navigation
- Main Page
- Documentation
- FAQ
- Downloads
- Glossary
- Index
- Recent changes
- Random page
- Help

search
Search
Go Search

tools
- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information

# Lazarus Tutorial

| Deutsch (de) | English (en) | español (es) | suomi (fi) | français (fr) | magyar (hu) | Bahasa Indonesia (id) | italiano (it) | 日本語 (ja) | 한국어 (ko) | македонски (mk) | Nederlands (nl) | português (pt) | русский (ru) | slovenčina (sk) | shqip (sq) | 中文(中國大陸) (zh_CN) | 中文(台灣) (zh_TW) |

Lazarus is a free and open source development tool for the Free Pascal compiler, which is also free and open source. The Lazarus Integrated Development Environment (IDE, see Screenshots) is a programming environment to create standalone graphical and console applications.

Lazarus currently runs on Linux, Mac OS X, FreeBSD and Win32 and provides a customizable source editor and visual form creation environment along with a package manager, debugger and complete GUI integration with the Free Pascal compiler.



## Getting Started

Get, install and launch Lazarus which will also make the Free Pascal Compiler available.

**Note:** On Linux Ubuntu at least, the command to start Lazarus from a console is "startlazarus". Otherwise, if you installed it from a Debian package, you should have a Lazarus menu entry under Application/Programming. (Issue: In Debian and Ubuntu the main binary and the package was renamed to "lazarus-ide" because the "tct" package already comes with a utility called "lazarus").

### Your first Lazarus Program!

From the Main Menu, choose Project-New Project-Application (or: File-New-Project-Application). A new GUI application is created, see also Form Tutorial.

Several windows will appear on the desktop: the Main Menu at the top, the 'Object Inspector' on the left, the 'Source Editor' occupying most of the desktop, and a ready-made 'Form1' window (form1) overlying the Source Editor.

To place a button on the form: on the top Menu window, underneath the menu line, is a row of tabs. If the 'Standard' tab is not already selected, select it by clicking with the mouse. Then find the Button icon (a rectangle with 'OK' on it) and click on that with the mouse. Then click on the Form1 window, somewhere to the left of the middle. A shadowed rectangle labelled 'Button1' will appear. Click again on the Button icon in the Standard tab, and click on the Form1 somewhere to the right of centre: a second rectangle labelled 'Button2' will appear.

Now click on Button1 to select it. The 'Object Inspector' window will now display the properties of the object Button1. Near the top is a property named 'Caption', with the displayed value 'Button1'. Click on that box, and change 'Button1' to 'Press'. If you hit ENTER or click in another box, you will see the label of the first button on Form1 change to 'Press'. Now click on the Events tab on the Object Inspector, to see the various events that can be associated with this button1. These include OnClick, OnEnter, OnExit etc. Select the box to the right of OnClick: a smaller box with three dots (... ellipsis) appears. When you click on this, you are taken automatically into the Source Editor and your cursor will be placed in a piece of code related to button1, starting:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {now type (or copy&paste):}    Button1.caption := 'Press again';
  {the editor has already completed the procedure with}
end;
```

Press F12 to select the Form1 window instead of the Source Editor.

Now edit the properties of Button2: click on Button2 to display its properties in the Object Inspector. Change its Caption property to 'Exit' instead of 'Button2'. Now select the Events tab, and click on the box for OnClick. Click on the ... ellipsis, and you will be taken into the Source Editor, in the middle of another procedure for button2:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  {now type (or copy&paste):}   Close;
  {the editor has already completed the procedure with}
end;
```

Now Press F12 to see the Form1 window again. You should save your work now (and frequently!!) by selecting from the Main Menu Project-Save Project As > your_selected_file_name.pas. Next, you will be asked to save a **L**azarus **P**roject **I**nformation file, with the suffix .lpi. (Note: For Lazarus-0.9.30 you will be asked for project name first and then file name). Choose a different name for this file, if both are identical you will get a "duplicate identifier" compile error (see chapter "The Lazarus files" at the end of this tutorial).

You are now ready to try to compile. The simplest way to compile is to select 'Run' from the main menu at the top, and then the 'Run' option on the sub-menu. Alternatively you could simply press F9. This will first compile and then (if all is well) link and execute your program.

Several text windows will appear and all sorts of compiler messages will be typed, but eventually your Form1 window will re-appear, but without the grid of dots; this is the actual main window of your application, and it is waiting for you to push buttons or otherwise interact with it.

Try clicking on the button labelled 'Press'. You will notice that it changes to 'Press again'. If you press it again, it will still say 'Press again'!!

Now click on the button marked 'Exit'. The window will close and the program will exit. The original Form1 window with the grid of dots will reappear, ready to accept more editing activity.

### Modify your Program

Re-open your saved Project and on the Form1 window click on the 'Press' button (Button1) to select it. Select the 'Events' tab on the Object Inspector, click on the right side box next to OnClick, click on the ... ellipsis, to go back to the appropriate point in the Source Editor.

