Введение

Начиная с 0.9.25 Lazarus полностью поддерживает юникод (Unicode) на всех платформах, исключая Gtk 1. На этой странице размещены инструкции для пользователей Lazarus, планы на будущее, описаны основные концепции и детали реализации.

Инструкции для пользователей

Хотя в Lazarus есть юникодные наборы виджетов, важно отметить, что не всё в юникоде. Кодировка строк и правильное преобразование между библиотеками, ожидающими разные кодировки, является ответственностью разработчика.

Обычно библиотекой используется единая кодировка строк (речь о динамической библиотеке DLL или пакете Lazarus). Каждая библиотека обычно ожидает одну кодировку, которой может быть юникод (UTF-8 для Lazarus) или ANSI (что означает системную кодировку, которая может быть UTF-8 или другой). RTL и FCL в FPC версий 2.4-2.6 ожидают строки ANSI.

Преобразование между юникодом и ANSI возможно при помощи функций **UTF8ToAnsi** и **AnsiToUTF8** модуля System или **UTF8ToSys** и **SysToUTF8** модуля FileUtil (Lazarus). Последние две мощнее (быстрее), но добавят к программе больше кода.

FPC не юникодный

Библиотеки Free Pascal времени исполнения (RTL) и свободных компонентов (FCL) в текущих версиях FPC (по 2.6.1) используют ANSI, поэтому требуется преобразование строк, получаемых из юникодных или переделываемых в юникодные библиотек (например, LCL).

Преобразование между ANSI и юникодом

Примеры:

Допустим, нужно передать в файловую функцию из RTL строку, полученную из TEdit:

```
var
  MyString: string; // в кодировке utf-8
begin
  MyString := MyTEdit.Text;
  SomeRTLRoutine(UTF8ToAnsi(MyString));
end;
```

И в обратную сторону:

```
var
  MyString: string; // в кодировке ANSI
begin
  MyString := SomeRTLRoutine;
  MyTEdit.Text := AnsiToUTF8(MyString);
end;
```

Важно: UTF8ToAnsi возвратит пустую строку, если строка в UTF-8 содержит неверные символы.

Важно: AnsiToUTF8 и UTF8ToAnsi требуют менеджера "широких" строк (widestring) в Linux, BSD и macOS. Можно воспользоваться функциями SysToUTF8 или UTF8ToSys (из модуля FileUtil) или добавить менеджер, включив cwstring одним из первых модулей в разделе uses программы.

Строки WideString и AnsiString

WideString -- это строковый тип, у которого базовый элемент данных имеет размер 2 байта. Строки этого типа почти всегда в кодировке UTF-16. См. Widestrings

Обратите внимание, что при доступе к WideString как к массиву, каждый его элемент представлен 2 байтами, а символ в UTF-16 может состоять из 1 или 2 элементов, занимающих соответственно 2 или 4 байта. Поэтому при доступе к WideString как к массиву **совершенно неправильно** ожидать соответствия "один элемент -- один символ", наличие в строке 4-х байтовых символов приведёт к ошибкам. Заметьте также, что UTF-16 как и UTF-8, допускает составные (decomposed) символы. Например, символ "Á" может быть представлен как одним элементом, так и парой: "A" + изменяющий акцент. Таким образом, включающий акцентированные буквы текст в юникоде часто может быть закодирован различными способами, Lazarus и FPC не обрабатывают такие случаи автоматически.

При передаче AnsiString в качестве WideString требуется преобразование кодировки.

```
var
  w: widestring;
begin
  w:='Über'; // неправильно, FPC преобразует системную кодовую страницу в
UTF16
  w:=UTF8ToUTF16('Über'); // правильно
  Button1.Caption:=UTF16ToUTF8(w);
end;
```

Работа со строками и символами UTF-8

В Lazarus до версии 0.9.30 включительно функции LCL для работы с UTF-8 находились в модуле LCLProc. В Lazarus 0.9.31 и последующих они в целях совместимости по-прежнему доступны при подключении LCLProc, но сам код работы с UTF-8 находится в модуле lazutf8 пакета lazutils.

