, aBitNumber :
integer) : integer;
begin
  result := aValue and not( 1 shl aBitNumber );
end;
    
```





In the function `ClearBit` above, the bit operands `AND` and `XOR` are used:

DEFINITIONS:

AND Bitwise operation

Binary AND Operator copies a bit to the result if it exists in both operands.

Xor Bitwise operation

Bitwise xor sets the bit to 1 where the corresponding bits in its operands are different, and to 0 if they are the same.

And with the `BitSet` function you can query whether a bit at a certain position is a 1:

```
// BitSet checks whether the specified position
// in the item has the value 1.                                1993

FUNCTION BitSet(pos, item: byte): Boolean; var bt : Byte;
BEGIN
  bt := $01; {bt get 00000001 }
  if Pos > 1 then { limitation }
    bt := bt SHL (Pos-1);
    { move bit no 1 pos-1 to the left }
  Result := (item and bt) > 0;
    { check whether both bits have the value 1 }
END;
```

or again much shorter (borrowed from the internet again) and perhaps another challenge for you to decipher the code below:

```
function isBitSet(AValue,
  ABitNumber:integer): boolean;
begin
  result:=odd(AValue shr ABitNumber);
end;
```

With these three functions above, a bit can be put in a binary number (give the value 1), remove one (give the value 0) or read its value (is it a 1 or a 0?) turn into. They are used in the code examples below.

LINK NODES

Now that we can define the route through the **scheme**, the next step in the development process is creating the **scheme**.

Again we first start formulating our ideas in pseudo code.

Suppose we want to connect two new nodes to a node C, one to the up pointer and the other to the down pointer. Linking nodes together is a bi-directional system: link node a to node b and link node b to node a. Otherwise you cannot walk back and forth through the schematic. But before we can link a new node it must first be created.

Now imagine that nodeC already exists and that nodeA and nodeB have just been created and are linked to nodeC.

Nodes A, B and C have the following three fields:

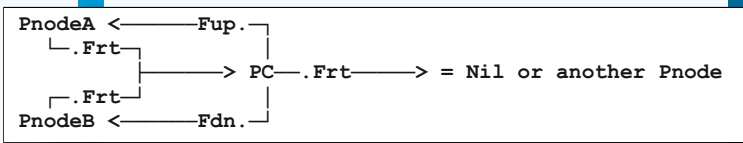
```
field: Fup { Nil: empty }
field: Fdn { Nil: empty }
field: Frt { Nil: empty }
```

If you want to connect `nodeA` to the up-pointer of `nodeC` and `nodeB` to the down-pointer of `nodeC` they have to get each other's pointer address (P ..) address.

In pseudo code:

```
nodeC.Fup → PnodeA
PnodeC ← Frt.nodeA
nodeC.Fdn → PnodeB
PnodeC ← Frt.nodeB
```

and schematic:



When using a Class, the **Constructor** method ensures the correct creation of new instances. What is an Instance anyway?

Info Instance (computer science)

In object-oriented programming (OOP), an instance is a concrete occurrence of any object, existing usually during the runtime of a computer program. The creation of an instance is called instantiation.

In class-based programming, objects are created from classes by subroutines called constructors, and destroyed by destructors. An object is an instance of a class, and may be called a class instance or class object; instantiation is then also known as construction.





Because you have to call the constructor for each new node instance, you can take the opportunity to immediately value all variables of the new node there. Our Constructor Create is given a number of parameters for this: the depth (dpth) in the **scheme** where the new node comes, the saved path (pscd) there and a number field (nmcd). In addition, the "pointer" fields are set to "no value" with the reserved word **Nil**.

```
// 2020
constructor TTrNd.Create(Adpth, Apscd, Anmcd : Byte);
begin
    Fdpth := Adpth;
    Fpscd := Apscd;
    Fnmcd := Anmcd;
    Fup := Nil;
    Fdn := Nil;
    Frt := Nil;
end;
```

Since no new memory is claimed in the constructor itself, it is not necessary to release it in the Destructor, and only the inherited destructor is called there using the reserved word **Inherited**.

```
// 2020
destructor TTrNd.Destroy;
begin
    Inherited;
end;
```

```
// 2020
// Note the use of the word var as prefix of the RootNode variable in the parameter list
PROCEDURE TTrNd.AddNode(Adpth, Apscd, Anmcd : Byte; var RootNode: TTrNd);
    var Newnode: TTrNd;
BEGIN
    NewNode := TTrNd.Create(Adpth, Apscd, Anmcd); // instance NewNode
    NewNode.SetFrt(RootNode); // newNode.Frt get the RootNode value
    RootNode := NewNode; // connect the new node at the calling node
END;
```

Code 10: The TTrNd.AddNode() class definition

What code is used here?
At first the info string: **// Note the use of the word var as prefix of the Rootnode in the parameter list of the procedure: var RootNode: TTrNd.**

The use of the prefix **var** before the parameter given to a routine deserves further explanation. The FPC provides the following definitions:

INFO
Difference between a Value- and a Variable parameter.

a) Value parameters
code snippets:

```
var
    Aparameter : AType
type
    Aprocedure(Aparameter: AType);
or
type
    Aprocedure(Aparameter: AType);
```

When parameters are declared as value parameters, the procedure gets a copy of the parameters that the calling statement passes. Any modifications to these parameters are purely local to the procedure's block, and do not propagate back to the calling block. So when the routine closes, the content of the variable is lost.

b) Variable parameters
code snippet:

```
var
    Aparameter : AType
type
    Aprocedure(var Aparameter: AType);
```

When parameters are declared as variable parameters with the word **var**, as prefix, the procedure or function accesses immediately the variable that the calling block passed in its parameter list. The procedure gets a pointer to the variable that was passed, and uses this pointer to access the variable's value (in this code snippet the variable **Aparameter** declared outside the routine definition). From this, it follows that any changes made to the parameter, will propagate back to the calling block. This mechanism can be used to pass values back

in procedures. Because of this, the calling block must pass a parameter of exactly the same type as the declared parameter's type. If it does not, the compiler will generate an error.





Briefly:

A value parameter has only meaning within the routine in which it is given as a parameter. Even if a variable with the same name already exists outside of this procedure!

A variable parameter must already exist as a variable outside the routine, in which it is given as a var parameter, and is used within the routine. So no copy is made! When this routine closes, the (possibly changed) value of this variable is retained.

The declaration as a var variable:

`var RootNode: TTrNd` in the above code ensures that the `RootNode` variable used in the `AddNode` procedure can be linked to `Newnode` and that `NewNode` can also be linked to `RootNode`.

In this way you can pass variables to a routine, the changes of which remain after the routine has ended.

Afterwards:

```
NewNode := TTrNd.Create(Adpth, Apscd, Anmcd);
```

This is the usual way in which you have a new instance generated in your code by the constructor (`Create`) of the relevant class type.

Note that it is the constructor of the class type `TTrNd` and not the constructor of the instance `NewNode` itself, because it does not yet exist!

Very logical but a mistake is easily made and the compiler will not warn you if you compile the following code line:

```
NewNode := NewNode.Create(Adpth, Apscd, Anmcd);
```

To demonstrate this, one line has been modified in the code of `TTr.Addnode ()` (See Code 10). See screenshot below (fig 19) where code line 165: `// NewNode := TTrNd.Create(Adpth, Apscd, Anmcd);` is temporarily replaced for line 166:

```
NewNode := NewNode.Create(Adpth, Apscd, Anmcd);
```

After that, the code was only recompiled (compile: (Ctrl + F9)).

```
160 | (**-----**)
    | // Note the use of the word var for the RootNode variable
    | PROCEDURE TTr.AddNode(Adpth, Apscd, Anmcd : Byte; var RootNode: TTrNd);
    |     var Newnode: TTrNd;
    |     BEGIN
165 | // NewNode := TTrNd.Create(Adpth, Apscd, Anmcd); // instantiate NewNode
    |     NewNode := NewNode.Create(Adpth, Apscd, Anmcd); // instantiate NewNode
    |     NewNode.SetFrt(RootNode); // give the field newNode.Frt the RootNode value
    |     RootNode := NewNode; // connect the new node to the calling node
    |     END;
170 | (**-----**)
    |
    |
158: 54 INS G:\Lazarus\testprojecten\clasrnd\utrclass.pas
```

Messages
...Compile Project, Target: clasrnd.exe: Success, Warnings: 56, Hints: 1

This also makes sense because the compiler cannot check for `NewNode` in this place whether or not it has already been created. Syntactics seems to make this code correct.

But if you then run this code then you get the following error (fig 20) and the program execution is aborted:

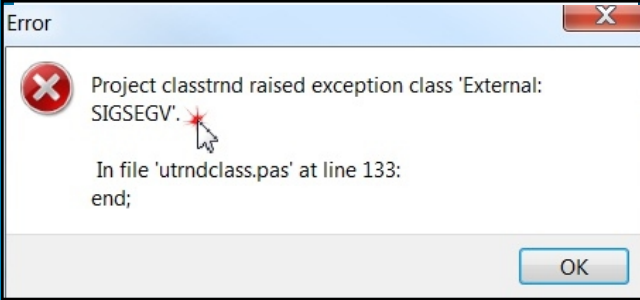


Fig 20: Exeption Class Error 'External: SIGSEGV'

What is striking here is that we made a "mistake" in unit `UtrClass.pas` on line 166, but the error message that the program stops talking about: "In file `utrndclass.pas` at line 133": a code line from another unit ! If we look at the code of the reported unit on line 133, we see the following there (See fig 21):

```
UTrNdClass utrclass
. (** TTrNd Class definitions -----)
125 | constructor TTrNd.Create(Adpth, Apscd, Anmcd);
. | begin
. |     Fdpth := Adpth;
. |     Fpscd := Apscd;
. |     Fnmcd := Anmcd;
130 |     Fup := Nil;
. |     Fdn := Nil;
. |     Frt := Nil;
133 | end;
. | (**-----)
```

fig 21: Execution fails here!





The TTrNd.Create () constructor is called here for the first time within the program. But now of the not yet created Class instantiation NewNode since we in the utrclass.pas unit deliberately have the wrong syntax in the code: `NewNode := NewNode.Create (Adpth, Apscd, Anmcd);` used (See fig 19). And this causes the fatal error here in the code.

INFO

If you get the above error (fig 20) then you have invoked a non-existent object somewhere, in any way. For example, you will also see it when you try to put a value in a field of a nonexistent object. The problem is that it is often difficult to find out where you made the crucial mistake in your code. As in the example above, this can be in a completely different unit than where the error message occurs and is therefore often difficult to detect.

But this aside, continue with our code: Second to last, the Frt field of the new node is linked to the var variable RootNode. We do this by using the Class method `SetFrt ()`.

```
// 2019
PROCEDURE TTrNd.SetFrt (ANode: TTrNd);
BEGIN
    Frt := ANode;
END;
```

Finally, the RootNode itself is assigned the value of the new node:

```
NewNode.SetFrt(RootNode);
// newNode.Frt get the RootNode value
RootNode := NewNode;
// connect the new node to the calling node
```

Ttree

Now that all the parts we need for creating a schematic have been designed, a container must be defined in which all those elements can be placed. In 1993 that was put in a (DOS Pascal) Object. Again with a corresponding pointer variable:

```
type
    PTr = ^TTr;
    TTr = Object
        Nds :PTrNd;

        constructor Init;
        destructor Done; virtual;

        procedure Add(nwndx: PTrNd);
        procedure MakeTree(ATrdpth: Byte);
        ...
END;
```

In the current Pascal we make it a Class again. This has been given the name Tree. If we follow the naming convention rules, that gives a prefix T for Type and with my preference for very short abbreviations we get **TTr (Type Tree)** as Class name from **TObject**. If it is a Class of the basic type **TObject**, you may also omit **TObject: TTr = class**.

In the private section, the root field of the tree is: **Froot** of type **TTrNd** (replaces the field **Nds** in the **DOS pascal** code). We use **Froot** to "fix" the schedule. It thus becomes the starting point of the virtual **scheme** that is further built entirely in memory. In the public section the constructor **Create** and the **destructor Destroy** must be added, supplemented with various own methods (procedures and functions). If you want to place more variables in this public section, they must be placed immediately before the constructor declaration.

As a rule, the variables are always listed first in each section, only then the methods. For example, if you look at the code definitions of a Form1 you have created with different components placed on it, you will see the same image: first the component (s) variable definitions and below that the methods are displayed.

The old **DOS code** looks like a modern Class (2020) like this:

```
type
TTr = class(TObject)
private
    Froot: TTrNd; // Tree start point
public
    // place any public fields here

    constructor Create;
    destructor Destroy; virtual;

    procedure AddNode(Adpth, Apscd, Anmcd : Byte);
var RootNode: TTrNd);
    function MakeTree(ATrdpth: Byte): TTrNd;

    function GiveTrdpth: Byte ;
    function GiveRt: Pointer;
    function CntNds: Word;

    procedure SetFroot(ANode: TTrNd);
    function GetFroot: TTrNd;
end;
```

Code 11: The TTr Class definition





We have already discussed the **AddNode ()** (See code 10) procedure above. This procedure is used in the **MakeTree ()** function to build the schema in memory.

NOTE: MakeTree () is defined as a function that returns type **TTrNd**.

Finally, the **Froot** field is assigned this value:

```
Tr.Froot := Tr.MakeTree();
```

so that later you can access the tree schema, which only exists in memory, via this field.

We have named the global **TTr** Class variable **Tree**:

```
var Tree : TTr;
```

Then we need to bring this "lifeless" thing to "life" using the Class constructor **Create**:

```
tree := Ttr.Create;
// and again NOT: Tree := Tree.Create;
```

If you forget this step, when you run the program you will get an error message that you want to do something with an object that does not exist and your program will crash (See Fig 19: **Exception Class Error "External: SIGSEGV"**).

As a final step, we need to ground the memory created **scheme** to the **Froot** field of the **Tree** variable so that we keep access to it. Since **MakeTree ()** is of type **TTrNd**, we can put it directly into the **SetFRoot** procedure, which accepts a variable of that type as a value parameter:

```
Tree.SetFroot(Tree.MakeTree());
// connect the in memory tree to the field var Froot
```

MakeTree

What's in the **MakeTree ()** function?

It starts with the variable **Treestart: TTrNd**. The value of this train start variable is returned to **TTr.Froot** as the result value of the function at the end of the code.

In the function itself, this is the starting point of the **scheme**.