Edit your code to read as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
{ Makes use of the Tag property, setting it to either 0 or 1}
begin
  if Button1.tag = 0 then
  begin
    Button1.caption := 'Press again';
    Button1.tag := 1;
  end else
  begin
    Button1.caption := 'Press';
    Button1.tag := 0;
  end;
end;
```

Save your work, re-compile and run. The left button will now toggle between two alternative messages.

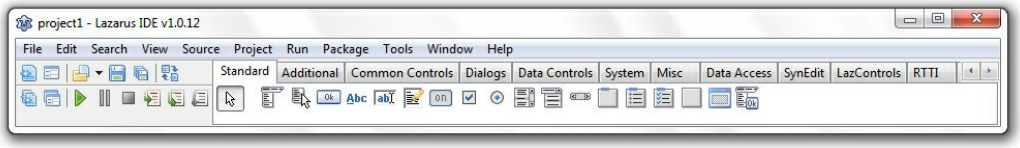
The rest is up to you!

If you prefer to write Console- or text-based Pascal programs (for example if you are following a basic Pascal programming course, or you need to write programs to use in batch mode or for system programming), you can still use Lazarus to edit, compile and run your programs. See Console Mode Pascal.

## The Editor Windows

When you launch Lazarus for the first time, a series of separate disconnected or 'floating' windows will appear on your desk-top.

The first, running right along the top of the desk-top, is titled **project1 - Lazarus IDE vXXXXXX** (which will subsequently be modified to reflect the name of your currently-open project). This is the main controlling window for your project, and contains the Main Menu and the Component Palette.




On the line below the title bar is the *Main Menu* with the usual entries for File, Edit, Search, View categories and so on, with a few selections that are specific to Lazarus. Below this on the left is a set of symbols (icons which take you rapidly to particular menu commands); and on the right is the Component Palette.

Under the Lazarus Editor window will appear the 'Object Inspector' window on the left, and the Lazarus 'Source Editor' on the right. There may be another smaller window, labelled 'Form1', overlying the Lazarus Source Editor window. If this is not visible immediately, it can be made to appear by pressing the F12 key, which toggles between the Source Editor view and the Form view. The Form window is the one on which you will construct the graphical interface for your application, while the Source Editor is the window which displays the Pascal code associated with the application which you are developing. The operation of the Object Inspector is discussed in more detail below while the Component Palette is described. Finally, there may also be a Message window in the lower of the screen: this one is used by Lazarus to show feedback to the programmer, for instance when compiling a program.

When you start a new project (or when you first launch Lazarus) a default Form will be constructed, which consists of a box in which there is a grid of dots to help you to position the various components of the form, and a bar along the top which contains the usual **Minimise**, **Maximise** and **Close buttons**. If you click with your mouse cursor anywhere in this box, you will see the properties of this form displayed in the Object Inspector on the left side of the desk-top.

Other windows that may become visible during your work: the 'Project Inspector', which contains details of the files included in your project, and allows you to add files to or delete files from your project; the **Messages** window, which displays compiler messages, errors or progress reports on your project; if Lazarus was launched from a terminal window, the original terminal remains visible and detailed compiler messages are also printed there.

## The Main Menu

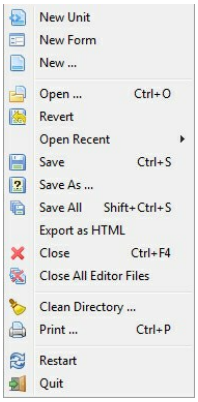
The Lazarus **main menu** contains the following entries:

File Edit Search View Source Project Run Package Tools Window Help

As usual, the options can be selected either by placing the mouse cursor over the menu option and clicking the left mouse button, or by typing Alt + F on the keyboard (provided the main menu window has focus: if it has not, hit Tab repeatedly to cycle focus through the various windows until the desired window has its title bar highlighted in colour).

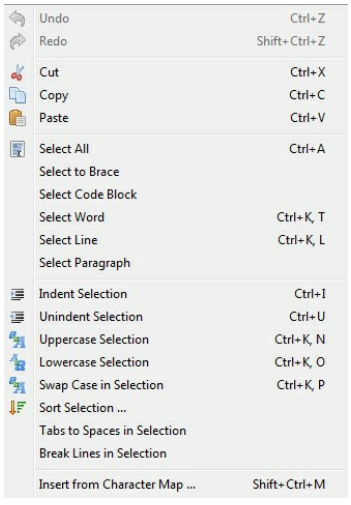
### File

- **New Unit**: Creates a new Unit file (Pascal Source).
- **New Form**: Creates a new Form: both the visual on-screen window and the associated Pascal source file.
- **New ...**: Offers a pop-up menu box with a variety of new document types to create.
- **Open**: Offers a pop-up Dialog Box to enable you to navigate the filesystem and choose an existing file to open.
- **Revert**: Abandon editing changes and restore the file to its original state.
- **Open Recent**: Offers a list of recently edited files, and an opportunity to select one.
- **Save**: Save the current file, using its original filename. If there is no name, the system will prompt for one (just like Save As).
- **Save As**: Allows you to choose a directory and filename for saving the current file.
- **Save All**: Saves all the files attached to the editor, not just the current one selected.
- **Export As HTML**: Export the contents of the current file to a new file in HTML format.
- **Close**: Closes the current file, prompting whether to save all editor changes.
- **Close all editor files**: Close all files currently open in the editor. Prompt for saving changes.
- **Clean directory**: Offers a dialog with a series of editable filters for removing files from the current directory. Useful for removing .bak files and remnants of former Delphi projects.
- **Print**: Uses the system printer to print the selected file. This menu item may not appear by default if you are not using Windows; you will then need to install $Lazdir/components/printers/design/printers4lazide.pas and re-compile the IDE
- **Restart**: Re-start Lazarus - useful if files have got hopelessly scrambled!
- **Quit**: Exit Lazarus, after prompting for saving all edited files.



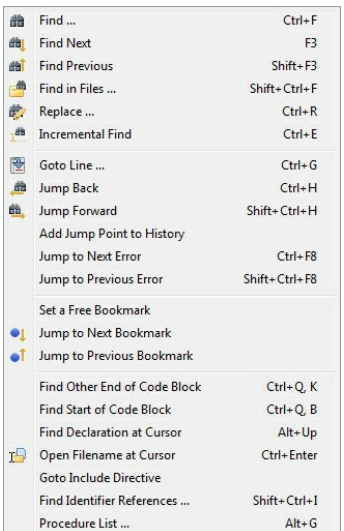

### Edit

- **Undo**: Undo the last edit action, leaving the Editor in the state just before the last action.
- **Redo**: Re-instates the last action that was reversed by Undo.
- **Cut**: Remove the selected text or other item and place it on the Clipboard.
- **Copy**: Make a copy of the selected text, leaving the original in place, and placing the copy on the Clipboard.
- **Paste**: Places the contents of the Clipboard at the cursor position. If text has been selected at the cursor position, the contents of the Clipboard will replace the selected text.
- **Select**: Allows selection of blocks of text. Options include select all, select to brace, select paragraph or line etc.
- **Indent Selection**: Move the selected text to the right by the amount specified in Environment -> Editor options -> General -> Block indent. This feature is useful for formatting your Pascal source code to show the underlying Block structure.
- **Unindent selection**: Removes one level of indenting, moving the text to the left by the amount specified in Block indent.
- **Uppercase Selection**: Convert selected text to uppercase.
- **Lowercase Selection**: Convert selected text to lowercase.
- **Swap Case in Selection**: Convert selected text to lowercase or uppercase.
- **Sort selection**: Sort lines (or words or paragraphs) alphabetically; options for ascending or descending order, case sensitive or insensitive. In the middle of program source code, of course, it makes no sense, but if you have a list you need to have sorted this will do the trick.
- **Tabs to Spaces in Selection**: Converts any tabs in the selected text to the number of spaces specified by Environment -> Editor options -> General -> Tab widths. The number of spaces is not a fixed quantity, but is the number needed to fill the remaining width of the tab.
- **Break Lines in Selection**: If any lines in the selected text are longer than 80 characters or the number specified in Environment -> Editor options -> Display -> Right Margin, then the line is broken at a word boundary and continued on the next line.
- **Insert from Character Map**: Allows insertion of non-keyboard symbols such as accented characters, picked from a pop-up character map.




### Search

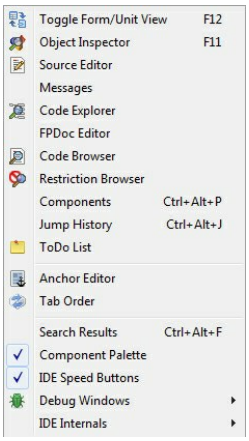
- **Find**: Similar to the facility in almost all graphic text editors: a pop-up dialog box appears allowing entry of a search text string, with options such as case sensitivity, whole words, origin, scope and direction of search.