Для выполнения операций над строками в UTF-8 используйте функции модуля lazutf8 вместо предоставляемых модулем SysUtils из FPC, поскольку SysUtils не готов для работы с юникодом, а

lazutf8 готов. Просто замените утилиты из SysUtils на эквивалентные из lazutf8, имеющие те же имена с префиксом UTF-8.

Учтите также, что перебор символов как элементов строки-массива не подходит для юникода. Эта особенность присуща не только UTF-8, юникод не предполагает фиксированный размер каждого символа. Возможны два основных способа перебора символов строки в кодировке UTF-8:

- побайтовый -- подходит для поиска подстроки или поиска ASCII символов в строке UTF-8, например, при разборе файлов XML.
- посимвольный -- подходит для экранных компонентов, таких как synedit, когда, например, нужно определить третий символ, отображаемый на экране.

Поиск подстроки

Благодаря особенностям UTF-8 для поиска подстроки подходят обычные функции поиска подстроки. Несмотря на мультибайтность, первый байт не может быть спутан со вторым. Поэтому поиск корректной подстроки в UTF-8 при помощи Pos всегда возвращает правильную позицию с точки зрения UTF-8:

```
uses lazutf8; // LCLProc для Lazarus 0.9.30 и ранее
...
procedure Where(SearchFor, aText: string);
var
  BytePos: LongInt;
  CharacterPos: LongInt;
begin
  BytePos:=Pos(SearchFor,aText);
  CharacterPos:=UTF8Length(PChar(aText),BytePos-1);
  writeln('Подстрока "',SearchFor,'" начинается в тексте "',aText,'"',
    ' с байта ',BytePos,', с символа ',CharacterPos);
end;
```

Из-за неоднозначности юникода возможно неожиданное поведение Pos() (как и любых сравнений), если одна из строк содержит составные (decomposed) символы, а другая -- их непосредственные коды. RTL не обрабатывает такие случаи автоматически.

Доступ к символам UTF-8

Юникодные символы могут различаться по длине, поэтому лучшее решение для посимвольного доступа -- последовательный перебор символов. Для перебора символов используйте следующий код:

```
uses lazutf8; // LCLProc для Lazarus 0.9.30 и ранее
...
procedure DoSomethingWithString(AnUTF8String: string);
var
  p: PChar;
```

```
  CharLen: integer;
  FirstByte, SecondByte, ThirdByte: Char;
begin
  p := PChar(AnUTF8String);
  repeat
    CharLen := UTF8CharacterLength(p);

    // У нас есть указатель на символ и его длина
    // Для побайтного доступа к UTF-8 символу:
    if CharLen >= 1 then FirstByte := P[0];
    if CharLen >= 2 then SecondByte := P[1];
    if CharLen >= 3 then ThirdByte := P[2];

    inc(p,CharLen);
  until (CharLen=0) or (p^ = #0);
end;
```

Доступ к n-му символу UTF-8

Иногда помимо последовательного требуется и произвольный доступ к символам UTF-8.

```
uses lazutf8; // LCLProc для Lazarus 0.9.30 и ранее
...
var
  AnUTF8String, NthChar: string;
begin
  NthChar := UTF8Copy(AnUTF8String, N, 1);
```

Получение кодов символов при помощи UTF8CharacterToUnicode

Пример демонстрирует получение 32-битного кода каждого символа строки в UTF-8:

```
uses lazutf8; // LCLProc для Lazarus 0.9.30 и ранее
...
procedure IterateUTF8Characters(const AnUTF8String: string);
var
  p: PChar;
  unicode: Cardinal;
  CharLen: integer;
begin
  p:=PChar(AnUTF8String);
  repeat
    unicode:=UTF8CharacterToUnicode(p,CharLen);
    writeln('Unicode=',unicode);
    inc(p,CharLen);
```

```
  until (CharLen=0) or (unicode=0);
end;
```

UTF-8 Copy, Length, LowerCase и т.п.

