Multiplatform Programming Guide/ru

From Free Pascal wiki

Jump to navigationJump to search

| Deutsch (de) | English (en) | español (es) | français (fr) | 日本語 (ja) | polski (pl) | **русский (ru)** | 中文（中国大陆） (zh_CN) |

Большинство LCL приложений работают в кроссплатформенном режиме без каких-либо дополнительных усилий.

Это руководство по написанию кроссплатформенных приложений с использованием Lazarus и Free Pascal. Он будет охватывать необходимые меры предосторожности, чтобы помочь в создании кроссплатформенной программы, готовой к Развертывание вашего приложения.

Введение в мультиплатформенное (кроссплатформенное) программирование

Сколько платформ вам нужно?

Чтобы ответить на этот вопрос, вы должны сначала определить, кто ваши потенциальные пользователи и как ваша программа будет использоваться. Этот вопрос зависит от того, где вы развертываете свое приложение.

Если вы разрабатываете стандартное настольное программное обеспечение в 2014 году, Microsoft Windows может быть самой важной платформой. Обратите внимание, что macOS и/или Linux набирают популярность и могут стать важной целью для вашего приложения.

Популярность различных операционных систем для настольных компьютеров отличается в зависимости от страны, типа используемого программного обеспечения и целевой аудитории; тут нет общего правила. Например, macOS довольно популярна в Северной Америке и Западной Европе, в то время как в Южной Америке компьютеры Mac в основном ограничены работой с видео и звуком.

Для многих контрактных проектов важна только одна платформа. Free Pascal и Lazarus вполне способны писать программы, ориентированные на конкретную платформу. Вы можете, например, получить доступ ко всему API Windows, чтобы написать хорошо интегрированную программу Windows.

Если вы разрабатываете программное обеспечение, которое будет работать на веб-сервере, обычно используется платформа Unix в одном из ее различных вариантов. В этом случае, возможно, только Linux, Solaris, * BSD и другие Unix-системы имеют смысл в качестве целевых платформ, хотя вы можете добавить поддержку Windows для полноты [картины].

Решив любые кросс-платформенные проблемы в своем дизайне, вы можете в значительной степени игнорировать другие платформы, так же, как и при разработке для одной платформы. Однако в какой-то момент вам нужно будет протестировать развертывание и запуск вашей программы на других платформах. Для этого будет полезно иметь неограниченный доступ к машинам с целевыми операционными системами. Если вам не нужны несколько физических компьютеров, попробуйте решения с двойной загрузкой или виртуальные машины, такие как VMware или Parallels.

Кроссплатформенное программирование

Работа с файлами и папками

При работе с файлами и папками важно использовать платформо-независимые разделители путей и [маркеры] конца строки. Вот список объявленных констант в Lazarus, которые будут использоваться при работе с файлами и папками.

- **PathSep**, **PathSeparator**: разделитель пути при добавлении нескольких путей вместе (';', ...)
- **PathDelim**, **DirectorySeparator**: разделитель каталогов для каждой платформы ('/', '\', ...)
- **LineEnding**: правильная последовательность символов окончания строки (#13#10 - CRLF, #10 - LF, ...)

Еще одна важная вещь, которую следует отметить, - это чувствительность к регистру [имен файлов и каталогов] файловой системы. В Windows имена файлов обычно не чувствительны к регистру, в то время как они обычно [регистрозависимы] на платформах Linux и BSD. Но если файловая система EXT2, EXT3 и т.д.

смонтирована в Windows, она будет чувствительна к регистру. Соответственно, файловая система FAT, смонтированная в Linux, не должна учитывать регистр символов.

Следует обратить особое внимание, что NTFS не чувствительна к регистру при использовании в Windows, но она чувствительна к регистру при монтировании ОС POSIX. Это может вызвать различные проблемы, в том числе потерю файлов, если файлы с одинаковыми именами файлов в разных случаях существуют в разделе NTFS, смонтированном в Windows. Разработчики должны рассмотреть возможность использования пользовательских функций для проверки и предотвращения создания нескольких файлов с одинаковыми именами в NTFS.

macOS по умолчанию использует имена файлов без учета регистра. Это может быть причиной досадных ошибок, поэтому любое переносимое приложение должно постоянно использовать имена файлов.