```
// 1993/2020
FUNCTION TTr.MakeTree(ATrdpth: word):TTrNd;

type
  treestart : TTrNd;
  ndcnt : word; { count nodes }
  ... { code 10 }

BEGIN
  NdCnt := 1; { first node }
  TreeStart := nil; { initiate treestart }
  AddNode(0,0,Ndcnt,treestart);
  { create 1ste 'ROOT' treestart-node instance }
  SetFRoot(treestart); { Froot becomes 1ste node }
  ... { code 11 }
END;
```

Code 12: The TTr.Maketree() methode definition

So far nothing special.

To link two new nodes to an existing node, we use a local, that is, a local procedure defined in the **Ttr.MakeTree ()** procedure:

```
PROCEDURE PreOrder(reflink: TTrNd);
```

This is in place of the first row of red dots in the code above.

The **reflink** parameter is the node to which two newly created nodes are linked. Because two new nodes are linked in this procedure, we need an auxiliary node variable (**tempnd**) in which we temporarily store the address of **reflink**. Then with **AddNode ()** a new node is attached to the **Up** and **Down** fields of the variable **TreeStart**.

The local **PreOrder** procedure:

```
// 1993/2020
PROCEDURE PreOrder(reflink: TTrNd);
  var tempnd : TTrNd;
BEGIN
  if Reflink.GetFdpth < ATrdpth then
  BEGIN // not yet at the requested depth: so make 2 new nodes
    with reflink do
    BEGIN
      // save the reflink-Ptr (Frt-link) link for the new
      // FUp/FDn-node^.Rt in the var tempnd
      tempnd := reflink;
      // save then reflink link here for reuse because...
      // ...it immediately gets in AddNode a new value
      // make the new Up root
      Inc(ndcnt); // increase node counter
      AddNode(Succ(GetFdpth), GetFpscd, ndcnt,tempnd);

      // give the new ptr to ^.UP
      SetFUp(tempnd); // make the link

      // make the new Dn root
      tempnd := reflink; // restore the tempnd link
      Inc(ndcnt); // increase node counter
      AddNode(Succ(GetFdpth),
        SetBit(Succ(GetFdpth),GetFpscd),ndcnt,tempnd);

      // give the new ptr to ^.DN
      SetFDn(tempnd); // make the link
    END; { if Reflink.GetFdpth < ATrdpth }
    ... { code 12 }
  END; { with reflink do }
END;
PreOrder(reflink: TTrNd);
```

Code 13: The TTr.MakeTree() local ProOrder() procedure definition

```
BEGIN { TTr.MakeTree }
...
// built the complete tree with Froot/TreeStart as starting point
Preorder(TreeStart);
...
END;
```

Code 14: The second last TTr.MakeTree() code line





The only difference in code between the new Up and Dn node is in the **SetBit()** function of the Dn node. This sets a 1 to **Succ (Reflink.GetFdpth)** in the pscd variable.

The **Succ** function (def: Return next element of ordinal type) returns the next depth value. So in fact one level deeper than that of the treenode to which the two new knots are attached. Exactly what is needed.

This is all it takes to create two new nodes and link them to an existing node.

The line with the red dots at the bottom of the **PreOrder()** procedure contains only the following two code lines:

```
(* First make the upper half of the tree, *)
PreOrder(Reflink.GetFUp);

(* than the lower half of the tree. *)
PreOrder(Reflink.GetFDn);

END;
END; { PreOrder(reflink: TTrNd)}
```

Code 15: The last 2 Preorder() code lines

These are only two lines of code, but these two lines ensure that the entire **scheme** is made "automatically".

Here the simpler code from **MakeTree**:

```
... { = PreOrder() see code 13 }
BEGIN { TTr.MakeTree }
NdCnt := 1;
TreeStart := nil; { initiate treestart }
AddNode(0,0,NdCnt,treestart);
{ create 1ste 'ROOT' treestart-node instance }
SetFRoot(treestart); { Froot becomes 1st node}

(* built tree with TreeStart as startpoint *)
Preorder(TreeStart);
Result := TreeStart;
{ return root node address as function result }
END;
```

Before you read on:

Now try for yourself whether you can follow the course of the **ProOrder (starttree)** procedure, say depth 3. You will probably need some paper with it!

In short, the **PreOrder** procedure is called twice again in its own code until a certain condition is met. If this condition is never met, the procedure will only stop if the computer crashes due to lack of memory space. It is therefore important to define this break condition well. This mechanism is an example of recursivity that we will come across here more often.

Because it is quite difficult to follow exactly what happens step by step in this procedure, below is a printout of a log file in which every step in creating the schedule is written.

In this way, any size of a schedule is created without worrying about it. The only limitation here is the memory of your computer (and the byte type of some variables). No matter how big or small the schedule is.

INFO log file

The expression behind Str is the text string that can be found in the above diagram. E.g: "(nr1) 1 [0] Finale." or "(13) 6 [3-101 = 5]".

The different numbers represent the following: The numbers in brackets e.g. (No. 5) correspond to the numbers in brackets (5) in the diagram above. They indicate the order in which the treenodes are created. In the code, the variable **ndcnt** (node counter) maintains the counter value.

The number for the square bracket: 2, indicates the order of the node at this depth level. The rank is counted over and over for each depth level. The number behind the square bracket: [2, indicates the Depth at which this node point is located in the diagram. These two numbers can be found in the log file as = **R2 / D2**.

After the dash that follows, the distance traveled is shown in the diagram in the form of a binary number: -10 followed by its decimal representation: = 2, closed with a].

So in the example (5) 2 [2-10 = 2]: "fifth node in rank 2 and depth 2 with 10 as binary number corresponding to the decimal number 2"

Each time the function **Preorder** is called, this is indicated in the log file with an IN indented 2 spaces further than that of the previous IN. When the function is ready, it says OUT and the next line of text jumps back 2 spaces.

So everything between two associated IN and OUT lines happens **WITHIN** one **Preorder** function call.

For example, the first call to **Preorder()**:
 "IN: (No. 1) = R1 / D0 Preorder (ATree.Froot): (No. 1) 1 [0] Finale."
 ends at the bottom last:

"OUT: End No. 1 = R1 / D0 Preorder (ATree.Froot): (No. 1) 1 [0] Final."

So everything in between already happens in the first **Preorder** call!

The log file on the next page →





```

LOGINIT
MakeTree with dept 3
Start MakeTree

Add Start Node: (0,0,1,TreeStart)
Tree.Froot := TreeStart
IN : (Nr 1) = R1/D0. Preorder(TreeStart): str = (nr1) 1[0] Final.
Add(new UP node): Str = (2) 1[1-0=0]
...new Up.Root node: Str = (nr1) 1[0] Final.
Add(new DN node): Str = (3) 2[1-1=1]
...new Dn.Root node: Str = (nr1) 1[0] Final.
Thus: TreeStart.Up: Str = (2) 1[1-0=0]
Thus: TreeStart.Dn: Str = (3) 2[1-1=1]

IN : (Nr 2) = R1/D1. Preorder(Reflink.FUp)
Add(new UP node): Str = (4) 1[2-00=0]
...new Up.Root node: Str = (2) 1[1-0=0]
Add(new DN node): Str = (5) 2[2-10=2]
...new Dn.Root node: Str = (2) 1[1-0=0]

IN : (Nr 4) = R1/D2. Preorder(Reflink.FUp)
Add(new UP node): Str = (6) 1[3-000=0]
...new Up.Root node: Str = (4) 1[2-00=0]
Add(new DN node): Str = (7) 2[3-100=4]
...new Dn.Root node: Str = (4) 1[2-00=0]

IN : (Nr 6) = R1/D3. Preorder(Reflink.FUp)
==> max depth: No new Up node needed
OUT: (Nr 6) = R1/D3. Preorder(Reflink.FUp)

IN : (Nr 7) = R2/D3. Preorder(Reflink.FUp)
==> max depth: No new Dn node needed
OUT: (Nr 7) = R2/D3. Preorder(Reflink.FUp)

OUT: (Nr 4) = R1/D2. Preorder(Reflink.FUp)

IN : (Nr 5) = R2/D2. Preorder(Reflink.FUp)
Add(new UP node): Str = (8) 3[3-010=2]
...new Up.Root node: Str = (5) 2[2-10=2]
Add(new DN node): Str = (9) 4[3-110=6]
...new Dn.Root node: Str = (5) 2[2-10=2]

IN : (Nr 8) = R3/D3. Preorder(Reflink.FUp)
==> max depth: No new Up node needed
OUT: (Nr 8) = R3/D3. Preorder(Reflink.FUp)

IN : (Nr 9) = R4/D3. Preorder(Reflink.FUp)
==> max depth: No new Dn node needed
OUT: (Nr 9) = R4/D3. Preorder(Reflink.FUp)

OUT: (Nr 5) = R2/D2. Preorder(Reflink.FUp)

OUT: (Nr 2) = R1/D1. Preorder(Reflink.FUp)

IN : (Nr 3) = R2/D1. Preorder(Reflink.FUp)
Add(new UP node): Str = (10) 3[2-01=1]
...new Up.Root node: Str = (3) 2[1-1=1]
Add(new DN node): Str = (11) 4[2-11=3]
...new Dn.Root node: Str = (3) 2[1-1=1]

IN : (Nr 10) = R3/D2. Preorder(Reflink.FUp)
Add(new UP node): Str = (12) 5[3-001=1]
...new Up.Root node: Str = (10) 3[2-01=1]
Add(new DN node): Str = (13) 6[3-101=5]
...new Dn.Root node: Str = (10) 3[2-01=1]

IN : (Nr 12) = R5/D3. Preorder(Reflink.FUp)
OUT: (Nr 12) = R5/D3. Preorder(Reflink.FUp)

IN : (Nr 13) = R6/D3. Preorder(Reflink.FUp)
==> max depth: No new Dn node needed
OUT: (Nr 13) = R6/D3. Preorder(Reflink.FUp)

OUT: (Nr 10) = R3/D2. Preorder(Reflink.FUp)

IN : (Nr 11) = R4/D2. Preorder(Reflink.FUp)
Add(new UP node): Str = (14) 7[3-011=3]
...new Up.Root node: Str = (11) 4[2-11=3]
Add(new DN node): Str = (15) 8[3-111=7]
...new Dn.Root node: Str = (11) 4[2-11=3]

IN : (Nr 14) = R7/D3. Preorder(Reflink.FUp)
==> max depth: No new Up node needed
OUT: (Nr 14) = R7/D3. Preorder(Reflink.FUp)

IN : (Nr 15) = R8/D3. Preorder(Reflink.FUp)
==> max depth: No new Dn node needed
OUT: (Nr 15) = R8/D3. Preorder(Reflink.FUp)

OUT: (Nr 11) = R4/D2. Preorder(Reflink.FUp)

OUT: (Nr 3) = R2/D1. Preorder(Reflink.FUp)

OUT: End Nr 1 = R1/D0. Preorder(TreeStart): str = (nr1) 1[0] Final.

Result := GetFroot: str = (nr1) 1[0] Final.

Node count is: 15

LOGDONE
    
```





A small piece of code example where the log file is written by LOGIN (log file, string) LOGOUT (log file, string) and LOG Text (log file, string):

INFO Logfile The procedures and functions used to create the log file are listed in the Std_log unit. To use this only for debugging, a custom compiler directive has been defined in the implementation section of the relevant unit:

```

. implementation
.
75 uses Std_log,
76     DisplayForm,
.     Fpbox;
.
. {$R *.lfm}
80 {$DEFINE usedebug} //self-defined compiler directive for the use of the logfile
    
```

fig 22: my own compiler directive in the implementation section of the program

By calling this directive usedebug in the code with: {\$IFDEF usedebug} and logging out with {\$ENDIF}; the code between them is executed at compile time. If you comment the line {DEFINE usedebug} in the implementation section by prefixing 2 //,

the compiler ignores the code between the usedebug call and call. The LOGINIT procedure is used to connect the Log file of type TEXT to a text file. The procedure LOGDONE (Logfile) to close this text file using Close (Logfile).

```

280 FUNCTION TTr.MakeTree(ATrdpth: word):TTrNd;
. BEGIN
.     {$IFDEF usedebug}
.     LOGINIT(datapad,'LOG-MakeTree.txt', Logfile,'MakeTree', true);
.     {$ENDIF};
.
.     |
284
285     {$IFDEF usedebug}
.     LOGDONE(Logfile);
.     {$ENDIF};
.
. END;
    
```

fig 23: LOGINIT() and LOGDONE()

```

FUNCTION TTr.MakeTree(ATrdpth: Byte):TTrNd;
PROCEDURE PreOrder(reflink: TTrNd);
.
.     {$IFDEF usedebug}
.     With Reflink.GetFUp do
230 Begin
.         Str(GiveRank(GetFpscD,GetFdpth),prnrank);
.         Str(GetFdpth,prndpth);
.         Str(GetFmcd,prncnt);
.         LOGIN(logfile,indent,'(Nr '+prncnt+') = R'+prnrank+'/D'+prndpth+
235             '. Preorder(Reflink.FUp)');
.     End;
.     {$ENDIF};
.
.     (* Make the top half of the tree first, *)
.     PreOrder(Reflink.GetFUp);
240
.     {$IFDEF usedebug}
.     With Reflink.GetFUP do
.     Begin
.         if GetFdpth >= ATrdpth then
245             LOGText(logfile,indent,' ==> max depth: No new Up node needed',0);
.         Str(GiveRank(GetFpscD,GetFdpth),prnrank);
.         Str(GetFdpth,prndpth);
.         Str(GetFmcd,prncnt);
.         LOGOUT(logfile,indent,'(Nr '+prncnt+') = R'+prnrank+'/D'+prndpth+
250             '. Preorder(Reflink.FUp)');
251             LOGText(logfile,indent,'',0);
.     End;
.
.     With Reflink.GetFDn do
255 Begin
.         Str(GiveRank(GetFpscD,GetFdpth),prnrank);
.         Str(GetFdpth,prndpth);
.         Str(GetFmcd,prncnt);
.         LOGIN(logfile,indent,'(Nr '+prncnt+') = R'+prnrank+'/D'+prndpth+
260             '. Preorder(Reflink.FUp)');
.     End;
.     {$ENDIF};
.     (* then the bottom half of the tree. *)
.     PreOrder(Reflink.GetFDn);
    
```

fig 24: LOGIN(), LOGOUT() and LOGText in progress





You will find these text lines in the log file above, for example at node 9 and the five lines below:

```

LOGIN() ==>      IN : (Nr 9) = R4/D3 Preorder(Reflink.FUp)
LOGText() ==>    ==> max depth: No new Up node needed
LOGOUT() ==>    OUT: (Nr 9) = R4/D3 Preorder(Reflink.FUp)
LOGText(leeg) ==>
LOGOUT() ==>    OUT: (Nr 5) = R2/D2 Preorder(Reflink.FUp)
LOGText(leeg) ==>
LOGOUT() ==>    OUT: (Nr 2) = R1/D1 Preorder(Reflink.FUp)
LOGText(leeg) ==>
    
```

Print Tree

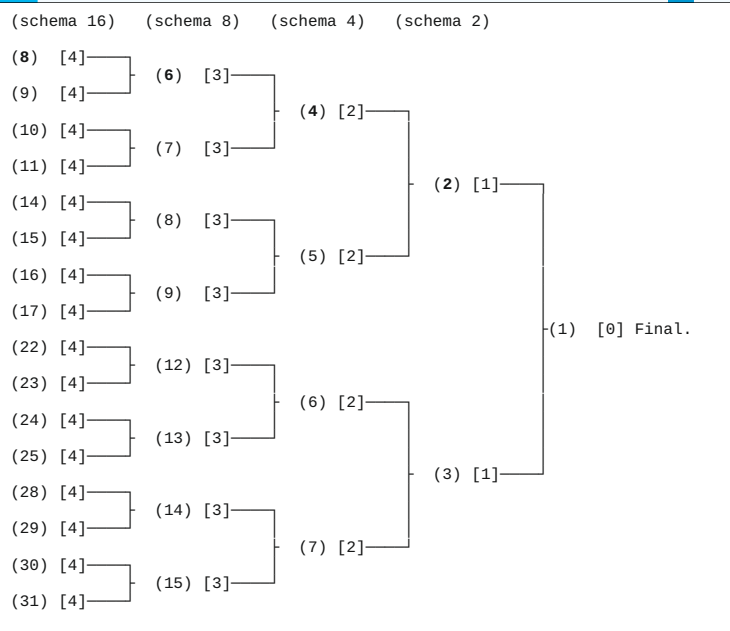
Now that the **scheme** tree can be built in memory, it must of course also be possible to display it. In the **Ftree** program written for this article, this is done in two ways: directly written as the lines of a memo component to the screen and in the background in the original way to a text file on the disk. The latter also takes account of printing to paper. The printing procedure must also be suitable for line printers.

That is why the preparation of the printing process is organized per line. What again proved to be an advantage when writing these lines to the lines of the memo component.

Since we don't want to send every attempt directly to the printer while developing the code, we first send it to a text file using the variable LST: Textfile. (Textfile: Text).

The starting point

Which node should I start with and does the node creation sequence provide a handle for this? Have a look at the combined **scheme** below; a **scheme16** with the creation order () at max depth [] per **scheme 16 / 8/4/2**. For **scheme8** this is the series (6,7,8,9,12,13,14,15) and for **scheme4** (4,5,6,7):



It contains between () the creation order of the nodes during the creation of a **scheme16**. Starting at the top left means for a **scheme16** with depth [4] starting with node (8) at the top left: (8) [4] - . For a **scheme8** 8 with depth [3] start with node (6) top left: (6) [3] - and for **scheme4** with depth [2] start with node (4) also top left: (4) [2] - , etc.





It is striking that the creation order of the nodes always depends on the **scheme** size. So, unless you write down all possible schedules, there is no use for the creation sequence when printing a **scheme**.

What would work? The creation of the **scheme** in memory was done using recursivity. Could printing also work with that so that you only have to specify the level and the schedule can be printed like this? Yes, that's possible. By now you will have understood that the schematic tree examples used in this article have not been assembled by hand for this article. They have been generated by a program and where necessary adapted for this article by hand. In that program recursivity is also used in various places, only slightly different.

When building the schema in memory, the recursivity **scheme** in pseudo code is like this:

```

Program
// declaration intern/local procedure
procedure Preorder()
begin
  if breaks-down-condition then
  begin
    do code;
    Preorder(up); // recursive
    Preorder(dn); // recursive
  end;
end;

Begin
  initialise;
  Preorder("start");
End;
    
```

Because of the order of the code to be executed and the recursive **call(s)** I then chose the name Preorder: first execute the code before calling the same procedure again. This call is not valid for printing because the print order for a schedule of 8 players (depth 3) would be the following series from the production order: no- 6,4,7,2,8,5,9,1, (etc).

As noted earlier, this sequence is specific to a **scheme8** with depth 3. This approach has therefore not been found suitable for a general code. What the nodes 8,6 and 4 in the above **scheme16** have in common is that their Up and Dn fields in their OWN **schemes** have the value NIL. After all, there is no higher level at this point in a **scheme16**, **scheme8**, or **scheme4**!

A general line in pseudo code could then read: "from the start node point (finalist) go all the way up to the left until the Up and Dn fields of the node concerned have the value NIL. If yes then do something. It follows that the recursivity order for printing is different. In pseudo code:

```

procedure Inorder();
begin
  if breaks-down-condition then
  begin
    Inorder(up); // recursive
    do code;
    Inorder(dn); // recursive
  end;
    
```





The "do code" is now between the two recursive function calls. I have therefore named this procedure InOrder.

If we look at the translation of this pseudo code into Pascal code and taking into account the above findings, we see:

The breaks down condition becomes:

```
If reflink <> Nil Then
  BEGIN
    Inorder(reflink.GetFUp,depth-1); { upper treenodes }
```

This means that as long as reflink is <> Nil, a new Inorder procedure is always called with the node point being the one who is on the Up link of reflink with one step lower value for the depth. If the Up-link is Nil (endpoint) then this Inorder () procedure is exited and we return to the previous InOrder () procedure at: do code.

When we are done with "do code" we will jump down in the next InOrder () procedure with the Dn link from Reflink. For example, we finish the whole schedule from top left, via the final node on the far right, to bottom left. This is how the differences in the production order of the Nodes per scheme size arise. And this works for any schedule of any size.

THE DO CODE

The code in "do code" here is considerably more complicated than that of having the schema tree generated in memory. In the original code, each part of a line is sent separately to the printer. To get a clear explanation, that code has also been used here. For the memo component, this information is first collected per line in a string variable before that line is presented to the memo.lines.Append (AString) at once. Note the use of Append () here.

So we start with the basics and expand it further and further until the schedule is complete on paper.

What kind of variables do we need?

```
//PrintTree: print a tree to a textfile
PROCEDURE TTr.PrintTree;
  var
    cntnd : word; // node counter
    trdpth : Byte; // tree depth
    s : string[25];
    // length printstr = standard distance between the
    // nodes at different tree depths.
```

The first 2 variables speak for themselves. For simplicity, the printing distance between two treenodes, anywhere in the schematic, is kept the same. The string s is used for this and has been given the length 25 by default: s: string [25]; (A more complicated alternative could be that the length of s depends on the length of the Node information string and thus changes in length per level.)

The PrintTree procedure starts with preparing the required data in the various variables. Then calls the Inorder () procedure and ends by specifying the number of treenodes printed:

```
BEGIN { TTr.printtree }
//Innit parameters
  trdpth := Givetrdepth; // give tree depth
  cntnd := 0; // initiation node counter
  s := ""; // initiation standard string s
  Fillchar(s[1],High(s),' ');
  // fills entire show string with spaces
  // print entire tree structure recursive
  Inorder(GetFRoot, trdpth);

  write(LST,'Number of nodes: ',cntnd);
END;
```

We go through the above code from top to bottom.

```
trdpth := Givetrdepth;
```

The Givetrdepth (Give Tree Depth) function is included as a method in the TTr class and declared as follows:

```
// 1993
FUNCTION TTr.Givetrdepth: Byte;

PROCEDURE PreOrder(reflink: TTrNd);
BEGIN
  if reflink <> Nil then BEGIN
    If reflink.GetFdpth > Result Then Inc(Result);
    Preorder(reflink.GetFUp);
    Preorder(reflink.GetFDn);
  END;
END;

BEGIN
  Result := 0;
  Preorder(GetFRoot);
END;
```





The `Givetrdepth()` function does nothing but run through the whole **scheme** looking for the highest value of `dpth` (via: `reflink.GetFdpth> Result`) and puts that value in the function result (Result). This also happens again via a recursive mechanism `PreOrder()`. This way **TTr** always “knows” what maximum depth a offered schedule has.

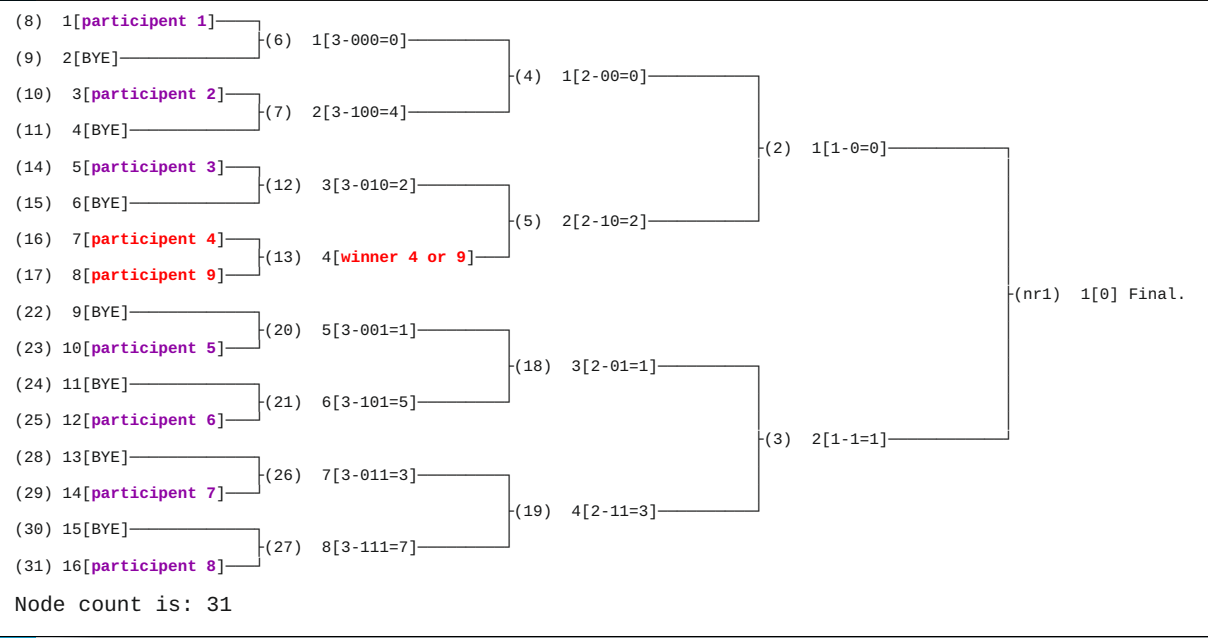
Now I hear you think that searching the whole tree is a lot of good: Just go all the way to the top left until the up link is Nil shouldn't it be enough? This is also the case for our example in this article, but suppose that there are 9 participants instead of 8. Then a **scheme8** of 8 participants (max place with **scheme** of depth 3) is not sufficient and a schedule 16 of depth 4 must be made. That's 16 playgrounds for which we only have nine participants. This means that seven players have a bye and therefore only start their match in the next round and 2 participants have to play a so-called preliminary round. For the distribution of 9 players over 16 places, there is an official distribution key which has been converted here into a constant `ByeArr16`:

```
{ 1 10 16}
{a} 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 these are ranking numbers, 16 places here}
{b} (0,1,0,5,0,4,0,8,7,0,3,0,6,0,2,0); these are the byes in consecutive number order for een schedule of 16 places}
```

Combine the two digit strings a and b above: 1st bye is in 2nd place, 2nd by 15th in the schematic, etc gives the following code implementation:

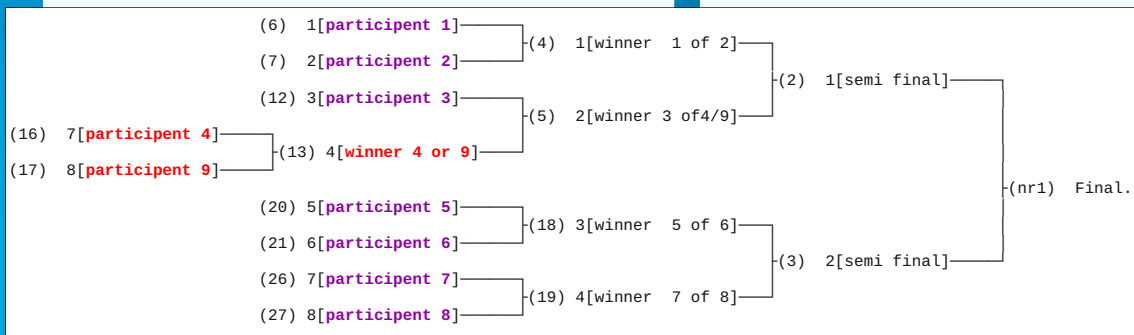
```
const byearr16 : array[1..8] of byte = (2,15,11,6,4,13,9,8); {these are the byes in consecutive number order }
```

There are 8 byes in the `byearr16`. If you use all 8 byes you will get automations on a **scheme8** of 8 places. In the example here we use $16-9 = 7$ byes. Then it follows from the `byearr16` that the first 7 ranking places (2,15,11,6,4,13,9) in the **scheme16** get a Bye. Ranking 8 (from **scheme16**) remains for participant 9. The other ranking places are allocated to the other 8 participants. The **scheme16** with 16 places and only 9 participants will look like this:





The places in this schedule that are unoccupied (BYE), so that the players "move on" (have a Bye) to the next round, can be removed from the schedule together with the empty players. The cleaned **scheme16** will then look like a **scheme8**, with only one preliminary round in this example:



Node count is:
 17 {calculation rule: $([8*2]-1) + 2$ }

In this situation you go in the cleaned schedule tree all the way to the top left and then set that at node nr 6, 3 is the maximum depth of the tree then you are wrong, because that is 4 at the treenodes 16 and 17! And because you never know in advance how many participants will participate, you never know in advance where exactly one or more preliminary rounds will be played and so you have to go through the whole tree in search of the highest value for the depth dpth.

The code after that:

```

cntnd := 0; {initiate node counter}
s := "";
Fillchar(s[1],High(s),' ');
{fills entire showstring with spaces}
    
```

The node counter (cntnd) is set to zero, the string s is cleared and then using the FillChar procedure from position one - s [1] - to one with the highest position of s - High(s) = 25, because defined as string [25] - provided with spaces. If we print the string s in the code, we actually print 25 times "nothing". Now that everything is ready, we step into the Procedure Inorder:

```

Inorder(GetFRoot, trdpth);
{print the entire tree structure recursively}
    
```

This procedure is given 2 parameters: the variable that gives the starting point of the "higher" **scheme** and the desired maximum depth of the **scheme** to be printed.

The InOrder Code

```

// InOrder() is used recursively! 1993
PROCEDURE InOrder(Reflink: TTrnd; depth: byte);
var i : byte; // loop counter
intern : Boolean; // 'insite' or 'outside' the schema
prnstr : string; // 'the collecting string to print'
BEGIN
If reflink <> Nil Then // the breaks down condition
BEGIN
1) Inorder(reflink.GetFUp,depth-1);
// at first the upper tree-nodes
...
    
```

The three variables speak for themselves. The first line of code: 1) should now also be clear. Then comes "The Code":

```

// Note: dpth is zero by the draw-root where the fields
// FUp and FDN are NIL !
if reflink.GetFdpth >= 0 then
BEGIN
for i := 1 to depth do write(LST,s);
// print the correct number of s units =
// equivalent to moving the print cursor

for i := 2 to depth do write(LST,' ');
// extra spaces for the (N-1) number of graphic char(s)
// used in the line above

if Reflink.GetFdpth <> trdpth Then
write(LST,hvl); // hvl = the graphic char+
...
    
```





When calling this function again within the Inorder procedure, 1 value is always subtracted from the depth. It then becomes zero when the highest level of the schedule is reached. This is useful here because we can read the diagram from the right (from **final = FRoot**) to the left, but we will print it from left to right! No matter how large the schedule is when the depth here becomes zero, you are always at the beginning of the lines to be printed.

In the 1st "for" loop, we always write units of 25 spaces (s), as many times as we are towards the final in the schedule. The second "for" loop adds, on the same line, some necessary extra spaces to ensure mutual line alignment. More about that later.

```
// Get the description of this node *)
s := Reflink.NdInfoStr(trdpth);
// Node Info Str(.,.) in unit UtrNdClass.pas!
write(LST,s);
```

NdInfoStr is a method of the Class **TTrNd** and returns a string with node info:

```
// NdInfoStr displays the available information from the node point as follows:
// (,creation sequence number, ), ascending order number of the schema at that level,
// [, treedepth, -, nmcd written as a binary string, =, nmcd written as a number value, ]
// eg maxdpth 7: dpth 7, draw-root 24 gives: (53) 24[7-1110100=116]
// eg maxdpth 7: dpth 4, in-between node 7 gives: (100) 7[4-0110=6]
// eg maxdpth 4: dpth 4, draw-root 7 gives: (16) 7[4-0110=6]
// eg maxdpth 4: dpth 2, in-between node 3 gives: (18) 3[2-01=1]
// eg maxdpth 4: dpth 0, in-between node 1 gives: (nb1) 1[0] Final
```

```
FUNCTION TTrNd.NdInfoStr:String;
var
  prndpth : string[2];
  prnrnk : string[3];
  prnpsc : string[3];
  prnnmcd : string[3];
BEGIN
  (* prepare the print strings *)
  Str(GiveRank(Fpsc,Fdpth):3,prnrnk); // returns the rank order as a string
  / :3 is the amount of digits of the number

  Str(GetFdpth,prndpth); // returns the depth as a string
  Str(GetFpsc,prnpsc); // returns the position_code bit order as a string
  Str(GetFnmcd,prnnmcd); // returns the num_code as a string
  (* concat for result *)
  if GetFdpth = 0 then // final
    Result := Concat('(nr',prnnmcd,')',prnrnk,['[,prndpth,'],' Final.')]
  else
    Result := Concat('((',prnnmcd,')',prnrnk,['[,prndpth,
    '- ',GiveBinstr(GetFpsc,GetFdpth),'=',prnpsc,']')];
END;
```

```
type
TTrNd = class(TObject)
...
public
...
  function NdInfoStr:String;
end; { TTrNd = Class() }

binstr = string[8]; // is the binary string representation
// of the byte value pscd. example: '01100101'
...
```

Code 16: Declaration of the type class TTrNd.NdInfoStr and binstr

And since Reflink is also an instance of that, you can request it directly by means of the dot reference:

```
Reflink.NdInfoStr() .
```

This function is specially designed to string as much information about the node as possible. How? You can figure that out yourself in the code below.





The function result above is composed of the different strings and punctuation marks using the internal **Concat** function and then assigned to Result. Two other functions are used in this function: **Giverank()** and **GiveBinStr()**. The first is discussed in detail later in this article. The code for the second follows. It converts a decimal byte value into a binary bit string of type **binstr**: `binstr = string[8];` (See code 16).

