Getting started with Lazarus

Michaël Van Canneyt

March 4, 2006

Abstract

Lazarus is a cross-platform 2-way RAD tool which can be used to develop almost any kind of program for Windows, Linux, Solaris or some of the BSD distributions. The focus is on GUI applications. In this and following articles, it will be explained how to get started with Lazarus.

1 Introduction

The Lazarus IDE has been around for quite a while now: although it's version number still has not reached 1.0, it considered stable enough for production software. The underlying compiler (Free Pascal) is stable since years. In fact, Lazarus is so stable that commercial companies are known to port their flagship products to Lazarus.

What does this RAD and 2-way tools mean ? Lazarus offers a complete development environment for Object Pascal:

- An highly advanced code editor.
- A GUI form designer with point-and-click interface.
- An integrated debugger.
- A plethora of components (code libraries) for all kinds of tasks, most of them cross-platform. These tasks range from GUI elements over TCP/IP communication to transparant database access to many known databases.

These three are highly integrated: changes in the form designer are reflected in the code and vice versa (this is where the 2-way comes from).

The generated sources can be compiled for any supported platform, and should function as-is, provided a few elementary precautions are taken.

2 Architecture

To make this possible in a cross-platform manner, Lazarus is built on top of 3 layers, which in turn are built on top of each other.

RTL The Run-Time Library of Free Pascal. This contains all functions to interact with the OS. It contains both platform-specific units as platform-independent units.

FCL The Free Component Library of Free Pascal. This is a set of all-purpose cross-platform classes and components: components for image treatment, process management, database access, web programming.

LCL The Lazarus Component Library. This is the GUI layer of Lazarus: it contains all visual code for lazarus.

The LCL abstracts the underlying GUI widget set: This means that it is the responsability of the Lazarus team to make sure that the visual components work and act the same on all supported GUI platforms, currently:

GTK1 This is the oldest supported widget set.

GTK2 This widget set is mostly functional, but still under development.

Windows This is the second oldest widget set, and as such completely functional.

Qt This widget set is still under heavy development.

Carbon Work on this native Mac OS widget set is in the beginning stages.

Note that applications with the GTK widget sets run perfectly on a KDE based desktop system, provided the GTK libraries are present. Vice versa, a Qt-based application will also run on a GNOME system.

3 Installation

Installing lazarus is possible in 2 ways: using an installer program, and by compiling it. For Windows, an installation package is available, just as for Linux (an rpm). The windows installation contains a complete Free Pascal installation. For Linux, a separate FPC install is available.

The more adventurous people, or people working on a platform for which no binary installation is available, can or should compile Lazarus from source. The lazarus website offers daily zips of the sources, or zips of the latest released version (currently 0.9.12).

To compile these, a recent Free Pascal compiler is needed (2.0.2 should do just fine). Compiling is simply a matter of typing 'make all' in the lazarus source directory. That is all what is needed.

All files can of course be downloaded from the official lazarus website:

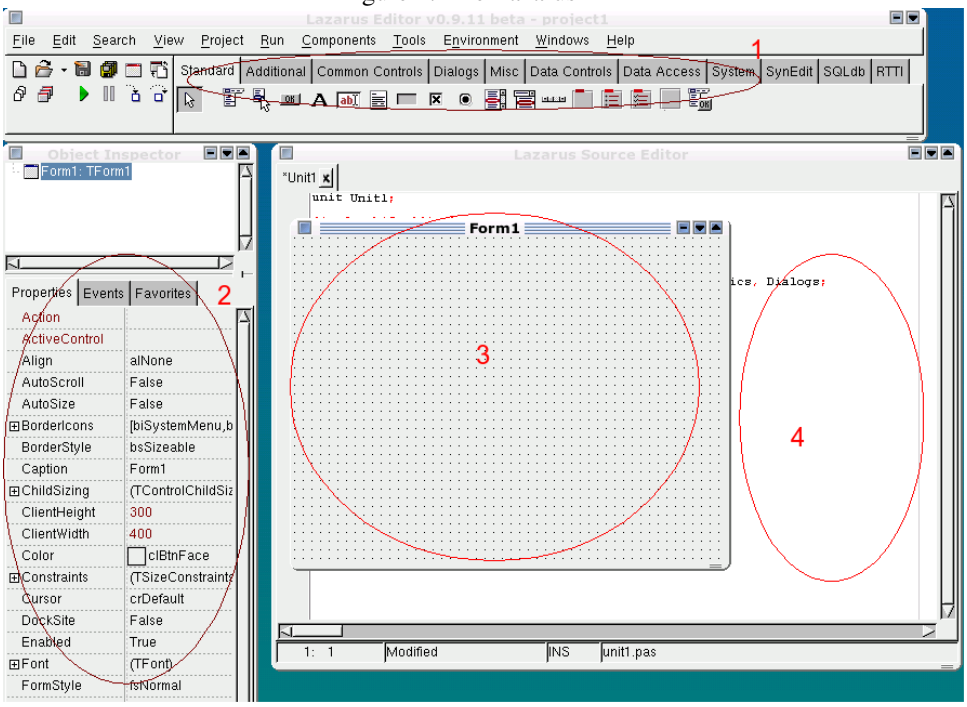
`http://www.lazarus.freepascal.org/`

4 The IDE: a quick tour

When Lazarus is first started, a window as in figure 1 on page 3 appears. It contains 4 main areas:

1. The component palette. Here one selects buttons, memos, edit controls, all other kinds of components, and drops them on the form designer: Select a control, and click on the form where the control should be placed.
2. The Object Inspector: If a component is selected in the form designer or in the component treeview above the inspector, then it's properties can be reviewed or set here.
3. The form designer: This is a WYSIWIG representation of the form as it will appear in runtime. For non-visual forms (Datamodules) this contains a visual representation of all components dropped on the datamodule; each component is then represented by an icon.

Figure 1: The Lazarus IDE



4. The code editor: this is the editor window where all code is written. Manipulating the form (adding, deleting or renaming components) will cause changes in the code editor. Adding event handlers in the Object Inspector will also cause changes in the code editor: methods will be created in the designer.

There exist many more windows (dialogs) in the IDE than the ones presented here; but the above four will be used most of all. The component palette contains a lot of standard GUI controls, and some non-visual components as well. However, by default not all controls and components are shown: The IDE can be extended with additional packages that add additional components on the component palette. These additional packages can also add new menu entries, introduce new dialogs, in short: add new functionality to the IDE.

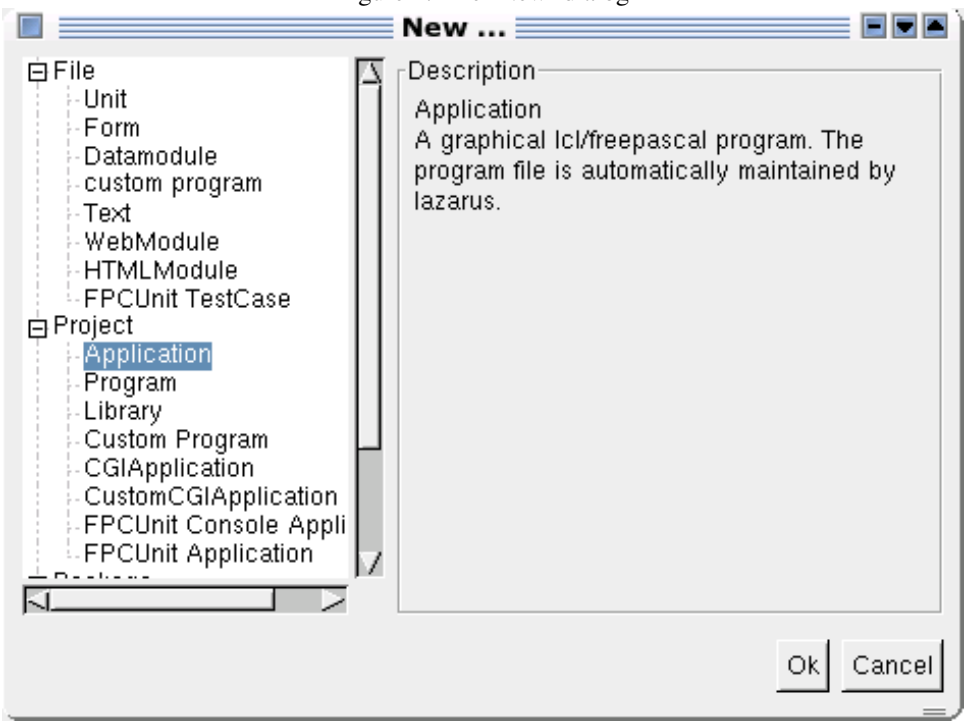
5 Starting a new project

When lazarus is started, it can do 2 things:

1. Load the last used project.
2. Start a new, blank, project. This project consists of a single application with a single form, as in figure 1 on page 3.

