Слинкин Д.А.

От Turbo Pascal к Free Pascal.
Часть 3.

Символьные и строковые типы данных. Поддержка стандарта Unicode.

В третьей части книги раскрываются изменения и нововведения Free Pascal в обработке символьных и строковых типов данных, механизмы поддержки стандарта Unicode.

Материал предназначен для учителей информатики, может быть полезен для преподавателей программирования, а также школьников и студентов младших курсов, обучающихся на IT-специальностях.

Оглавление

Введение

Перед Вами - третья часть учебно-методического пособия "От Turbo Pascal к Free Pascal", полностью посвященная особенностям работы с символьными и строковыми типами данных. Много внимания уделено стандарту Unicode, особенностях использования Unicode-кодировок с различными строковыми типами Free Pascal.

Изменения в строковых типах, обеспечившие прозрачную поддержку строк практически неограниченного размера, поставили Free Pascal по удобству обработки текста в один ряд с современными динамическими языками (Perl, Python, PHP, JavaScript и др.), не лишая при этом высокой скорости работы программы.

Поддержка стандарта Unicode и всех кодировок, предназначенных для представления символов данного стандарта, позволяют обеспечивать полную интернационализацию программ на Free Pascal, использовать Free Pascal для создания лингвистических программ, программ представления и обработки математических формул и т. д.

Как и в предыдущих частях пособия, излагаемый материал сопровождается большим количеством примеров. Версия используемого компилятора Free Pascal - 2.6.2, режим совместимости по умолчанию - **objfpc**. В качестве базовой операционной системы используется Linux, дополнительной - Windows (в ситуациях, когда программа функционирует по разному в различных ОС). Для визуализации текста программы применяется Geany. Результат запуска программы чаще всего демонстрируется в терминале, генерируемом Geany, иногда - в отдельных терминалах Windows и Linux.

Кодировки и стандарт Unicode.

В основе машинного хранения, обработки и визуализации текста лежат способы представления в памяти ЭВМ символов различных языков, математических операций, цифр, пиктограмм и т.п. Понятие "**кодировка**" заключает в себе описание указанных способов. Другими словами, каждая кодировка содержит список поддерживаемых символов, а также информацию о соответствии каждого символа набору машинных данных. Этот набор характеризуется **базовым объемом**, обычно - 8, 16, 32 бита, а также **шириной**, фиксированной или переменной. Например, 16-разрядная (двубайтовая) кодировка фиксированной ширины кодирует каждый символ 16 битами, а 8-разрядная (однобайтовая) кодировка переменной ширины кодирует каждый символ одним или несколькими подряд идущими 8-разрядными значениями. Среди устоявшихся терминов, описывающих особенности кодировок, следует выделить: **SBCS**[1] (Single Byte Character Set) - термин, определяющий 8-

1 http://en.wikipedia.org/wiki/SBCS

разрядные кодировки фиксированной ширины; **DBCS**[2] (Double Byte Character Set) - термин, определяющий 16-разрядные кодировки фиксированной или (в некоторых источниках) переменной ширины; **MBCS** [3] (Multi Byte Character Set) - термин, определяющий 8-разрядные или (в некоторых источниках) 16-разрядные кодировки переменной ширины. Некоторые производители программного обеспечения выделяют в отдельный класс кодировки **Unicode**. К сожалению, часто возникает ситуация, когда одну и ту-же кодировку в разных источниках описывают различными терминами. Чтобы избежать путаницы, в дальнейшем все рассматриваемые кодировки я буду характеризовать только **объемом** и **шириной**.

Для визуального представления символов кодировки компьютерная система должна иметь в своем составе один или несколько шрифтов. Современные системы обычно содержат несколько шрифтов разных типов, каждый из которых, среди прочих параметров, характеризуется и кодировкой.

Однобайтовые кодировки фиксированной ширины

Количество символов, поддерживаемых кодировкой, может быть различным. Одна из первых широко распространенных кодировок ASCII[4,5], разработанная Американским национальным институтом стандартов, сокращенно ANSI[6], поддерживает 128 символов, среди которых присутствует более трех десятков символов, предназначенных для управления устройствами ввода-вывода (табуляция, перевод строки, возврат каретки, звуковой сигнал и т. д.), символы латинского алфавита в нижнем и верхнем регистрах, цифры, знаки препинания и математических операций, скобки различных видов и т. д. Каждому символу кодировки ASCII сопоставлено числовое значение-код (например, управляющий символ звукового сигнала имеет код 7, латинская буква "A" - код 65, пробельный символ - код 32 и т. д.). Минимальный объем памяти, необходимый для хранения одного символа ASCII, составляет 7 битов, что позволяет большинству компьютерных систем отводить на каждый символ ASCII ровно 1 байт, заполняя восьмой бит нулем. В результате, ASCII стала прародительницей целого ряда 8-разрядных кодировок фиксированной ширины.

Свое дальнейшее развитие ASCII получила в национальных кодировках, некоторые из которых изменяли символы ASCII, но большинство - оставляли ASCII неизменной, задействуя восьмой бит для представления символов национального алфавита, псевдографики и других знаков. Таким образом обеспечивалась поддержка дополнительных 128 символов. Большой набор кодировок, основанный на ASCII, был принят Международной организацией по

2 http://en.wikipedia.org/wiki/DBCS
3 http://en.wikipedia.org/wiki/Multi-byte_character_set
4 http://ru.wikipedia.org/wiki/ASCII
5 http://www.unicode.org/Public/MAPPINGS/VENDORS/MISC/US-ASCII-QUOTES.TXT
6 http://www.ansi.org/

стандартизации (ISO)[7] и Международной электротехнической комиссией (IEC)[8] в качестве стандарта ISO/IEC-8859[9,10]. Кириллица описана в 5 части стандарта[11,12], однако в России не получила большого распространения. Набор кодировок ISO/IEC-8859 часто называют ANSI-кодировками (например - в ОС Windows), не смотря на то, что организация ANSI имеет к их разработке только косвенное отношение.

В компьютерах IBM PC с 80-х годов 20-го столетия использовалась основанная на ASCII кодировка CP437[13,14], шрифт для визуализации которой размещался обычно в ПЗУ видеоадаптера и тем самым был доступен на любом компьютере данной платформы вне зависимости от используемой операционной системы. Одной из важных характеристик CP437 было наличие символов псевдографики. Фактически, совместимость новых кодировок с псевдографическим набором символом CP437 на долгие годы стала фактором, определяющим востребованность и распространенность каждой из них. Причиной этого являлось большое количество электронной документации в кодировке CP437, в которой для построения рисунков, графиков и таблиц активно применялась псевдографика.

В Советском Союзе, а впоследствии в России на компьютерах IBM PC, в операционной системе DOS, активно использовалась кодировка CP866[15,16] (другое название - IBM866), которая характеризовалась совместимостью с CP437 в области символов псевдографики. Для обеспечения указанной совместимости разработчикам кодировки пришлось "разорвать" последовательность букв нижнего регистра русского алфавита на две части, первая из которых включает символы от "а" (код 160) до "п" (код 175), вторая - от "р" (код 224) до "я" (код 239). Символы "Ё" (код 240) и "ё" (код 241) и также были вынесены со своих естественных позиций. На сегодняшний день терминал ОС Windows по умолчанию поддерживает кодировку CP866, растровые шрифты терминала также предназначены для визуализации символов указанной кодировки.

Справа приведен пример, демонстрирующий вывод последовательности символов из диапазона кодов 160-239. Так как ОС Windows по умолчанию устанавливает для терминала кодировку CP866, то запуск программы в данной ОС покажет разрывность диапазона русских символов, при этом внутри разрыва будет представлен набор символов псевдографики, применяющихся для

866smallrus.pp

```
1  var c:char=#160;
2  begin
3   while c<>#240 do begin
4    write(c); inc(c);
5   end;
6   writeln;
7  end.
```

7 http://www.iso.org/
8 http://www.iec.org/
9 http://ru.wikipedia.org/wiki/ISO_8859
10 http://www.unicode.org/Public/MAPPINGS/ISO8859/
11 http://ru.wikipedia.org/wiki/ISO_8859-5
12 http://www.unicode.org/Public/MAPPINGS/ISO8859/8859-5.TXT
13 http://ru.wikipedia.org/wiki/CP437
14 http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/PC/CP437.TXT
15 http://ru.wikipedia.org/wiki/Cp866
16 http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/PC/CP866.TXT

построения таблиц в тексте:

```
Z:\cp866>866small rus.exe
абвгдежзийклмноп▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌рстуфхцчшщъыьэюя
```

Запуск программы в ОС Linux требует дополнительных манипуляций. В большинстве современных Linux-систем используется Unicode-кодировка UTF-8, не совместимая с CP866. Однако, с учетом того, что в стандарте Unicode представлены все известные на сегодняшний день символы, присутствующие в других кодировках, не составляет сложности организовать перевод символов CP866 в UTF-8. Для этого можно воспользоваться Linux-утилитой iconv[17,18]:

```
[slinkin@sdanout cp866]$ ./866smallrus | iconv -f cp866
абвгдежзийклмноп▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌рстуфхцчшщъыьэюя
```

При запуске перенаправляем вывод программы на ввод утилиты iconv, которой в свою очередь указываем, что направляемый ей текст - в кодировке CP866. Iconv переводит полученный текст в текущую кодировку терминала (UTF-8) и выводит на экран.

