UTF-8 & Co.
- Multi-Byte-Zeichenverarbeitung -

Karsten Brodmann

April 2017

1 Gewaltig viele Zeichen

„Zu schön für unsere Ohren und gewaltig viel Noten, lieber Mozart.“ Dieser Ausspruch Josephs II. über die «Entführung aus dem Serail» war wohl weniger Kritik, als vielmehr die nach Worten suchende Reaktion auf die bestürzende Überfülle der neuen Eindrücke. Es war ein Qualitätsurteil, das sich mit einem quantitativen Argument behelfen musste. Mozart konnte daher auch witzig kontern: „Grad so viel Noten, Eure Majestät, als nötig sind.“

1.1 Computer und ihre Zeichensätze: Wohl und Wehe

Noch vor wenigen Jahren waren wir froh und glücklich, wenn wir mittels unserer Tastatur die Zeichen eingeben konnten, die wir zur Erstellung von Texten in unserer jeweiligen Landessprache benötigten. 256 verschiedene Zeichen standen, inklusive der nicht druckbaren Steuerzeichen, zur Verfügung. Das war in der Regel hinreichend. Unangenehm wurde es nur, wenn wir Texte in anderen Landessprachen bekamen, die Zeichen enthielten, die nicht in unserem Zeichensatz enthalten waren. Statt des Originalzeichens stand dann ein Zeichen aus unserem Zeichensatz an dessen Stelle und wollte da, schon rein optisch, gar nicht passen.

Heute passiert das, zumindest bei Programmen mit grafischer Benutzeroberfläche, im Allgemeinen nicht mehr. Alles ist gut und wird schön dargestellt. Wir haben mehrere tausend Zeichen zur Auswahl. Es gibt zwar immer noch hinreichend Fälle im Alltagsleben, in denen doch nicht alles super ist, aber im Großen und Ganzen ist aus Anwendersicht die Welt weitestgehend in Ordnung.

Programmierer und insbesondere Programmieranfänger raufen sich dagegen die Haare. Hatten wir früher verlässlich ein einzelnes Byte zur Zeichenkodierung, so sind wir heute mit so genannten *Multi-Byte-Zeichensätzen* konfrontiert. Der ehedem zur Zeichendarstellung verwendete Datentyp, bei Free Pascal beispielsweise `Char`, funktioniert nicht mehr zuverlässig. Ein `Char`, besitzt nur ein Byte, eine Eigenschaft, die er sich mit den meisten Zeichendatentypen anderer Programmiersprachen teilt. Wollen wir einzelne Zeichen untersuchen, so treffen wir vielfach auf Zeichen, die sich mit diesem Datentyp nicht darstellen lassen. Noch viel schlimmer ist es, dass wir ohne penible Analyse dessen, was man uns vorsetzt, nicht einmal wissen, wie viele Byte ein Zeichen überhaupt hat. In der weit verbreiteten UTF-8-Kodierung können Zeichen ein bis vier Bytes belegen. Das macht die Welt nicht eben einfacher! - Die einschlägigen Internetforen sind voll mit Fragen zur Verarbeitung von Multi-Byte-Zeichen.

Soweit das Stimmungsbild zu Zeichen und deren Verarbeitung auf dem Computer. Gehen wir aber Schritt für Schritt vor. Beginnen wir mit der Historie und schauen uns an, wieso sich was wie entwickelt hat. Sodann schauen wir uns an, was so manchem vorzeitig graue Haare zu bescheren droht: *Multi-Byte-Zeichensätze*. Sie sind der Schlüssel zur heutigen Zeichenvielfalt.

Bezogen auf die tägliche Programmierpraxis sehen wir uns, zumindest ausschnittsweise, an, welche Kodierungen es gibt und wie wir als Programmierer mit ihnen umzugehen haben. Als Programmiersprache verwende ich hier Free Pascal unter macOS. Der notwendige Compiler ist für Microsoft Windows, Gnu/Linux und maOS gratis erhältlich und verhält sich auf den unterstützten Plattformen weitestgehend identisch. So hat der Leser gute Chancen, die hier gezeigten Beispiele am eigenen Rechner auszuprobieren. Der sittliche Nährwert meiner Ausführungen ist aber analog auch auf andere Programmiersprachen übertragbar, auch wenn ich hier zum Teil etwas detaillierter auf Fre Pascal und dessen Bibliotheksfunktionen eingehe. Insbesondere Delphi-Programmierer können die hier vorgestellten Beispiele nahezu 1:1 verwenden.