- **Find Next, Find Previous**: Search again for the previously entered text string, in the specified direction.
- **Find in Files**: Search for text string in files: pop-up dialog with options all open files, all files in Project, or all directories; masks available for selecting file types.
- **Replace**: Similar to **Find**; shows pop-up dialog with place to enter search text string and replacement text, and options for case sensitivity, direction etc.
- **Incremental Find**: Search for the string while you are entering the search string. Example: after you choose "Incremental Find" if you press "l" the first "l" will be highlighted. If then you press "a", the editor will find the next "la" and so on.
- **Goto Line**: Move editing cursor to specified line in file.
- **Jump Back**: Jump to previous position. Everytime jumping to an error or find declaration the IDE saves the current source position. With this function you can jump back in the history.
- **Jump Forward**: Jump to next position. Undoes a Jump back.
- **Add Jump Point to History**: Add the current source position to the jump hsitory.
- **Jump to Next Error**, **Jump to Previous Error**: Jump to the positions in the source file of the next or previous reported error.
- **Set a Free Bookmark**: mark the current line where the cursor is located with the next available (free) numbered bookmark, and add this to the list of bookmarks. Note that a pop-up menu (obtained by right-clicking with the mouse on the appropriate line of the source file) gives a larger range of Bookmark options, allowing the number of a bookmark to be specified, or allowing the user to jump to a numbered bookmark, not just the next or previous ones.
- **Jump to Next Bookmark**, **Jump to Previous Bookmark**: Jump to next or previous bookmark in the numerical sequence.
- **Find Other End of Code Block**: If positioned on a **begin**, finds the corresponding **end** or vice versa.
- **Find Code Block Start**: Moves to the **begin** of the procedure or function in which the cursor is placed.
- **Find Declaration at Cursor**: Finds the place at which the selected identifier is declared. This may be in the same file or another file already open in the Editor; if the file is not open, it will be opened (so if a procedure or function is declared, for example, in classesh.inc , this will be opened in the Editor). (More)
- **Open Filename at Cursor**: Opens the file whose name is selected at the cursor. Useful for looking at Include files or the files containing other Units used in the project.
- **Goto Include Directive**: If the cursor is positioned in a file which is Included in another file, goes to the place in the other file that called the Include file.



- **Find Identifier References**: Produces a list of all the lines in the current file, or the current project or all attached files, in which an identifier is mentioned.
- **Procedure List**: Produces a list of all Procedures and Functions in the current file, with the line numbers where they are defined.

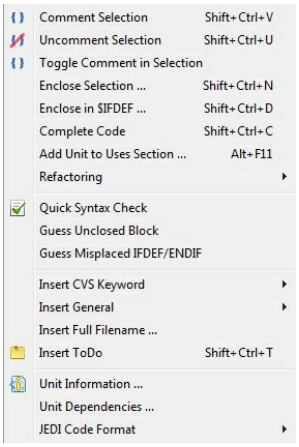
### View

Controls the display of various windows and panels on the screen.



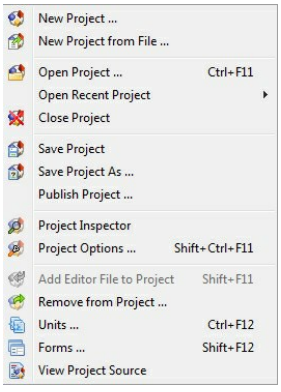

- **Object Inspector**: The window that usually occupies the left side of the Desktop, and displays the features of the Form which is on the desktop. Clicking with the mouse on any component of the form will cause the details of that component to be displayed in the Object Inspector. There is a panel at the top which shows the tree-structure of the current project, and the components of the form may optionally be selected in this panel: this will also cause the corresponding details to be displayed in the Object Inspector. The main lower panel has two tabs which allow selection of either **Properties** or **Events** to be displayed. Selection of **Properties** causes features such as name, color, caption, font, size etc to be displayed: there are two columns, the left showing the property, and the right showing the value associated with that property. Selection of **Events** displays two columns: the left lists the possible events such as MouseClick or KeyDown associated with that component, and the right shows the action that results from that event. If there is no action defined, then clicking in the appropriate box or on the
  …
  button causes the Source Editor to be displayed, with the cursor already positioned in a dummy Procedure declaration, waiting for event-defining code to be typed in.
- **Source Editor**: The main window in which source code is edited. Its behaviour is very like that of most other graphical text editors, so that the mouse can move the cursor over the displayed text, and clicking with the left mouse button while dragging the mouse will select and highlight text. Right clicking with the mouse displays a pop-up menu, it includes the usual Edit Cut, Copy or Paste functions, *Find Declaration* and *Open File at Cursor*. The top of the Source Editor window has a number of tabs, corresponding to the files that are open for the current project; clicking on any tab makes that file visible, and you can move easily from file to file, copying and pasting between files and performing most of the normal editing functions. The Source Editor performs color syntax highlighting on the code, with different colors for punctuation marks, comments, string constants etc. It will also maintain the level of indentation from line to line as you type in code, until you change the indentation. The function and appearance of the Source Editor are very configurable from the Main Menu by selecting Environment -> Editor options and then selecting one of several tabs in the pop-up dialog box.
- **Code Explorer**: A window usually placed on the right of the Desktop which displays, in tree form, the structure of the code in the current unit or program. It usually opens with just the Unit name and branches for Interface and Implementation sections, but clicking on the
  +
  box to the left of any branch will open up its sub-branches or twigs, in more and more detail until individual constants, types and variables are displayed as well as procedure and function declarations. If you change the file displayed in the main Source Editor window, you need to click on the Refresh button of the Code Explorer to display the structure of the new file.
- **Units...**: Opens a pop-up dialog window with a list of the unit files in the current project.Clicking with the mouse on a filename selects that file; click on Open to display that file in the Source Editor. Checking the Multi box allows several files to be selected simultaneously, and they will all be opened in the Source Editor (but only one at a time will be displayed). This Menu Option is rather like the Project -> Project Inspector option, but only displays the list of Unit files and allows them to be opened.
- **Forms...**: Opens a pop-up dialog window with a list of the Forms in the current project, and allows the selection of one or more of them for display.
- **View Unit Dependencies**: Opens a pop-up dialog window that shows, in a tree-like manner, the structure of dependencies of the currently open unit file. Most of the files listed as dependencies will have their own
  +
  boxes, which allow the dependencies of the individual files to be explored, often in a highly recursive manner.
- **Toggle form / unit view F12**: Toggles whether the Source Editor or the current Form is placed on the top layer of the Desktop, and given focus. If the Source Editor has focus, then you can edit the source code; if the Form is given focus, you can manipulate the components on the desktop and edit the appearance of the Form. The easiest way to toggle the display between Editor and Form is to use the F12 key on the keyboard, but the same effect is achieved by selecting this option on the Main Menu.
- **Messages**: A window that displays compiler messages, showing the progress of a successful compilation or listing the errors found.
- **Search Results**: A window that displays the results of find in files.