```
// GiveBinstr converts a byte into a string of ones en zeros.
// givebinstr: b is the byte to be converted, d indicates
// the number of bits put in a string. Max 8.
```

```
FUNCTION GiveBinstr(b,d: byte): binstr;
var
  i, bt : byte;
  s : binstr;
BEGIN
  s := "";
  bt := $01;
  for i := 1 to d do
  BEGIN
    if (b and bt) > 0
    then s := '1'+s // there is a 1 in b
    else s := '0'+s; // there is a 0 in b
    bt := bt shl 1; // move bits one position to the left
  END;
  Result := s;
END;
```

If you still want to display larger than **scheme255** of depth 8, the type of **binstr** will also have to be changed, e.g. to `binstr = string [16];` Then the string S is sent to the text file. Still without line cover!

If we look at the name of the **binstr** type, it turns out that this was an unfortunate choice.

Free Pascal has an internal function of the same name that does the same as the 1993 function **GiveBinStr()** does here: **Binstr()**.

Info Binstr(aValue, aCnt): shortstring.

BinStr returns a string with the binary representation of **aValue**. The string has at most **aCnt** characters. (i.e. only the **cnt** rightmost bits are taken into account) To have a complete representation of any longint-type value, 32 bits are needed, i.e. **aCnt=32**.

But this aside. Because the information string of the node point at different depth levels is not the same length, S is now supplemented as necessary with dashes (the constant char **hrz**: —) to the chosen standard length of 25 chars.

The node counter then keeps track of the number of node created. The code:

```
if Reflink.GetFdpth > 0 then
// exclude the finale node (= Froot) here
  for i := 1 to (High(s)-Length(s)) do
    write(LST,hrz);
// adds up to standard length(25) with dashes {-}
  Inc(cntnd);
```

Note the use of the standard functions **High(s)** and **Length(s)**:

High(s) returns the maximum length of s while

Length(s) returns the current length of s!

A node string currently looks something like this:

(7) 2[3-100=4]—————

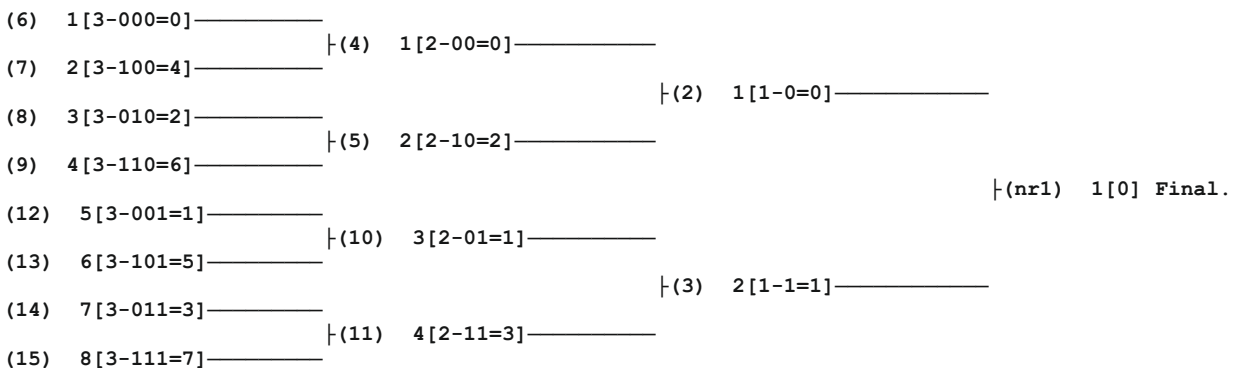
After this:

```
if (trdpth-depth) <> 0 then
With reflink do
BEGIN
...

```

Below is code that needs further explanation.

For that we first look at what kind of **scheme8** would be printed so far:





In the above diagram, all treenodes with the exception of their content are equal: (nr) followed by position code nr [depth-pcode = decimal number] followed by more or less fill marks so that the string has a length of 25 characters. This is already starting to look similar to our previous schedules. What are we still missing? To make that clearer, 25 dots have been placed in the string s instead of spaces for the following example: s = '.....' and the still missing four graphic line elements with different basic text characters (\> / +) are filled in:

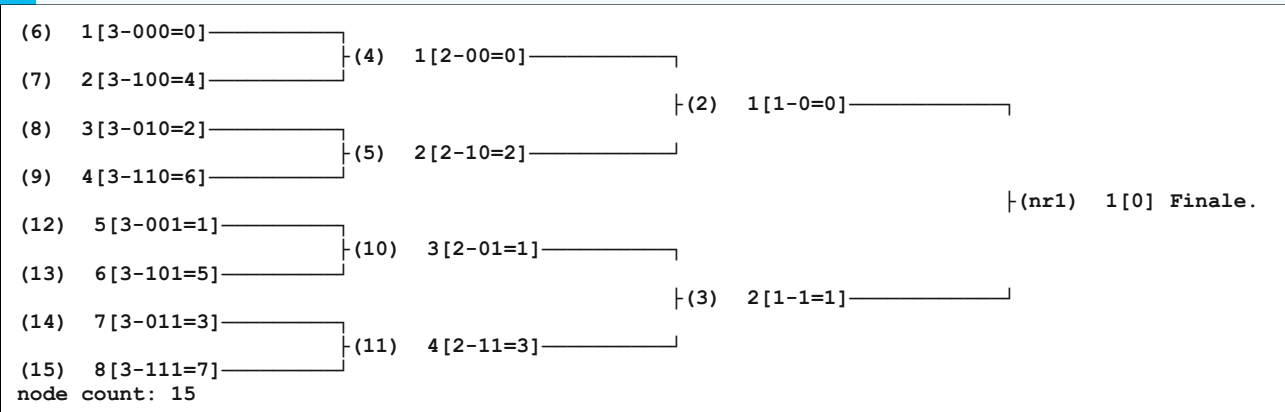
```
(6)  1[3-000=0]-----\
.....>(4)  1[2-00=0]-----\
(7)  2[3-100=4]-----/.....|
.....+>(2)  1[1-0=0]-----\
(8)  3[3-010=2]-----\.....|
.....>(5)  2[2-10=2]-----/.....|
(9)  4[3-110=6]-----/.....+.....|
.....++>(nr1)  1[0] Finale.
(12) 5[3-001=1]-----\.....+.....|
.....>(10)  3[2-01=1]-----\.....|
(13) 6[3-101=5]-----/.....|.....|
.....+>(3)  2[1-1=1]-----/
(14) 7[3-011=3]-----\.....|
.....>(11)  4[2-11=3]-----/
(15) 8[3-111=7]-----/
Node count: 15
```

fig 25: schema printed with the help of standard text elements \> / |

If you compare this with the diagram above, we will miss the following four characters \, >, /, | and a plus sign: just like the dot a visible substitute for a space. If you look closely at the last diagram, the line segments of the different nodes are now NO longer exactly the same: there are with a \ or a / or a >. The question arises here how the code knows when and which of these 3 different graphical elements should be added to a node point string. The solution lies in the binary representation of the position code. Take for instance that of node (8): 010. This symbolizes that the route to this node point from the final point on the right side of the schedule 1x up (0), then 1X down (1) and again 1x up (0) runs from right to left by default. These numbers indicate when a node point is an Up or a Dn node: If the bit in place depth in the pscd is a 1 then it is a Dn node otherwise it is an Up node. Converted to code you will get the following:

```
(* if a node at (Trdpth-depth) level has a bit =1, then write graphic element {+} *)
  if BitSet((trdpth-depth),GetFpscd) then // this is a Dn node
    write(LST,hup) {+}
(* else: (Trdpth-depth) level has a bit =0, so write here graphic element {+}*)
  else // this is a Up node
    write(LST,hdn); {+}
```

The schedule will now look like this:





If you now look at figure 25 above, you will see that after placing the lock (┘ or ┘) sign in the print string [25], some nodes are followed by 1 or more blocks with dots, sometimes with a plus sign in between, sometimes with one or more plus signs at the end of the series of blocks and others not at all.

Again the question arises how the code knows where to place which graphic character in the right place. Where should there be vertical lines in the text line to be printed to complete the diagram? "Of course" between the (┘ or ┘) sign and the | sign on the line below and above it, but how does the code "know" for each line in the **scheme** where that position is exactly in that line? Finally, the code does not remember what it printed before and does not "know" what happens next. **Once again, the key to the solution lies in the position code.**

A binary code is read from right to left by default and is therefore filled from right to left with the different routes. Again the previous example of node (8) whose binary **pscde** is 010. This route says that from the final on the right side of the schedule you have to go 1x up (0), then 1x down (1) and 1x up again, by default reading from right to left. This in itself does not provide more information than what we already know: a 1 stands for a Dn node and a 0 stands for an Up node.

It had been intriguing me for quite a long time before I woke up one night and I knew I'd found the solution. It turned out not to be in the numbers 0 and 1 itself but in the alternation of the 1 and the 0 in the series of the binary number.

And because the schedule is going to be printed from left to right, you MUST also see this variation from left to right, starting at the **Dpth** level of the node point itself. If the neighbour to the right changes sign in the sequence, that is, from a 1 to a 0 or vice versa, the route at that location nods downwards or upwards in the diagram: so the line to be printed at that location is cut a vertical line of the schematic!

Check that in the schedule, it is always correct! The code therefore does not have to remember anything at all about what it printed before or what it comes after, it already has the necessary information in the data of the node point at its disposal.

Before we can use the code for this, a number of variables have to be reinitialized here in the code. We will use a print string: **prnstr** as a collection string and make **s** again a **string[25]** filled with spaces. In addition, the boolean **Intern** is set to **False**:

```
Intern := FALSE;           { initiate }
SetLength(prnstr,0);      { initiate }
SetLength (s,High(s));
{ High(s) give's back max length of s }
Fillchar(s[1],High(s),' ');
{ fills entire showstring(25) with spaces }
```

Then follows the code for placing the vertical graphic line element (s) in the correct position in the line to be printed:

```
// Question: have line elements to be placed
// further on as a rule?

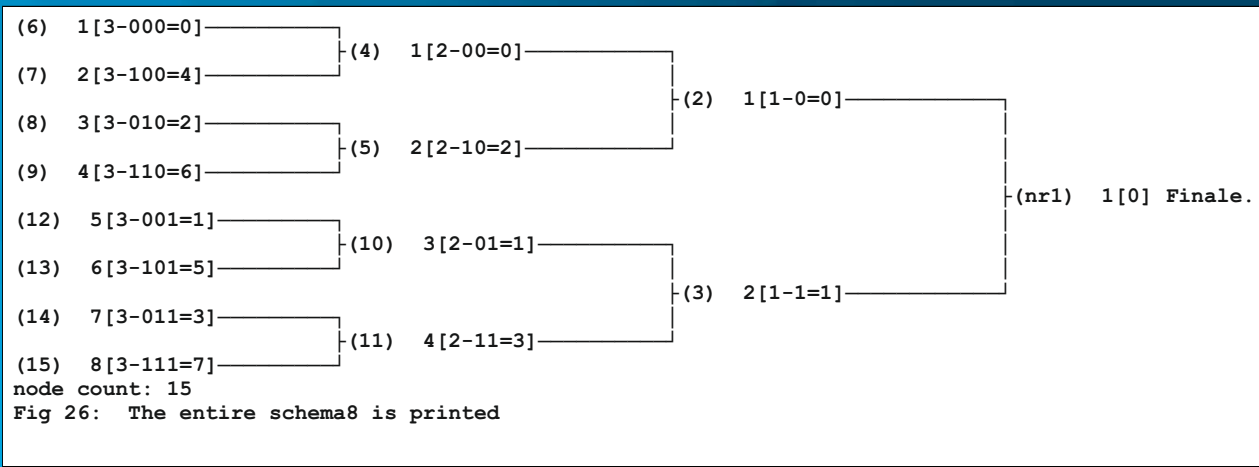
for i := 2 to GetFdpth do
  BEGIN
  // is there a sign change (0=>1 of 1=>0)
  // at position i in the position-code?
  if (BitSet(i,GetFpscde) <> BitSet(i-1,GetFpscde))
  then
    BEGIN { place graphic element (vrt=|) in prnstr }
    prnstr := Concat(s+vrt,prnstr);
    // concat prnstr after s with the graphical element |
    // in between
    intern := TRUE;
    END
  else
    // concat prnstr after s with a space char instead of
    // the graphical element | in between
    if intern then prnstr := Concat(s+' ',prnstr);
    END; { For.... }

  if intern then write(LST,prnstr);
```

In the above code, the boolean **Internal** is used. The reason for this is that the vertical line elements only have to be placed "within" the **scheme**, of course they no longer serve any purpose. **Internal** is set to **True** if **s** is still within the schedule and only if **Internal** is **True** is printing.

See fig 25 Page 34 of this article: only the enclosed area is provided with strings **s** (here filled with dots). Now the whole **scheme** can be printed complete with all graphic line elements:





The Inorder procedure ends with the following code:

```

END; // if (trdpth-n) <> 0 then With reflink do
END; // if reflink.Givedpth >= 0

writeln(LST); // now writeline
Inorder(reflink.GetFdn,depth-1);
// make down nodes

END; // If reflink <> Nil
END; // Inorder()
    
```

A log file has also been kept for the printing process, in which almost every step is saved. For clarity, the string *s* is also filled here with dots instead of spaces.

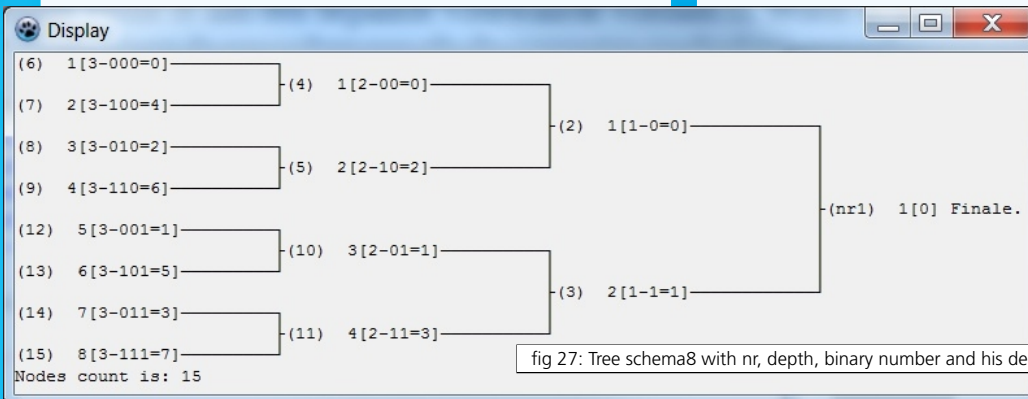
Even for a small **scheme** of 8 participants, it produces a very large log file. Therefore, only part of the chart8, with the last few lines of the bottom is shown here. Since you can download the complete code you will be able to have a closer look.

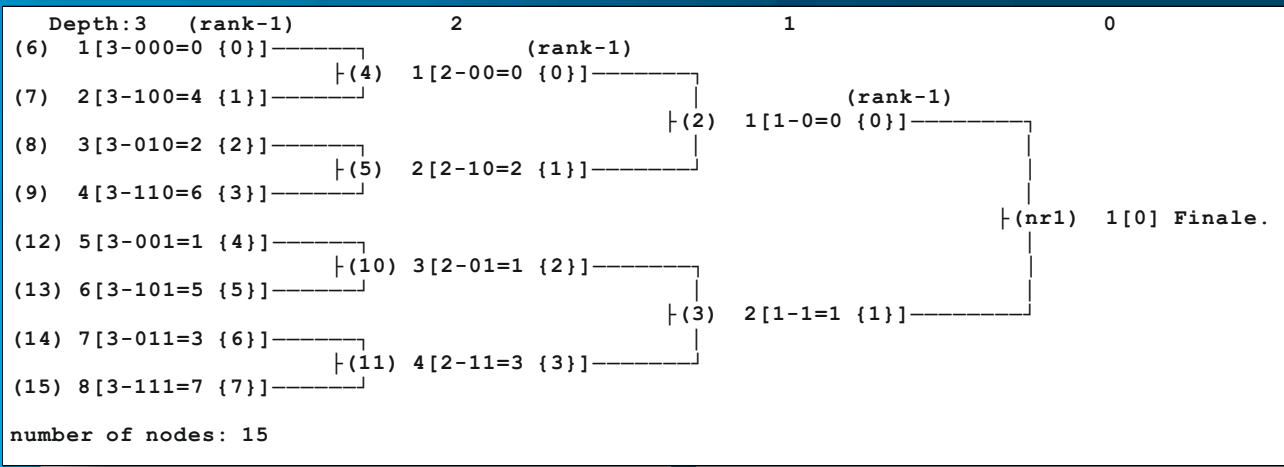
And this is how the **scheme8** with a depth of 3 and node information, "printed" on the canvas of a Form, looks like:

RANKING OF PARTICIPANTS

In a knock-out **scheme** each node has a rank order number. This number is used in the lottery draw and thus determines which players will face each other where in the schedule. This number runs from the top node to the left to the bottom node to the left from 1 to 4, 8, 16, 32, 64, 128 etc. depending on the size of the **scheme**. These ranking numbers are generated by the code. Here again the question arises how the code "knows" what the rank number of a node point is, regardless of the level at which the treenodes act. We again take the **scheme8** used for this as an example:

In this **scheme8** those numbers run on level 3 from 1 [...] to 8 [...], on level 2 from 1 [...] to 4 [...] and on level 1 from 1 [...] to 2 [...]. The question is how the code gets these numbers in the right place. And again the key is hidden in the position code (pscd). If you look in the pscd for the binary numbers for all numbers 2 of the entire **scheme**, you will see the following information in the corresponding series: 100 = 4, 10 = 2 and 1 = 1 with their decimal number behind it: 4.2, 1.





It then follows that the position code itself does not give the correct information about the ranking number (it should always be 2 here) of these treenodes.

And yet that information is hidden in it.

Take the pscd binary number from node point 2 at depth 3. It reads 100 and is decimal 4, read from right to left by default. Now read the binary position code number at depth 3 from left to right, starting at the correct position (here depth 3). Then it says $1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 = 1$. And those of depth 2: $1 \times 2^0 + 0 \times 2^1 = 1$, and of depth 1: $1 \times 2^0 = 1$. Now the results are the same, namely 1.

If we now put the other decimal value read from left to right in the entire diagram above, instead of the current decimal value, you will see that this always, at any depth, is part of the nicely increasing series of 0,1,2,3,4,5, etc.

Since schedules always start to count with 1, we have to add the value 1 for each result to make the ranking numbers correct. As I mentioned at the beginning of this article, it was a happy choice to choose the Up towards the value 0. This makes the ranking numbers easily in the right place.

The ranking numbers are in the node info string in the diagram just before the brackets. For example with (7) 2[...] —

In code it will look like this:

```
(** GiveRank convert the pscd byte value into a tree depth *
 * dependent ranking value that is numbered ascending *
 * from top to bottom, starting with the value 1. date: 1993 **)
```

```
FUNCTION GiveRank(Apscd, Adpth: Byte): Word;
var i, j : Word;
BEGIN
  j := 0;
  // If from Hi to Lo the bit in the pscd has the value one ...
  // ... then sum the read byte value with function result.
  for i:= Adpth downto 1 do
    // read the bit value from left to right out !
    if BitSet(i,Apscd) then
      Inc(j,SetBit((Adpth-i+1),0));

  Result := j+1;
  { compensate the function result for the start value zero}
END; { GiveRank }
```

GRAPHIC LINE ELEMENTS

Now something about the graphic line elements used. These are not ordinary **AnsiChar** elements but unicode **WideChar** characters. They are declared as constants in the interface section of the relevant unit so that they are available for all code in the implementation section of that unit. And of course for any other code if this unit is included in the **uses** clause.

```
const vrt : widechar = #9474; {= |}
      hrz : widechar = #9472; {= -}
      hup : widechar = #9496; {= +}
      hdn : widechar = #9488; {= +}
      hvl : widechar = #9500; {= +}
```





Where does this Encoding information come from? After some research on the internet, the following website was found:

unicode-table.com/en/#box-Drawing

What is a widechar?

Widechar character

A wide character is a computer character datatype that generally has a size greater than the traditional 8-bit character. The increased datatype size allows for the use of larger coded character sets.

A wide character is a 2-byte multilingual character code. Any character in use in modern computing worldwide, including technical symbols and special publishing characters, can be represented according to the Unicode specification as a wide character.

WideChar

A variable of type WideChar, which has a synonym of UnicodeChar (type UnicodeChar = WideChar;), is exactly 2 bytes in size, and usually contains one Unicode character in UTF-16 encoding. As it is impossible to encode all Unicode code points (a code point normally corresponds to a character) in 2 bytes, two WideChars may be needed to encode a single code point.

As of version 3 of Free Pascal, the Char datatype is a synonym for an AnsiChar. However, in the future the Free Pascal compiler may consider Char a synonym for WideChar.

Unicode chars

When printing a **scheme**, we use AnsiChar (= 1 byte) characters for the plain text and WideChar (= 2 byte) for the line elements. Actually, that is not done because you should not actually put them together because you could lose half of the WideChar. If you still want to merge them, the Pascal compiler of the ShortString makes a WideString so that all information is retained.

In the message window of the Lazarus IDE the following text will appear:

Warning: Implicit string type conversion from "ShortString" to "WideString"

Below a code example of which the compiler issues a warning in the message window:

```

(* Question: have line elements to be placed further on as a rule? *)
for i := {1}2 to Givedpth do
BEGIN
  { is there a sign change (0=>1 of 1=>0) at position i in the position-code?}
  if (BitSet(i,Givepscd) <> BitSet(i-1,Givepscd)) then
  BEGIN { concat prnstr after s with the graphical element | in between }
463   prnstr := Concat(s,vrt,prnstr); (|) ( plak prnstr achteraan s )
  intern := TRUE;
465   END
  else { concat prnstr after s with a space char instead of the graphical }
      { element | in between }
      if intern then prnstr := Concat(s,' ',prnstr);
  END; {for}
    
```

Messages

...Compile Project, Target: ObjectTree.exe: Success, Warnings: 21, Hints: 1

- ▲ Stdtree.pas(463,34) Warning: Implicit string type conversion from "showstr" to "WideString"
- ▲ Stdtree.pas(463,40) Warning: Implicit string type conversion from "AnsiString" to "WideString"
- ▲ Stdtree.pas(463,27) Warning: Implicit string type conversion with potential data loss from "WideString" to "AnsiString"

fig 28: warning: Compiler warnings about ShortString en WideString "abuse"





Since Delphi 7 (2007), Delphi has been fully standardized for the use of unicode chars. For Lazarus you need to add the lazUTF8 unit to the uses clause if you want to use unicode chars in your code:

```
unit FTree;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, Forms, Controls, Graphics,
  Dialogs, StdCtrls, lazUTF8;
...trls, lazUTF8;
...
end;
```

THE TREE PROGRAM

Now that all this preliminary work has been done, the code can be executed. A simple program - classtrnd-has been written for that. With which the depth to be printed can be set to a maximum of eight and the schedule, printed in different ways, is written to a text file. In addition, it can also be sent as a demonstration to the canvas of the form.

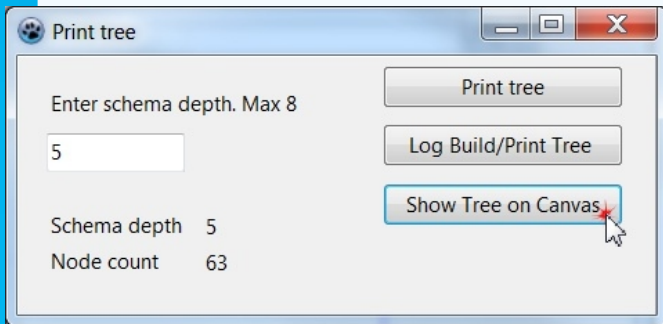


fig 29: The classtrnd program

The form has two scroll bars that allow you to scroll to the invisible portion of a schedule when the **scheme** is too large for the window.

The final but important code
Now that the schedule has been printed, the work is done and all the memory used must be released again before the program can be closed. That also happens recursively and again in a different order. First you go to the nodes themselves to check there whether both the Up and Dn fields are Nil and then the memory for that node itself can only be released. Hence the name PostOrder.

In the old code, the release of the memory happens with the counterpart of New, Dispose. Dispose () is called 2x here. The first time, the destructor ndx.Done of the index is called first, which clears the memory of the index and then the pointer itself is released with Dispose command. The second Dispose releases the memory for the node itself.

The old code

```
// TTr.Destroy 1993
// Destructor for tree's nodes
// developed 20 februari 1993

DESTRUCTOR TTr.Done;

PROCEDURE PostOrder(reflink : PTrnd);
BEGIN
  if reflink <> NIL then
  BEGIN
    PostOrder(reflink^.Up);
    PostOrder(reflink^.Dn);
    // both nodes (Up and Dn) are Nil so:
    Dispose(reflink^.ndx,Done);
    // first ndx.Done, then the reflink
    Dispose(reflink);
  END;
END;

BEGIN
  PostOrder(treestart);
END;
```

Because the new code combines the two old records into one Class, the memory used by the node can also be released at once. This happens when using a Class using the Free method. Finally, the ancestor Class memory itself is released by calling the reserved word Inherited.

```
(*-- TTr.Destroy -----2020--*
 * Destructor for tree's nodes
 *-----**)

DESTRUCTOR TTr.Destroy;

PROCEDURE PostOrder(reflink : TTrnd);
BEGIN
  if reflink <> NIL then
  BEGIN
    PostOrder(reflink.GetFUp);
    PostOrder(reflink.GetFDn);
    // both nodes (Up and Dn) are Nil so:
    reflink.Free;
  END;
END;

BEGIN
  PostOrder(GetFRoot);
  inherited;
END;
```

And with this all the used memory is released and the program can be closed.





LAZARUS SPECIAL EDITION 2.0.6

THIS PROGRAM IS FREE (YOU CAN DOWNLOAD IT HERE:
(<https://www.blaisepascalmagazine.eu/9372-2/>)
or from our website

Colourbuttons,(HS) Free including Code,

Webcore (TMS) Free, fully functional, no code

kbmMemtable(Standard Version)

Components4DevelopersFree, fully functional, no code

You can unpack this zip file and simply copy it to any directory,
even a USB stick and it will work.

You can compile & add other components to this version.

Do NOT do: CleanUp and Build from the Lazarus Menu.

Then you will damage the files for this version, because it can NOT
recompile the sources.

But you probably will never have to...

Be sure to have a copy of this on your system

IMPORTANT:

Request a trial license:

<https://www.tmssoftware.com/site/trialkey.asp>

You need to install the TMS Webcore Trial

TMSWEBCoreXE12_BIN.zip

You can install only Webcore for Lazarus or install it as well for Delphi.

Lazarus TMSDemo Projecten

(<https://www.blaisepascalmagazine.eu/wp-content/uploads/2019/12/>

LazarusTMSDemoProjecten.zip) or go to our website

Align \ Anchors \ Bootstrap \ DataModule \ Dataset \ DBGrid \
EditAutoComplete \ FilePicker \ Formhosting \
Forminheritance \ Frames \ GridPanel \ HTML \ ImageZoom \
IndexedDB \ MainMenu \ MessageDialogs \ Multiform \
PaintBox \ Pictures \ PushNotifications \ Regular Expressions \
ResponsiveGrid \ ResponsiveGridPanel \ RichEditor \ Simple \
ableControl \ Themes \ Treeview \ Accordion \ Upload \
WebCrypto

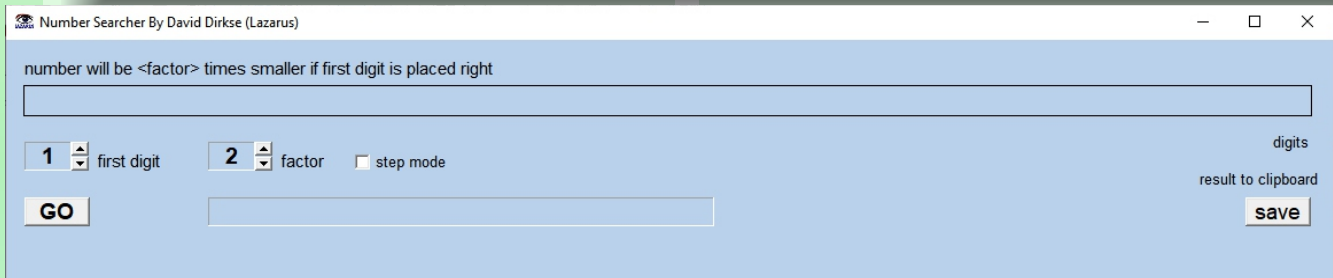
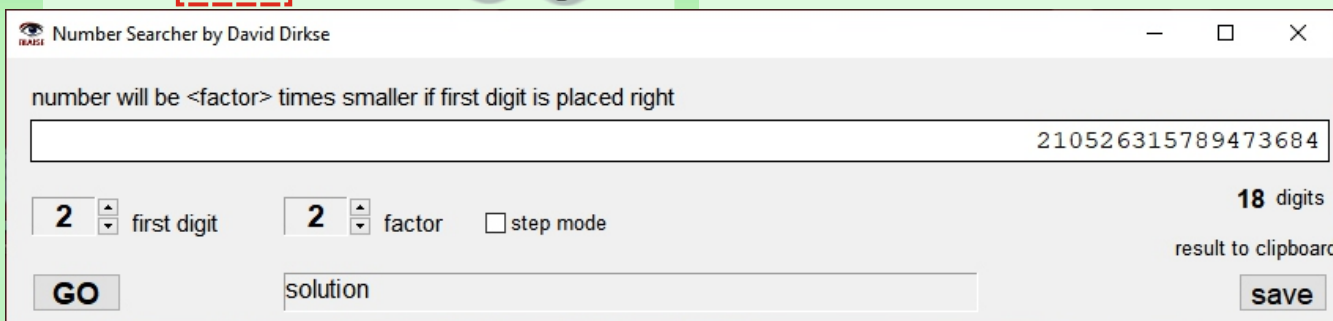
Lazarus206KbmMemtable_ChangeTool

<https://www.blaisepascalmagazine.eu/wp-content/uploads/2019/12/>

KbmMemtable_ChangeTool.zip

starter

expert



There is also a Lazarus version

INTRODUCTION

A well known math puzzle is this:
A number has 2 as the lowest digit.
If this digit is moved to the left of the number the effect is multiplication by 2.

Similar: A number has 2 as the leftmost digit.
If this digit is moved right of the number the effect is division by 2.

The funny thing about this puzzle is that it requires only primary school calculation however even math teachers have problems solving it.

After some tries by hand the question arises if such a number exists anyhow.

Also : are there digits other then 2 possible and : are there other factors possible than 2?

This article describes a Delphi program that searches for number having these properties.

The algorithm.

For explanation we use digit 2 as a start and also a multiplication factor of 2.

Note: a carry is written as (..)

To start 2
 $2 \times 2 = 4$ 42
 $2 \times 4 = 8$ 842
 $2 \times 8 = 16$ (1) 6842
 $2 \times 6 = 12 + 1 = 13$ (1) 36842
 $2 \times 3 = 6 + 1 = 7$ 736842
 Finally 105... 736842
 $2 \times 1 = 2$ 2105... 736842

Now the first and the last digit are equal (and no carry exists)

Discard the lowest "2" digit and we have found a number that divides by 2 if the first digit is moved right of the number. The picture above shows the program at work. UpDown controls, associated with statictext components, select the first digit and the multiplication factor.

The **GO** button starts the search.

The **SAVE** button copies the result to the clipboard, which allows for pasting in text editors.

If step mode is checked, the program stops after each iteration, showing intermediate results. The number is stored in byte array $A[0...120]$ Integer N is the index of array $A[]$.

Initialization:.

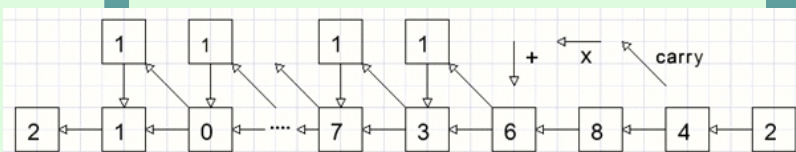
$N := 0;$
 $A[0] := \text{start digit}.$

Step 1:

$A[N]$ is multiplied by 2, the result placed in $A[N+1]$ (lower digit) and $A[N+2]$ (carry).

Step 2:

$N := N + 1;$
 Steps 1 and 2 are repeated until $N = 120$ (no solution) or $A[N] = A[0]$ and $A[N+1] = 0$ (no carry).



This code does the job:

```
.....
procedure TForm1.GObtnClick(Sender: TObject);
var carry,i,f,h,N : byte;
    hit : boolean;
begin
  activecontrol := nil;
  for i := 0 to maxN do A[i] := 0;
  displayA(0);
  A[0] := NUpDown.Position; //starting digit
  f := factorUpDown.Position; //factor
  //--
  N := 0; //array A index
  repeat
    inc(N);
    A[N] := A[N-1]*f + A[N];
    h := 0;
    while A[N+h] >= 10 do //handle carries
      begin
        carry := A[N+h] div 10;
        A[N+h] := A[N+h] mod 10;
        inc(h);
        A[N+h] := A[n+h] + carry;
      end;
    hit := (A[N] = A[0]) and (h=0);
  until (N = maxN) or hit;
  //--

  if hit then begin
    msgText.Caption := 'solution';
    displayA(N);
  end
  else begin
    msgText.Caption := 'no solution';
    label4.Caption := "";
  end;
end;
```

At counter p = 3 ". " is placed for clarity, separating triple digits.