1.2 Geschichte der Zeichensätze

1.2.1 7-Bit-ASCII

Obgleich der Ur-Vater des digitalen Computers, Konrad Zuse, ein Deutscher war, sind Computer eindeutig amerikanisch geprägt. Und so beginnt unsere Geschichte im Jahr 1963. Der *American Standard Code for Information Interchange* (ASCII, 7-Bit-ASCII) ist eine 7-Bit-Zeichenkodierung und wurde am 17. Juni 1963 von der *American Standards Association* (ASA) als Standard ASA X3.4-1963 veröffentlicht. Diese Zeichenkodierung definiert 128 Zeichen. Das sind 33 nicht druckbare und 95 druckbare Zeichen.

Tabelle 1.1: 7-Bit-ASCCI-Code (hexadezimal)

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	lf	vt	ff	cr	so	si
1	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

Die druckbaren Zeichen umfassen das lateinische Alphabet in Groß- und Kleinbuchstaben, die zehn arabischen Ziffern sowie einige Satz- und wenige Sonderzeichen. Der Zeichenvorrat entspricht dem einer Tastatur für die englische Sprache.

Die nicht druckbaren Steuerzeichen enthalten Ausgabezeichen wie Wagenrücklauf oder Tabulator, Protokollzeichen und ähnliches mehr.

Jedem Zeichen ist ein Bitmuster aus 7 Bit zugeordnet. Das ergibt $2^7 = 128$ verschiedene Bitmuster, die auch als die ganzen Zahlen von 0 bis 127 (hexadezimal: 00–7F) interpretiert werden können. Das Zeichen `A` ist beispielsweise als (`$41`[1]) kodiert. Der Hexadezimalwert entspricht dem dezimalen Wert 65.

[1] Pascal-Notation

In nicht-englischen Sprachen verwendete Sonderzeichen – beispielsweise die deutschen Umlaute – sind im 7-Bit-ASCII-Zeichensatz nicht enthalten.

Auch damals besaß ein Byte natürlich acht Bit. Das für 7-Bit-ASCII nicht benutzte achte Bit konnte für Fehlerkorrekturzwecke, als das so genannte *Paritätsbit*, verwendet werden. Das war sehr praktisch, wenn man die Korrektheit einer Datenübertragung prüfen wollte.

Hinweis: Man beachte: Alle Buchstaben, groß oder klein, sowie auch alle Ziffern sind geordnet. Diese Regel gilt grundsätzlich für alle Zeichensatzkodierungen, auch für moderne Multi-Byte-Zeichensatzkodierungen! Ausnahme: Die thailändische Zeichenkodierung ist im Unicode nach visuellen Kriterien angeordnet.

1.2.2 8-Bit-ASCII

Auch in nicht-englischen Sprachräumen verspürten die Menschen das Bedürfnis, ihre landestypischen Zeichen in Texten verwenden zu wollen. Im so genannten *erweiterten ASCII-Code*, der 8 Bit nutzt, schaffte das achte Bit, unter Wegfall der Pariätsprüfung, den Kodierungsfreiraum für nationale Sonderzeichen. Diese Erweiterungen sind mit dem ursprünglichen 7-Bit-ASCII weitgehend kompatibel. Das heißt, die ersten 128 Zeichen sind in der Regel identisch kodiert. Die einfachsten Erweiterungen sind Kodierungen mit landes- und sprachspezifischen Zeichen, die nicht im lateinischen Grundalphabet enthalten sind.

Bis zur Erweiterung des ASCII-Codes war die Welt einfach. Man konnte nur nicht alles darstellen, weil man es eben mit sieben Bit nicht kodieren konnte. Programmiertechnisch war das achte Bit aber insoweit unproblematisch, als der Standarddatentyp für Zeichen acht Bit hatte und alle Bits des erweiterten ASCII-Codes darin Platz fanden.

Erste Probleme rührten jedoch daher, dass verschiedene Computerhersteller unterschiedliche, eigene 8-Bit-Codes entwickelten. Der *Codepage* 437 genannte Code war lange Zeit der am weitesten verbreitete. Er kam auf dem IBM-PC unter englischem MS-DOS vor. Auch heute findet er sich noch im englischen Kommandozeilenfenster von Microsoft Windows wieder. In deutschen Installation ist dagegen die Codepage 850 vorherrschend.