RTL-функции файлов используют системную кодировку для имен файлов. Под Windows это одна из кодовых страниц Windows, в то время как Linux, BSD и macOS обычно используют UTF-8. Модуль **FileUtil** [библиотеки] LCL предоставляет файловые функции, которые принимают строки UTF-8, как и остальная часть LCL.

```
// [функциям] AnsiToUTF8 и UTF8ToAnsi нужен менеджер широких строк (widestringmanager)
под Linux, BSD, macOS,
// но обычно эти ОС используют UTF-8 в качестве системной кодировки, поэтому [там]
менеджер широких строк
// не нужен.
function NeedRTLAnsi: boolean;// true, если системная кодировка не UTF-8
procedure SetNeedRTLAnsi(NewValue: boolean);
function UTF8ToSys(const s: string): string;// как UTF8ToAnsi, но более не зависим от
widestringmanager
function SysToUTF8(const s: string): string;// как AnsiToUTF8, но более не зависим от
widestringmanager
function UTF8ToConsole(const s: string): string;// преобразовывает строку UTF8 в
консольную кодировку (используется Write, WriteLn)

// файловые операции
function FileExistsUTF8(const Filename: string): boolean;
function FileAgeUTF8(const FileName: string): Longint;
function DirectoryExistsUTF8(const Directory: string): Boolean;
function ExpandFileNameUTF8(const FileName: string): string;
function ExpandUNCFileNameUTF8(const FileName: string): string;
function ExtractShortPathNameUTF8(Const FileName : String) : String;
function FindFirstUTF8(const Path: string; Attr: Longint; out Rslt: TSearchRec):
Longint;
function FindNextUTF8(var Rslt: TSearchRec): Longint;
procedure FindCloseUTF8(var F: TSearchrec);
function FileSetDateUTF8(const FileName: String; Age: Longint): Longint;
function FileGetAttrUTF8(const FileName: String): Longint;
function FileSetAttrUTF8(const Filename: String; Attr: longint): Longint;
function DeleteFileUTF8(const FileName: String): Boolean;
function RenameFileUTF8(const OldName, NewName: String): Boolean;
function FileSearchUTF8(const Name, DirList : String): String;
function FileIsReadOnlyUTF8(const FileName: String): Boolean;
function GetCurrentDirUTF8: String;
function SetCurrentDirUTF8(const NewDir: String): Boolean;
function CreateDirUTF8(const NewDir: String): Boolean;
function RemoveDirUTF8(const Dir: String): Boolean;
function ForceDirectoriesUTF8(const Dir: string): Boolean;

// окружение
function ParamStrUTF8(Param: Integer): string;
function GetEnvironmentStringUTF8(Index: Integer): string;
function GetEnvironmentVariableUTF8(const EnvVar: string): String;
function GetAppConfigDirUTF8(Global: Boolean): string;

// другое
function SysErrorMessageUTF8(ErrorCode: Integer): String;
```

Прим.перев.: после появления поддержки юникода на уровне компилятора (FPC 2.7.1) вместо AnsiToUTF8, UTF8ToAnsi, SysToUTF8, UTF8ToAnsi для работы с WinAPI рекомендуется использовать функции **UTF8ToWinCP** и **WinCPToUTF8**. Подробнее здесь: RTL с кодовой страницей UTF-8 по умолчанию

Пустые имена файлов и двойные разделители путей

Существуют различия в обработке имен файлов/каталогов в Windows по сравнению с Linux/Unix/Unix-подобными системами.

- Windows позволяет пустые имена файлов. Вот почему FileExistsUTF8 ('..\') проверяет в Windows в родительском каталоге наличие файла без имени.
- В Linux/Unix/Unix-подобных системах пустой файл сопоставляется с каталогом, а каталоги рассматриваются как файлы. Это означает, что FileExistsUTF8 ('../') в Unix проверяет наличие родительского каталога, что обычно приводит к значению true.

Двойные разделители пути в именах файлов также обрабатываются по-разному:

- Windows: 'C:\' не то же самое, что 'C:\\'
- Unix-подобные OS: путь '/usr//' совпадает с '/usr/'. Если '/usr' является каталогом, то даже все три равнозначны.

Это важно при объединении имен файлов. Например:

```
FullFilename:=FilePath+PathDelim+ShortFilename; // может привести к двум PathDelims, которые дают разные результаты под Windows и Linux
FullFilename:=AppendPathDelim(FilePath)+ShortFilename); // создает только один PathDelim
FullFilename:=TrimFilename(FilePath+PathDelim+ShortFilename); // создает только один PathDelim и делает еще несколько чисток
```

Прим.перев.: для "нормализации" количества и вида разделителей в путях папок и файлов можно использовать следующие функции из модуля `LazFileUtils`

```
function CleanAndExpandDirectory(const Filename: string): string;
function CleanAndExpandFilename(const Filename: string): string;
```

Функция TrimFilename заменяет разделители двойных путей одиночными и сокращает пути '..'. Например /usr//lib/../src обрезается до /usr/src.

Если вы хотите узнать, существует ли каталог, используйте **DirectoryExistsUTF8**.

Другая распространенная задача - проверить, существует ли часть пути имени файла. Вы можете получить путь с помощью ExtractFilePath, но он будет содержать разделитель пути.

- Под Unix-подобной системой вы можете просто использовать FileExistsUTF8 в пути. Например, FileExistsUTF8('/home/user/') вернет true, если каталог /home/user существует.
- В Windows вы должны использовать функцию DirectoryExistsUTF8, но перед этим вы должны удалить разделитель пути, например, с помощью функции ChompPathDelim.

В Unix-подобных системах корневым каталогом является '/', а использование функции ChompPathDelim создаст пустую строку. Функция DirPathExists работает как функция DirectoryExistsUTF8, но обрезает заданный путь.

Обратите внимание, что Unix/Linux использует символ '~' (тильда) для обозначения домашнего каталога, обычно '/home/jim/' для пользователя с именем jim. Так что '~/myapp/myfile' и '/home/jim/myapp/myfile' идентичны в командной строке и в скриптах. Тем не менее, тильда не будет автоматически расширяться Lazarus'ом. Необходимо использовать ExpandFileNameUTF8('~/myapp/myfile'), чтобы получить полный путь.

Прим.перев.: Если code completion "не находит" выше описанные функции, добавьте в секцию uses модули **LazUTF8** и **LazFileUtils**.