```
var s : string;
    i,n,p : byte;
begin
  s := "";
  n := maxN;
  while (A[n] = 0) and (n > 0) do dec(n);
  //find first non zero digit
  p := 0;
  for i := n downto 1 do
    begin
      if p = 3 then begin
        s := s + '.';
        p := 0;
      end;
      s := s + char(A[i] + ord('0'));
      inc(p);
    end;
  end;
```

Step mode

If **stepMode** is selected, after each iteration then variable **stopflag** is set **true** followed by :

```
while stopFlag do
  application.ProcessMessages;
Pressing <SPACE BAR> resets the stopflag so
the proces continues.
```

This needs the **keyPreview** property set to **true** in the **form1 object inspector**.

The search procedure started with

```
Activecontrol := nil.
```

This prevents the <SPACE> character being send to the GO button which has the focus after pressing.

Display of **array A[]** use a paintbox for display. Reason is that this allows for the display of digits in different colors.

Carries are displayed in red.

Courier new font is used, having characters with a fixed width.

A[1] is placed right, character position x is decremented to display the next characters left. procedure **displayA (n : byte)**; does the job. **n** is the number of digits.

Any non zero character from a higher index of n from **A[n]** is painted in red.

Saving the result to the clipboard.

Clipbrd unit is added to the **uses** clause.

With the result saved in string s, this statement does the job:

```
clipboard.AsText := s;
```

But first **A[]** must be copied to **s**.

The highest digit is transferred first.

Please refer to the source code for more details.

An astonishing property of the answer

(**for factor 11**) is this:

Split the number in halves and add then.

The result shows only "9" digits: 99.....99.





Official Embarcadero partner in Benelux,
France and French speaking countries

Contact us for free advice, promotions and quotations,
license management questions,
hiring developers,
training, consultancy and much more!

Watch our new French YouTube channel with training
video's:

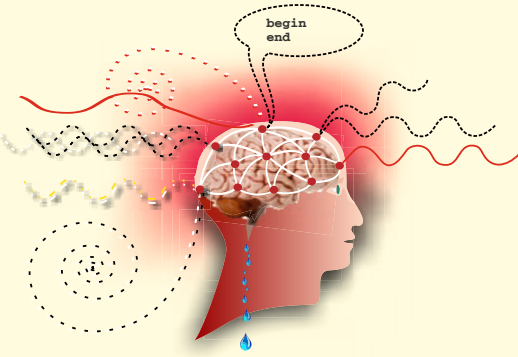
Regardez notre nouvelle chaîne YouTube en français. Vous
y trouverez des vidéos de formation pour Delphi



<https://bit.ly/3aMG8sx>

www.barnsten.com / info@barnsten.com
France: Téléphone +33 (0)9 72 19 28 87
Benelux: Telefoon +31 (0)2 35 42 22 27





MIND-READING AI IS ABLE TO TRANSLATE THOUGHTS INTO WORDS USING A BRAIN IMPLANT.

This article is based on a publication in "NewScientist"

Some types of Artificial Intelligence can accurately translate thoughts into sentences of limited size through their learning patterns - a vocabulary of ± 250 words.

The system can greatly help us make a start in speech recovery, for example for people cut off from communication for any reason.

Joseph Makin and a number of colleagues from the **University of California at San Francisco** used in-depth learning algorithms to study the brain signals of multiple women while speaking language.

The women, who all have epilepsy as a clinical picture, were already in the condition that electrodes had been inserted into their brains because of their illness in order to be able to monitor possible attacks.

Each patient individually was asked to read a number of sentences while the Development Team recorded the brain activity.

The vocabulary was based on 250 unique words, spoken in complete sentences.

The **OT (Research Team)** fed this brain activity data to a **neural network algorithm**, and through an ever-repeating pattern, the **AI network** attempts to discover frequently occurring patterns that can be linked to repetitive speech properties, such as vowels, consonants, timbre, hiccups and coherence of the entire sentence.

These previously recorded patterns were then presented to another neural network also intended for this task, attempting to convert them into words and form a sentence.

Each patient repeated the sentences at least twice with the last repetition not being part of the previously obtained training data, allowing the system to be put to the test:

When a person speaks the same sentence repeatedly, the corresponding brain activity will be similar but not one-on-one.

It is not really important to remember the brain activity of these sentences. The **AI network** has to register what is comparable, so that it can extrapolate the meaning and content. In the four patients, the best softening of the **AI network** turned out to make an average number of errors of only 3 percent.

The use of a small number of short sentences makes it easier for the AI network to discover syntactic similarities.

The team tried to convert the brain signals into individual words rather than whole sentences, but that resulted in an increased error rate of 38 percent. The **AI network** thus clearly teaches facts about which words fit together, and the conclusion is not just which neural activity is mapped to which words.

Unfortunately, this makes it difficult to expand the system to a larger vocabulary as each new word produces more possible sentences, which is bad for the overall learning curve.

Presumably, it will have to go the same length of time as **OCR** that also took years to get to great accuracy - and this is a much more difficult task.

About 250 words can of course make sense for people who cannot speak.

The average word knowledge per person is much greater. The average person's vocabulary develops from about 300 words at two years of age, through 5,000 words at five years old, to about 17,000 words at 12 years of age.

It stays around this number of words for the rest of most (average) people's lives, with the result that this is about the same number of words as the words recorded by a popular newspaper in the course of producing the daily editions, while a graduate might have a vocabulary that is almost twice as large (23,000 words).

Shakespeare, had one of the largest recorded vocabulary of an English writer at about 30,000 words. But him we cant make speak again.





PREFACE

One of the very nice things about users is that they sometimes request features that I did not originally think about.

This article explains a new one of those, namely binding to **TStrings** (including **TStringList** and other **TStrings** descendants). The shown features will be available in the upcoming release of kbmMW.

Binding to a TStringList

First let us see what it can do, then I will show how to do it:

So how is it done?

First we create a TStringList:

```
FStrings:=TStringList.Create;
FStrings.AddObject('Name1=Item 1',TObject(1000));
FStrings.AddObject('Name2=Item 2',TObject(2000));
FStrings.AddObject('Name3=Item 3',TObject(3000));
FStrings.AddObject('Name4=Item 4',TObject(4000));
FStrings.AddObject('Name5=Item 5',TObject(5000));
Binding.DefineData('strings',FStrings);
```

I have also defined a placeholder for the **TStringList** instance, called strings. This makes it easy to access it later on, and even to replace it with another **TStrings** descendant on the fly.

This time I show how to bind from code, but using the textual expression method:

```
Binding.Bind('{@strings.#strings, to:Edit10.Text, twoWay:true}',self);
Binding.Bind('{@strings.#name, to:Edit11.Text, twoWay:true}',self);
Binding.Bind('{@strings.#value, to:Edit12.Text, twoWay:true}',self);
Binding.Bind('{@strings.#objects, to:Edit13.Text, twoWay:true}',self);
```

The four **TEdit**'s could have been bound up using design-time binding as have been shown earlier, or by binding specifically to the **FStrings** variable. But I do recommend using the placeholder technique since it makes it so easy to separate binding from data, and thus to replace data with something else.

As you probably have noticed, the source reference in the binding use a syntax like **@strings.#strings**. Obviously we want to bind between the placeholder strings (which refers to the **FStrings:TStringList** instance), and the Edit controls.

To tell what part of the **TStrings** item we want to bind to, we refer to either **TStrings**'s properties directly, or what is more relevant in this case, we refer to a couple of specialized binding members, namely **#strings**, **#name**, **#value** and **#objects**.

#strings refers to that we want to bind to the complete strings value (name, separator and value if such are defined), for the current position by referred to by the bindings navigator.

#name refers to that we only want to bind to the name part.

#value refers to that we only want to bind to the value part.

#objects refers to that we want to bind to the objects property. It is treated as an integer or int64 value depending on if you are running on a 32 or 64 bit CPU. The reason for this, is that the Objects property often is used as a strings item integer tag value.

Remember... if you like **kbmMW** or what you read here, share it with your friends and colleagues.

Also remember you can get going totally for free by downloading kbmMW Community Edition, which can be used for teaching, personal use and even commercial use (terms apply). kbmMW

Community Edition do not include source, and only supports latest version of Delphi in 32 bit mode, it however contains most features found in kbmMW Enterprise Edition except those that require compilation of the kbmMW source code.

Happy binding

Kim/C4D



Independent Tests of Anti-Virus Software



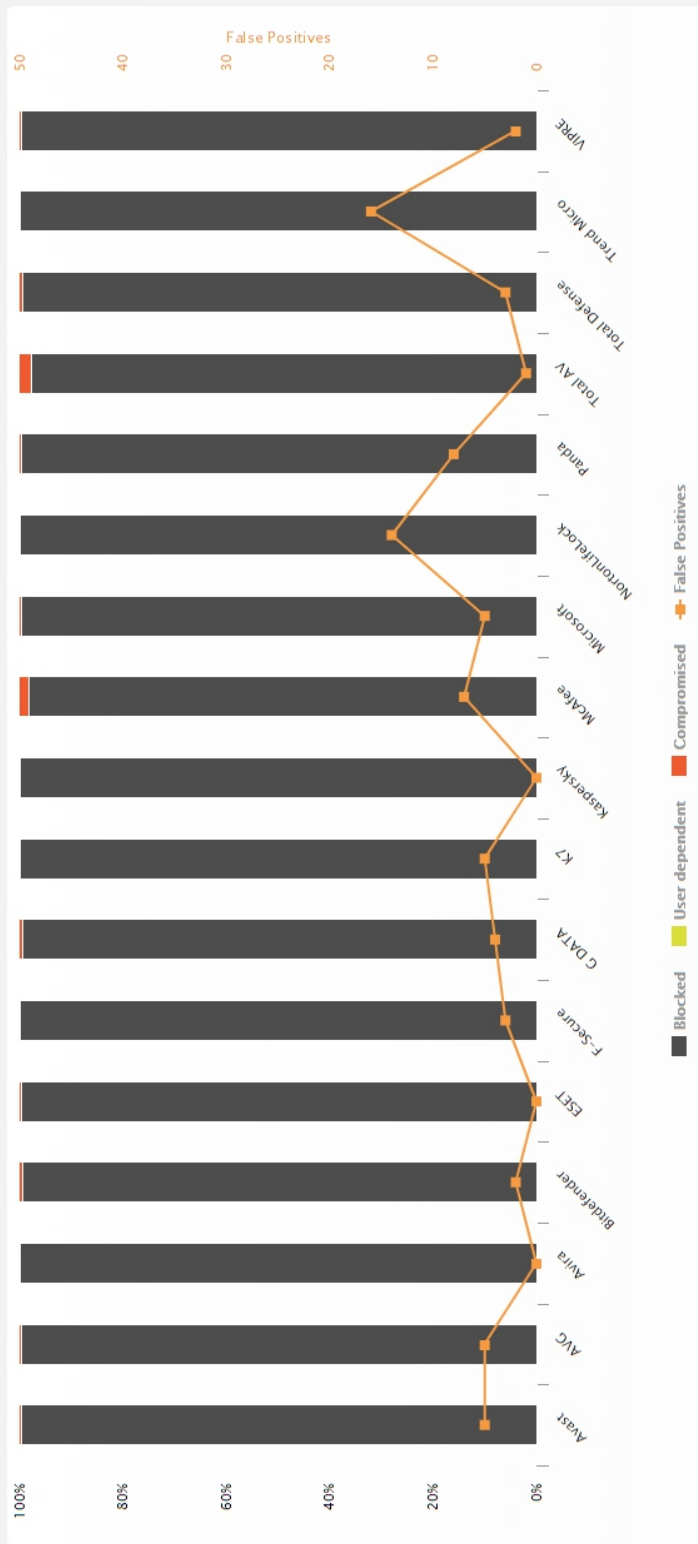
- Search
- Press
- About
- Wiki
- Blog
- Awards

Switch to Enterprise Area

- Report
- Export

Consumer Test Charts

Consumer ▼ Real-World Protection Test ▼ 2020 ▼ by vendor ▼ 0 - 100% ▼



Do you want to create a link to this chart with the selected data? [Create Link](#)

Terminology:

- Blocked ... Malware was successfully blocked by AV
- User Dependent ... The user had the option to allow the execution of the malware
- Compromised ... Malware compromised the system
- False Positive ... A clean sample was wrongly detected as malicious

We would like to point out that while some products may sometimes be able to reach 100% protection rates in a test, it does not mean that these products will always protect against all threats on the web. It just means that they were able to block 100% of the widespread malicious samples used in a test.



Hot off the press



FastMM5 changes towards commercial licensing. Be aware of the new rules

FastMM5

Homepage: <https://github.com/pleriche/FastMM5>

FastMM is a fast replacement memory manager for Embarcadero Delphi applications that scales well across multiple threads and CPU cores, is not prone to memory fragmentation, and supports shared memory without the use of external .DLL files.

Version 5 is a complete rewrite of FastMM. It is designed from the ground up to simultaneously keep the strengths and address the shortcomings of version 4.992:

- Multithreaded scaling across multiple CPU cores is massively improved, without memory usage blowout. It can be configured to scale close to linearly for any number of CPU cores.
- In the Fastcode memory manager benchmark tool FastMM 5 scores 15% higher than FastMM 4.992 on the single threaded benchmarks, and 30% higher on the multithreaded benchmarks. (I7-8700K CPU, EnableMMX and AssumeMultithreaded options enabled.)
- It is fully configurable runtime. There is no need to change conditional defines and recompile to change options. (It is however backward compatible with many of the version 4 conditional defines.) Debug mode uses the same debug support library as version 4 (FastMM_FullDebugMode.dll) by default, but custom stack trace routines are also supported. Call FastMM_EnterDebugMode to switch to debug mode ("FullDebugMode") and call FastMM_ExitDebugMode to return to performance mode. Calls may be nested, in which case debug mode will be exited after the last FastMM_ExitDebugMode call.
- Supports 8, 16, 32 or 64 byte alignment of all blocks. Call FastMM_EnterMinimumAddressAlignment to request a minimum block alignment, and FastMM_ExitMinimumAddressAlignment to rescind a prior request. Calls may be nested, in which case the coarsest alignment request will be in effect.
- All event notifications (errors, memory leak messages, etc.) may be routed to the debugger (via OutputDebugString), a log file, the screen or any combination of the three. Messages are built using templates containing mail-merge tokens. Templates may be changed runtime to facilitate different layouts and/or translation into any language. Templates fully support Unicode, and the log file may be configured to be written in UTF-8 or UTF-16 format, with or without a BOM.
- It may be configured runtime to favour speed, memory usage efficiency or a blend of the two via the FastMM_SetOptimizationStrategy call.

Once payment has been made at <https://www.paypal.me/fastmm> (paypal@leriche.org), please send an e-mail to fastmm@leriche.org for confirmation. Support is available for users with a commercial licence via the same e-mail address.



BY KIM MADSEN

starter expert DX 



Some people may have wondered if I have fallen off the face of the earth as I have been less vocal the last couple of weeks.

It has nothing to do with the dreadful COVID-19 infection I suppose most of us, one way or the other, are affected by. It rather has to do with being overly busy with various things.

One of the things, that relate to **kbmMW** and **kbmMemTable**, is the development of a brand new **COMPILETOOL** which will be included in next release of **kbmMemTable** and **kbmMW**.

The purpose of the CompileTool is ... TA DAAA.... to compile stuff

So what's new about that? Not really much... but let me explain the rationale behind my apparent brain damage.



THE COMPILE TOOL

Compiling and installing **kbmMemTable**, has in **Delphi** always been fairly easy. In **C++Builder** only mode, not so much, partly because the **C++** only environment diverge more and more from what **kbmMemTable** originally supported, and the matching **C++** project files.

To complicate matters even more, **kbmMW** can be a pain to install in **Delphi** due to **kbmMW's** ability to seamlessly integrate with loads of 3rdparty stuff. Paths and requirements and more needs to be provided. In **C++** only mode it is even more complex to get it going, and despite the Compile Tool helping much on the situation, it is not fully solved with **kbmMW** yet, because **C++Builder** exhibits random crashes and unexplained compile/link errors (internal errors).

But one of the things **Delphi** did very well.. in the early days (I suppose until **XE** got to see the light), was to consistently tell you that some 3rdparty packages should be referenced to compile **kbmMW's** packages nicely.

Delphi was even nice enough to add the relevant requirements to the **kbmMW** package so everything just worked.

Unfortunately, **Delphi** has stopped to do that stably since many years. I have reported it to Embarcadero on numerous occasions, and they have acknowledged the issue, but have not been able to figure out why it has stopped working. Mind you.. sometimes it works... but then suddenly it does not, usually when that happens, it stops working for good in my experience.

So the Compile Tool has as goal to:

- Know everything about the units and requirements of the project in hand
- Be able to manage knowledge about 3rdparty libraries/packages
- Produce correct valid project files for the project that the Compile Tool has been prepared for.
- Compile and install the projects automatically
- Be able to recompile and restart itself, so it is up to date with whatever settings you may have made in for example **kbmMWConfig.inc** and thus based on those settings, is able to produce correct project files.

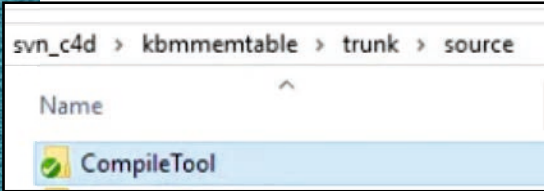
At first I created it to make compilation and installation of **kbmMW** easier, but it soon dawned to me that **kbmMemTable** should be supported too (standalone), and that it as such could be a generic tool that will work for other developers too.

(Currently it is not released with license for other 3rdparty developers to use it, but ping me if you have interest in that.)



COMPILE TOOL FOR KBMMEMTABLE

Let us have a look at the **Compile Tool** for **kbmMemTable**. It will usually be found, as **CompileTool.exe**, in the source directory of the project for which it is supposed to support. Further the source of the **Compile Tool** will be found in the subdirectory **CompileTool** under the managed projects source directory.



If you loose **CompileTool.exe**, it can be recompiled by opening and building the project in the **CompileTool** directory.

You can not use the **CompileTool source/executable** from a different project (like **kbmMW**), because they contain different settings, specially in the **uCompileToolFeatures.pas** file, which is specialized for each project.

If you start **CompileTool.exe** on a computer on which **Delphi**, **C++Builder** or **RAD Studio** is not installed, you will get an exception and the tool will shut down.



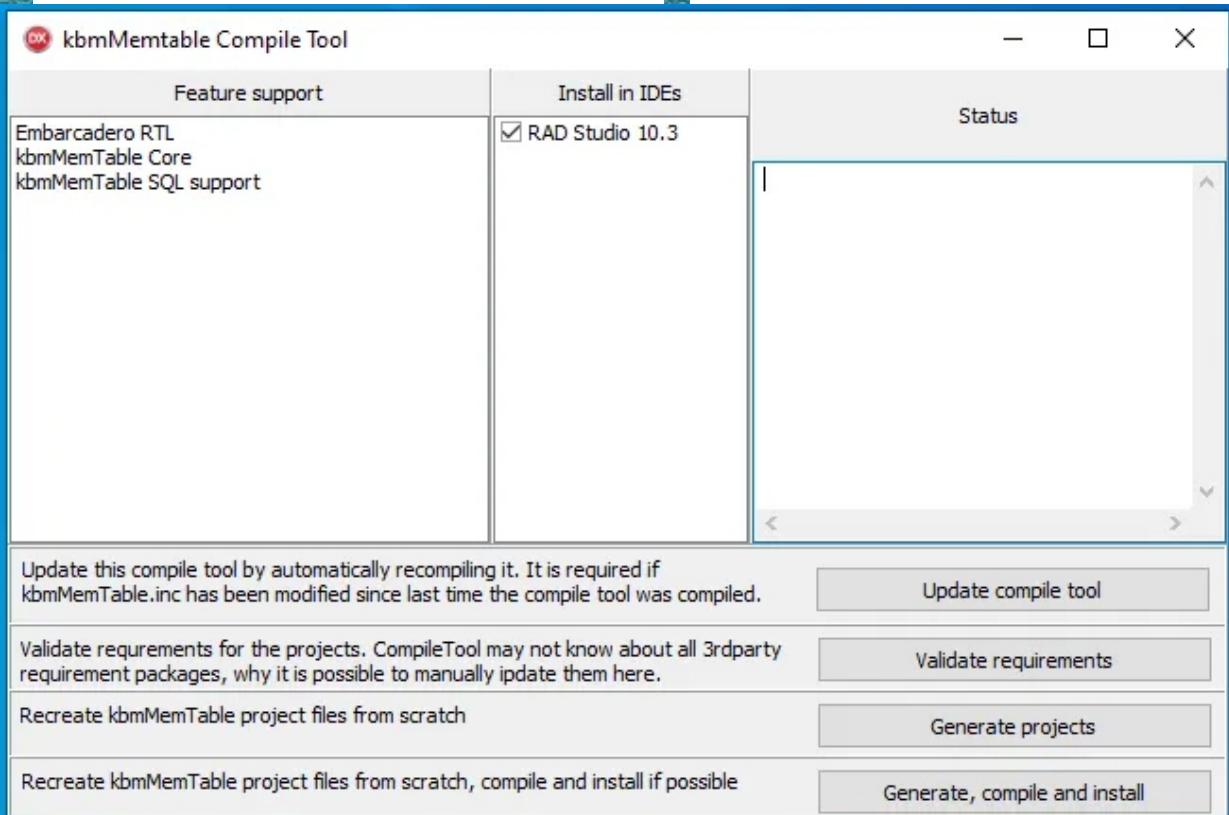
So let us start it on a computer with **RAD Studio** installed.

On the left side, a list of supported features for the project, is listed. In the case of **kbmMemTable** it is pretty simple, and will not change, since all features are available for all versions of **kbmMemTable**. However look later for how the **Compile Tool** for **kbmMW** looks.

In the center, the discovered versions of **Delphi**, **C++Builder** or **RAD Studio** is listed. You can select, in which of them, **kbmMemTable** should be installed.

On the right side, you can follow the status of what is happening.

At the bottom, various buttons are available.



1 Update compile tool – Will rebuild the Compile Tool itself, and restart it with the newly compiled executable. For kbmMemTable it is not often needed to do, but in the case of kbmMW, you will want to do that, everytime you have changed something in **kbmMWConfig.inc**.

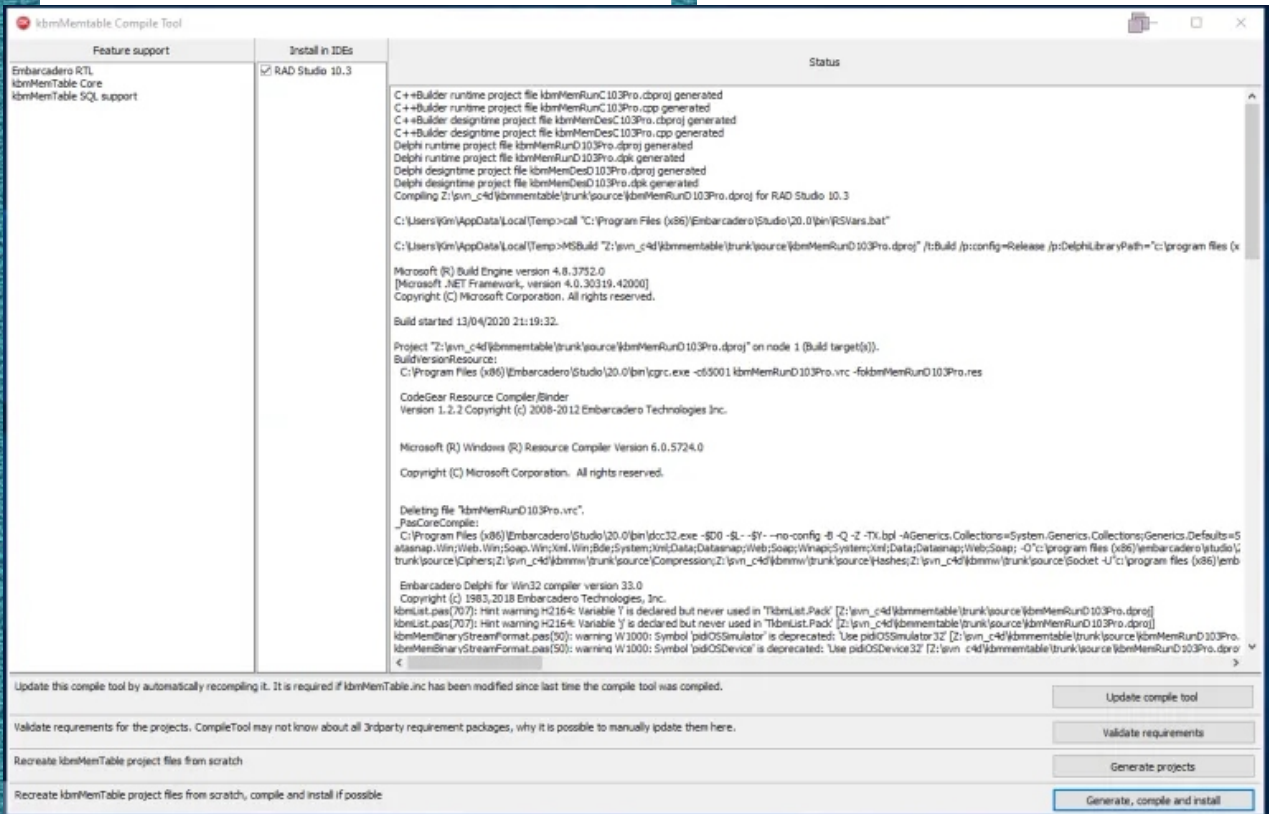
2 Validate requirements – It will show a dialog, where 3rdparty requirements, which the Compile Tool has not been able to resolve itself, can be defined. The definitions made here will be remembered for next time in the file **CompileTool.ini**, making it easier to recompile/install without having to reconfigure each time. The **.ini** file will never be overwritten by kbmMemTable or kbmMW installations. kbmMemTable will usually not need any settings in this dialog. See section about kbmMW installation further down for more information.

3 Generate projects – It will produce new kbmMemTable project files matching the selected IDE.

The project files will automatically be written to the parent directory (which is the kbmMemTable source directory).

4 Generate, compile and install – It will generate projects, as described above, compile the projects and automatically install the resulting packages in the IDE, for all selected IDE's. Usually you will be required to close the selected IDE before being able to compile and install. You can start with the option **-F: CompileTool.exe -F** to override the requirement to stop the IDE. However the packages will not show up until you restart the IDE later on, and if the packages already was in use by the IDE you will get compile/linker errors. This is an example.

The result. You may notice that there are various paths shown in the status. Those paths are automatically picked up from your current installation, and provided for the compiler by the Compile Tool.



After it succeeds, you can close the tool, and start the IDE. Now kbmMemTable will be available and installed with all paths for Windows 32 compilation, correctly setup automatically.

So let us look at how it works when installing kbmMW.



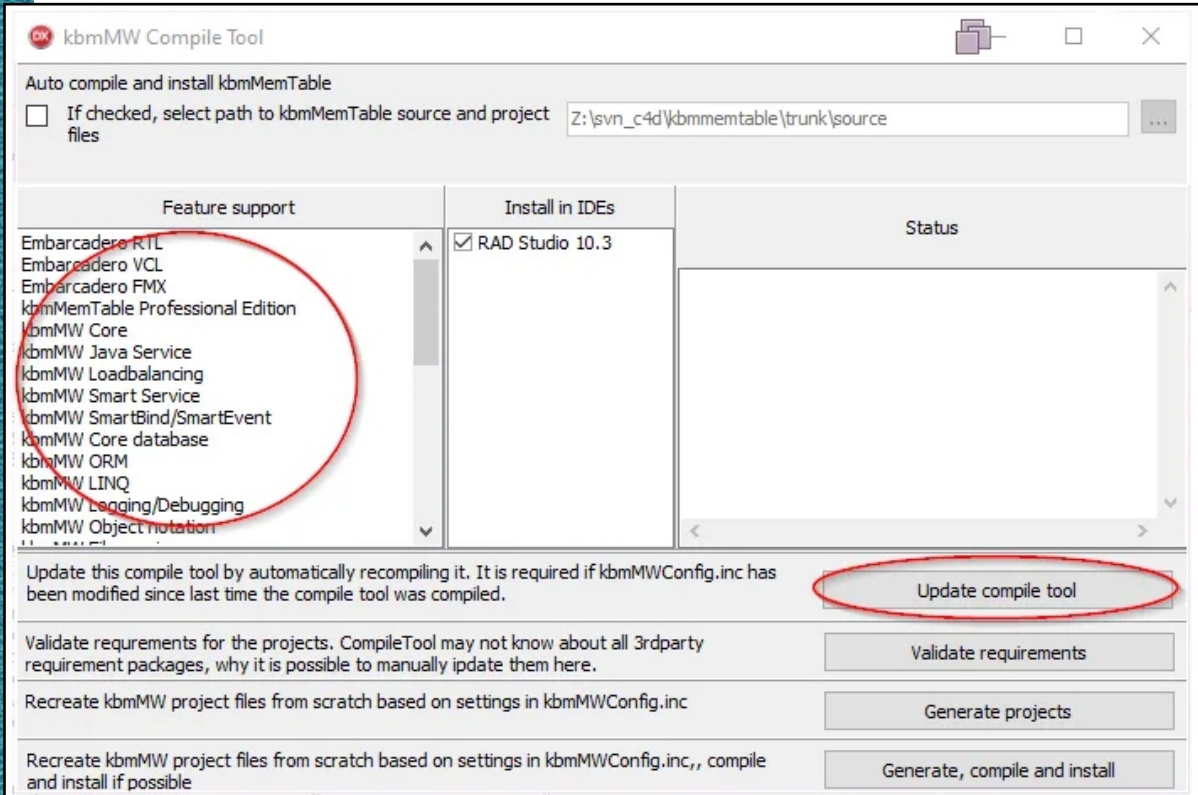
COMPILE TOOL FOR KBMMW

In this case, I have, for the demo, opened `kbmMWConfig.inc`, uncommented the line:

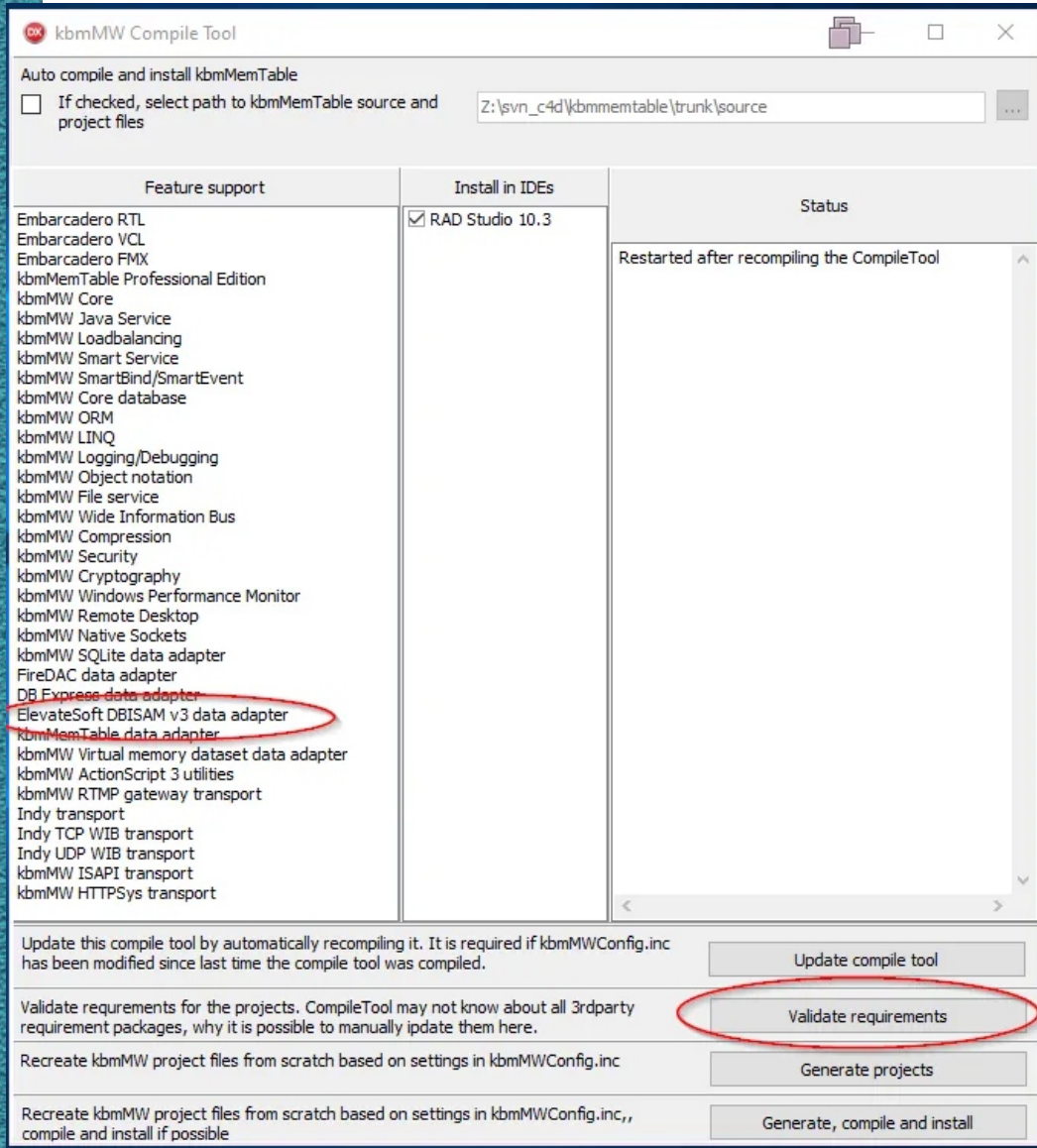
```
{$DEFINE KBMMW_DBISAM3_SUPPORT} // DBISAM 3 support.
```

and saved the file again, which essentially tells kbmMW that we want full support for DBISAM v3. (FTR there are similar defines for 36 other databases too, incl. DBISAM v4, ElevateDB, NexusDB and many many more).

Since this is a new setting, I first start the Compile Tool, where the Feature support at this time do not include DBISAM3, and let it recompile itself.

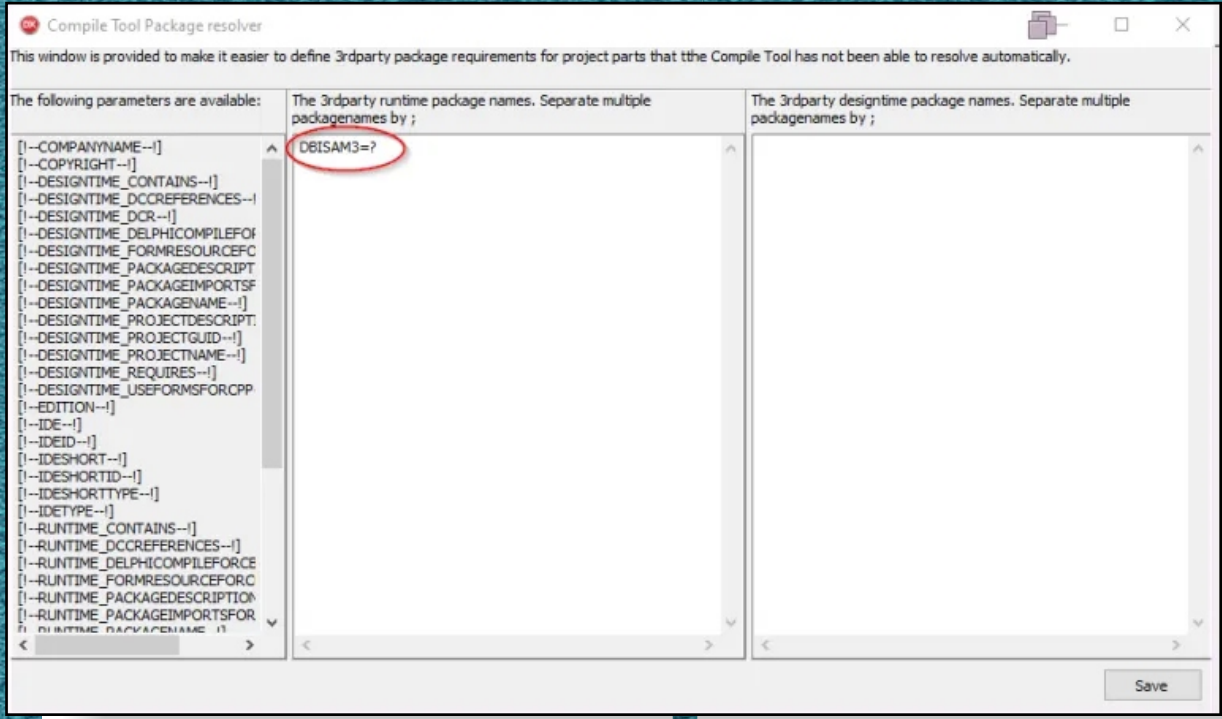


When it restarts, it looks like this:



Now the Compile Tool recognize the request for feature support for ElevateSoft DBISAM v3. Since it is a new 3rdparty tool that kbmMW should support, compared to previous settings, we want to check the requirements dialog by clicking on the Validate requirements button.





You can see that there are a runtime package requirement that is currently unresolved. To resolve it, simply type the name of the DBISAM v3 runtime package, typically something like db300d20 or something along those lines. As the structure of the package name can vary wildly between various 3rdparty projects, it is left for you to type the right value. Remember that this value will be used for all the IDE's that the Compile Tool will compile for.

You can include parameters in the name.
 E.g.
 BISAM3=db300[!--IDESHORTTYPE--!][!--IDESHORTID--!] which will replace [!--IDESHORTTYPE--!] with D for Delphi or C for C++Builder and [!--IDESHORTID--!] with 20 for Delphi 10.3.

If you need to refer to multiple packages/requirements for the DBISAM3 selection, you can separate those with a semicolon ;

Click save, and the Compile Tool will remember your settings, also for next time you start the Compile Tool.

Clicking either Create projects or Create, compile and install, will ensure the kbmMW project files contains the relevant requirements.

Also notice that there is an extra Prerequisites section at the top of the Compile Tools window. It is there because kbmMW requires compilation and installation of kbmMemTable beforehand. You can point out where its source is, and click the checkbox, then it will automatically recompile and install it when you recompile and install kbmMW via the Create, compile and install but.



BEHIND THE SCENES

So what is happening behind the scenes? Well... I told about the `uCompileToolFeatures.pas` file which is special for each project. It is in that file of the Compile Tool sources, where the project specialities are defined.