- **Debug windows**: Opens a pop-up menu with several options for operating and configuring the Debugger. See below where the debugger is described.

### Source



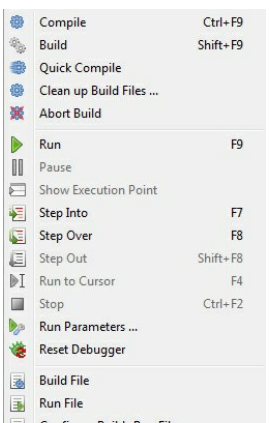

- **Comment Selection**: Comment out the currently selected text.
- **Uncomment Selection**: Uncomment the currently selected text.
- **Toggle Comment in Selection**: Comment out or uncomment the currently selected text.
- **Enclose Selection...**: Enclose the currently selected text with a statement such as Try...Finally.
- **Enclose in $IFDEF...**: Enclose the currently selected text with an IFDEF (conditional defines) statement.
- **Complete Code**:
- **Add Unit to Uses Section**: Add one or more Units to the uses section in the current file.
- **Refactoring**: Open the Refactoring sub-menu.
- **Quick Syntax Check**: Run a quick syntax check.
- **Guess Enclosed Block**: Let the editor end the enclosed block, such as a missing "End" in an "If" statement.
- **Guess Misplaced IFDEF/ENDIF**: Let the editor correct a misplaced conditional defines statement.
- **Insert CVS Keyword...**: Insert a CVS keyword, such as "Author" or "Date".
- **Insert General**: Insert a general value, such as a "GPL Statement".
- **Insert Full Filename...**: Insert the filename.
- **Insert ToDo**: Insert a ToDo comment.
- **Unit Information...**: Display information about the current unit.
- **Unit Dependencies...**: Display the dependencies for the current unit.
- **JEDI Code Format**: View the JEDI Code Format options for the project.

### Project



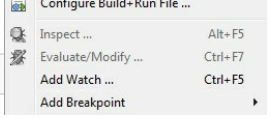

- **New Project**: Create a new project. A pop-up dialog window appears offering a choice of types of project to create.
- **New Project from File**: A Navigation dialog window appears, alowing selection of a file from which to create a new project.
- **Open Project** Open a project which has already been created and saved. A navigation dialog appears with a list of Lazarus Project Information (.lpi) files from which a project may be chosen.
- **Open Recent Project**: Displays a pop-up list of recent projects on which you have been working, and allows selection of one of these.
- **Close Project** Close the current project.
- **Save Project**: Similar to File -> Save: all the files of the current project are saved; if they have not previously been saved, there is a prompt for filename(s)- similar to Save Project As...
- **Save Project As...**: Prompts for filename to save project. A default filename of Project1.lpi is offered, but you should choose your own filename. Lazarus will not permit you to use the same name for the Project file and the Unit File (see below).
- **Publish Project**: Creates a copy of the whole project. If you want to send someone just the sources and compiler settings of your code, this function is your friend. A normal project directory contains a lot of information. Most of it is not needed to be published: the .lpi file contains session information (like caret position and bookmarks of closed units) and the project directory contains a lot of .ppu, .o files and the executable. To create a lpi file with only the base information and only the sources, along with all sub directories use "Publish Project". In the dialog you can setup the exclude and include filter, and with the command after you can compress the output into one archive. See Lazarus IDE Tools
- **Project Inspector**: Opens a pop-up dialog with a tree-like display of the files in the current project. Allows you to add, remove or open selected files, or change options of the project.
- **Project Options...**: Opens a pop-up dialog window with tabs for setting options for Application (Title, Output Target file name), Forms (allowing you to select among the available forms, make them Auto-create at start of application) and Info (specifying whether editor information should be saved for closed files, or only for project files).
- **Compiler options ...**: (Recently moved here from the Run Menu). Opens a multi-page tabbed window which allows configuration of the compiler. Tabs include **Paths** which allows definition of search paths for units, include files, libraries etc, as well as allowing choice of widget type for the forms (gtk, gnome, win32); **Parsing** which allows choice of rules for parsing source programs, **Code** which allows choice of optimisation for faster or smaller programs, choice of target processor, types of checks, heap size etc; **Linking** allowing choice of whether or how to use debugging, static or dynamic libraries, and whether to pass options through to the linker; **Messages** to define what type of messages should be generated during error conditions; **Other** which allows decision to use default configuration file (fpc.cfg) or some other file; **Inherited** which shows a tree structure diagram to indicate how options have been inherited from units already incorporated; **Compilation** which allows definition of commands to be executed before or after the compiler is launched and can allow use of Make files.
- **Add Editor File to Project**: Add the file currently being edited to the Project
- **Remove from Project**: Gives a pop-up menu of files available for removal from project.
- **Units...**: View a list of all the units in the project.
- **Forms...**: View a list of all the forms in the project.
- **View Project Source**: No matter which file you are editing, takes you back to the main program file (.lpr)or the main .pas file if there is no .lpr.

### Run




- **Compile**: Compiles all the files in the project.
- **Build**: Builds all files in the project.
- **Quick Compile**: Compiles the files in the project that have changed.
- **Clean up Build Files**: Cleans up the build files in the project.
- **Abort build**: Stop the build process once it is running - either you have remembered that you did something silly and want to stop the build, or the system seems to be taking far too long and something is obviously wrong.
- **Run**: This is the usual way to launch the compiler and, if compilation is successful, to start execution of the application. What actually happens is that Lazarus saves a copy of your files, then starts the compiler and linker, then begins execution of the final linked binary program.
- **Pause**: Suspend execution of the currently running program. This may allow you to inspect any output that has been generated; execution may be resumed by selecting **Run** again.
- **Show Execution Point**: Shows the current execution point.
- **Step Into**: Used in conjunction with the debugger, causes execution of the program one step at a time up to a bookmarked point in the source.
- **Step Over**: Causes stepwise execution up to the statement marked, then skips the marked statement, and continues execution at normal speed. Useful in trying to isolate a statement that introduces a logical error.
- **Step Out**: Step out of the current code block.
- **Run to cursor**: Causes execution at normal speed (ie NOT one statement at a time) until either the statement is reached where the cursor is located or the current procedure is exited; then stops. Does not stop, if location is reached during recursion. Resume execution at normal speed by selecting **Run**.
- **Stop**: Cease execution of the running program. Cannot be resumed by selecting **Run**; this will start the program again from the beginning (re-compiling if necessary).
- **Run Parameters**: Opens a multi-page pop-up window which allows command-line options and parameters to be passed to the program to be executed; allows selection of display to run program (eg a remote X terminal may be used in Linux); some system Environment variables may be overridden.
  One very important use of this sub-menu is to activate a terminal window in which conventional Pascal console input/output is displayed. If you are developing a console-mode Pascal

program (ie one that doesn't use the Graphical User Interface with its forms, buttons and boxes) then you should check the box for "Use launching application". The first time you do this and try the Compile/Run sequence, you will probably get a rude message to say