В модуле lazutf8 (LCLProc для Lazarus 0.9.30 и ранее) реализованы практически все операции над строками в UTF-8. Ниже приведён список утилит из lazutf8.pas:

```
function UTF8CharacterLength(p: PChar): integer;
function UTF8Length(const s: string): PtrInt;
function UTF8Length(p: PChar; ByteCount: PtrInt): PtrInt;
function UTF8CharacterToUnicode(p: PChar; out CharLen: integer): Cardinal;
function UnicodeToUTF8(u: cardinal; Buf: PChar): integer; inline;
function UnicodeToUTF8SkipErrors(u: cardinal; Buf: PChar): integer;
function UnicodeToUTF8(u: cardinal): shortstring; inline;
function UTF8ToDoubleByteString(const s: string): string;
function UTF8ToDoubleByte(UTF8Str: PChar; Len: PtrInt; DBStr: PByte): PtrInt;
function UTF8FindNearestCharStart(UTF8Str: PChar; Len: integer;
                                  BytePos: integer): integer;
// поиск n-го символа UTF-8, без учёта двунаправленности (BIDI)
function UTF8CharStart(UTF8Str: PChar; Len, CharIndex: PtrInt): PChar;
// поск байтового индекса n-го символа UTF-8, без учёта двунаправленности
(BIDI) (длина подстроки в байтах)
function UTF8CharToByteIndex(UTF8Str: PChar; Len, CharIndex: PtrInt): PtrInt;
procedure UTF8FixBroken(P: PChar);
function UTF8CharacterStrictLength(P: PChar): integer;
function UTF8CStringToUTF8String(SourceStart: PChar; SourceLen: PtrInt) :
string;
function UTF8Pos(const SearchForText, SearchInText: string): PtrInt;
function UTF8Copy(const s: string; StartCharIndex, CharCount: PtrInt):
string;
procedure UTF8Delete(var s: String; StartCharIndex, CharCount: PtrInt);
procedure UTF8Insert(const source: String; var s: string; StartCharIndex:
PtrInt);

function UTF8LowerCase(const AInStr: string; ALanguage: string=''): string;
function UTF8UpperCase(const AInStr: string; ALanguage: string=''): string;
function FindInvalidUTF8Character(p: PChar; Count: PtrInt;
                                  StopOnNonASCII: Boolean = false): PtrInt;
function ValidUTF8String(const s: String): String;

procedure AssignUTF8ListToAnsi(UTF8List, AnsiList: TStrings);

//функции сравнения
```

```
function UTF8CompareStr(const S1, S2: string): Integer;
function UTF8CompareText(const S1, S2: string): Integer;
```

Работа с именами файлов и каталогов

Элементы управления и функции Lazarus ожидают имена файлов и каталогов в кодировке UTF-8, но RTL использует для таких имён строки в ANSI.

Например, пусть кнопка назначает текущий каталог свойству Directory элемента TFileListBox. Функция RTL GetCurrentDir использует ANSI, а не юникод, поэтому требуется преобразование:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FileListBox1.Directory:=SysToUTF8(GetCurrentDir);
  // или с помощью функциий из модуля FileUtil
  FileListBox1.Directory:=GetCurrentDirUTF8;
end;
```

Модуль FileUtil содержит общие файловые функции в варианте UTF-8:

```
// основные функции как в RTL, но в кодировке UTF-8
// вместо системной

// AnsiToUTF8 и UTF8ToAnsi требуют менеджер широких строк в Linux, BSD,
macOS,
// но эти ОС обычно используют в качестве системной кодировки UTF-8,
// и тогда менеджер не нужен.
function NeedRTLAnsi: boolean;// истина если системная кодировка не UTF-8
procedure SetNeedRTLAnsi(NewValue: boolean);
function UTF8ToSys(const s: string): string;// как UTF8ToAnsi, но более
независима от менеджера широких строк
function SysToUTF8(const s: string): string;// как AnsiToUTF8, но более
независима от менеджера широких строк