Среди кириллических однобайтовых кодировок фиксированной ширины, основанных ASCII, следует выделить еще две, получивших широкое распространение в РФ. Это кодировки KOI8-R[19,20,21] и CP1251[22,23]. Кодировка KOI8-R характеризуется достаточно экзотичным размещением кириллических символов, которые расположены таким образом, что при обнулении старшего бита у кода русского символа получается код визуально и фонетически идентичного или близкого к нему символа ASCII. Фактически, это давало возможность прочитать текст даже при утере по тем или иным причинам старшего бита у каждого байта текста (например - при передаче данных через некоторые виды сервисов в сети Интернет, либо на компьютерных системах, где поддерживалась только 7-разрядная кодировка ASCII). Как и CP866, KOI8-R совместима с CP437 в области символов псевдографики.

Справа приведен пример, который демонстрирует преобразование текста из кодировки KOI8-R в ASCII обнулением старшего бита. В строках 5-8 с использованием кодировки KOI8-R сформирована фраза "**КОИ8Р - сброс старшего бита**". В строках 9-10 обнуляется старший бит у каждого символа текста, результат выводится на терминал. Запуск программы наглядно показывает возможность чтения текста, полученного с помощью данного преобразования:

koi8toascii.pp

```
1  {$mode objfpc}
2   var s: string;
3       i: byte;
4  begin
5   s:=#235#239#233#56#242#32#45#32;
6   s+=#211#194#210#207#211#32;
7   s+=#211#212#193#210#219#197#199#207#32;
8   s+=#194#201#212#193;
9   for i:=1 to length(s) do
10   write(chr(ord(s[i]) and 127));
11  writeln;
12 end.
```

```
[slinkin@sdanout koi8r]$ ./koi8toascii
koi8r - SBROS STAR[EGO BITA
```

KOI8-R является одной из самых долгоживущих кириллических кодировок, она активно использовалась для русификации UNIX-систем, в том числе и в ОС

17 http://ru.wikipedia.org/wiki/Iconv
18 https://www.gnu.org/savannah-checkouts/gnu/libiconv/documentation/libiconv-1.13/iconv.1.html
19 http://ru.wikipedia.org/wiki/КОИ-8
20 http://tools.ietf.org/rfc/rfc1489.txt
21 http://www.unicode.org/Public/MAPPINGS/VENDORS/MISC/KOI8-R.TXT
22 http://ru.wikipedia.org/wiki/CP1251
23 http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP1251.TXT

Linux, однако в современных UNIX-системах постепенно вытесняется Unicode-кодировкой UTF-8.

Кодировка CP1251 в основном используется в русскоязычных версиях ОС Windows и по терминологии корпорации Microsoft является одной из ANSI-кодировок. В отличие от CP866 и KOI8-R, в ней отсутствуют символы псевдографики, русские символы расположены в алфавитном порядке в самом конце кодовой таблицы (коды от 192 до 255), за исключением символов Ё (код 168) и (код 184).

В Советском Союзе, а затем в России было принято несколько стандартов 8-разрядного кодирования символов. ГОСТ 19768-74 среди прочего описывал кодовую таблицу КОИ-8 (сокращение по первым буквам идентифицирующей фразы "Код Обмена Информацией" и количеству задействованных в кодировке битов), которая фактически соответствовала распространенной уже в то время кодировке KOI8-R. К сожалению, 19768-74 не действует в настоящее время, так как впоследствии был заменен стандартом ГОСТ 19768-93[24], в котором были приведены кодировки ДКОИ К1 и ДКОИ К2, несовместимые ни с одной из распространенных в настоящее время кодировок. Ситуацию улучшил ГОСТ Р 34.303-92[25], описывающий несколько кодовых таблиц, среди которых следует выделить: КОИ-8 В1, соответствующую ISO/IEC-8859-5 и носящую название "Основная кодировка ГОСТ", и КОИ-8 Н2, в целом соответствующую CP866, известную в настоящее время под именем "Альтернативная кодировка ГОСТ". Существует еще несколько стандартов ГОСТ, описывающих семи- и восьмибитные кодировки, наборы символов и правила их использования в машинной обработке, однако востребованность данных стандартов в практической разработке программ в настоящее время невелика.

Использование любой из вышеописанных кодировок характеризуется следующими особенностями:

1. Один символ кодировки хранится в одном байте оперативной памяти, поэтому получение доступа к символу по его позиции в тексте является элементарной задачей. Столь-же простыми являются задачи подсчета, копирования, удаления, перемещения символов, так как все они в конечном итоге означают манипуляции с эквивалентным набором байтов памяти. Фактически, правило **"один байт = один символ"** обеспечивает прямой доступ к любому символу вне зависимости от объема текста.

2. Поддержка нескольких кодировок налагает множество дополнительных требований на систему, основными из которых являются: а) наличие шрифтов для изображения символов в соответствующей кодировке; б) поддержка специфичных для каждой кодировки раскладок клавиатуры.

3. Преобразование текста из одной кодировки в другую не всегда возможно - во

24 http://standartgost.ru/ГОСТ 19768-93
25 http://standartgost.ru/ГОСТ Р 34.303-92

многие кодировках присутствуют уникальные символы, не встречающиеся в других кодировках (например, кодировки CP866 и CP1251 содержат символ "**№**", который отсутствует в кодировках ASCII, CP437 и KOI8-R).

Возможности обработки текста, которые предлагал в свое время Turbo Pascal, относились исключительно к 8-разрядным кодировкам фиксированной ширины, о которых шла речь в данном параграфе. Free Pascal дополнительно имеет встроенные и библиотечные средства для манипуляций с текстом в 8-разрядных кодировках переменной ширины (в основном - UTF-8), а также в 16-разрядных кодировках фиксированной и переменной ширины.

Кодировки Unicode

Однобайтовые кодировки фиксированной ширины не могут представлять более 256 символов. Это ограничение не позволяет описать полный набор символов многих языков, прежде всего - восточно-азиатской группы, где распространена иероглифическая письменность. Кодировки, решающие данную проблему (в большинстве своем однобайтовые и двубайтовые переменной ширины), известны под аббревиатурой **CJK-кодировки** (Chinese, Japanese, Korean) или **CJKV-кодировки** (CJK+ Vietnamese)[26]. Поддержка CJK-кодировок в значительной степени усложняет программное обеспечение обработки текста и ввода-вывода данных, не решая, впрочем, проблемы универсализации обработки текста.

Большое количество разнообразных стандартов кодировок, совместимых и несовместимых друг с другом, послужило толчком к разработке единого стандарта для всех возможных наборов символов. Данный стандарт носит название **Unicode**[27] и разрабатывается некоммерческой организацией "Консорциум Юникода"(Unicode Consortium)[28]. Еще один международный стандарт, предназначенный для поддержки всех известных символов, разрабатывается организациями ISO/IEC и носит название **ISO/IEC 10646**[29]. Последние версии Unicode и ISO/IEC 10646 практически полностью совместимы друг с другом, различаются только области применимости стандартов.

В настоящее время стандарт Unicode включает в себя **более 100 тысяч символов**, а потенциально - более одного миллиона, каждый из которых идентифицируется числовым значением - кодом символа, который, в свою очередь, описывается в одной из трех форм: «U+*xxxx*» (для кодов 0…FFFF), или «U+*xxxxx*» (для кодов 10000…FFFFF), или «U+*xxxxxx*» (для кодов 100000…10FFFF), где *xxx* — шестнадцатеричные цифры. Unicode вобрал в себя все известные на момент его разработки символы из различных кодировок.