Auch bei späteren Standards wie ISO 8859 wurden acht Bits verwendet. Dabei existieren mehrere Varianten, wie zum Beispiel ISO 8859-1 für die westeuropäischen Sprachen. Deutschsprachige Versionen von Microsoft Windows, mit Ausnahme des Kommandozeilenfensters, verwenden die auf ISO 8859-1 aufbauende Kodierung Windows-1252. Das ist der Grund, weshalb zum Beispiel die deutschen Umlaute falsch aussehen, wenn Textdateien in der Kommandozeilenumgebung erstellt und unter der grafischen Windows-Oberfläche betrachtet werden.

Aber die jetzt möglichen unterschiedlichen Kodierungen führten natürlich grundsätzlich zu ersten Problemen, Zeichen richtig zu interpretieren. Man musste wissen, welche Kodierung bei der Texterstellung verwendet wurde, um ein Zeichen korrekt wiederzugeben. Die Umkodierung von einer Quell-Codepage in eine Ziel-Codepage war/ist nur bedingt möglich. Ist ein Zeichen in beiden Codepages enthalten, jedoch lediglich anders kodiert, so kann es umkodiert und auf dem Zielsystem korrekt dargestellt werden. Zeichen der Quell-Codepage, die in der Ziel-Codepage nicht enthalten sind, bleiben auf der Strecke.

Kurz: Mit acht Bit konnte/kann zwar jeder irgendwie die für ihn gebräuchlichsten Zeichen verwenden, der Datenaustausch ist aber nur bedingt möglich. Es lässt sich leider nicht jedes Zeichen einer Codepage in eine andere übersetzen. Die Zeichen sind in der Ziel-Codepage einfach nicht definiert. Das war/ist für Anwender wie auch Programmierer problematisch. Der Programmieranfänger merkt das in der Regel nicht. Solange er nur für sich und in seiner Sprach- und Zeichenwelt programmiert, ist alles einfach. Er bemerkt die Problematik erst dann, wenn er mit Internationalisierung und ähnlichen Sachverhalten konfrontiert wird.

Und so gelangen wir in die Gegenwart. Der internationale Daten- und Informationsaustausch erfordert es, unterschiedlichste Zeichen von hüben nach drüben transportieren und korrekt darstellen zu können. Mit einer 8-Bit-Zeichenkodierung ist das nicht möglich, weshalb wir heute mit so genannten *Multi-Byte-Zeichensätzen* konfrontiert sind. Je nach Implementierung verwenden diese Zeichensätze ein bis vier Byte zur Darstellung eines Zeichens. Die Grundlage hierfür bildet der so genannte *Unicode*.

1.2.3 Unicode

Bis hierhin haben wir Zeichensatz, Zeichenkodierung und deren Implementierung nicht streng getrennt. Wenn wir darüber nachdenken, wird uns jedoch klar, dass zum Beispiel eine Codepage 850 beides definierte, den Zeichensatz (verfügbare Zeichen) und deren Kodierung (Bitmuster) sowie dessen Implementierung in einem Byte.

In den herkömmlichen 8-Bit-Zeichenkodierungen sind nach Abzug der Steuerzeichen 95 Zeichen bei 7-Bit-ASCII und 191 Zeichen bei den 8-Bit-Zeichensätzen als Schrift- und Sonderzeichen darstellbar. Diese Zeichenkodierungen erlauben nur wenige Sprachen gleichzeitig im selben Text darzustellen. Will man mehr, so besteht der Workaround darin, in einem Textdokument verschiedene Schriften mit unterschiedlichen Zeichensätzen zu verwenden. Das geht natürlich nur in formatierten Textdokumenten, wie beispielsweise Textverarbeitungen wie Microsoft Word oder LibreOffice. Reine Textdateien, die keine Schriftartunterstützung besitzen, gehen leer aus. Das dies keine optimale Ausgangsbasis für den internationalen Datenaustausch ist, leuchtet wohl jedem ein.