// операции с файлами
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;
{$IFNDEF VER2_2_0}
function ExtractShortPathNameUTF8(Const FileName : String) : String;
{$ENDIF}
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;
```

Восточно-азиатские языки в Windows

Шрифт элементов управления пользовательского интерфейса по умолчанию (Tahoma) в Windows XP в состоянии правильно отображать несколько письменностей/алфавитов/языков, включая арабский, русский (кириллица) и западноевропейские языки (латинский/греческий алфавиты), но не восточно-азиатские как китайский, японский и корейский.

Для корректного отображения указанных языков стандартным шрифтом достаточно в Панели управления выбрать Региональные настройки, щёлкнуть вкладку Языки и установить поддержку восточно-азиатских языков. Естественно, что в версиях Windows XP на указанных языках этот пакет включён по умолчанию. Дополнительные инструкции здесь.

Более позднее версии Windows скорее всего не требуют установки дополнений для поддержки указанных языков.

Особенности Free Pascal

UTF-8 и исходные файлы -- отсутствие BOM

Когда в созданном в Lazarus файле исходного кода используются не-ASCII символы, он сохраняется в UTF-8. При этом **не** используется **BOM** (Byte Order Mark -- маркер порядка байтов). Кодировку можно изменить при помощи правого щелчка по редактору кода / File Settings / Encoding. Помимо того, что файлы в UTF-8 не обязаны иметь BOM, причина его отсутствия скорее в том, как FPC обрабатывает строки AnsiString. LCL использует AnsiString для совместимости и UTF-8 для переносимости.

Примечание. Некоторые текстовые редакторы в MS Windows могут полагать, что файлы используют системную кодовую страницу (кодировку OEM) и отображают в них неправильные символы. Не добавляйте BOM. Если добавить BOM, то придётся изменить все строковые присваивания.

Пример:

```
Button1.Caption := 'Über';
```

Когда отсутствует BOM (и кодировка не задана параметром) компилятор считает, что строка в системной кодировке и копирует каждый байт в строку без преобразований. LCL ожидает строки именно в таком виде.