```
"xterm: Can't execvp /usr/share/lazarus//tools/runwait.sh: Permission denied".
```

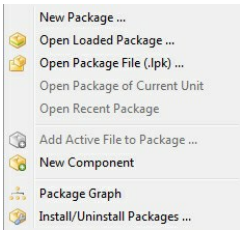
If this happens, you need to change the permissions on the appropriate file (for example using chmod +x *filename*, or using the Windows utility for changing permissions); you might have to do this as root. After this, each time you launch you program, a console box will appear and all your text i/o (readln, writeln etc) will appear in it.
After your program has finished execution, a message "Press enter" appears on the screen. Thus any output your program generated will remain on the screen until you have had a chance to read it; after you press 'enter' the console window closes.
**Note:** as for the current version, there is no prepared console command for Windows users. Until the Lazarus team adressess that, the following line should work (on WinXP -- someone please update for other Windows versions).

```
C:\Windows\system32\cmd.exe /C ${TargetCmdLine}
```

See the separate tutorial on Console Mode Pascal programming.

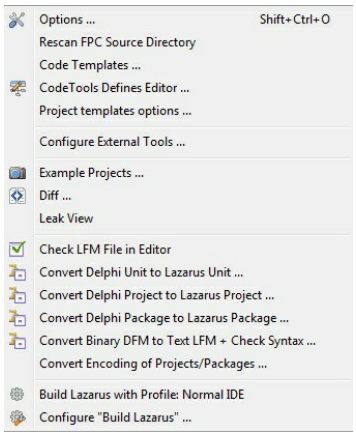
- **Reset debugger**: Restores the debugger to its original state, so that breakpoints and values of variables etc are forgotten.
- **Build file**: Compile (build) just the file that is currently open in the Editor.
- **Run file**: Compile, link and execute just the currently open file.
- **Configure Build + Run File**: Opens a multi-page tabbed window with options to allow for build of just this file when **Build Project** is selected, allows selection of the working directory, the use of various Macros, etc. Then Builds and Runs the file.
  These last three options enable you to open (and maintain) a test project. Use File -> Open to open an .lpr file, pressing cancel on the next dialog to open this file as "normal source" file.
- **Inspect...**: Inspect a value when the program is paused.
- **Evaluate/Modify...**: Evaluate or modify an expression or value when the program is paused.
- **Add Watch...**: Add a variable to the watch list.
- **Add Breakpoint...**: Add a breakpoint, which will will pause execution of the program at that line of code.

### Package




- **New Package...**: Create a new package.
- **Open Loaded Package...**: Open one of the files in the selected package.
- **Open Package File (.lpk)...**: Displays a list of installed packages, with an invitation to open one or more of them, or to select various general or compiler options.
- **Open Package of Current Unit**: Open the package for the unit currently in the editor.
- **Open Recent Package**: Open a package that was opened recently.
- **Add Active File to Package...**: Place the file (currently in the editor) into a package.
- **New Component**: Create a new component.
- **Package Graph**: Displays a showing the relationships of the packages currently being used (if you aren't using any other packages, the Lazarus package and the FCL and LCL will be displayed).
- **Install/Uninstall Packages...**: Install or uninstall one or more packages.

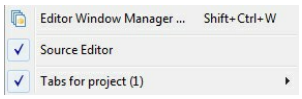
### Tools




- **Options...**: View and changed the options and settings in the Lazarus IDE.
- **Re-scan FPC Source directory** Looks through the directory again. Lazarus uses the fpc sources to generate correct event handlers and while looking for declarations. If somebody changes the directory in the environment options, then this directory is rescanned, to make sure Lazarus uses the version stored in that location. But if this directory has changed without Lazarus noticing, then you may get some errors when designing forms or doing "Find declaration". If you get such an error, you can do two things:
  1. Check the FPC source directory setting in the environment option.
  2. Re-scan FPC source directory.
- **Code Templates...**: View the code templates that are available.
- **CodeTools Defines Editor...**: Edit the code templates.
- **Project templates options...**: Set the options for the project templates.
- **Configure external tools**: Allows the user to add various external tools (usually macros) to the toolkit
- **Example Projects...**: View the example projects that are available.
- **Diff**: Allows comparison between two files (or, usually, two versions of the same file) to find differences. Options to ignore white space at beginning or end of lines or differences in line termination: CR+LF versus LF). Useful for checking if there have been changes since last CVS update etc.
- **Leak View**: View the heap trace output.
- **Check LFM File in Editor**: Allows inspection of the LFM file which contains the settings that describe the current form
- **Convert Delphi Unit to Lazarus Unit**: Helps in porting Delphi applications to Lazarus; makes the necessary changes to the source file. See Lazarus For Delphi Users and Code Conversion Guide.
- **Convert Delphi Project to Lazarus Project**: For porting from Delphi to Lazarus: converts a Delphi project to Lazarus.
- **Convert Delphi Package to Lazarus Package**: For porting from Delphi to Lazarus: converts a Delphi package to Lazarus.
- **Convert DFM file to LFM**: For porting from Delphi to Lazarus: converts the Form Description files from Delphi to Lazarus.
- **Convert Encoding of Projects/Packages...**:
- **Build Lazarus with Profile: Normal IDE**: Launches a re-build of Lazarus from the most recently downloaded or updated SVN files. Hit the button and sit back to watch it happen! (track the process on your **Messages** window).
- **Configure "Build Lazarus"**: Allows the user to determine which parts of Lazarus should be re-built, and how. For example, you could select to have just the LCL re-built, or to have everything except the examples built; you can select which LCL interface to use (ie which set of widgets), and you can select the target operating system and specify a different target directory.

### Window

Contains a list of the currently opened files and the available windows such as **Source Editor**, **Object Inspector** and **Project Inspector**. Clicking on the name of one of the windows brings it to the foreground and gives it focus.




### Help

At present this has three selections:


- **Online Help** which at present opens a browser window that contains a picture of the running cheetah and a few links to the Lazarus, Free Pascal and Wiki websites
- **Reporting a bug** opens the wiki page, which describe the bug reporting procedure
- **About Lazarus** Displays a pop-up box with some information about Lazarus.

Help can be configured with [Tools|Options|Help]

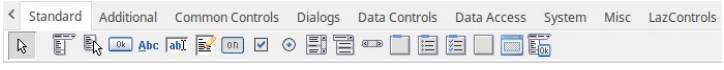
## The Button bar

A small toolbar area on the left of the main editor window, just below the Main Menu and to the left of the Component Palette, contains a set of buttons which replicate frequently-used Main Menu selections:

**New unit**, **Open** (with a down-arrow to display a drop-down list of recently used files), **Save**, **Save all**, **New Form**, **Toggle Form/Unit** (ie show either form or source code of Unit), **View Units**, **View Forms**, **Run** (ie compile and Run), **Pause**, **Step Into**, **Step over** (the last two are Debugger functions).

## The Component Palette

The **Component Palette** of the IDE is a tabbed toolbar which displays a large number of icons representing commonly used components for building Forms.




| Component Palette |
| --- |
| Standard - Additional - Common Controls - Dialogs - Data Controls - Data Access - System - Misc - LazControls - RTTI - SQLdb - Pascal Script - SynEdit - Chart - IPro |

Each tab causes the display of a different set of icons, representing a functional group of components. The left-most icon in each tabbed group is an obliquely leftward-facing arrow, called the Selection Tool.

If you allow the mouse cursor to hover over any of the icons on the Component Palette, without clicking on the icon, the title of that component will pop-up. Note that each title begins with a 'T' - this signifies 'Type' or more accurately 'Class' of the component. When you select a component for inclusion in a form, the Class is added to the **type** section of the **interface** part of the Unit (usually as part of the overall TForm1), and an **instance** of that class is added to the **var** section (usually as the variable Form1). Any **Methods** that you design to be used by the Form or its Components (ie Procedures or Functions) will be placed in the **implementation** part of the Unit.

### How To Use the Palette

To use the palette, there must be an open form on view in the editor (if there isn't one, select File -> New Form). Click on the icon in the appropriate tab of the palette for the component you want to use, then click on the form, near where you want the component to appear. When the desired component appears, you can select it by clicking with the mouse. Once selected on the form, the object is also selected in the Object Inspector window, where you can edit its properties and events.

Adjustments to the visual appearance of an object can be made either by altering the picture itself on the Form using the mouse, or by changing the relevant Property in the Object Editor for that component.

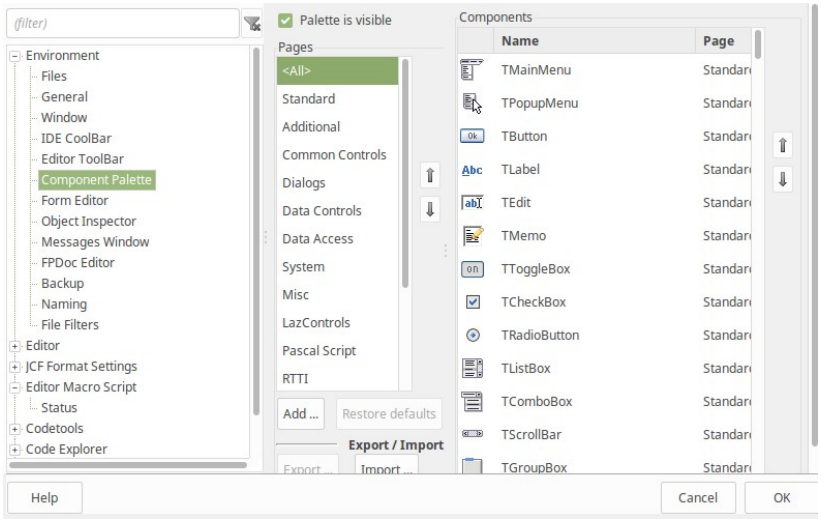
If you install additional components, either those you have written yourself, or some coming as a package from some other source, then extra tabs with the relevant icons will appear in your Component Palette. These new components can be selected and used on your forms in the same way as those supplied by default.

In the following list of the Components, you will find links to files that contain descriptions of the Units in which they are found. If you want to find out about the properties of a particular component, it is often worth looking at the

Inheritance of that component and then inspecting the properties of the base type from which it is derived. For example, to understand TMaskEdit it is also useful to examine TCustomMaskEdit.

### Customization

Using the menu [Tools|Options] or Crtl-Shift-O it is possible to change layout and visibility of the component palette items.




### Examples

Several useful dialog procedures or functions don't appear on the palette, but are easily used as direct calls from your source program.

For several good examples of the use of Components see the $LazarusPath/lazarus/examples subdirectory of your source installation. Many of the programs show how to use dialogs and other components directly without using the IDE and component palette or having a separate form definition file: all the components are fully and explicitly defined in the main Pascal program. Other example programs make full use of the IDE.

Remember to open these examples as projects, with the .lpi file. Opening the .pas source code file and hitting RUN will just append this source file to whatever project you opened last.

Some examples don't work straight away: you may need to play about with paths and permissions of files or directories. If you want to compile any of the examples, make sure that you have read/write/execute permissions for the files and directories, or copy the files to a directory where you do have the appropriate permissions.

Try running the 'testall' program to see a menu of the available components together with small example test forms for most of them; then inspect the code to find out how they work!

## How to use Common Controls

The Units **StdCtrls**, **ComCtrls** and **ExtCtrls** contain definitions and descriptions of many of the most commonly used controls for constructing Forms and other Objects in Lazarus Applications.

Many of the final target controls that the application developer wants to use, such as **TButton**, **TMemo**, **TScrollBar** etc, have a corresponding ancestor class such as **TCustomButton**, **TCustomMemo** or **TCustomScrollBar**. Several of the properties and methods relevant to the final target control are defined (and explained) more fully in the **TCustomXXX** class, and are **inherited** by the final target control.

If you drop a component on the form editor you don't need to add code explicitly to create it. The component is automatically created by the IDE together with the form, and destroyed when the form is destroyed. However, if you create the component yourself by code don't forget to free it when it is no longer needed.

### Ways to Set Properties

If you place a component on the Form Designer and look at the Object Inspector, you can observe the properties change as you move the component around.

For example, if you place a button (TButton) on the form, click on it to select it, then move it around the form with the mouse, you can watch the values of Top and Left change in the Object Inspector to reflect the new position. If you use the object's re-sizing bars to adjust its size, you can watch the Height and Width properties change as well.

On the other hand, by using the Object Inspector, you can select the value associated with a property such as height, and type in a new value; you can watch the size of the object on the form change to reflect the new value.

You can also explicitly change the properties of the object in code by typing (in the appropriate Implementation section of the Source editor), for example