Im Oktober 1991 wurde deshalb, nach mehrjähriger Entwicklungszeit, die Version 1.0.0 des so genannten *Unicode-Standards* veröffentlicht. Diese erste Version kodierte nur die europäischen, nahöstlichen und indischen Schriften. Version 1.0.1 kodierte, etwa acht Monate später, auch schon ostasiatische Schriftzeichen. Mit der Veröffentlichung von Unicode 2.0 im Juli 1996 wurde der Standard von ursprünglich 65.536 Zeichen auf die heutigen 1.114.112 Codepunkte erweitert. Das entspricht einer Kodierung von U+0000 bis U+10FFFF. Zur Kodierung von Zeichen stehen 1.111.998 Codepunkte zur Verfügung, die aber bislang noch nicht vollständig genutzt werden. Man beachte dabei: Ein Codepunkt ist nicht zwingend ein kodiertes Zeichen!

Codepunkte und Zeichen

Jedes im Unicode-Standard kodierte elementare Zeichen ist einem so genannten *Codepunkt* zugeordnet. Diese werden hexadezimal und mindestens vierstellig, also nötigenfalls mit führen-

den Nullen, und mit einem vorangestellten `U+` dargestellt. Beispiel: U+03A8 für das Zeichen Ψ.

Eine schöne Unicode-Zeichentabelle findet sich unter `http://unicode-table.com/de/#control-character`.

Der gesamte vom Unicode-Standard beschriebene Bereich umfasst, wir haben es bereits gesagt, 1.114.112 Codepunkte (U+0000 bis U+10FFFF). Er ist in 17 Ebenen zu je 65.536 Zeichen unterteilt. Von den 17 Ebenen werden zur Zeit nur sechs genutzt. Namentlich erwähnen möchte ich die erste Ebene (Ebene 0), die *BMP*-Ebene (Basic Multilingual Plane) heißt. Hier finden sich alle europäischen Sonderzeichen wieder. Die beiden letzten Ebenen (Ebene 15 und 16) bilden die so genannten *PUP* (Private Use Planes). Hier liegen die *PUA-A* und *PUA-B* (Private Use Area A und B). Diese speziellen Bereiche sind für die private Nutzung reserviert, wie der Name unschwer verrät. Hier werden niemals Codepunkte für in Unicode standardisierte Zeichen zugewiesen. Diese können jedoch für privat definierte Zeichen verwendet werden. Hierzu müssen sich die Erzeuger und Nutzer der Zeichen absprechen, wenn sie derartige Zeichen in ihren Texten verwenden.

Einige Bereiche sind nicht für die Zeichenkodierung zulässig. Für Zeichen stehen lediglich 1.111.998 Codepunkte zur Verfügung. Die Anzahl der tatsächlich zugewiesenen Codepunkte ist jedoch deutlich niedriger, nimmt aber immer noch zu. So wurde beispielsweise erst jüngst, im Juni 2015, unter anderem das Cherokee-Alphabet aufgenommen. - Es ist wirklich interessant, was es alles gibt.

Star Trek Fans mögen allerdings traurig sein, denn Klingonisch ist offiziell nicht im Unicode erfasst. Dank der beliebten Fernsehserie „The Big Bang Theory“ gibt es bestimmt mehr Menschen, die sich mit Klingonisch befassen, als es Interessenten für das Cherokee-Alphabet gibt. Umgekehrt sind die Cherokee natürlich reale Mitmenschen unserer Erdengesellschaft - reale Klingonen wurden bislang nicht auf der Erde gesichtet. *ConScript Unicode Registry* ist ein Projekt, welches sich um die Zuordnung von Codepunkten in der PUA des Unicodes bemüht. Dort verortet das Projekt das klingonische Alphabet im Bereich U+F8D0 bis U+F8FF.

 - Das ist Klingonisch und heißt: „Ich habe Kopfschmerzen.“

Kodierung

Neben dem eigentlichen Zeichensatz sind auch eine Reihe von Kodierungen definiert, die den Unicode-Zeichensatz implementieren und die benutzt werden können, um den vollen Zugriff auf alle Unicode-Zeichen zu haben. Sie werden *Unicode Transformation Format* (UTF) genannt. Am weitesten verbreitet sind UTF-32, UTF-16 und UTF-8.

1.3 Unicode Transformation Format (UTF)

1.3.1 UTF-32

Bei der UTF-32-Kodierung wird jedes Zeichen mit vier Byte (32 Bit) kodiert. Diese Kodierung ist, aufgrund ihrer konstanten Kodierungslänge je Zeichen, die einfachste Kodierung. UTF-16

und UTF-8 besitzen variable Byteanzahlen je Zeichen.