```
// исходный файл сохранён в UTF без BOM
if FileExists('Über.txt') then ; // неправильно, FileExists ожидает системную кодировку
if FileExistsUTF8('Über.txt') then ; // правильно
```

Основы юникода

Стандарт Unicode сопоставляет целые от 0 до 10FFFF(h) символам. Каждое такое сопоставление именуется кодовой точкой (code point). Другими словами, юникодные символы определены для кодовых точек от U+000000 до U+10FFFF (от 0 до 1 114 111).

Для представления кодовых точек как уникальных байтовых последовательностей существуют три основных схемы. Эти схемы называют форматами юникодных преобразований: UTF-8, UTF-16 и UTF-32. Преобразование между всеми ними возможно. Их основные свойства:

	UTF-8	UTF-16	UTF-32
Наименьшая кодовая точка [16]	000000	000000	000000
Наибольшая кодовая точка [16]	10FFFF	10FFFF	10FFFF
Размер кодового элемента [бит]	8	16	32
Минимум байт на символ	1	2	4
Максимум байт на символ	4	4	4

UTF-8 обладает несколькими важными и полезными свойствами: Содержимое интерпретируется как последовательность байтов, поэтому понятие порядка байт "от младшего" или "от старшего" отсутствует. Юникодные символы от U+0000 до U+007F (ASCII) кодируются байтами от 00h до 7Fh (совместимо с ASCII). Это значит, что файлы и строки, содержащие только 7-битные символы ASCII, кодируются одинаково в ASCII и в UTF-8. Все символы больше U+007F кодируются как последовательность нескольких байтов, у каждого из которых старшие биты установлены в 1. Последовательность байт одного символа заведомо не содержится в более длинной последовательности байт другого символа. Это позволяет без затруднений искать подстроки. Первый байт мультибайтовой последовательности (представляющей не-ASCII символ) всегда принадлежит диапазону от C0h до FDh и показывает количество последующих байт для этого символа. Все последующие байты мультибайтной последовательности принадлежат диапазону от 80h до BFh. Это позволяет легко выполнять ресинхронизацию и способствует надежности.

UTF-16 обладает следующими важными свойствами: Используется одно 16-битное слово для кодирования символов от U+0000 до U+d7ff и два 16-битных слова для кодирования всех остальных символов.

В заключение, любой юникодный символ возможно представить как один 4-байтный/32-битный элемент в **UTF-32**.

См. также: Unicode FAQ - Basic questions, Unicode FAQ - UTF-8, UTF-16, UTF-32 & BOM, Wikipedia: UTF-8 [1]

Детали реализации

Lazarus/LCL обычно использует только UTF-8

Начиная с Lazarus 0.9.31 интерфейс GTK1 объявлен устаревшим, все интерфейсы LCL совместимы с юникодом, Lazarus и LCL используют строки только в кодировке UTF-8, если явно не указано, что функция/процедура принимает другие кодировки.

Перевод интерфейса Win32 на юникод

Обзор

Сперва, и это самое важное, во избежание поломки существующего интерфейса ANSI следует все юникодные интерфейсные модификации заключить в IFDEF WindowsUnicodeSupport. После стабилизации модификаций все IFDEF-ы будут убраны и останется только юникодная часть. К тому моменту все существующие программы, использующие символы ANSI, потребуют миграции на юникод.

Win9x не поддерживает юникод

Платформы <= Win9x основаны на стандартах кодовых страниц ISO и поддерживают юникод только частично. Платформы Windows, начиная с WinNT, и Windows CE юникод поддерживают полностью.

Win 9x и NT предлагают два параллельных набора функций API: *A -- старый под ANSI, *W -- новый юникодный. Функции *W принимают в качестве параметров широкие строки, то есть строки в кодировке UTF-16. В Windows 9x присутствуют все функции *W, но имеют пустые реализации. Поэтому они ничего не делают. Но некоторые из них реализованы полностью даже в 9x и перечислены ниже в разделе "Широкие функции, присутствующие в Windows 9x". Это свойство актуально, поскольку позволяет иметь одно приложение для Win9x и WinNT и во время его исполнения определять, какой набор API использовать.

Windows CE использует только широкие функции API.

Широкие функции, присутствующие в Windows 9x

Некоторые функции широкого API присутствуют в Windows 9x. Вот список таких функций: http://support.microsoft.com/kb/210341

Пример преобразования:

```
GetTextExtentPoint32(hdcNewBitmap, LPSTR(ButtonCaption),
Length(ButtonCaption), TextSize);
```

Станет:

```
{$ifdef WindowsUnicodeSupport}
  GetTextExtentPoint32W(hdcNewBitmap, PWideChar(Utf8Decode(ButtonCaption)),
Length(WideCaption), TextSize);
{$else}
  GetTextExtentPoint32(hdcNewBitmap, LPSTR(ButtonCaption),
Length(ButtonCaption), TextSize);
{$endif}
```

Функции, требующие версий Ansi и Wide

Пример преобразования:

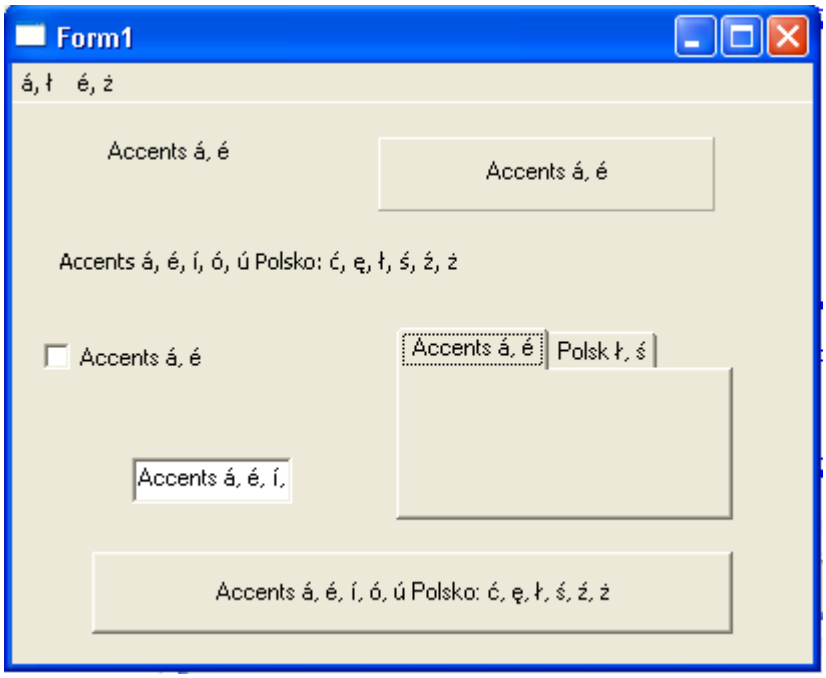
```
function TGDIWindow.GetTitle: String;
var
 l: Integer;
begin
   l := Windows.GetWindowTextLength(Handle);
   SetLength(Result, l);
   Windows.GetWindowText(Handle, @Result[1], l);
end;
```

Станет:

```
function TGDIWindow.GetTitle: String;
var
  l: Integer;
  AnsiBuffer: string;
  WideBuffer: WideString;
begin

{$ifdef WindowsUnicodeSupport}

if UnicodeEnabledOS then
begin
  l := Windows.GetWindowTextLengthW(Handle);
  SetLength(WideBuffer, l);
  l := Windows.GetWindowTextW(Handle, @WideBuffer[1], l);
```

```
  SetLength(WideBuffer, l);
  Result := Utf8Encode(WideBuffer);
end
else
begin
  l := Windows.GetWindowTextLength(Handle);
  SetLength(AnsiBuffer, l);
  l := Windows.GetWindowText(Handle, @AnsiBuffer[1], l);
  SetLength(AnsiBuffer, l);
  Result := AnsiToUtf8(AnsiBuffer);
end;

{$else}

   l := Windows.GetWindowTextLength(Handle);
   SetLength(Result, l);
   Windows.GetWindowText(Handle, @Result[1], l);

{$endif}

end;
```

Снимки экрана



Кодовые страницы FPC

Почему LCL не использует кодовые страницы для исходных текстов?

-Fcutf8 и {$codepage utf8} существуют не первый год.

Следующие программы требуют FPC 2.7.1. Описанные трудности существуют и на более старых компиляторах.

Есть в использовании -Fcutf8 и {$codepage utf8} несколько ловушек. Директивы работают, если DefaultSystemCodePage из RTL возвращает CP_UTF8. Иначе компилятор перекодирует строки. Например, в Linux умолчание для RTL -- CP_ACP, что по умолчанию ISO_8859-1. RTL самостоятельно **не** читает язык из переменных окружения. Поэтому умолчанием будет ISO_8859-1. Это означает, что строковые константы в UTF-8 будут преобразованы компилятором.

Откомпилируйте с -Fcutf8 и запустите в Linux с LANG в utf-8:

```
program project1;
{$mode objfpc}{$H+}
begin
  writeln(DefaultSystemCodePage,' ',CP_UTF8);
  writeln('ä');
end.
```

Результат:

```
0 65001
Ã¤
```

LCL использует widestringmanager -- менеджер широких строк (в настоящее время cwstring), который назначает DefaultSystemCodePage. Тоже самое можно сделать и в не-LCL программах:

```
program project1;
{$mode objfpc}{$H+}
uses cwstring;
begin
  writeln(DefaultSystemCodePage,' ',CP_UTF8);
  writeln('ä');
end.
```

Результат:

```
65001 65001
ä
```

Приведённый выше код прост и понятен, жаль только, что работает подобное лишь в простейших случаях. Посмотрим на программу ниже:

```
program project1;
{$mode objfpc}{$H+}
uses cwstring;
var
  a,b,c: string;
begin
```

```
  writeln(DefaultSystemCodePage,' ',CP_UTF8);
  a:='ä'; b:='='#$C3#$A4; // #$C3#$A4 это UTF-8 для ä
  c:=       'ä='#$C3#$A4;
  writeln(a,b); // writes ä=ä
  writeln(c);   // writes ä=Ã¤
end.
```