To start a new project, either the blank project can be used, or a new project can be started. This is done through the **'File|New'** menu item. When this menu item is selected, the dialog in figure 2 on page 4 appears. Here, there are many entries to start new forms or projects or tests: all have in common that one ore more new files will be created in the editor, and in the form designer. The number of items that appears here depends on what packages are installed in the IDE.

Figure 2: The 'New' dialog



To start a new GUI application, the 'Application' item under 'Projects' must be chosen. If the current project was modified, the IDE will first ask to save the project, before starting the new project.

An empty form is created (called `Form1`) and the corresponding unit is started in the editor unit1, which is what is shown in figure 1 on page 3. Each form in an application is a class which descendends from `TForm`: the unit `unit1` contains the declaration of the form class. The class name of the form is the name of the form, prepended with T, so the class name is `TForm1` in the above case. The properties of the form which are displayed in the object inspector, are the published properties of class `TForm`. The declaration will look therefore something like this:

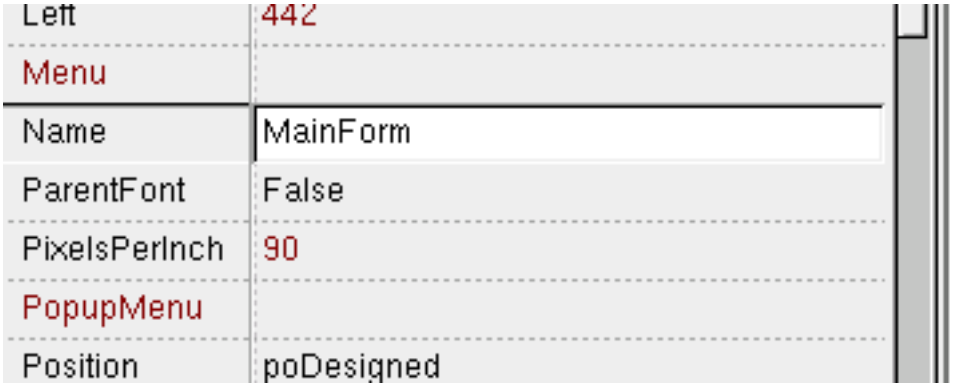
```
Type
  TForm1 = Class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;
```

For ease of use in simple applications, a global variable is also declared which will contain a reference to the instance of `TForm1` when the application is running:

```
var
  Form1: TForm1;
```

This is it. The project is ready to be compiled and run: pressing F9 or selecting the menu item **'Run|Run'** will compile the project, and, if everything went correctly, the program will be run, displaying an empty window.

Figure 3: Setting the 'Name' property



As soon as the window is closed, the program ends. This is true, regardless of how many forms (Windows) are open: as soon as the 'Main' window is closed, the program will end.

Technically, this means that the GUI message loop is ended, and program execution stops.

Note that the project and the units are saved in a temporary location if they have not been saved manually.

6 Manupilating the form

Now, obviously, this is not very useful. First of all, `Form1` is not a very descriptive name of the form. So, the form will be renamed. To do this, the Object Inspector can be used. Clicking on the form selects it, and the properties of the form are displayed in the Object Inspector. In the object inspector, changing the 'Name' property to 'MainForm' (as in figure 3 on page 5) has several effects:

1. The form's caption is set to 'MainForm'
2. The form's name is set to 'MainForm'
3. The form's class name is set to 'TMainForm'
4. The global variable name for the form instance is set to 'MainForm', and it's type is changed to 'TMainForm'.

These effects can be immediatly verified in the form designer and IDE code editor window.

To make things even better, the 'Caption' property of the form can be changed to 'Hello, world!': doing this does not change anything in the code window: the properties are stored in a separate file, the form file. These properties are automatically applied when the program is run.

Again the program can be compiled and run. It will not look very different: the only difference is that the caption on the window is changed.

7 Adding controls

To add some more functionality, a button will be added to the program. To do this, the following steps are needed:

1. In the component palette, select the 'Standard' tab. Click the button icon; it wil appear pressed ('down').
2. Click on the form, where the button should be placed. Lazarus places the button on the form.

The button is now on the form, and can be manipulated: it can be renamed to something more descriptive than `Button1` - for example `BPress`. It's size can be changed, anchors can be set, it can be given a descriptive caption.