Кодировка текста

Текстовые файлы часто кодируются в текущей кодировке системы. В Windows это обычно одна из кодовых страниц Windows, в то время как Linux, BSD и macOS обычно используют UTF-8. Не существует 100%-го правила, чтобы узнать, какую кодировку использует текстовый файл. Модуль LCL **Iconvencoding** имеет функцию для угадывания кодировки:

```
function GuessEncoding(const s: string): string;
function GetDefaultTextEncoding: string;
```

И он содержит функции для преобразования из одной кодировки в другую:

```
function ConvertEncoding(const s, FromEncoding, ToEncoding: string): string;

function UTF8BOMToUTF8(const s: string): string; // UTF8 с BOM
function ISO_8859_1ToUTF8(const s: string): string; // Центральная Европа
function CP1250ToUTF8(const s: string): string; // Центральная Европа
function CP1251ToUTF8(const s: string): string; // кириллица
function CP1252ToUTF8(const s: string): string; // latin 1
...
function UTF8ToUTF8BOM(const s: string): string; // UTF8 с BOM
function UTF8ToISO_8859_1(const s: string): string; // Центральная Европа
function UTF8ToCP1250(const s: string): string; // Центральная Европа
function UTF8ToCP1251(const s: string): string; // кириллица
function UTF8ToCP1252(const s: string): string; // latin 1
...
```

Например, чтобы загрузить текстовый файл и преобразовать его в UTF-8, вы можете использовать:

```
var
  sl: TStringList;
  OriginalText: String;
  TextAsUTF8: String;
begin
  sl:=TStringList.Create;
  try
    sl.LoadFromFile('sometext.txt'); // осторожно: это изменяет конец строки на конец
строки ОСи
    OriginalText:=sl.Text;

TextAsUTF8:=ConvertEncoding(OriginalText,GuessEncoding(OriginalText),EncodingUTF8);
    ...
  finally
    sl.Free;
  end;
end;
```

А для сохранения текстового файла в системной кодировке вы можете использовать:

```
sl.Text:=ConvertEncoding(TextAsUTF8,EncodingUTF8,GetDefaultTextEncoding);
sl.SaveToFile('sometext.txt');
```

Конфигурационные файлы

Вы можете использовать функцию GetAppConfigDir из модуля SysUtils, чтобы найти подходящее место для хранения файлов конфигурации в другой системе. Функция имеет один параметр, называемый Global. Если [его значение] - True, то возвращаемый каталог является глобальным каталогом, то есть действительным для всех пользователей в системе. Если параметр Global имеет значение false, тогда каталог является специфическим для пользователя, выполняющего программу. В системах, которые не поддерживают многопользовательские среды, эти два каталога могут быть одинаковыми.

Существует также [функция] GetAppConfigFile, который возвращает соответствующее имя для файла конфигурации приложения. Вы можете использовать это так:

```
ConfigFilePath := GetAppConfigFile(False);
```

Ниже приведены примеры вывода функций пути по умолчанию в различных системах:

```
program project1;
```

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

uses
  SysUtils;

begin
  WriteLn(GetAppConfigDir(True));
  WriteLn(GetAppConfigDir(False));
  WriteLn(GetAppConfigFile(True));
  WriteLn(GetAppConfigFile(False));
end.
```

Вывод в системе GNU/Linux с FPC 2.2.2. Обратите внимание, что использование True является ошибкой, что уже исправлено в 2.2.3:

```
/etc/project1/
/home/user/.config/project1/
/etc/project1.cfg
/home/user/.config/project1.cfg
```

Вы можете заметить, что глобальные файлы конфигурации хранятся в каталоге /etc, а локальные конфигурации хранятся в скрытой папке в домашнем каталоге пользователя. Каталоги, имя которых начинается с точки (.), скрыты в Linux. Вы можете создать каталог в месте, возвращаемом GetAppConfigDir, а затем сохранить там файлы конфигурации.

Примечание: Обычные пользователи не могут писать в каталог /etc. Это могут делать только пользователи с правами администратора.

Вывод на последние версии Windows с FPC 3.0.0 +:

```
C:\ProgramData\project1\
C:\Users\user\AppData\Local\project1\
C:\ProgramData\project1\project1.cfg
C:\Users\user\AppData\Local\project1\project1.cfg
```

Обратите внимание, что до FPC 2.2.4 функция использовала каталог, в котором приложение должно было хранить глобальные конфигурации в Windows.

Вывод на Windows 98 с FPC 2.2.0:

```
C:\Program Files\PROJECT1
C:\Windows\Local Settings\Application Data\PROJECT1
C:\Program Files\PROJECT1\PROJECT1.cfg
C:\Windows\Local Settings\Application Data\PROJECT1\PROJECT1.cfg
```

Вывод на macOS с FPC 2.2.0:

```
/etc
/Users/user/.config/project1
/etc/project1.cfg
/Users/user/.config/project1/project1.cfg
```

Примечание: Использование UPX мешает использованию функций GetAppConfigDir и GetAppConfigFile.

Прим.перев.: Очевидно, имеется ввиду библиотека упаковщика исполняемых файлов - UPX (the Ultimate Packer for eXecutables)

Примечание: В macOS файлы конфигурации в большинстве случаев являются файлами предпочтений, которые должны быть файлами XML с окончанием ".plist" и храниться в /Library/Preferences или ~/Library/Preferences с именами, взятыми из поля "Bundle identifier"(Идентификатор пакета) в Info.plist пакета приложения. Использование вызовов Carbon CFPreference ... вероятно, самый простой способ добиться этого. Файлы .config в каталоге User являются нарушением указаний по программированию.

Файлы данных и ресурсов

Очень распространенный вопрос - где хранить файлы данных, которые могут понадобиться приложению, такие как изображения, музыка, файлы XML, файлы базы данных, файлы справки и т.д. К сожалению, нет кроссплатформенной функции, которая бы обеспечивала лучшее место для поиска файлов данных. Решение состоит в том, чтобы реализовать по-разному на каждой платформе, используя [директивы условной компиляции] IFDEF.

Windows

В Windows данные приложения, которые изменяет программа, должны быть помещены не в каталог приложения (например, C:\Program Files\), а в определенном месте (см., например, Как написать приложение для Windows XP, в разделе "Classify Application Data"). Windows Vista и новее активно применяют это (пользователи имеют доступ на запись в эти каталоги только при использовании повышения [прав] или отключения UAC), но используют механизм перенаправления папок для поддержки старых, неправильно запрограммированных приложений. Просто чтение, а не запись данных из каталогов приложений все равно будет работать.

Короче говоря: используйте такую папку:

```
OpDirLocal:= GetEnvironmentVariableUTF8('appdata')+'\MyAppName';
```

См. Советы по программированию Windows: получение специальных папок (Мои документы, Рабочий стол, локальные данные приложения и т.д.).

Unix/Linux

В большинстве Unix-систем (таких как Linux, FreeBSD, OpenBSD, Solaris) файлы данных приложения расположены в фиксированном месте, которое может выглядеть примерно так: /usr/share/app_name или /opt/app_name.

Данные приложения, которые должны быть записаны приложением, часто хранятся в таких местах, как /var/<programname>, с установленными соответствующими разрешениями.

Специфичные для пользователя конфигурации/данные для чтения/записи обычно хранятся где-то в домашнем каталоге пользователя (например, в ~/.myfancyprogram).

Как получить путь к этому домашнему каталогу:

```
OpDirLocal:= GetEnvironmentVariableUTF8('HOME')+'/.myappname';
```

macOS

macOS является исключением среди UNIX. Приложение публикуется в связке - каталоге с расширением "app", которое обрабатывается файловым менеджером как файл (вы также можете сделать "cd path/myapp.app"). Ваши файлы ресурсов должны находиться внутри пакета. Если пакет - "path/MyApp.app", то:

- исполняемый файл - "path/MyApp.app/Contents/MacOS/myapp"
- каталог ресурсов - "path/MyApp.app/Contents/Resources"

Сохраните файлы конфигурации в домашний каталог:

```
OpDirLocal:= GetEnvironmentVariableUTF8('HOME')+'/.myappname';
```

Читайте ресурсы с:

```
OpDirRes:= ExtractFileDir(ExtractFileDir(Application.ExeName))+'/Resources';
```

Предупреждение: никогда не используйте `paramstr(0)` на любой платформе Unix для определения расположения исполняемого файла, так как это соглашение Dos-Windows-OS/2 и имеет несколько концептуальных проблем, которые не могут быть решены с помощью эмуляции на других платформ.

Единственное, что paramstr(0) гарантированно возвращает на платформах Unix, это имя, под которым была запущена программа. Каталог, в котором он находится, и имя фактического бинарного файла (в случае, если он был запущен с использованием символической ссылки) не обязательно будут доступны через paramstr(0).

32/64 bit

Обнаружение разрядности ОСи во время выполнения

Хотя вы можете контролировать, компилируете ли вы 32- или 64-битную версию с помощью определений компилятора, иногда вы захотите знать, какова разрядность операционной системы. Например, если вы запускаете 32-битную программу Lazarus в 64-битной Windows, вы можете запустить внешнюю программу из каталога Program Files (x86) для 32-битных программ или вы можете захотеть получить различную информацию пользователя: мне это нужно в моей программе установки Лазаруса LazUpdater, чтобы предложить пользователю выбор 32- и 64-битных компиляторов. Код: пример определения Windows x32-x64.

Обнаружение разрядности внешней библиотеки перед ее загрузкой

Если вы хотите загрузить функции из динамической библиотеки в вашу программу, она должна иметь ту же разрядность, что и ваше приложение. В 64-битной Windows ваше приложение может быть 32-битным или 64-битным, и в вашей системе могут быть 32-битные и 64-битные библиотеки. Поэтому вы можете проверить, является ли разрядность dll такой же, как разрядность вашего приложения, прежде чем загружать dll динамически. Вот функция, которая проверяет разрядность dll, (предоставленная на форуме пользователем GetMem):

```
uses {..., } JwaWindows;

function GetPEType(const APath: WideString): Byte;
const
  PE_UNKNOWN = 0; // если файл не является корректной dll, возвращается 0
 // PE_16BIT   = 1; // не поддерживается этой функцией
  PE_32BIT   = 2;
  PE_64BIT   = 3;
var
  hFile, hFileMap: THandle;
  PMapView: Pointer;
  PIDH: PImageDosHeader;
  PINTH: PImageNtHeaders;
  Base: Pointer;
begin
  Result := PE_UNKNOWN;

  hFile := CreateFileW(PWideChar(APath), GENERIC_READ, FILE_SHARE_READ, nil,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
  if hFile = INVALID_HANDLE_VALUE then
  begin
    CloseHandle(hFile);
    Exit;
  end;

  hFileMap  := CreateFileMapping(hFile, nil, PAGE_READONLY, 0, 0, nil);
  if hFileMap = 0 then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    Exit;
  end;

  PMapView := MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 0);
  if PMapView = nil then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    Exit;
  end;
```