```
65001 65001
ä=ä
ä=Ã¤
```

Посмотрите, работают строковые константы в UTF-8, строковые константы с кодами UTF-8, но не их комбинация. Приведённый выше код скомпилирован с -Fcutf8 и использует cwstring для установки DefaultSystemCodePage в CP_UTF8.

Так что же пошло не так?

Компилятор обрабатывает строковые не-ASCII константы (здесь -- ä) как широкие строки (UCS2, не UTF-16).

Одурачить компилятор при помощи 'ä='+#$C3#$A4 не выйдет. Придётся определить две отдельные строковые константы.

Использование символов не из диапазона UCS-2 приведёт к

```
Fatal: illegal character "'🙃 ($F0)
```

Но можно указать их в кодах UTF-16: *#$D834#$DD1E*. Да, прочитали правильно. Указание кодовой страницы при помощи -Fcutf8 или {$codepage utf8} действительно определяет смесь UTF-8 и UTF-16.

Теперь откомпилируйте приведённый выше код без -Fcutf8:

```
65001 65001
ä=ä
ä=ä
```

Вот это да, всё как хотелось. Можно даже смешивать строковые константы в ASCII и не-ASCII. Без указания кодовой страницы компилятор просто сохраняет строковые константы как последовательности байтов. А это и есть UTF-8.

В этом одна из причин того, почему LCL приложения не используют флаги кодировок.

В msegui реализована экосистема широких строк, которая работает лучше, чем кодовые страницы.

RTL с кодовой страницей UTF-8 по умолчанию

Эксперимент -- требуется тестирование под Windows!

Обычно RTL использует для строк системную кодовую страницу (например, FileExists или TStringList.LoadFromFile). Под Windows RTL не использует юникод, вам недоступны символы вне вашей языковой группы. LCL работает с кодировкой UTF-8, поддерживающей все символы юникода. Под Linux и macOS системной кодовой страницей обычно является UTF-8, поэтому для них RTL по умолчанию использует CP_UTF8.

Начиная с FPC 2.7.1 кодовая страница, используемая RTL по умолчанию, может быть изменена на UTF-8 (CP_UTF8). Таким образом, пользователи Windows теперь смогут использовать в RTL строки UTF-8. Для тестирования добавьте -dEnableUTF8RTL в *Lazarus build options* (параметры сборки проекта) и пересоберите проект.

- Например, FileExists и aStringList.LoadFromFile(Filename) теперь полностью поддерживают юникод. Полный перечень функций, уже поддерживающих юникод полностью, смотрите по ссылке:

http://wiki.freepascal.org/FPC_Unicode_support#RTL_changes

- *AnsiToUTF8, UTF8ToAnsi, SysToUTF8, UTF8ToAnsi* не действуют. Они использовались преимущественно для вышеуказанных функций, которые больше не требуют преобразований аргументов и результата.
- Многочисленные явные вызовы *UTF8Encode* и *UTF8Decode* больше не нужны, поскольку при присваивании строки в *UnicodeString* строке String и наоборот компилятор сделает это автоматически.
- При обращении к WinAPI необходимо использовать функции с "W" либо пользоваться функциями **UTF8ToWinCP** и **WinCPToUTF8**.
- Включить новый режим можно перекомпилировав Lazarus с **-dEnableUTF8RTL**.
- Если используете строковые литералы с **WideString**, **UnicodeString** или **UTF8String**, то теперь исходные тексты **должны** быть в соответствующей кодировке. Например, можно использовать исходные тексты в UTF-8 (поведение Lazarus по умолчанию) и передавать компилятору параметр **-FcUTF8**.
- "String" и "UTF8String" -- разные типы. При присваивании *String* переменной в *UTF8String* компилятор добавляет код проверки соответствия кодировок. Это повлечёт увеличение времени исполнения и размера машинного кода.

Дополнительную информацию о новой поддержке юникода в FPC см. по ссылке:
http://wiki.freepascal.org/FPC_Unicode_support

См. также

- UTF-8 - Описание строк в UTF-8