26 http://en.wikipedia.org/wiki/CJK_characters
27 http://ru.wikipedia.org/wiki/Unicode
28 http://www.unicode.org/
29 http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

Фактически, описание набора символов любой кодировки подразумевает его сопоставление набору символов Unicode[30]. Кроме набора символов, Unicode определяет множество правил их обработки. Например, в стандарте Unicode существуют специальные символы - диакритические знаки[31], которые представляют собой надстрочные или подстрочные символы, изменяющие или уточняющие значение других символов. Правила их обработки гласят, что диакритический знак ставится в тексте после модифицируемого символа, а целевая система должна визуализировать оба символа как один.

Хранение символов Unicode в памяти ЭВМ обеспечиваются **кодировками Unicode**. Благодаря большому числу поддерживаемых символов, не существует однобайтовых кодировок Unicode фиксированной ширины, каждый символ, в зависимости от используемой кодировки, может описываться одним или несколькими подряд идущими 8-, 16- или 32-разрядными значениями. Все кодировки Unicode совместимы между собой, преобразование текста из одной в другую производится без потерь. Рассмотрим вкратце некоторые распространенные кодировки Unicode.

UTF-8[32,33]**(Unicode Transformation Format)** - пожалуй самая популярная и востребованная 8-разрядная кодировка переменной ширины. UTF-8 совместима с ASCII, в отличии от большинства других Unicode-кодировок, что позволяет использовать ее в обработке англоязычных текстов без каких-либо предварительных манипуляций. Все символы Unicode, эквивалентные ASCII, хранятся с использованием UTF-8 в одном байте, остальные символы Unicode могут занимать от 2 до 4 байтов. Технически возможно кодирование символов длиной в 5 и 6 байтов, однако стандарт Unicode в настоящее время это не предусматривает. Первые 128 символов стандарта Unicode соответствуют ASCII и UTF-8, кодируются 1 байтом, остальные символы Unicode кодируются 2, 3 или 4 байтами с помощью специализированного алгоритма преобразования.

UTF-16[34,35] - 16-разрядная кодировка переменной ширины. Каждый символ занимает 2 или 4 подряд идущих байта. UTF-16 **не совместима** с ASCII, текст в UTF-16 не может быть корректно обработан большинством старых программ, поддерживающих только ASCII или основанные на ASCII 8-разрядные кодировки фиксированной ширины. UTF-16 представлена двумя разновидностями: **UTF-16LE** и **UTF-16BE**, где окончания LE и BE являются сокращениями от аббревиатур little-endian и big-endian, которые в свою очередь определяют порядок расположения байт в оперативной памяти для многобайтовых значений. Первые 65536 (от 0 до $FFFF) символов стандарта Unicode кодируются в одном 16-разрядном символе UTF-16 без

30 http://www.unicode.org/Public/MAPPINGS/
31 http://ru.wikipedia.org/wiki/Диакритические_знаки
32 http://ru.wikipedia.org/wiki/UTF-8
33 https://tools.ietf.org/html/rfc3629
34 http://ru.wikipedia.org/wiki/UTF-16
35 http://www.ietf.org/rfc/rfc2781.txt

преобразований. Этот набор символов описывается терминами "**базовый план**", Basic Multilingual Plane, BMP[36]. Для хранения символов, код Unicode которых превышает $FFFF, используется специализированный алгоритм преобразования, который создает так называемую "**суррогатную пару**" UTF-16, представляющую собой два 16-разрядных символа UTF-16 из диапазона от $D800 до $DFFF.

UTF-32[37,38] - 32-разрядная кодировка фиксированной ширины. Это самая простая в реализации кодировка, каждый символ Unicode кодируется в ней без преобразования одним 32-битным значением. Совместимость с ASCII **отсутствует**. Как и UTF-16, представлена двумя разновидностями: **UTF-32LE** и **UTF-32BE**. При скурпулезном следовании стандарту Unicode, UTF-32 может рассматриваться как кодировка переменной ширины в случае, например, использования в тексте диакритических знаков или визуализации дробей одним символом с помощью нескольких символов Unicode (значение числителя, знак дроби (U+2044), значение знаменателя) [39].

Стандарт ISO/IEC 10646 предлагает к использованию несколько кодировок, полностью или частично совместимых с кодировками Unicode. Рассмотрим некоторые из них:

UCS-2 (Universal Coded Character Set) - 16-разрядная кодировка фиксированной ширины. Кодирует только подмножество символов базового плана Unicode. Часто определяется как "UTF-16 без суррогатных пар"[40]. Разновидности - **UCS-2LE** и **UCS-2BE**.

UCS-4 - кодировка, идентичная UTF-32. Разновидности - **UCS-4LE** и **UCS-4BE**.

Для идентификации кодировки текста разработчики Unicode предлагают использовать так называемый **BOM**-маркер[41] (Byte Order Mark). Это символ Unicode U+FEFF, который должен быть первым символом текста, что позволит определить кодировку благодаря различному представлению указанного символа в различных кодировках. Например, BOM в UTF-8 кодируется в трех байтах $EF, $BB, $BF; в UTF-16LE - в двух байтах $FF, $FE; в UTF-32LE - в четырех байтах $FF, $FE, $00, $00 и т. д. К сожалению, BOM не позволяет в общем случае отличить кодировки UTF-16LE и UTF-32LE, т. к. один символ BOM в UTF-32LE выглядит как два подряд идущих символа в UTF-16LE: BOM и U+0000. Поэтому разработчики соответствующего программного обеспечения не рекомендуют использовать U+0000 в начале текста, и однозначно обрабатывают последовательность $FF, $FE, $00, $00 как BOM в кодировке UTF-32LE.

36 http://en.wikipedia.org/wiki/Plane_(Unicode)
37 http://ru.wikipedia.org/wiki/UTF-32
38 http://www.unicode.org/reports/tr19/tr19-9.html
39 http://ru.wikipedia.org/wiki/Дроби_в_Юникоде
40 http://www.unicode.org/faq/basic_q.html#14
41 http://ru.wikipedia.org/wiki/Маркер_последовательности_байтов

Манипуляции с текстом в кодировках Unicode наталкиваются на множество серьезных проблем. Действительно, при использовании кодировок переменной ширины обработка такого текста может быть только последовательной, прямой доступ к символу становится невозможен. Также невозможно определение количества символов Unicode в тексте без его полной последовательной обработки, усложняются вырезка, копирование, удаление элементов текста. В то-же время упрощается поддержка Unicode-систем, где теперь не требуется наличия множество локалей и специфичных для каждой кодировки шрифтов. Системе достаточно поддерживать одну произвольную кодировку Unicode, получая тем самым доступ ко всем возможностям данного стандарта.

Free Pascal позволяет обрабатывать текст в нескольких Unicode-кодировках. 8-разрядная кодировка UTF-8 обслуживаться типами Char и String, а также некоторыми новыми типами, о которых пойдет речь далее. Появились также специализированные типы данных для обработки текста в кодировке UTF-16, средства преобразования из UTF-8, а также из некоторых 8-разрядных кодировок фиксированной ширины в UTF-16 и обратно. Поддержка UTF-32 реализована только на самом начальном уровне.