```
  PIDH := PImageDosHeader(PMapView);
  if PIDH^.e_magic <> IMAGE_DOS_SIGNATURE then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    UnmapViewOfFile(PMapView);
    Exit;
  end;

  Base := PIDH;
  PINTH := PIMAGENTHEADERS(Base + LongWord(PIDH^.e_lfanew));
  if PINTH^.Signature = IMAGE_NT_SIGNATURE then
  begin
    case PINTH^.OptionalHeader.Magic of
      $10b: Result := PE_32BIT;
      $20b: Result := PE_64BIT
    end;
  end;

  CloseHandle(hFile);
  CloseHandle(hFileMap);
  UnmapViewOfFile(PMapView);
end;

//Теперь, если вы компилируете свое приложение для 32-битной и 64-битной windows, вы
можете проверить, является ли разрядность dll такой же, как у вашего приложения:
function IsCorrectBitness(const APath: WideString): Boolean;
begin
  {$ifdef CPU32}
    Result := GetPEType(APath) = 2; //приложение скомпилировано как 32-битное, мы
спрашиваем, возвращает ли GetPeType 2
  {$endif}
  {$ifdef CPU64}
    Result := GetPEType(APath) = 3; //приложение скомпилировано как 64-битное, мы
спрашиваем, возвращает ли GetPeType 3
  {$endif}
end;
```

Приведение типов Pointer/Integer

Для указателей под 64-бит требуется 8 байтов вместо 4-х на 32-битных. Тип 'Integer' остается 32-битным на всех платформах для совместимости. Это означает, что вы не можете приводить тип pointers к целым числам и обратно.

FPC задает для этого два типа: PtrInt и PtrUInt. PtrInt представляет собой 32-битное целое число со знаком на 32-битных платформах и 64-битное целое число со знаком на 64-битных платформах. То же самое для PtrUInt, но только - это целое число без знака.

Используйте для кода, который должен работать с Delphi и FPC:

```
 {$IFNDEF FPC}
 type
   PtrInt = integer;
   PtrUInt = cardinal;
 {$ENDIF}
```

Замените все **integer(SomePointerOrObject)** на **PtrInt(SomePointerOrObject)**.

Порядок байтов

Платформы Intel имеют младший порядок байтов, это означает, что младший байт стоит первым. Например, два байта слова $1234 хранятся как $34 $12 в системах с младшим порядком байтов. В системах со старшим порядком байтов, таких как powerpc, два байта слова $1234 хранятся как $12 $34. Разница важна при чтении файлов, созданных в других системах.

Прим.перев.: подробнее о порядке байтов можно почитать в Википедии

Используйте для кода, который должен работать в обоих [случаях]:

```
{$IFDEF ENDIAN_BIG}
...
{$ELSE}
...
{$ENDIF}
```

И наоборот для ENDIAN_LITTLE.

Системный модуль предоставляет достаточно много функций преобразования порядка байтов, таких как SwapEndian, BEtoN (со старшим порядком байтов в текущий), LEtoN (с младшим порядком байтов в текущий), NtoBE (с текущим порядком байтов в старший) и NtoLE (с текущим порядком байтов в младший).

Libc и другие специальные модули

Избегайте устаревших модулей, таких как "oldlinux" и "libc", которые не поддерживаются вне linux/i386.

Ассемблер

Избегайте использования ассемблера.

Директивы компилятора

```
{$ifdef CPU32}
...напишите здесь код для 32-битных процессоров
{$ENDIF}
{$ifdef CPU64}
...напишите здесь код для 64-битных процессоров
{$ENDIF}
```

Проекты, пакеты и пути поиска

Проекты и пакеты Lazarus предназначены для нескольких платформ. Обычно вы можете просто скопировать проект и необходимые пакеты на другой компьютер и скомпилировать их там. Вам не нужно создавать по отдельному проекту для каждой платформы.

Несколько советов для достижения этого.

Компилятор создает для каждого модуля ppu-файл с таким же именем. Этот ppu-файл может быть использован другими проектами и пакетами. Исходные файлы модулей (например, unit1.pas) не должны быть общими. Просто дайте компилятору выходной каталог модуля, в котором нужно создавать файлы ppu. IDE делает это по умолчанию, поэтому здесь вам нечего делать.

Каждый файл модуля должен быть частью **одного** проекта или пакета. Если файл модуля используется только одним проектом, добавьте его в этот проект. В противном случае добавьте его в пакет. Если вы еще не создали пакет для общих модулей, см. здесь: создание пакета для ваших общих модулей.

Каждый проект и каждый пакет должны иметь **разобщенные каталоги** - они не должны совместно использовать каталоги. В противном случае вы должны быть экспертом в искусстве поиска путей компилятора. Если вы не являетесь экспертом или если те, кто может использовать ваш проект/пакет, не являются экспертами: не делитесь каталогами между проектами/пакетами.

Платформозависимые модули

Например, модуль wintricks.pas должен использоваться только под Windows. В разделе uses используйте:

```
uses
  Classes, SysUtils
  {$IFDEF Windows}
  ,WinTricks
  {$ENDIF}
  ;
```

Если модуль является частью пакета, вы также должны выбрать модуль в редакторе пакетов и снять флажок *Use unit*(Использовать модуль).

См. также платформозависимые модули

Платформозависимые пути поиска

Когда вы ориентируетесь на несколько платформ и обращаетесь к операционной системе напрямую, вы быстро устаете от бесконечных конструкций IFDEF. Одним из решений, которое часто используется в исходниках FPC и Lazarus, является использование include-файлов. Создайте по одному подкаталогу для [каждой] целевой [ОС]. Например, win32, linux, bsd, darwin. Поместите в каждый каталог включаемый файл с тем же именем. Затем используйте макрос во включаемом пути. Модуль может использовать обычную включающую директиву. Пример для одного include-файла для каждого набора виджетов LCL:

Создайте один файл для каждого набора виджетов, который вы хотите поддерживать:

```
win32/example.inc
gtk/example.inc
gtk2/example.inc
carbon/example.inc
```

Вам не нужно добавлять файлы в пакет или проект. Добавьте включаемый путь поиска *$(LCLWidgetType)* в параметры компилятора вашего пакета или проекта.

В вашем модуле используйте директиву:

```
{$I example.inc}
```

Вот некоторые полезные макросы и общие значения:

- LCLWidgetType: win32, gtk, gtk2, qt, carbon, fpgui, nogui
- TargetOS: linux, win32, win64, wince, freebsd, netbsd, openbsd, darwin (многое другое)
- TargetCPU: i386, x86_64, arm, powerpc, sparc
- SrcOS: win, unix

Вы можете использовать макрос $Env() для использования переменных окружения.

И конечно вы можете использовать комбинации. Например, LCL использует:

```
$(LazarusDir)/lcl/units/$(TargetCPU)-$(TargetOS);$(LazarusDir)/lcl/units/$(TargetCPU)-$(TargetOS)/$(L
```

Смотрите здесь полный список макросов: макросы в путях и именах файлов

Аппаратно- / пользователь-зависимые пути поиска

Например, у вас есть две машины Windows, Stan'a и Oliver'a. На [машине] Stan'a ваши модули находятся в *C:\units*, а на [машине] Oliver'a ваши модули находятся в *D:\path*. Модули принадлежат пакету *SharedStuff*, который представляет собой *C:\units\sharedstuff.lpk* на [машине] Stan'a и *D:\path\sharedstuff.lpk* на [машине] Oliver'a.

После того, как вы открыли lpk в IDE или в lazbuild, путь автоматически сохраняется в его файлах конфигурации (packagefiles.xml). При компиляции проекта, для которого требуется пакет SharedStuff, среда IDE и lazbuild знают, где он находится. Так что никакой [дополнительной] конфигурации не требуется.

Если вы хотите развернуть пакет на нескольких компьютерах или для всех пользователей компьютера (например, бассейн для студентов), вы можете добавить файл lpl в исходный каталог lazarus. Смотрите packager/globallinks для [получения] примеров.

Различия языковой среды

Некоторые функции из Free Pascal, такие как StrToFloat, ведут себя по-разному в зависимости от текущей языковой среды [ОСи]. Например, в США десятичный разделитель обычно равен ".", но во многих странах Европы и Южной Америки это ",". Это может быть проблемой, поскольку иногда желательно, чтобы эти функции вели себя фиксированным образом, независимо от языковых настроек операционной системы. Примером является формат файла с десятичными точками, который всегда должен интерпретироваться одинаково.

В следующих разделах объясняется, как это сделать.

StrToFloat

Новый набор настроек формата, которые устанавливают фиксированный десятичный разделитель, может быть создан с помощью следующего кода:

```
// в вашем файле проекта .lpr
uses
...
{$IFDEF UNIX}
clocale
{ Требуется в Linux/Unix для поддержки форматирования. Должен быть одним из первых
(вероятно, после cthreads?)}
{$ENDIF}
```

и:

```
// в вашем коде:
var
  FPointSeparator, FCommaSeparator: TFormatSettings;
begin
  // Настройки формата для преобразования строки в число с плавающей точкой
  FPointSeparator := DefaultFormatSettings;
  FPointSeparator.DecimalSeparator := '.';
  FPointSeparator.ThousandSeparator := '#';// отключаем тысячный разделитель
  FCommaSeparator := DefaultFormatSettings;
  FCommaSeparator.DecimalSeparator := ',';
  FCommaSeparator.ThousandSeparator := '#';// отключаем тысячный разделитель
```

Позже вы можете использовать настройки этого формата при вызове StrToFloat, например:

```
// Эта функция работает как StrToFloat, но просто пробует два возможных десятичных
разделителя
// Это позволит избежать исключения, если формат строки не соответствует локали
function AnSemantico.StringToFloat(AStr: string): Double;
begin
  if Pos('.', AStr) > 0 then Result := StrToFloat(AStr, FPointSeparator)
  else Result := StrToFloat(AStr, FCommaSeparator);
end;
```

Gtk2 и маскировка исключений FPU

Библиотека Gtk2 изменяет значение по умолчанию маски исключений FPU (модуль [для операций] с плавающей запятой). Следствием этого является то, что некоторые исключения с плавающей точкой не возникают, если приложение использует библиотеку Gtk2. Это означает, что, например, если вы разрабатываете приложение LCL в Windows с набором виджетов win32/64 (который по умолчанию для Windows) и планируете компилировать для Linux (где Gtk2 является набором виджетов по умолчанию), вы должны помнить об этой несовместимости.

После этой темы форума и ответов на этот багрепорт стало ясно, что с этим ничего не поделаешь, поэтому мы должны знать, в чем эти различия.

Поэтому давайте проведем тест:

```
uses
  ..., math,...