Vorteile

UTF-32 gewährleistet (weitestgehend) wahlfreien Zugriff auf ein bestimmtes Zeichen innerhalb einer Zeichenkette. Die Adresse des n-ten Zeichens kann durch einfache Zeigerarithmetik berechnet werden. Im $\mathcal{O}$-Kalkül (Big-O-Kalkül) beträgt der Aufwand hierfür $\mathcal{O}(1)$, ist also konstant. Zudem ist es einfach möglich, anhand der Größe eines Dokuments in Bytes, die Anzahl der enthaltenen Zeichen zu ermitteln. Hierzu ist die Anzahl der Bytes einfach durch 4 zu dividieren.

Diese soeben beschriebenen Eigenschaften relativieren sich allerdings dadurch, dass ein Unicode-Zeichen nicht zwingend einem Schriftzeichen entsprechen muss, was beispielsweise bei Ligaturen der Fall ist. UTF-Kodierungen kodieren immer nur einen Codepunkt. Entspricht dieser Codepunkt keinem Zeichen, funktioniert der wahlfreie Zugriff natürlich nicht mehr. In Bezug auf reine Textdateien ist das jedoch zu vernachlässigen, so dass die beschriebenen Vorteile durchaus real existieren.

Nachteile

Der augenscheinlichste Nachteil von UTF-32 ist der Speicherbedarf. Jedes Zeichen belegt 32 Bit (4 Byte). Texte, die überwiegend aus lateinischen Buchstaben bestehen, benötigen in UTF-32 in etwa den vierfachen Speicherplatz gegenüber UTF-8 (siehe unten). Nihct benötigte Bytes werden mit NUllen aufgefüllt (Null-Bytes). UTF-32 wird daher selten zum Speichern auf Datenträgern verwendet.

Ein weiterer Nachteil, der, wie ich meine, noch ausschlaggebender als der Speicherverbrauch ist, ist die fehlende Abwärtskompatibilität zu ASCII. Im Gegensatz zu UTF-8, welches abwärtskompatibel zu ASCII ist, benötigt man in jedem Fall einen UTF-32-fähigen Editor, um auch einfache Texte, die lediglich ASCII-Zeichen verwenden, bearbeiten zu können. Andernfalls sind alle Zeichen durch drei Null-Bytes getrennt, über die der Editor stolpert.

1.3.2 UTF-16

Bei der UTF-16-Kodierung wird jedem Unicode-Zeichen eine speziell kodierte Kette von ein oder zwei 16-Bit-Einheiten zugeordnet, so dass sich, wie auch bei den anderen UTF-Formaten, alle Unicode-Zeichen abbilden lassen.

UTF-16 wird beispielsweise für die interne Zeichendarstellung der Betriebssysteme Microsoft Windows und macOS/OS X verwendet. Auch einige Software-Entwicklungsumgebungen/-Frameworks nutzen UTF-16. Das sind beispielsweise Java oder .NET).

Vorteile

Die meisten in der Realität gebräuchlichen Zeichen sind mit zwei Byte kodiert, können also ohne aufwendige Rechnerei in UTF-16 genutzt werden.

Wird ein UTF-16-kodierter Text als ASCII interpretiert, sind lateinische Buchstaben erkennbar, wenn auch durch Null-Bytes getrennt. Ungeachtet der Null-Bytes, ist der Text aber lesbar.

Nachteile

Aufgrund der Kodierung aller Zeichen der BMP in zwei Bytes, benötigt die UTF-16-Kodierung rund doppelt so viel Speicherplatz wie UTF-8 oder eine geeignete ISO-8859-Kodierung bei Texten, welche hauptsächlich aus lateinischen Buchstaben bestehen. Werden jedoch viele BMP-Zeichen jenseits des Codepoints U+007F kodiert, etwa chinesische Schriftzeichen, benötigt UTF-16 vergleichbar viel oder gar weniger Speicher. Demgegenüber entsteht jedoch der Aufwand, analysieren zu müssen, ob ein Zeichen zwei oder vier Byte besitzt.