These actions have also made a change in the code window:

```
TMainForm = class(TForm)
  BPress: TButton;
private
  { private declarations }
public
  { public declarations }
end;
```

The button appears as a published field in the form declaration, using the name that was given to it. Note that this places restrictions on the names that can be given to controls on forms: these names should be valid pascal identifiers. At run-time, the `BPress` field contains the instance of the button. The field is initialized automatically by the streaming system.

Changing the caption, setting the size, setting anchors: all these actions do not change the source file. They do change the form file, where all properties to the components are recorded.

Once more, the form can be run, and the button can be pressed. Nothing happens, however.

8 Event handlers

GUI Programs should respond to user actions: they are not 'linear' programs, they react to user events: a button is pressed, an item in a list is selected, text is typed.

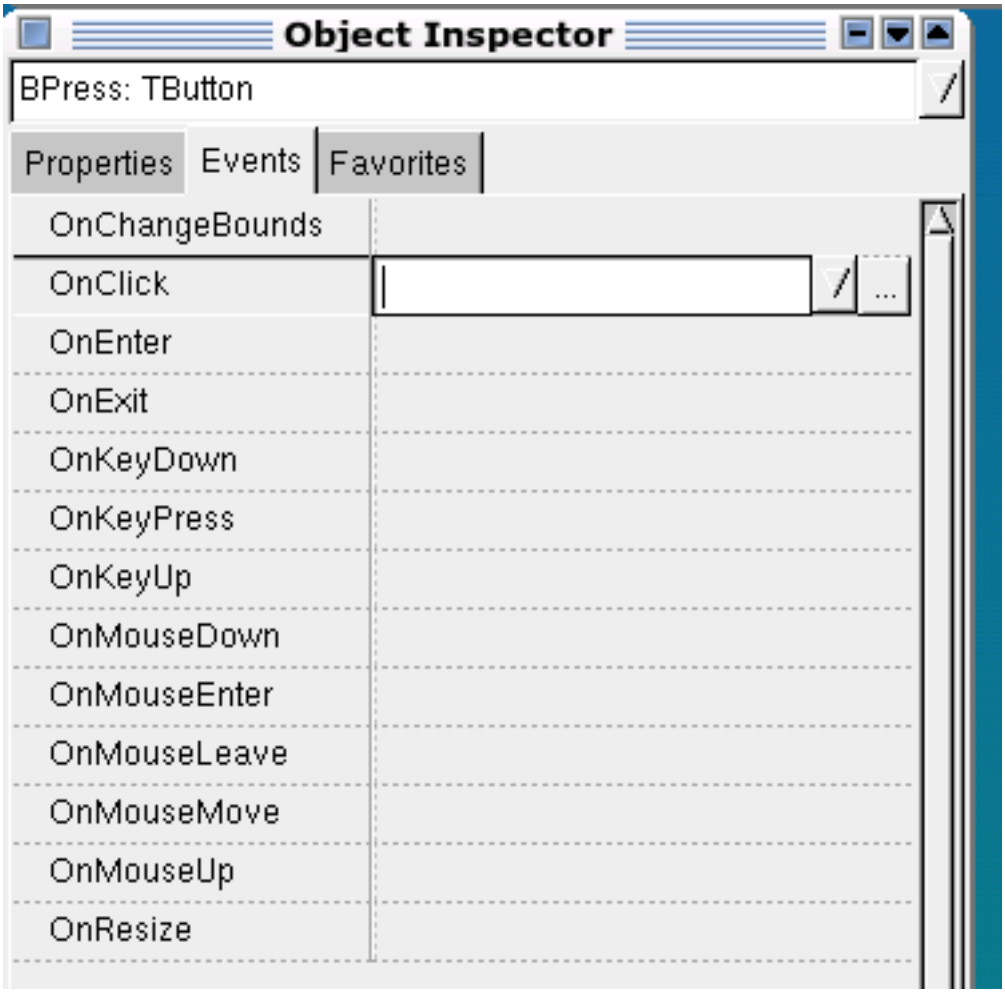
Reaction to such user-triggered events is done in event handlers. Most, if not all, components contain a lot of event handlers which are triggered at various points.

Technically, an event is a variable or property of procedural type: if the variable is set, the procedure it points to will be executed. The basic procedural type used throughout the LCL is `TNotifyEvent`:

```
TNotifyEvent = Procedure(Sender : TObject) of object;
```

It's a method pointer, because all event handlers will be implemented as methods of the form class. `TNotifyEvent` has a single parameter: `Sender`. When the event handler is called, `Sender` will contain a pointer to the instance that triggered the event. It's of type `TObject`, so that it can be used for any component.