Символьные типы

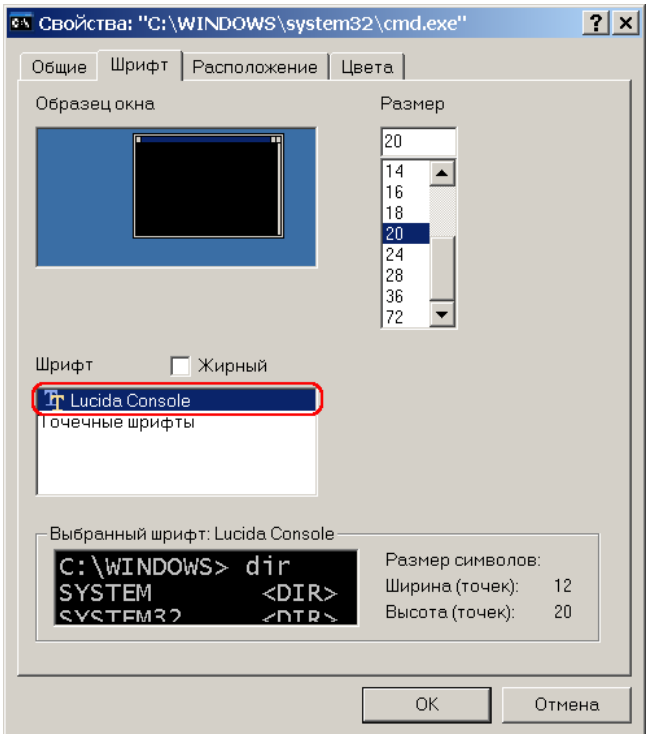
Free Pascal вводит несколько псевдонимов для символьного типа Char, определяя их в модуле System: **AnsiChar**, **TAnsiChar**. Это однобайтовые типы данных, применяемые для хранения символов 8-битных кодировок фиксированной ширины (CP866, CP1251, KOI8-R и т.д.), изменение размера и способов обработки которых в последующих версиях языка не предусматривается. Для типа Char в дальнейшем предполагается ввести изменение размера в зависимости от платформы таким-же или схожим образом, как это реализовано для типа Integer[42,43]. В результате, тип Char будет соответствовать либо типу AnsiChar, либо типу WideChar, о котором пойдет речь ниже. В текущей версии языка (2.6.2) Char соответствует типу AnsiChar.

Free Pascal также вводит новый символьный тип WideChar, так называемый **широкий символьный тип**. Это **двубайтовый** тип, позволяющий хранить один из символов базового плана стандарта Unicode в кодировке UCS2. Строки на основе WideChar позволяют хранить уже полный набор символов Unicode в кодировке UTF-16.

WideChar и основанные на нем строки чаще всего используются в операционной системе Windows, многие API-функции которой требуют строковых параметров в кодировке UTF-16LE.

42 http://freepascal.org/docs-html/ref/refsu7.html
43 http://freepascal.org/docs-html/ref/refsu5.html#x27-27003r2

Операционные системы Windows и Linux на уровне библиотечных функций и в графическом интерфейсе поддерживают достаточно большое количество кодировок, в том числе и некоторые кодировки Unicode. Однако **в терминалах** обеих операционных систем из Unicode-кодировок без особых проблем поддерживается только UTF-8. Поэтому в программах, использующих символы и строки Unicode, для организации вывода на экран следует перекодировать обрабатываемый текст в UTF-8 или готовить его непосредственно в редакторе, поддерживающем UTF-8. В ОС Linux (с учетом штатной поддержки UTF-8 в терминале) этого обычно достаточно. В ОС Windows следует проделать еще два дополнительных действия:



1) выбрать для отображения символов терминала векторный шрифт Lucida Console, как показано на примере справа;

2) выполнить в терминале команду смены кодировки на UTF-8: **chcp 65001**.

Рассмотрим пример визуализации некоторых символов Unicode в операционных системах Windows и Linux. В стандарте Unicode определено, среди всего прочего, 256 математических символов с кодами от U+2200 до U+22FF. Пример справа выводит этот набор на экран. В строках 1-3 производится подключение менеджера широких строк, если целевой ОС является UNIX-подобная операционная система, в нашем случае ОС Linux (в ОС Windows существует свой собственный автоматически подключаемый менеджер). Цикл в строках 8-12 перебирает все Unicode-символы из диапазона от U+2200 до U+22FF, формирует из каждого символа строку в кодировке UTF-8 и выводит ее на экран, отделяя символы друг от друга пробелом (строки 9-10). Обратите внимание на преобразование типов от WideChar к UnicodeString в строке 9. UnicodeString - это один из широких строковых типов данных, поддерживаемых Free Pascal, содержащий набор символов типа WideChar.

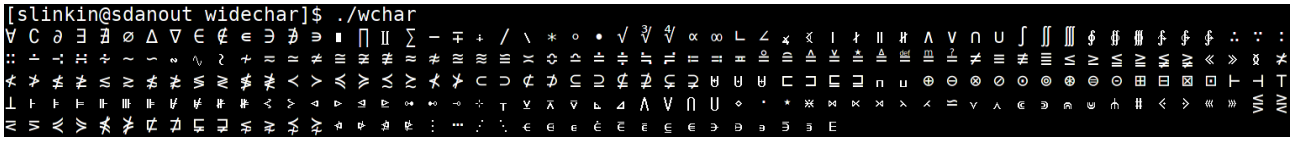
wchar.pp

```
1  {$IFNDEF WINDOWS}
2   uses cwstring;
3  {$ENDIF}
4  Var
5    WCH : widechar = #$2200;
6      s : string;
7  begin
8   while WCH<#$2300 do begin
9    s:=UTF8Encode(UnicodeString(WCH));
10   write(s,' ');
11   inc(WCH);
12  end;
13  writeln;
14 end.
```

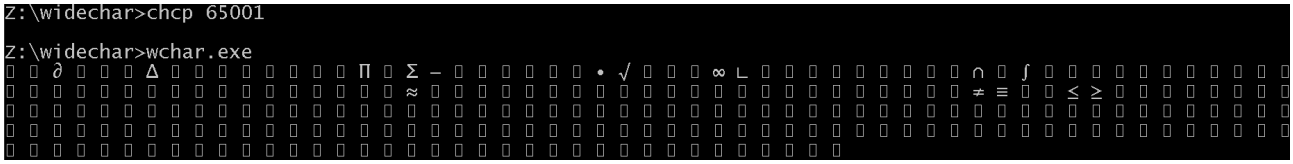
Для корректной визуализации символов требуется, чтобы шрифты и кодовая страница терминала операционной системы поддерживали Unicode. Терминалы операционных систем Windows и Linux поддерживают напрямую или путем простых предварительных действий Unicode-кодировку UTF-8, однако не поддерживают без достаточно сложных манипуляций кодировку Unicode-кодировку UTF-16, совместно с которой используется тип WideChar. Поэтому для вывода на экран символа данного типа последовательно выполняется его

перевод в строку типа UnicodeString с последующим преобразованием полученной строки из UTF-16 в UTF-8 с помощью функции UTF8Encode (строка 5).

Результат работы программы в ОС Linux:



К сожалению, поддержка Unicode консольными шрифтами в ОС Windows обеспечивается на самом минимальном уровне. Для проверки данного факта сначала переключаем кодировку терминала на UTF-8 с помощью команды chcp 65001, а затем запускаем программу:



В модуле System для удобства программиста определено несколько типов-псевдонимов для WideChar: **WChar**, **UCS2Char**, **UnicodeChar**. Их использование ничем не отличается от манипуляций с WideChar.

Для возможности работы с Unicode-кодировкой UTF-32 и кодировкой UCS-4 стандарта вводится тип данных UCS4Char, который фактически является целочисленным типом поддиапазона из промежутка от 0 до $10FFFF. Не существует встроенных в язык средств обработки как UCS4Char, так и типов на его основе, вместо этого существует несколько функций для преобразования текста из UTF-32 в UTF-16 и обратно.

Строковые типы данных

В Turbo Pascal было известно два типа строк - String и PChar. Основным недостатком типа String был фиксированный максимальный размер, не превышающий 255 символов, что ограничивало применение данного типа в обработке длинных строк. Тип PChar представляет собой указатель на набор символов, завершающийся нулем. Это стандартный для многих языков программирования строковый тип, однако необходимость строго контролировать распределение памяти для строк, невозможность быстро определить длину строки, отсутствие простых механизмов объединения строк - все это усложняло и усложняет работу с данным типом.

Free Pascal вводит четыре новых строковых типа: **ShortString**, **AnsiString**, **UnicodeString** и **WideString**, а также модифицирует тип String. Часто тип ShortString называют "короткой строкой", а типы AnsiString, UnicodeString и WideString - "длинными строками".

Тип ShortString полностью эквивалентен типу String в Turbo Pascal. Тип String во Free Pascal соответствует либо типу ShortString, либо AnsiString.

Если в программе имеется прямое указание длины строки при определении типа или переменной (например: var x:string[20];), тип String соответствует

ShortString. В противном случае задействуется директива компилятора {$H}. При установке директивы компилятора {$H-} тип String соответствует ShortString. При установке директивы компилятора {$H+} тип String соответствует AnsiString, а в будущих версиях FreePascal может соответствовать WideString для некоторых платформ.

В дальнейшем, для описания общих характеристик AnsiString и ShortString будем говорить "**тип String**".