{...}

var
  FPUException: TFPUException;
  FPUExceptionMask: TFPUExceptionMask;
begin
  FPUExceptionMask := GetExceptionMask;
  for FPUException := Low(TFPUException) to High(TFPUException) do begin
    write(FPUException, ' - ');
    if not (FPUException in FPUExceptionMask) then
      write('not ');

    writeln('masked!');
```

```
  end;
  readln;
end.
```

Наша простая программа получит значение FPC по умолчанию:

```
 exInvalidOp - not masked!
 exDenormalized - masked!
 exZeroDivide - not masked!
 exOverflow - not masked!
 exUnderflow - masked!
 exPrecision - masked!
```

Однако в Gtk2 только exOverflow не маскируется.

Следствием этого является то, что исключения EInvalidOp и EZeroDivide не возникают, если приложение ссылается на библиотеку Gtk2! Обычно, деление ненулевого значения на ноль возбуждает исключение EZeroDivide, а деление нуля на ноль возбуждает исключение EInvalidOp. Например, такой код:

```
var
  X, A, B: Double;
// ...

try
  X := A / B;
  // блок кода 1
except
  // блок кода 2
end;
// ...
```

при компиляции в приложении с набором виджетов Gtk2 примет другое направление. На виндовом widgetset'e, когда **B** равно нулю, будет сгенерировано исключение (EZeroDivide или EInvalidOp, в зависимости от того, равен ли **A** нулю или нет) и будет выполнен «блок кода 2». На Gtk2 **X** становится Infinity (положительным бесконечным числом), NegInfinity (отрицательным бесконечным числом) или NaN (не числовым значением) и «блок кода 1» будет выполнен.

Мы можем придумать разные способы преодоления этого несоответствия. В большинстве случаев вы можете просто проверить, равен ли **B** нулю, и не пытаться делить в этом случае. Однако иногда вам потребуется другой подход. Итак, взгляните на следующие примеры:

```
uses
  ..., math,...

//...
var
  X, A, B: Double;
  Ind: Boolean;
// ...
try
  X := A / B;
  Ind := IsInfinite(X) or IsNan(X); // на gtk2 мы "падаем" здесь
except
  Ind := True; // на windows мы "падаем" здесь, когда B равен нулю
end;
if Ind then begin
  // блок кода 2
end else begin
  // блок кода 1
end;
// ...
```

Или:

```
uses
  ..., math,...
```

```
//...
var
  X, A, B: Double;
  FPUExceptionMask: TFPUExceptionMask;
// ...

try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]); // демаскируем
  try
    X := A / B;
  finally
    SetExceptionMask(FPUExceptionMask); // немедленно возвращаем предыдущую
маскировку, мы не должны позволять вызывать Gtk2 без маски
  end;
  // блок кода 1
except
  // блок кода 2
end;
// ...
```

Будьте осторожны, не делайте что-то подобное (вызовите LCL с все еще удаленной маской):

```
try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]);
  try
    Edit1.Text := FloatToStr(A / B); // НЕТ! Настройка присвоение значения тексту
Edit'a сводится к внутренним элементам набора виджетов, и Gtk2 API нельзя вызывать без
маски!
  finally
    SetExceptionMask(FPUExceptionMask);
  end;
  // блок кода 1
except
  // блок кода 2
end;
// ...
```

Но используйте вспомогательную переменную:

```
try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]);
  try
    X := A / B; // Сначала мы устанавливаем вспомогательную переменную X
  finally
    SetExceptionMask(FPUExceptionMask);
  end;
  Edit1.Text := FloatToStr(X); // Теперь мы можем присвоить значение текста Edit'a.
  // code block 1
except
  // code block 2
end;
// ...
```

Во всех ситуациях при разработке приложений LCL очень важно знать об этом и иметь в виду, что некоторые операции с плавающей запятой могут идти по-разному с разными наборами виджетов. Тогда вы можете придумать подходящий способ обойти это, но это не должно остаться незамеченным.

Проблемы при переходе с Windows на *nix и т.д.

Проблемы, специфичные для Linux, macOS, Android и другие Unix'ы, описаны здесь. Не все предметы могут применяться ко всем платформам.

В Unix нет "каталога приложений"

Многие программисты используют вызов ExtractFilePath(ParamStr(0)) или Application.ExeName для получения местоположения исполняемого файла, а затем поиска необходимых файлов для выполнения программы (изображений, файлов XML, файлов базы данных и т.д.) на основе расположения исполняемого файла. Это неправильно в Unix. Строка в ParamStr(0) может содержать каталог, отличный от каталога исполняемого файла, и он также зависит от различных программ командной оболочки (sh, bash и т.д.).

Даже если Application.ExeName на самом деле может знать каталог, в котором находится исполняемый файл, этот файл может быть символической ссылкой, так что вы можете вместо этого получить каталог ссылки (в зависимости от версии ядра Linux вы можете получить каталог ссылки или самого бинарного файла программы).

Чтобы избежать этого, прочитайте разделы о файлах конфигурации и файлах данных.

Обхождение без Windows COM Automation

В Windows COM Automation - это мощный способ не только удаленного манипулирования другими программами, но и предоставления другим программам возможности манипулировать вашей программой. С помощью Delphi вы можете сделать вашу программу одновременно клиентом автоматизации COM и сервером автоматизации COM, то есть она может манипулировать другими программами и, в свою очередь, [позволять] манипулировать [собой] другим программам. Например, см. Использование COM Automation для работы с OpenOffice и Microsoft Office.

Альтернатива [COM для] macOS

К сожалению, COM Automation недоступна в macOS и Linux. Однако вы можете имитировать некоторые функции COM Automation в macOS, используя AppleScript.

AppleScript в некотором роде похож на COM Automation. Например, вы можете писать скрипты, которые манипулируют другими программами. Вот очень простой пример AppleScript, который запускает NeoOffice (версия OpenOffice.org для Mac):