Figure 4: Events in the object inspector



So, is it necessary to create event handlers and connect them to the necessary variables or properties ? No. The IDE does all this. On the tab 'Events' in the object inspector, a list is presented of possible events for the currently selected component, as shown in figure 4 on page 7. When selecting an event in the list, there are 2 options:

1. Select an existing event handler: 3 buttons can have the same event handler connected to their 'OnClick' event. The handler can distinguish which button fired the event through the 'Sender' argument.
2. Create a new event handler: Simply clicking the ellipsis button will create a new event handler with a predefined name: The name of the component, followed by the event name. The used name can be changed by first typing the desired name, and only then clicking the ellipsis button.

As for component names, the event handler names are used for methods of the form, hence they must be valid pascal identifiers.

For the button dropped on the form, the `OnClick` event handler will be called `ShowHello`. Pressing the ellipsis changes the code of the unit:

```
  TMainForm = class(TForm)
    BPress: TButton;
    procedure ShowHello(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{ TMainForm }

procedure TMainForm.ShowHello(Sender: TObject);
begin

end;
```

An empty method is created, ready to be filled with code.

The method handler is inserted automatically inserted in the first section of the form declaration: This section is reserved for published methods and fields, and is the only section that the form designer and object inspector will use. Moving a method out of this section will make it unavailable in the object inspector. Inversely, moving a private method to the published section of the form declaration makes it available in the object inspector.

The reason for the exclusive use of the Published section is that the streaming system uses the information in the form file to look up the method in the RTTI (Run-Time Type Information) generated for the form. The RTTI is only generated for published sections, so only published methods are usable.

Now that a method exists, some code can be inserted:

```
ResourceString
  SCloseForm = 'Would you like to close the window ?';

procedure TMainForm.ShowHello(Sender: TObject);

Var
  mr : TModalResult;

begin
  mr:=MessageDlg(SCloseForm,mtConfirmation,[mbYes,mbNo],0);
  if (mr=mrYes) then
    Close;
end;
```

The `MessageDLG` function is a standard LCL function, which pops up a dialog window with an icon, a text, and some buttons. The kind of icon depends on the second argument, which can be one of the following:

mtInformation for an informational message.

mtConfirmation for a confirmation message.

Figure 5: The 'OnClick' handler at work

mtWarning for a warning message.

mtError for an error message.

Which buttons are shown is determined by the third argument, a set. This set also determines the result of the `MessageDLG` function: The function returns a modal result which can be one of the following:

mrYes if the `mbYes` button was pressed.

mrNo if the `mbNo` button was pressed.

mrOK if the `mbOK` button was pressed.

mrCancel if the `mbCancel` button was pressed.

mrAbort if the `mbAbort` button was pressed.

mrRetry if the `mbRetry` button was pressed.

mrIgnore if the `mbIgnore` button was pressed.

mrAll if the `mbAll` button was pressed.

mrYesToAll if the `mbYesToAll` button was pressed.

mrNoToAll if the `mbNoToAll` button was pressed.

mrNone if the dialog was closed by directly closing the dialog window.

In the above example, pressing the `OK` button will call the `Close` method of the form, forcing the form to be closed, and terminating the program.

The result of the above `OnCLick` handler is shown in figure 5 on page 9.

9 Other events

Not all events are GUI events, triggered by user actions; Throughout the LCL there are many other events which fullfill various tasks. For instance the `OnCreate` event of a form is triggered when a form is instantiated. The `OnDestroy` event is triggered when this instance is destroyed again: the perfect locations to allocate and deallocate resources.

The `OnCloseQuery` event is triggered when the form is closed: it can be used to prevent the form from closing, for instance when data still needs to be saved.

The `OnPaint` event, exposed by many controls, can be used to do some custom drawing: this event is triggered by the GUI system, when it decides that some control needs to be redrawn on screen.

There are many more events, all of them triggered at one point or another: it's outside the scope of this article to discuss them all. Most of them have fairly descriptive names, so it should be easy to determine their function.

10 Conclusion

In this first article, we've created a small program, which doesn't do much, but which does demonstrate how to work with the IDE and how to create visual programs that react to user actions. In subsequent articles more detailed information about controls, their properties and how they interact, will be discussed.