```
Form1.Button1.Height := 48;
```

If you type this new value into the Source Editor and then look back at the Form Designer, you will see that the button on the Form has taken the new size. The new value will also be shown in the Object Inspector.

In summary, there are usually about three different ways to determine each property of an object:

- by using the mouse on the form,
- by setting the values in the Object Inspector,
- or explicitly by writing code in the editor.

### Common Properties

The components defined in these Units have several properties that are common to most of them, and other properties that are specific to the individual components. We shall describe the most common ones here. Unusual or control-specific properties will be described for the individual controls.

Additional Help can always be obtained by selecting a property or keyword, in either the Object Inspector or the Source Editor, and pressing F1. You will be taken by your Help browser to the appropriate page in the documentation.

If the description of a property on that page is insufficient, you can navigate to the corresponding description in the ancestor classes, by selecting the links in the Inheritance listing or by selecting the ancestor Type in the declaration of the object.

**Some commonly listed properties**

| Property | Meaning |
|---|---|
| Action | The main action or event associated with the object. For example selecting an 'Exit' Button might cause the 'Close' action |
| Align | Defines the way in which an object is to be lined up with the parent object. Possible values are alTop (placed at the top and using the full available width), alBottom, alLeft (placed at the left and using the full available height), alRight. alNone (place anywhere on parent control) or alClient (takes all available space next to controls aligned to top, bottom, left or right) |
| Anchor | Used to keep a control a certain distance from the defined edges of a parent control, when the parent is resized. For example **[akBottom, akRight]** will keep the control a fixed distance from the bottom right corner. |
| AutoSelect | When True, an editing control will select all its text when it receives focus or when the Enter key is pressed. |
| AutoSelected | True indicate that the edit or combobox control has just performed an AutoSelect operation so that subsequent mouse-clicks and keystrokes proceed normally without selecting the text. |
| BorderSpacing | The space around the edge between an **Anchored** control and its parent. |
| Caption | The text that is displayed on or near the control; it should preferably give some clue as to the function of the control, or an instruction such as 'Close' or 'Execute'. By default Caption is set to be the same as the 'Name' property, and the application programmer should substitute meaningful text instead of the default values. |
| CharCase | Indicates how text is displayed in a text editing control: Normal (retaining the case of the letters typed by the user), converted to uppercase, or converted to lowercase |
| Constraints | Sets the minimum and maximum sizes for a control. If a control is resized the new dimensions are always within the ranges given here. You should take care when setting these options that they do not conflict with the Anchors and Align settings. |
| Color | The Colour to be used to draw the control or to write the text it contains. |
| Enabled | A Boolean property to determine whether or not a control is capable of being selected and performing an action. If it is not **Enabled**, it is often **Grayed** out on the Form. |
| Font | The Font to be used for writing the text associated with the control - either the caption or label, or the text-strings contained within the control. The entry on the Object Inspector usually has a (+) box on the left, and selecting this box reveals further options such as character set, colour and size. |
| Hint | A short piece of informative pop-up text that appears if the mouse-cursor hovers over the control. See the ShowHint property. |
| Items | The list of 'Things' that the object contains, such as a group of images, a series of lines of text, a number of actions in an actionlist, etc |
| Lines | An array of strings, containing the textual data in controls with more than a single line of data, such as an Edit-Box or a Combo-Box. The array is zero-indexed, ie the lines are numbered [0..numLines-1] |
| Name | The identifier by which the control is known in the program. The IDE gives it a default name based on the underlying type, for example successive instances of TBitButton would be named Form1.BitBitton1 and Form1.BitButton2; it is up to the application programmer to give them more meaningful names such as ExitButton or OKButton. By default the Name of the control is applied to the Caption for the control, but the text of the Caption may be changed separately. |
| PopUpMenu | A window containing context-sensitive menu information that pops up when the right mouse button is clicked on the object. |
| Position (or Top, Left) | Determines where the control is located on the parent form or window |
| ReadOnly | Boolean property which, if True, signifies that the contents of the control can be read by the user or the calling routine, but cannot be written or changed. |
| ShowHint | Allows a small window containing a context-sensitive Help or other description to be displayed when the mouse cursor 'hovers' over the control. See the Hint property. |
| Size (or Height and Width) | The dimensions of the control |
| Style | The options available for Style depend upon the sort of Control being considered: for instance the Style may be defined by TFormStyle, TBorderStyle, TButtonStyle etc. |
| TabOrder | Integer defining where in the sequence of tabs on the Form this control is to lie |
| TabStop | Boolean property which if True places this control in the sequence of objects that the user can reach by successively pressing the Tab key |

| | |
|---|---|
| Text | The String of Text that represents the actual data that this control contains. Applies particularly to Text, Memo and StringList types of object. Most of the editing operations (such as **Select**, **Clear**, **Cut**, **Copy**) are performed in this part of the object, which holds the actual string being edited. If the control contains more than a single line of text, for example **TMemo** or **TComboBox**, then the textual elements are arranged as an array of strings (zero-indexed, ie numbered from [0..numLines-1]) in **Lines**. |
| Visible | If true, the object can be seen on the Form; if False, object is hidden |
| WordWrap | Logical flag to show whether or not word-wrap is enabled, ie if a word comes close to the end of a line and is going to be too long for the line, it is wrapped down to the next line. |

### Event Actions

Many actions are commonly listed in the 'Events' tab of the Object Inspector. If you select an entry in the list, a ComboBox appears with a DropDown list showing any actions that have aleady been defined, and allowing you to choose one to be associated with this event. Alternatively you can select the ellipsis (three dots ...) and you will be taken to an area of the Source Editor where you can begin typing your own action instructions for the selected event.

While a large number of events is available for any given control, in practice it is only necessary to populate a few of them. For most controls, it is sufficient to provide coding for 'OnClick'; for more complex controls it may be necessary also to provide for 'OnEntry' (when the mouse cursor enters the Control and gives it focus) and 'OnExit' (when the mouse cursor leaves the Control; or you may need to write an event handler for 'OnChange' or 'OnScroll', depending on the nature of the particular control with which you are dealing.

The pop-up menu that appears when you right-click an object in the Form Designer has, as its first item: 'Create default event' and selecting this option will have the same effect as selecting the ellipsis in the Object Inspector for the default event, usually OnClick: you are taken to the Implementation area of the Source Editor where you can type the code for the event handler.