Символы в переменных типа AnsiString, ShortString и PChar имеют тип Char (AnsiChar в текущей версии Free Pascal). Таким образом, в строках указанных типов можно хранить текст в любых 8-разрядных кодировках. Из кодировок фиксированной ширины в России на сегодняшний день чаще всего применяются ASCII, CP1251, CP866, ранее - KOI8-R; из кодировок переменной ширины - UTF-8.

Строки AnsiString

Для упрощения обработки строк произвольной длины во Free Pascal определяется тип AnsiString. Все функции, операторы и операции обработки строк, действительные для ShortString, действительны и для AnsiString. В то-же время размер такой строки ограничен только положительным диапазоном целочисленного типа, соответствующего разрядности целевой операционной системы. Например, для 32-разрядных систем максимальная длина строки равна high(longint), а для 64-разрядных - high(int64). Конечно, задачи с применением гигабайтных и тем более экзабайтных строк встречаются крайне редко, гораздо чаще приходится манипулировать строками в несколько килобайт или мегабайт.

В примере справа демонстрируется использование AnsiString для объединения всех строк файла в одну строку. Имя исходного файла должно передаваться параметром командной строки. Имя результирующего файла формируется из исходного с добавлением расширения ".out". На своем компьютере в ОС Linux запуск данной программы я произвел следующим образом:

```
[slinkin@sdanout strings]$ ./ansiFRW ansiFRW.pp
```

Запуск программы из командной строки ОС Windows в моей системе выглядит несколько по другому:

```
Z:\strings>ansiFRW.exe ansiFRW.pp
```

ansiFRW.pp

```
1  {$mode objfpc}
2  var s,res:ansistring;
3      f:text;
4  begin
5   if paramcount<1 then exit;
6   assign(f,paramstr(1));
7   reset(f);
8   res:='';
9   while not eof(f) do begin
10   readln(f,s);
11   res:=res+' '+s;
12  end;
13  close(f);
14  assign(f,paramstr(1)+'.out');
15  rewrite(f);
16  writeln(f,res);
17  close(f);
18 end.
```

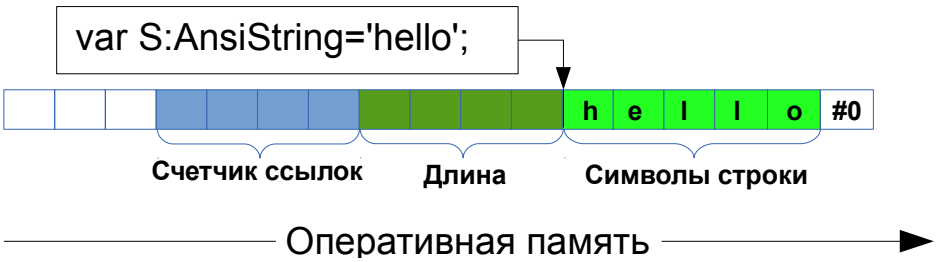
И в том и в другом случае в качестве исходного файла был использован текст данной программы (его размер превышает 255 байт). В результате был получен файл с именем ansiFRW.pp.out, объединяющий все строки программы в одну строку. Если заменить тип AnsiString (строка 2)

на ShortString, то при обработке даже небольших файлов возникнет ошибка: результирующая строка будет урезана до 255 байт. При использовании AnsiString размеры исходного файла ограничены только объемом доступной приложению оперативной памяти, так как ограничения на максимальную длину AnsiString обычно значительно превышают данный объем.

На прикладном уровне ShortString и AnsiString различаются в манипуляциях длиной строки. Актуальная длина строки типа ShortString может быть получена и изменена как с помощью нулевого символа строки, так и с помощью функции Length и процедуры SetLength. Для AnsiString - **только** с помощью Length и SetLength, доступ к нулевому символу строки заблокирован. Кроме этого, перед доступом к символам AnsiString, строке должна быть установлена актуальная длина тем или иным способом (например - вызовом SetLength или присвоением одной строки другой).

На системном уровне ShortString и AnsiString отличаются кардинально. Если ShortString - это просто последовательность символов, фактически не отличающаяся от небольшого массива, то AnsiString является указателем на последовательность символов, которая завершается нулевым значением, адресуя также с отрицательным смещением два целочисленных значения - длину строки без завершающего нуля и так называемый счетчик ссылок, который определяет количество переменных AnsiString, ссылающихся на ту-же самую область памяти. В будущих версиях Free Pascal структура AnsiString может быть изменена, с сохранением обратной совместимости. Например, в разрабатываемом на начало 2014 года Free Pascal 2.7.1 в структуру строки добавляется ее кодировка и количество байт на символ.

Следующий пример демонстрирует расположение пятисимвольной строки S в оперативной памяти 32-разрядной системы.



Таким образом, строка S будет занимать 4(счетчик ссылок)+4(длина)+5(символы) +1(завершающий нуль)=14 байт оперативной памяти. Для 64-разрядных систем строка S будет занимать 8(счетчик ссылок)+8(длина)+5(символы)+1(завершающий нуль)=22 байта оперативной памяти.

Управление как счетчиком ссылок, так и длиной строки производится автоматически, без вмешательства программиста, что иногда может приводить к замедлению работы программы в самых неожиданных местах. Дело в том, что даже простейшие операции над строками могут порождать множество системных манипуляций, связанных с выделением и освобождением оперативной памяти, переносом данных из одних областей памяти в другие и т. п.

В примере справа генерируется переменная AnsiString размером в 1 гигабайт, замеряется время генерации (строки 6-10). Затем, производится копирование строки, замеряется время на эту операцию (строки 11-14). Следующим действием изменяется первый символ полученной копии, также с замером затраченного времени (строки 15-18). Последним действием производится измерение скорости повторного изменения первого символа строки-копии (строки 19-22).

ansisStrangeSpeed.pp

```
1  {$mode objfpc}
2  uses sysutils,dateutils;
3  var s1,s2:ansistring; i:dword;
4      dt:tdatetime;
5  begin
6   dt:=now(); s1:='test';
7   for i:=1 to 28 do s1:=s1+s1;
8   writeln('Длина строки: ',length(s1));
9   write('Время на генерацию строки: ');
10  writeln(millisecondsbetween(now(),dt));
11  dt:=now();
12  s2:=s1;
13  write('Время на копирование строки: ');
14  writeln(millisecondsbetween(now(),dt));
15  dt:=now();
16  s2[1]:='1';
17  write('Время на первое изменение символа строки: ');
18  writeln(millisecondsbetween(now(),dt));
19  dt:=now();
20  s2[1]:='2';
21  write('Время на второе изменение символа строки: ');
22  writeln(millisecondsbetween(now(),dt));
23 end.
```

Один из полученных результатов продемонстрирован ниже:

```
Длина строки: 1073741824
Время на генерацию строки: 1129
Время на копирование строки: 0
Время на первое изменение символа строки: 546
Время на второе изменение символа строки: 0
```

В данном случае наблюдаем два интересных факта. Первый указывает на крайне малое время, затраченное программой на копирование гигабайтной строки. Второй - демонстрирует резкий провал в скорости работы программы при первом изменении строки-копии.

Причины такого странного поведения программы заключаются в особенностях обработки длинных строк. Присвоение одной строки другой не приводит к копированию содержимого строки. Вместо этого копируется только адрес строки и увеличивается ее счетчик ссылок, после чего исходная строка и строка-копия указывают на одну и ту-же область оперативной памяти. Если дальнейшие манипуляции предусматривают доступ к полученным строкам только для чтения, в оперативной памяти так и останется только один набор символов. Однако при первом-же изменении любой из строк будет произведено их физическое разделение в оперативной памяти, что и продемонстрировано в строках 15-18 программы. Данный механизм описывается термином "**copy-on-write**".

Работа данной программы серьезно зависит от объема свободной физической оперативной памяти системы. На моих системах время на генерацию строки колебалось от 0.7 до 5 секунд, а время на первую модификацию строки - от 0.3 до 1.5 секунд. Дело в том, что манипуляции с такими объемами оперативной памяти, в случае нехватки физической - задействуют виртуальную память системы, которая расположена на жестком диске, что и служит причиной резкого снижения скорости работы программы.

В целом, скорость манипуляций с AnsiString-переменными в несколько раз ниже, чем с переменными типа ShortString, даже если размер строки не превышает 255 символов.