1.3.3 UTF-8

UTF-8 wird vor allem von Unix- oder Unix-ähnlichen Betriebssystemen wie GNU/Linux oder macOS/OS X, sowie in verschiedenen Internetdiensten (E-Mail, WWW) genutzt.

UTF-8 ist die am weitesten verbreitete Kodierung für Unicode-Zeichen. Die Kodierung wurde im September 1992 von Ken Thompson und Rob Pike bei Arbeiten am Plan-9-Betriebssystem festgelegt. Ursprünglich als FSS-UTF bezeichnet, wurde sie später, im Rahmen der Standardisierung, in UTF-8 umbenannt.

Vorteile

UTF-8 ist in den ersten 128 Zeichen (Kodierung 0 bis 127) identisch mit ASCII. Für die Kodierung englischsprachiger Texte reicht jeweils ein Byte pro Zeichen. Daher können solche Texte auch mit nicht-UTF-8-fähigen Texteditoren ohne Beeinträchtigung bearbeitet werden, was einen der Gründe für den Status als De-facto-Standard-Zeichenkodierung des Internets und damit verbundener Dokumenttypen darstellt. Aktuell, Frühjahr 2017, verwenden mehr als 80% aller Websites UTF-8.

Nachteile

In anderen Sprachen ist der Speicherbedarf in Bytes pro Zeichen größer, wenn diese vom ASCII-Zeichensatz abweichen. Bereits die deutschen Umlaute erfordern zwei Bytes. Kyrillische, fernöstliche und Sprachen aus dem afrikanischen Raum belegen bis zu 4 Bytes je Zeichen. Da die Verarbeitung von UTF-8 als Multi-Byte-Zeichenfolge wegen der notwendigen Analyse jedes Bytes, im Vergleich zu Zeichenkodierungen mit fester Byteanzahl je Zeichen, mehr

Rechenaufwand und auch mehr Speicherplatz erfordert, werden abhängig vom Einsatzszenario auch andere UTF-Kodierungen zur Abbildung von Unicode-Zeichensätzen verwendet: Microsoft Windows verwendet, als sicherlich meistgenutztes Desktop-Betriebssystem, intern UTF-16. Das ist ein Kompromiss zwischen dem Speicherbedarf von UTF-32 und der aufwendigen Zeichenanalyse bei UTF-8.

1.4 UTF-8-Zeichenkodierung

Unicode-Zeichen mit Werten aus dem Bereich von 0 bis 127 (0 bis 7F hexadezimal) werden in der UTF-8-Kodierung als ein Byte mit dem gleichen Wert wiedergegeben. Daher sind alle Daten, für die ausschließlich 7-Bit-ASCII-Zeichen verwendet werden, in beiden Darstellungen identisch.

Unicode-Zeichen größer als 127, also zum Beispiel deutsche Umlaute und andere Sonderzeichen, werden in Byteketten von zwei bis vier Byte kodiert.

Tabelle 1.2: UTF-8-Zeichenkodierung

Unicode-Bereich (hexadezimal)	UTF-8-Kodierung (binär)
0000 0000 – 0000 007F	0xxxxxxx
0000 0080 – 0000 07FF	110xxxxx 10xxxxxx
0000 0800 – 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 – 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Das erste Byte beginnt eines Multi-Byte-Zeichens beginnt immer mit 11. Die zweite Eins ist das Flag dafür, dass noch ein weiteres Byte folgt. Hat ein Zeichen mehr als zwei Bytes, folgen weitere Flags. Die Flags sind von den Daten durch eine 0 getrennt. Alle folgenden Bytes beginnen mit 10. Die xxxxx stehen für die Bits des Unicode-Zeichenwerts. Mit 0 beginnende Byte einer Zeichenkette sind also 7-Bit-ASCII–Zeichen, mit 11 beginnende beginnende Bytes stellen das erste Byte eines Multi-Byte-Zeichens dar. Die Folgebytes beginnen mit 10. Die Anzahl der höchstwertigen 1-en, der bis zu vier Flag-Bits, des ersten Bytes eines Multi-Byte-Zeichens, geben Auskunft über die Byte-Anzahl des Zeichens.

1.5 Schriftarten

Ob das entsprechende Unicode-Zeichen auch tatsächlich korrekt am Bildschirm erscheint, hängt davon ab, ob die verwendete Schriftart eine Glyphe für das gewünschte Zeichen (also eine Grafik für die gewünschte Zeichennummer) enthält. Oftmals, zum Beispiel unter Windows, wird, falls die verwendete Schrift ein Zeichen nicht enthält, nach Möglichkeit ein Zeichen aus einer anderen Schrift eingefügt.