A common strategy in Object-Oriented programming is to provide an **ActionList** with the facility for entering, removing or editing a number of pre-defined actions from which the most appropriate can be selected to use in any particular instance.

**Some commonly listed Actions**

| Action | Meaning |
|---|---|
| OnChange | Action to be taken if any change is detected (eg mouse move, mouse click, key press, edit text, alter picture, etc) |
| OnClick | Action to be taken when the (first, usually left) mouse button is clicked. This is commonly the main or default action of the control; for example clicking on a button or checkbox initiates the action associated with the checkbox. It may alternatively initate a process of selection, for instance in a TextBox or Memo, or signal the beginning of painting with a Pen or Brush. |
| Click | A method to emulate in code the effect of clicking on a control. This method is most often found in Button-type controls (TButton, TBitBtn, TSpeedButton etc). A procedure can be written that calls the same code as the OnClick action. This facility can be particularly useful if the activation of one control by clicking causes a cascade of other controls to be activated, and the Click method can be used to initiate the action rather than having the user explicitly click on a lot of controls. |
| OnDragDrop | Action to be taken during Drag-Drop manoeuvres, ie when the mouse is used to 'capture' an item or some text etc and move it around the screen to a new location. |
| OnEditingDone | Action to be taken when the user has finished all edits/modifications to the control. This is often used to validate the control content (e.g. check if an entered text is a valid IP address) |
| OnEntry | Action to be taken when the mouse cursor enters the area occupied by the object, usually transfering focus to that object. This might include changes in the appearance of the object such as highlighting or raising the border. |
| OnExit | Action to be taken when the mouse moves out of the area of the object, usually transferring focus out of the object. |
| OnKeyPress | Action to be taken for any key-press. Subtly different from OnKeyDown, which simply responds to a key being down, whether or not it was already down when focus was given to this control. OnKeyPress requires that a key becomes pressed while focus is in this control. |
| OnKeyDown | Action to be taken if a key is down while focus is in this control. Subtly different from OnKeyPress - for example the key might already have been down when focus entered this control, whereas OnKeyPress requires the key to become pressed while focus is in the control. |
| OnKeyUp | Action to be taken if a key is up (ie not pressed) while focus is in this control. |
| OnMouseMove | Action to be taken if the mouse cursor moves while focus is in this control. |
| OnMouseDown | Action to be taken if the mouse button is down while focus is in this control. |
| OnMouseUp | Action to be taken if the mouse button is up while the cursor is over this control. Implies that the mouse button was previously down and has been released. The case where the cursor enters the control but the mouse button has not yet been pressed is covered by OnEntry or OnMouseEnter. |
| OnResize | Action to be taken when the control is resized. Might include re-alignment of text or selection of a different font size etc. |

### Constructors & Destructors

These are two special methods associated with each control:

- **Constructors:** such as **Create** allocate memory and system resources needed by the object. They also call the constructor of any sub-objects present in the class.
- **Destructors:** remove the object and de-allocate memory and other resources. If you call **Destroy** for an object which hasn't being initialized yet it will generate an error. Always use the **Free** method to deallocate objects, because it checks whether an object's value is **nil** before invoking **Destroy**.

Take the following precautions when creating your own **Destroy** method:

- Declare **Destroy** with the **override** directive, because it is a **virtual** method.
- Always call **'inherited Destroy;'** as the last thing on the destructor code
- Be aware that an **exception** may be raised on the **constructor** in case there is not enough memory to create an object, or something else goes wrong. If the **exception** is not handled inside the constructor, the object will be only partially built. In this case **Destroy** will be called when you weren't expecting it, so your destructor must check if the resources were really allocated before disposing of them.
- Remember to call **Free** for all objects created on the constructor.

## Menu controls

| Deutsch (de) | English (en) | español (es) | suomi (fi) | français (fr) | magyar (hu) | Bahasa Indonesia (id) | italiano (it) | 日本語 (ja) | 한국어 (ko) | македонски (mk) | Nederlands (nl) | português (pt) | русский (ru) | slovenčina (sk) | shqip (sq) | 中文（中國大陸） (zh_CN) | 中文（台灣） (zh_TW) |

### TMainMenu

TMainMenu is the Main Menu that appears at the top of most forms; form designers can customise by choosing various menu items.

TMainMenu is a non-visual component: that is, if the icon is selected from the Component Palette and placed on the form, it will not appear at run time. Instead, a menu bar with a structure defined by the Menu Editor will appear.

### TPopupMenu

TPopupMenu is a menu window that pops up with pertinent, usually context-sensitive, details and choices when the right mouse button is clicked on a control if the popupmenu is linked to the PopupMenu property of that component, thus providing a context sensitive menu for that component.

### Menu editor

To see the Menu Editor, right-click on the Main Menu or Popup Menu icon on your Form. A pop-up box appears that invites you to enter items into the Menu bar.

An edit box is displayed, containing a button labeled "New Item1". If you right-click on that box, a pop-up menu is displayed that allows you to add a new item before or after (along the same level) or create a sub-menu with the opportunity to add further items below (or above) the new item in a downward column.

Any of the TMenuItems that you add can be configured using the Object Inspector.

At the least you should give each item a *Caption* which will appear on the Menu Bar. The caption should indicate the activity to be selected, such as "File Open" or "Close", "Run" or "Quit". You may also wish to give it a more meaningful *Name*.

If you want a particular letter in the Caption to be associated with a shortcut key, that letter should be preceded by an ampersand (&). The Menu item at run-time will appear with the shortcut letter underlined, and hitting that letter key will have the same effect as selecting the menu item. Alternatively you can choose a shortcut key sequence (such as Ctrl + C for Copy or Ctrl + V for Paste - the standard keyboard shortcuts) with the *ShortCut* property of the TMenuItem.

### ActionList use

It is often helpful to use the Menu controls in conjunction with a TActionList which contains a series of standard or customised TActions. Menu items can be linked in the Object Inspector to *actions* on the list, and the same actions can be linked to TButtons, TToolButtons, TSpeedButtons etc. It is obviously more efficient to re-use the same code to respond to the various events, rather than writing separate *OnClick* event handlers for each individual control.

By default, a number of standard actions are pre-loaded from *StdActns* or, if DataAware controls are used, from *DBActns*. These actions can be chosen using the ActionList editor which appears when you right-click on the TActionList icon on the Form Designer.

## The Debugger

*Still to be written.*

1) Make sure you read the setup: Debugger_Setup

2) See also category: IDE Window - Debug

3) Read limitations: GDB_Debugger_Tips#Inspecting_data_types_.28Watch.2FHint.29 This page also helps with some platform/version specific issues

## The Lazarus files