В примере справа демонстрируется многократное присвоение и объединение строк отдельно для типов AnsiString (строки 8-11) и ShortString (строки 14-17), при этом размер результирующей строки и обоих случаях не выходит за пределы типа ShortString. Программа замеряет время работы обеих участков кода и выводит его на экран. Результат одного из замеров приведен ниже:

```
6133
2307
```

shortVSansi.pp

```
1  {$mode objfpc}
2  uses sysutils,dateutils;
3  var sA:ansistring;
4      sS:shortstring;
5      i:dword; dt:tdatetime;
6  begin
7   dt:=now();
8   for i:=1 to 100000000 do begin
9    sA:='To be, or not to be:';
10   sA:=sA+' that is the question:';
11  end;
12  writeln(millisecondsbetween(now(),dt));
13  dt:=now();
14  for i:=1 to 100000000 do begin
15   sS:='To be, or not to be:';
16   sS:=ss+' that is the question:';
17  end;
18  writeln(millisecondsbetween(now(),dt));
19  end.
```

Видим, что манипуляции с типом ShortString выполняются в 2.5 раза быстрее. Тестирование на различных аппаратно-программных системах показало отличия в скорости обработки ShortString и AnsiString от 2 до 3 раз. Таким образом нецелесообразно использовать тип AnsiString, если точно известно, что длина строк при обработке не превысит 255 символов.

Переменные AnsiString и PChar прозрачно преобразуются друг в друга прямым приведением типов, хотя с системной точки зрения это совершенно разные операции. Приведение к типу PChar просто копирует в переменную PChar указатель на область памяти, где содержатся символы строки AnsiString. Это становится возможным благодаря тому, что фактически AnsiString только расширяет представление PChar в оперативной памяти, добавляя два поля (длину и счетчик ссылок) слева от набора символов.

Нижеприведенный пример демонстрирует неожиданный побочный эффект от модификации строки PChar, полученной приведением типов из AnsiString.

ansisPChar.pp

```
1  {$mode objfpc}
2  var s1,s2:ansistring;
3      ps:pchar;
4  begin
5   s1:='test'; s2:=s1; s2[1]:='w'; ps:=pchar(s1); ps[0]:='r';
6   writeln('s1= ',s1,'; s2= ',s2,'; ps= ',ps);
7   s1:='test'; s2:=s1; ps:=pchar(s1); ps[0]:='r';
8   writeln('s1= ',s1,'; s2= ',s2,'; ps= ',ps);
9  end.
```

В программе заданы переменные s1 и s2 типа AnsiString и переменная ps типа PChar (строки 2-3). В строке 5 программы создается s1, ее содержимое копируется в s2 с модификацией первого символа с целью продемонстрировать различие переменных s1 и s2. Затем с помощью приведения типа переменной ps присваивается s1 и модифицируется начальный символ строки ps с целью продемонстрировать факт хранения символов ps и s1 в одной области памяти. В строке 6 содержимое s1, s2 и ps выводится на экран.

Действия в строке 7 программы отличаются только отсутствием манипуляций над первым символом переменной s2. Следовательно ожидается, что ее содержимое будет равно **'test'**. В строке 8 содержимое всех переменных снова выводится на экран.

Результат работы программы:

```
s1= rest; s2= west; ps= rest
s1= rest; s2= rest; ps= rest
```

Обращает на себя внимание второе значение во второй строке: модификация переменной ps привела к изменению не только переменной s1, **но и s2**, вместо ожидаемого **s2= test** было получено **s2= rest**. Причиной некорректного с прикладной точки зрения поведения программы является применение механизма "copy-on-write" для длинных строк.

Таким образом, если модифицируется строка PChar, созданная из AnsiString приведением типов, то это может привести к модификации других строк, созданных ранее присвоением из исходной строки AnsiString.

В текущей версии FreePascal отсутствуют встроенные механизмы для решения данной проблемы. В качестве временного решения рекомендую принудительно разделять строки AnsiString сразу после их присвоения. Для этого достаточно присвоить значение любому символу результирующей строки так, как это сделано в строке 5 вышеприведенного примера, либо еще проще: **s2[1]:=s2[1]** (для сохранения неизменным содержимого результирующей строки). Кроме этого, существует функция UniqueString, также решающая данную проблему.

Приведение типа от PChar к AnsiString требует больше манипуляций, чем в противоположном случае. AnsiString обладает более сложной структурой, чем PChar, поэтому при таком присвоении Free Pascal выделяет память для новой строки и копирует туда символы исходной. Таким образом, если приведение типа от AnsiString к PChar - процесс практически мгновенный (копируется только указатель на строку), то приведение типа от PChar к AnsiString занимает достаточно много времени (выделяется оперативная память и копируется все содержимое строки).

Строки UnicodeString и WideString.

В текущей версии Free Pascal тип String предназначен для хранения однобайтовых значений типа Char или AnsiChar. Это означает, что среди всех кодировок, предназначенных для работы с символами Unicode, для типа String можно использовать только кодировку UTF-8, в которой для кодирования символа используется от 1 до 4 байтов. Для поддержки всех разновидностей кодировок UTF-16 и UCS2 Free Pascal предлагает строковые типы **UnicodeString** и **WideString**.

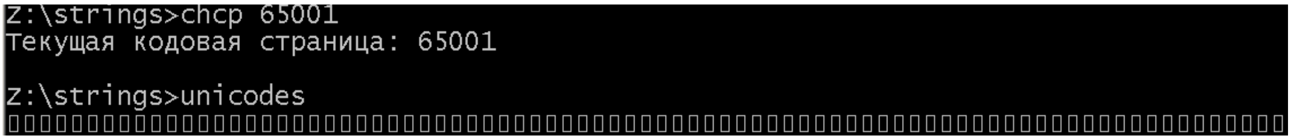
Структура UnicodeString практически полностью эквивалентна AnsiString. Разница заключается в объеме памяти на один символ - в UnicodeString хранятся символы типа WideChar.

Пример справа демонстрирует формирование Uncode-переменной, состоящей из 80 рунических символов. В строках 6-9 к unicode-переменной **ws** добавляется очередной рунический символ **wch**, код которого на каждой итерации цикла увеличивается на единицу. В строке 10 содержимое переменной ws преобразуется из utf-16 в utf-8 с помощью функции

unicodes.pp

```
1  {$mode objfpc}
2  Var
3    WS: unicodestring='';
4    wch: widechar=#$16A0;
5  begin
6   while wch<=#$16F0 do begin
7    ws:=ws+wch;
8    inc(wch);
9   end;
10  writeln(UTF8Encode(ws));
11 end.
```

UTF8Encode и выводится на экран. Программа предназначена для запуска в ОС Linux, где обычно UTF-8 является штатной кодировкой терминала. В русскоязычной ОС Windows кодировкой терминала является обычно кодировка CP-866 и, не смотря на то, что встроенная команда chcp позволяет сменить кодировку терминала Windows на UTF-8 (chcp 65001), ее поддержка на уровне шрифтов крайне ограничена. Например, запуск данной программы под Windows XP, с предварительной сменой кодировки на UTF-8, приведет к выводу набора прямоугольников:



Аналогичный запуск под Linux дает корректный результат:

Free Pascal поддерживает корректный ввод-вывод содержимого переменных UnicodeString с помощью процедур read, readln, write и writeln, а также преобразование между String и UnicodeString при объединении, присвоении, сравнении строк разных типов друг с другом. При этом автоматически производится преобразование строк из основной кодировки или в основную кодировку операционной системы с помощью специализированного менеджера строк Unicode, индивидуально реализованного для различных операционных систем. В ОС Windows для этой цели используется соответствующее API, не требующего от программиста дополнительных манипуляций, в ОС Linux тот-же результат достигается подключением модуля **cwstring**, который должен быть первым среди всех подключаемых модулей.

Информацию об основной кодировке менеджер Unicode получает от операционной системы, вызывая соответствующие системные функции. Для русскоязычной Windows это обычно кодировка CP-1251, для Linux - UTF-8. Не рекомендуется смешивать UnicodeString и String при манипуляциях в ОС Windows, так как это может привести к утере некоторых символов во время преобразования между UTF-16 и CP-1251. В Linux, с учетом использования UTF-8, такие преобразования всегда корректны, так как UTF-8 и UTF-16 являются кодировками Unicode.