```
tell application "NeoOffice"
  launch
end tell
```

Приложение, разработанное для работы с AppleScript, предоставляет «словарь» классов и команд, которые можно использовать с приложением, аналогично классам сервера Windows Automation. Однако даже такие приложения, как NeoOffice, которые не предоставляют словарь, все равно будут реагировать на команды "launch" (запуск), "activate" (активация) и "quit" (выход). AppleScript может быть запущен из редактора сценариев macOS или Finder или даже преобразован в приложение, которое вы можете поместить на док-станцию, как любое другое приложение. Вы также можете запустить AppleScript из своей программы, как в этом примере:

```
fpsystem('myscript.applescript');
```

Предполагается, что скрипт находится в указанном файле. Вы также можете запускать сценарии на лету из своего приложения с помощью команды macOS OsaScript:

```
fpsystem('osascript -e '#39'tell application "NeoOffice"'#39 +
      ' -e '#39'launch'#39' -e '#39'end tell'#39);
      {Note use of #39 to single-quote the parameters}
```

Однако эти примеры являются эквивалентом следующей команды Open:

```
fpsystem('open -a NeoOffice');
```

Аналогично, в macOS вы можете эмулировать команды оболочки Windows для запуска веб-браузера и запуска почтового клиента с:

```
fpsystem('open -a safari
"http://gigaset.com/shc/0,1935,hq_en_0_141387_rArNrNrNrN,00.html"');
```

и

```
fpsystem('open -a mail "mailto:ss4200@invalid.org"');
```

что предполагает, [что это] достаточно безопасно, что в системе macOS будут установлены приложения Safari и Mail. Конечно, вы никогда не должны делать подобные предположения, и для двух предыдущих примеров вы можете просто положиться на macOS, чтобы поступить правильно и выбрать веб-браузер пользователя по умолчанию и почтовый клиент, если вы вместо этого используете эти варианты:

```
fpsystem('open "http://gigaset.com/shc/0,1935,hq_en_0_141387_rArNrNrNrN,00.html"');
```

и

```
fpsystem('open "mailto:ss4200@invalid.org"');
```

Не забудьте включить модуль Unix в вашу секцию uses, если вы используете `fpsystem` или `shell` (взаимозаменяемые).

Настоящая сила AppleScript заключается в удаленном управлении программами для создания и открытия документов и автоматизации других действий. Сколько вы можете сделать с [этой] программой, зависит от того, насколько обширным является ее словарь AppleScript (если он есть). Например, программы Microsoft Office X не очень пригодны для использования с AppleScript, тогда как более новые программы Office 2004 полностью переписали словари AppleScript, которые во многих отношениях сравниваются с тем, что доступно через серверы Windows Office Automation.

Альтернативы для Linux

В то время как оболочка Linux поддерживают сложные сценарии командной строки, тип сценариев ограничен тем, что может быть передано программе из командной строки. Не существует единого, унифицированного способа доступа к внутренним классам и командам программы в Linux, как в COM Automation Windows и AppleScript macOS. Однако отдельные среды рабочего стола (GNOME/KDE) и прикладные среды часто предоставляют такие методы межпроцессорного взаимодействия. В GNOME см. Bonobo Components. KDE имеет фреймворк KParts, DCOP. OpenOffice имеет платформонезависимый API для удаленного управления офисом (google OpenOffice SDK) - хотя вам, вероятно, придется написать связующий код на другом языке, который имеет привязки (например, Python), чтобы использовать его. Кроме того, в некоторых приложениях «режимы сервера» активируются специальными параметрами командной строки, которые позволяют управлять ими из другого процесса. Также возможно (Borland сделал это с помощью браузера документов Kylix) «встроить» одно окно приложения X верхнего уровня в другое, используя XReparentWindow (я думаю).

Как и в Windows, многие программы для macOS и Linux состоят из нескольких библиотечных файлов (расширения .dylib и .so). Иногда эти библиотеки разработаны так, что вы также можете использовать их в программах, которые вы пишете. Хотя это может быть способом добавления некоторых функций внешней программы к вашей программе, на самом деле это не то же самое, что запуск самой внешней программы и управление ею. Вместо этого ваша программа просто ссылается на библиотеку внешней программы и использует ее подобно тому, как она использует любую библиотеку программирования.

Альтернативы для функций Windows API

Многие программы Windows широко используют Windows API. В кроссплатформенных приложениях функции Win API в модуле Windows не должны использоваться или должны быть заключены в условную компиляцию (например, {$IFDEF MSWINDOWS}).

К счастью, многие из часто используемых функций Windows API реализованы многоплатформенным способом в модуле lclintf. Это может быть решением для программ, которые в значительной степени зависят от Windows API, хотя лучшее решение - заменить эти вызовы настоящими кроссплатформенными компонентами из LCL. Например, вы можете заменить вызовы функций рисования GDI вызовами методов объекта TCanvas.

Коды клавиш

К счастью, обнаружение кодов клавиш (например, по событиям KeyUp) является переносимым: см. LCL Key Handling.

Установка вашего приложения

См. Развертывание вашего приложения.

Смотри также