```
    (Thanks to Kevin Whitefoot.)
    (Additions by Giuseppe Ridinò, User:Kirkpatc and Tom Lisjac)
```

When you save you will actually be saving two files:

```
   xxx.pas and yyy.lpr
```

(You save more than that but those are the ones you get to name). The project file (lpr) and the unit file (pas) must not have the same name because Lazarus will helpfully rename the unit (inside the source code) to the same as the unit file name and the program to the name of the project file (it needs to do this or the compiler will probably not be able to find the unit later when referred to in the project file). Of course to be consistent it changes all the occurrences of unit1 to xxx.

So if you are saving a project called **again**, trying to save again.pas and again.lpr fails because unit names and program names are in the same name space resulting in a duplicate name error.

So here is what I ended up with:

e:/lazarus/kj/lazhello:

```
total 4740  free 76500
-rwxrwxrwx   1 kjwh     root  4618697 Mar 24 11:19 again.exe
-rw-rw-rw-   1 kjwh     root     3002 Mar 24 11:21 again.lpi
-rw-rw-rw-   1 kjwh     root      190 Mar 24 11:18 again.lpr
-rw-rw-rw-   1 kjwh     root      506 Mar 24 11:08 againu.lfm
-rw-rw-rw-   1 kjwh     root      679 Mar 24 11:08 againu.lrs
-rw-rw-rw-   1 kjwh     root      677 Mar 24 11:08 againu.pas
-rw-rw-rw-   1 kjwh     root     2124 Mar 24 11:08 againu.ppu
-rwxrwxrwx   1 kjwh     root      335 Mar 24 11:07 ppas.bat
```

Note that there are many more files than the two that I thought I was saving.

Here is a brief note about each file:

**again.exe:** The main program binary executable. Win32 adds an "exe" extension. Linux has none (just the name of the program). This file will be huge in Linux due to the inclusion of debugging symbols. Run the "strip" utility to remove them and substantially shrink the executable size.

**again.lpi:** (Lazarus Project Information). This is the main information file of a Lazarus project; the equivalent Delphi main file of an application will be the .dpr file. It is stored in an XML format and contains instructions about all the libraries and units required to build the executable file.

**again.lpr:** The main program source file or master file. Despite its Lazarus specific extension it is in fact a perfectly normal Pascal source file. It has a uses clause that lets the compiler find all the units it needs. Note that the program statement does not have to name the program the same as the file name. This file is usually fairly small, with just a few statements to initialise, build the forms, run and close the application. Most of the work is done in the unit source files, with suffix '.pas'

**againu.lfm:** This is where Lazarus stores the layout of the form unit, in human readable form. It reflects the properties of the various components, as set in the Object Inspector. Each object description starts with a line:

```
object xxxx
  then there follows a list of properties
  (including embedded or nested objects) then an
end
```

line. Lazarus uses this file to generate a resource file (.lrs) that is included in the initialisation section of the againu.pas unit. Delphi dfm files can be converted to lfm format in the Lazarus IDE using the Tools->Convert DFM file to LFM utility.

**againu.lrs:** This is the generated resource file which contains the instructions to the program for building the form (if you look in the main Unit file, you will see in the initialization section the line

```
{$i againu.lrs}
```

which instructs the program to load the resource file). Note that it is not a Windows resource file.

**againu.pas:** The unit that contains the code for the form; this is usually the only file that the application programmer needs to edit or inspect, and contains any code specifically supplied by the programmer (especially event handlers).

**againu.ppu:** This is the compiled unit which gets linked into the executable file together with any other units named in the Uses section.

**ppas.bat:** This is a simple script that links the program to produce the executable. If compilation is successful, it is deleted by the compiler.

## See also

- Lazarus IDE Tools

Categories: Lazarus | Tutorials