Пример справа демонстрирует опасность переноса текста между переменными AnsiString и UnicodeString. Программа предназначена для запуска как в ОС Windows, так и в ОС Linux, поэтому строках 1-3 производится подключение модуля cwstring в случае, если Windows не является целевой платформой. Строка 7 содержит константу с символами кириллического и греческого алфавитов в кодировке UTF-8. С помощью функции utf8Decode константа переводится в UTF-16 и присваивается UnicodeString-переменной **sw**. Вывод на консоль данной переменной и шестнадцатеричных кодов ее символов производится в строках 8-10.

mixAsUs.pp

```
1  uses {$IFNDEF WINDOWS}
2         cwstring,
3        {$ENDIF} sysutils;
4  var sw:unicodestring; s:ansistring;
5      i:integer;
6  begin
7   sw:=utf8Decode('Альфа(α) и Омега(ω)');
8   writeln(sw);
9   for i:=1 to length(sw) do
10   write(' ',intToHex(ord(sw[i]),0));
11  writeln;
12  s:=sw; sw:=s;
13  writeln(sw);
14  for i:=1 to length(sw) do
15   write(' ',intToHex(ord(sw[i]),0));
16  writeln;
17 end.
```

Строка 12 копирует **sw** в AnsiString-переменную **s** и обратно для демонстрации возможных искажений исходного текста в процессе изменения его кодировки.

Строки 13-15 снова выводят на консоль переменную **sw** и набор кодов ее символов.

Результаты запуска программы в операционных системах Linux и Windows отличаются достаточно серьезно.

Запуск в ОС Windows, во первых - требует изменения базовой кодировки терминала (CP-866) на системную (CP-1251), что реализуется предварительным выполнением команды **chcp 1251**, а во вторых - приводит к утере греческих символов, так как их поддержка в CP-1251 отсутствует:

```
Z:\strings>chcp 1251
Текущая кодовая страница: 1251

Z:\strings>mixAsUs.exe
Альфа(?) и Омега(?)
 410 43B 44C 444 430 28 3B1 29 20 438 20 41E 43C 435 433 430 28 3C9 29
Альфа(?) и Омега(?)
 410 43B 44C 444 430 28 3F 29 20 438 20 41E 43C 435 433 430 28 3F 29
```

Замена греческих символов (коды $3B1 и $3C9) на вопросительные знаки заметна уже в первой строке вывода, но данная замена никак не сказывается на содержимом переменной **sw**, что доказывается второй строкой вывода. Фактически, преобразование текста при выводе на консоль было произведено во время передачи переменной **sw** в процедуру writeln (строка 8). Дальнейший перенос текста из UnicodeString в AnsiString и обратно (строка 12) вызвал автоматическое перекодирование вида UTF-16 -> CP-1251 -> UTF-16, в результате чего содержимое переменной **sw** изменилось - греческие символы были утеряны.

Запуск в ОС Linux показывает устойчивость текста к преобразованиям вида UTF-16 -> UTF-8 -> UTF-16:

```
[slinkin@sdanout strings]$ ./mixAsUs
Альфа(α) и Омега(ω)
 410 43B 44C 444 430 28 3B1 29 20 438 20 41E 43C 435 433 430 28 3C9 29
Альфа(α) и Омега(ω)
 410 43B 44C 444 430 28 3B1 29 20 438 20 41E 43C 435 433 430 28 3C9 29
```

По умолчанию FreePascal не производит никаких манипуляций со строковыми константами при компиляции программы, оставляет их неизменными. Однако существует возможность потребовать от компилятора автоматически перекодировать содержимое строковых констант в кодировку UCS2 (UTF-16 без суррогатных пар). Для этого следует указать кодировку исходного кода программы с помощью директивы компилятора **codepage**. Директива codepage поддерживает поддерживает параметры UTF8, CP1251, CP866. Например, использование в начале программы директивы **{$codepage UTF8}** потребует от компилятора преобразовать все используемые в программе строковые константы из кодировки UTF-8 в UCS2.

Пример справа демонстрирует использование директивы codepage. Текст программы набран в кодировке UTF-8 с применением кириллических и греческих букв в используемой строковой константе (строка 7). Директива {$codepage UTF8} (строка 1) обеспечила преобразование указанной константы из UTF-8 в UCS2. Запуск программы в ОС Windows дает следующий результат:

useCP.pp

```
1  {$codepage UTF8}
2  {$IFNDEF WINDOWS}
3   uses cwstring;
4  {$ENDIF}
5   var sw:unicodestring;
6  begin
7   sw:='Альфа(α) и Омега(ω)';
8   writeln(sw);
9  end.
```

```
Z:\strings>chcp 1251
Текущая кодовая страница: 1251

Z:\strings>usecp.exe
Альфа(?) и Омега(?)
```

В отличие от предыдущего примера, в программе не потребовалось использовать функцию **utf8Decode**.

Исключение директивы **codepage** из текста программы приведет к ожидаемо некорректному результату:

```
Z:\strings>usecp.exe
РђР»СЊС„Р°(О±) Рё РћРјРµРіР°(О©)
```

Применение директивы **codepage** упрощает создание программ с активным использованием строк Unicode, так как не требует принудительного вызова функций перекодирования utf8Decode и аналогичных. Однако существуют ситуации, когда применение **codepage** не оправдано. Это связано, во первых, с ограничениями целевой кодировки UCS2 по сравнению с UTF-16, а именно - отсутствием поддержки суррогатных пар, что ограничивает набор поддерживаемых символов и в некоторых случаях может стать причиной отказа от использования указанной директивы. Во вторых, **codepage** ограничивает исходный код программы только одной кодировкой, не предусматривается одновременное существование в программе строковых констант в различных кодировках. В третьих, применение **codepage** понижает эффективность манипуляций с AnsiString и ShortString, если в выражениях с переменными данного типа присутствуют строковые константы (фактически, в таком выражении будут присутствовать дополнительные преобразования строковых констант из UCS2 в системную кодировку).

В примере справа производится попытка вывести на консоль строку, содержащую знак скрипичного ключа. Данный знак имеет код Unicode U+1D11E, в UTF-8 кодируется 4 символами, в - UTF-16 - двумя символами (суррогатной парой), в UCS2 - отсутствует. Компиляция программы приводит к фатальной ошибке:

useCPmus.pp

```
1  {$codepage UTF8}
2  {$IFNDEF WINDOWS}
3   uses cwstring;
4  {$ENDIF}
5   var s:ansistring;
6  begin
7   s:='Скрипичный ключ: 𝄞';
8   writeln(s);
9  end.
```

```
Целевая ОС: Linux for x86-64
Компиляция useCPmus.pp
useCPmus.pp(7,5) Ошибка: Обнаружен код UTF-8, превышающий 65535
useCPmus.pp(7,5) Ошибка: Неверная UTF-8 строка
useCPmus.pp(7,5) Фатально: Не закрыта строковая константа
Фатально: Компиляция прервана
```

Удаление из программы строки **{$codepage UTF8}** позволит ее скомпилировать и запустить:

```
[slinkin@sdanout strings]$ ./useCPmus
Скрипичный ключ: 𝄞
```

Еще одним типом данных, поддерживающим UTF-16, является **WideString**. На всех платформах, кроме ОС Windows, **WideString** эквивалентен UnicodeString. В ОС Windows **WideString** обладает более простой структурой, не поддерживает счетчик ссылок и представляет собой указатель на массив символов WideChar. **WideString** применяется исключительно для обмена данными при использовании API-функций Windows.

Обработка текста в кодировках Unicode.

Одной из основных проблем в обработке String и UnicodeString в кодировках UTF-8 и UTF-16 является переменная длина символа. В результате становится нетривиальными задачи доступа к символу строки по его символьной (визуальной) позиции, замене одного символа в строке на другой, удаление из строки определенного количества символов и т. д. Например, замена одного символа на другой в тексте должна предваряться определением количества байт (UTF-8) или слов (UTF-16), занимаемых обоими символами в соответствующей кодировке и сдвигом оставшейся части строки, если размеры символов не равны друг другу.