```
type
TkbmCTFeatures = class
const
    LOWEST_SUPPORTED_BDS_VERSION = 12.0; // XE5
public
class function GetText(const ATextInfo:TkbmCTTextInfo):string;
class procedure RegisterRuntimeFeatures(const AInfo:TProjectInfos);
class procedure RegisterDesignTimeFeatures(const AInfo:TProjectInfos);
class function BuildParameters(const AMain:TfrmMain; const ACpp:boolean; const AIDE:TIDEInfo):TStringList;
class function GenerateProjectFileName(const ACpp:boolean; const AIDE:TIDEInfo;
    const ADesignTime:boolean):string;
end;
```

It contains a class definition which groups a few methods which should be defined for a project.

```
class function TkbmCTFeatures.GetText(const ATextInfo:TkbmCTTextInfo):string;
begin
    case ATextInfo of
        cttiCaption:                Result:='kbmMW Compile Tool';
        cttiRebuildToolCaption:    Result:=
            'Update this compile tool by automatically recompiling it.
            It is required if kbmMWConfig.inc has been modified
            since last time the compile tool was compiled.';
        cttiRecreateProjectsCaption: Result:=
            'Recreate kbmMW project files from scratch
            based on settings in kbmMWConfig.inc';
        cttiRecreateInstallCaption: Result:=
            'Recreate kbmMW project files from scratch based on settings
            in kbmMWConfig.inc,, compile and install if possible';
        cttiPrerequisiteCaption:   Result:=Auto compile and install kbmMemTable';
        cttiPrerequisiteExplanation: Result:=
            'If checked, select path to kbmMemTable source and project files';
    end;
end;
```

The above specifies the texts to be shown and thus can be configured for other projects (like `kbmMemTable` which Compile Tool has a similar section).

```
class function TkbmCTFeatures.BuildParameters(const AMain:TfrmMain; const ACpp:boolean;
const AIDE:TIDEInfo):TStringList;
```

This method builds relevant parameters that must exist for the project file generation. It includes version numbers, project names and descriptions and more. The method will be called multiple times during project generation.




```

class procedure TkbmCTFeatures.RegisterRuntimeFeatures(const AInfo:TProjectInfos);
begin
  AInfo.AddProjectInfo('RTL','Embarcadero RTL','rtl','',true);

  AInfo.AddProjectInfo('VCL','Embarcadero VCL','vcl;vclimg','',true);

  AInfo.AddProjectInfo('FMX','Embarcadero FMX','fmx','',true);

{$IFDEF KBMMW_ENTERPRISE_EDITION}
  AInfo.AddProjectInfo('KBMMEMTABLE','kbmMemTable Professional Edition','kbmMemRun[!--IDE--!]Pro','',true);
{$ELSE}
{$IFDEF KBMMW_PROFESSIONAL_EDITION}
  AInfo.AddProjectInfo('KBMMEMTABLE','kbmMemTable Professional Edition','kbmMemRun[!--IDE--!]Pro','',true);
{$ELSE}
  AInfo.AddProjectInfo('KBMMEMTABLE','kbmMemTable Standard Edition','kbmMemRun[!--IDE--!]Std','',true);
{$ENDIF}
{$ENDIF}
...
{$IFDEF KBMMW_DBISAM3_SUPPORT}
  AInfo.AddProjectInfo('DBISAM3','ElevateSoft DBISAM v3 data adapter','?','kbmMWDBISAM3',true);
{$ENDIF}
...

```

This section defines all the features that can exist in a **kbmMW runtime package**, and their library requirements and units, including the DBISAM v3 option.

AddProjectInfo takes 5 arguments:

- ❶ The unique ID of the project part. For example KBMMEMTABLE. Any ID can be used, as long as it is unique.
- ❷ The descriptive name of the project part.
- ❸ The libraries that are required for the project part, separated by semicolon and without file extensions. If it is an empty string, there are no requirements for that particular project part. If it is a question mark, it is unknown, and thus can be handled by the user in the **Compile Tool package resolver dialog**. It is allowed to include paths if needed, but recommended to only use relative paths from the Source directory. Further it is legal to prefix each library with either < or }. Doing so will ensure to sort the item first (<) or last (}), rather just according to its regular name, when project files are generated.
- ❹ The unit names (without extension) that are to be part of this project part. Multiple unit names can be specified separated by semicolon (;). If the unit also encompasses a form or datamodule file, use this syntax:
unitname=formname:formclass. Eg.
kbmMWCustomJavaService=kbmMWCustomJavaService:TkbmMWSimpleService; }JNI
- ❺ A boolean indicating if a requirement is mandatory for this project part. It is used for validation/warning that the project file may not have been generated correctly, if the requirement value has not been made available.



```

class procedure TkbmCTFeatures.RegisterDesignTimeFeatures(const AInfo:TProjectInfos);
begin
  AInfo.AddProjectInfo('RTL','Embarcadero RTL','rtl','',true);

  AInfo.AddProjectInfo('VCL','Embarcadero VCL','vcl;vclimg;vclx','',true);

  AInfo.AddProjectInfo('IDE','Embarcadero IDE','designide','',true);

  AInfo.AddProjectInfo('FMX','Embarcadero FMX','fmx','',true);

{$IFDEF KBMMW_LICENSE_DATABASE}
  AInfo.AddProjectInfo('KBMMW core database','kbmMW Core database','dbrtl;vcldb','',true);
{$ENDIF}
...

```

This section defines all the features that can exist in a kbmMW designtime package, and their library requirements and units.

It follows the same explanation as for the runtime part.

```

class function TkbmCTFeatures.GenerateProjectFileName(const ACpp:boolean; const AIDE:TIDEInfo;
const ADesignTime:boolean):string;
...

```

The above code is used for creating the relevant project file name for a specific IDE. The file name should not include any file extensions. The method will be called multiple times during project creation.

FINAL PLEA

If you have reached here, then you are qualified to assist me making the Compile Tool better. Since kbmMW supports so many 3rdparty tools, I have for now only specified library requirements in the Compile Tool for some of them. For the remaining, I have left it for you to define, simply because I do not know the naming rules for all the various 3rdparty libraries. Please comment here on this thread if you have some pet libraries that kbmMW supports and that you would like the Compile Tool to know about, preferably with the complete description of how the library name is defined. I.e. what each part of the name consists of.

If the name do not change based on the version of the 3rdparty library, it most likely will be possible to add automatic support in the Compile Tool for that particular library, making it even easier to compile and install kbmMW.



KBMMW PROFESSIONAL AND ENTERPRISE EDITION V. 5.10.20 RELEASED!

- RAD Studio XE2 to 10.3 Rio supported
- Win32, Win64, Linux64, Android, IOS 32, IOS 64 and OSX client and server support
- Native high performance 100% developer defined application server
- Full support for centralized and distributed load balancing and failover
- Advanced ORM/OPF support including support of existing databases
- Advanced logging support
- Advanced configuration framework
- Advanced scheduling support for easy access to multithread programming
- Advanced smart service and clients for very easy publication of functionality
- High quality random functions.
- High quality pronounceable password generators.
- High performance LZ4 and Jpeg compression
- Complete object notation framework including full support for YAML, BSON, Messagepack, JSON and XML
- Advanced object and value marshalling to and from YAML, BSON, Messagepack, JSON and XML
- High performance native TCP transport support
- High performance HTTPS transport for Windows.
- CORS support in REST/HTML services.
- Native PHP, Java, OCX, ANSI C, C#, Apache Flex client support!

kbmMemTable is the fastest and most feature rich in memory table for Embarcadero products.

- **Easily supports large datasets with millions of records**
- **Easy data streaming support**
- **Optional to use native SQL engine**
- **Supports nested transactions and undo**
- **Native and fast build in M/D, aggregation/grouping, range selection features**
- **Advanced indexing features for extreme performance**

- ◆ **NEW! SmartBind now fully supports VCL, FMX, including image/graphics and TListView**
- ◆ **NEW! SmartBind data generators and data proxies for easy separation of data sharing concerns in modular applications**
- ◆ **NEW! SmartEvent for easy separation of event and execution workflow based concerns for the ultimate in modular application design**
- ◆ **NEW! Native highly scalable TCP server transport now also supports REST**
- ◆ **Significant improvements and fixes in many areas including**
 - ◆ RTTI
 - ◆ Scheduler
 - ◆ LINQ
 - ◆ Object Notation
 - ◆ ORM
- High speed, unified database access (35+ supported database APIs) with connection pooling, metadata and data caching on all tiers
- Multi head access to the application server, via REST/AJAX, native binary, Publish/Subscribe, SOAP, XML, RTMP from web browsers, embedded devices, linked application servers, PCs, mobile devices, Java systems and many more clients
- Complete support for hosting FastCGI based applications (PHP/Ruby/Perl/Python typically)
- Native complete AMQP 0.91 support (Advanced Message Queuing Protocol)
- Complete end 2 end secure brandable Remote Desktop with near realtime HD video, 8 monitor support, texture detection, compression and clipboard sharing.
- Bundling kbmMemTable Professional which is the fastest and most feature rich in memory table for Embarcadero products.

 **COMPONENTS 4 DEVELOPERS**