Следующий пример демонстрирует невозможность в общем случае прямого доступа к конкретному символу Unicode в тексте по его символьной (визуальной) позиции. Задача программы заключается в поиске позиций символов-пробелов в строках AnsiString в кодировке UTF-8 и UnicodeString в кодировке UTF-16. Программа предназначена для запуска в ОС Linux, однако может быть запущена и в Windows, при соблюдении некоторых условий и с небольшими ограничениями.

utf124.pp

```
1  {$IFNDEF WINDOWS} uses cwstring;{$ENDIF}
2
3  procedure findSpaces(s:ansistring);
4    var sw:unicodestring; i:integer;
5   begin
6      writeln('------------------------');
7      writeln('Текст: ',s);
8      writeln('Размер в байтах(кодировка UTF-8): ',length(s));
9      write('Позиции пробелов(кодировка UTF-8): ');
10     for i:=1 to length(s) do if s[i]=' ' then write(i:3); writeln;
11     sw:=utf8Decode(s);
12     writeln('Размер в словах(кодировка UTF-16): ',length(sw));
13     write('Позиции пробелов(кодировка UTF-16): ');
14     for i:=1 to length(sw) do if sw[i]=' ' then write(i:3); writeln;
15  end;
16
17 const comedyRoadMovie='It''s a Mad, Mad, Mad, Mad World';
18       proVerb='Поспешай не торопясь';
19       flashRoyal='🂪 🂫 🂭 🂮 🂡';
20 begin
21  findSpaces(comedyRoadMovie);
22  findSpaces(proVerb);
23  findSpaces(flashRoyal);
24 end.
```

Функция findSpaces (строки 3-15) принимает в качестве параметра переменную AnsiString в кодировке UTF-8 и выводит на консоль ее содержимое (строка 7), размер (строка 8), позиции пробелов (9-10). Затем исходный текст перекодируется из UTF-8 в UTF-16 и сохраняется в переменной UnicodeString (строка 11). В строках 12-14 на консоль последовательно выводятся размер полученной UnicodeString и позиции пробелов в ней. Размер и и полученные позиции пробелов для AnsiString определяется в байтах, так как размер типа Char (AnsiChar) - 1 байт. Размер и полученные позиции для UnicodeString определяются в

словах, так как размер типа WideChar - 2 байта (одно машинное слово).

В программе определяются три строковые константы в кодировке UTF-8. Первая из них (comedyRoadMovie, строка 17) содержит символы только из нижней части кодовой таблицы ASCII. Вторая (proVerb, строка 18) - как символы-пробелы, так и символы кириллицы, которые в UTF-8 кодируются двумя байтами. Третья (flashRoyal, строка 19) кроме символов-пробелов содержит символы игральных карт, которые в UTF-8 кодируются четырьмя байтами, а в UTF-16 - двумя двубайтными словами (суррогатной парой).

Основная программа вызывает findSpaces для трех вышеописанных констант.

Для запуска программы в ОС Windows следует предварительно сменить кодировку терминала на UTF-8 командой chcp 65001 и выбрать для отображения векторный шрифт. Однако даже в этом случае визуальное представление третьей строки не будет корректным из-за ограниченной поддержки Unicode в базовых шрифтах.

```
[slinkin@sdanout strings]$ ./utf124
------------------------
Текст: It's a Mad, Mad, Mad, Mad World
Размер в байтах(кодировка UTF-8): 31
Позиции пробелов(кодировка UTF-8):   5  7 12 17 22 26
Размер в словах(кодировка UTF-16): 31
Позиции пробелов(кодировка UTF-16):   5  7 12 17 22 26
------------------------
Текст: Поспешай не торопясь
Размер в байтах(кодировка UTF-8): 38
Позиции пробелов(кодировка UTF-8):  17 22
Размер в словах(кодировка UTF-16): 20
Позиции пробелов(кодировка UTF-16):   9 12
------------------------
Текст: 🂪 🂫 🂹 🂾 🃁
Размер в байтах(кодировка UTF-8): 24
Позиции пробелов(кодировка UTF-8):   5 10 15 20
Размер в словах(кодировка UTF-16): 14
Позиции пробелов(кодировка UTF-16):   3  6  9 12
```

Справа демонстрируется результат запуска программы в ОС Linux:

Видим, что размер и позиции пробелов первой строки одинаковы в UTF-8 и UTF-16, а также полностью соответствуют визуальному представлению строки.

Для второй строки ее визуальному представлению соответствует только кодировка UTF-16, так как в ней и пробелы и символы кириллицы занимают по одному слову.

Визуальное представление третьей строки не соответствует результатам ее обработки в UTF-8 и UTF-16. Визуально строка содержит 9 символов Unicode: 5 карточных символов с 4 пробелами между ними. Однако в UTF-8 она кодируется 24 байтами, а в UTF-16 - 14 словами. Символьные позиции пробелов (2 4 6 8), также не соответствуют результатам, полученным для UTF-8 и UTF-16.

Таким образом, в общем случае нельзя осуществить прямой доступ к конкретному символу Unicode текста, зная его символьную позицию. Вместо этого следует последовательно перебирать содержимое строки, анализируя символы в соответствии с правилами кодирования UTF-8 или UTF-16.

Корректная обработка символьной позиции текста в кодировках Unicode реализована в нескольких сторонних программных библиотеках, среди которых наиболее известным является модуль lazutf8 из проекта Lazarus. Модуль поставляется в исходном и скомпилированном виде совместно со средой Lazarus. На моей основной 64-разрядной Linux-системе он доступен по адресу /usr/lib64/lazarus/components/lazutils/lazutf8.pas, на виртуальной машине с Windows XP - по адресу c:\lazarus\components\lazutils\lazutf8.pas. Модуль содержит множество процедур и функций для обработки текста в кодировке UTF-8, не зависит от других модулей и может использоваться отдельно от среды Lazarus. К сожалению для обработки UnicodeString не существует столь-же полной библиотеки. Причина этого, по видимому, в невысокой востребованности обработки суррогатных пар UTF-16 в прикладных системах.

Следующий пример демонстрирует модификацию предыдущей программы поиска позиций пробелов в Unicode-тексте. Вместо исследования UnicodeString производится повторный анализ AnsiString с использованием функций из модуля lazutf8 (строки 10-14).

utf124laz.pp

```
1  uses lazutf8;
2
3  procedure findSpaces(s:ansistring);
4    var i:integer;
5   begin
6      writeln('------------------------');
7      writeln('Текст: ',s);
8      writeln('Размер в байтах(кодировка UTF-8): ',length(s));
9      write('Байтовые позиции пробелов(кодировка UTF-8): ');
10     for i:=1 to length(s) do if s[i]=' ' then write(i:3); writeln;
11     writeln('Размер в символах(кодировка UTF-8): ',utf8length(s));
12     write('Символьные позиции пробелов(кодировка UTF-8): ');
13     i:=utf8pos(' ',s);
14     while i<>0 do begin write(i:3); i:=utf8pos(' ',s,i+1); end; writeln;
15  end;
16
17 const comedyRoadMovie='It''s a Mad, Mad, Mad, Mad World';
18       proVerb='Поспешай не торопясь';
19       flashRoyal='🂪 🂫 🂭 🂮 🂡';
20 begin
21  findSpaces(comedyRoadMovie);
22  findSpaces(proVerb);
23  findSpaces(flashRoyal);
24 end.
```

Результат запуска программы демонстрирует корректность доступа к символьной позиции для каждой из трех строк при использовании возможностей модуля lazutf8.

```
[slinkin@sdanout strings]$ ./utf124laz
------------------------
Текст: It's a Mad, Mad, Mad, Mad World
Размер в байтах(кодировка UTF-8): 31
Байтовые позиции пробелов(кодировка UTF-8):   5  7 12 17 22 26
Размер в символах(кодировка UTF-8): 31
Символьные позиции пробелов(кодировка UTF-8):   5  7 12 17 22 26
------------------------
Текст: Поспешай не торопясь
Размер в байтах(кодировка UTF-8): 38
Байтовые позиции пробелов(кодировка UTF-8):  17 22
Размер в символах(кодировка UTF-8): 20
Символьные позиции пробелов(кодировка UTF-8):   9 12
------------------------
Текст: 🂪 🂫 🂭 🂮 🂡
Размер в байтах(кодировка UTF-8): 24
Байтовые позиции пробелов(кодировка UTF-8):   5 10 15 20
Размер в символах(кодировка UTF-8): 9
Символьные позиции пробелов(кодировка UTF-8):   2  4  6  8
```

Заключение

Обработка текста является одной из важнейших задач программирования и Free Pascal предоставляет достаточно возможностей для ее решения, в том числе благодаря поддержке стандарта Unicode. Не менее серьезные возможности предлагает Free Pascal в обработке сложных составных структур данных, к которым относятся массивы и записи. В следующей части пособия будут рассмотрены нововведения в обработке указанных структур.