Mittlerweile hat der Coderaum von Unicode einen Umfang angenommen (mehr als 100.000 Schriftzeichen), der sich nicht mehr vollständig in einer Schriftdatei unterbringen lässt. Die

heute gängigsten Schriftdateiformate, TrueType und OpenType, können maximal 65.536 Glyphen enthalten. Unicode-Konformität einer Schrift bedeutet also nicht, dass der komplette Zeichensatz enthalten ist, sondern lediglich, dass die darin enthaltenen Zeichen normgerecht kodiert sind.

2 UTF-8 und Programmierpraxis

Nun wissen wir einiges über Unicode und insbesondere die UTF-8-Kodierung. Jetzt wollen wir uns ansehen, wie die diesbezügliche Programmierpraxis mit Free Pascal aussieht und wie wir das neu erworbene Wissen umsetzen können. Wie bereits eingangs gesagt, sind die im Folgenden besprochenen Techniken auch auf andere Programmiersprachen übertragbar. Free Pascal dient nur als Beispiel.

2.1 Free Pascal und Zeichenkodierung

Die folgenden Ausführungen beziehen sich ausschließlich auf die Verarbeitung von Multi-Byte-Zeichen. Die Beispielprogramme wurden auf UTF-8-Systemen erstellt und getestet. Das waren Ubuntu 14.04 LTS (64 Bit) und macOS 10.12. Für Windows-Programmierer, die Konsolenanwendungen schreiben, sind die folgenden Ausführungen nur bedingt nachvollziehbar, beispielsweise dann, wenn sie UTF-8-kodierte Textdateien einlesen und dekodieren wollen oder die Codepage 65001 eingestellt haben. Linux und macOS arbeiten durchweg mit UTF-8.

Free Pascal und andere Programmiersprachen treffen grundsätzlich keine Annahmen bezüglich irgendeiner Zeichenkodierung. So wie wir unsere Zeichen in der Konsole oder im Editor, zum Erfassen der Quelltexte, eingeben, so werden sie von Free Pascal entgegengenommen und verarbeitet.

2.1.1 Einfache Ein- und Ausgabe von Zeichenketten

Betrachten wir das folgende Beispielprogramm:

```
1  program EinfacheZeichenketten;
2    { Ein- und Ausgabe einfacher Zeichenketten
3      (ohne WideString-Manager)
4    }
5  var
6    s: String;  // Short-/Ansi-Zeichenkette
7  begin
8    writeLn('Zeichenkette mit Umlauten: äöüÄÖÜ und ß');
9    write('Bitte noch eine Zeichenkette mit Umlauten: ');
10   readLn(s);
11   writeLn('Ihre Eingabe: ', s);
12   writeLn('Die Eingabe war ', Length(s), ' Byte lang.')
13 end.
```

Listing 2.1: einfachezeichenketten.pas

Der Testlauf ist interessant. Ich meine damit nicht, dass ich mich im ersten Lauf der Aufforderung widersetzt habe, irgendwas mit Umlauten einzugeben. Nein, das Interessante ist, keinerlei Vorkehrungen für irgendwelche Sonderzeichen getroffen zu haben und dennoch ein vollkommen korrekt arbeitendes Programm zu haben. Das war sicherlich, nach den obigen Ausführungen zu den verschiedenen Kodierungen, nicht jedem klar. Es ist aber auch keine Leistung von Free Pascal. Unsere Betriebssystemumgebung sorgt dafür. Free Pascal fungiert einfach als transparente Durchreiche, wenn wir so wollen. Es nimmt die eingegebenen Zeichen (Byte-Folgen) entgegen, speichert sie unverändert und gibt sie ebenso unverändert wieder aus.

```
$ ./einfachezeichenketten
Zeichenkette mit Umlauten: äöüÄÖÜ und ß
Bitte noch eine Zeichenkette mit Umlauten: abc
Ihre Eingabe: abc
Die Eingabe war 3 Byte lang.
$ ./einfachezeichenketten
Zeichenkette mit Umlauten: äöüÄÖÜ und ß
Bitte noch eine Zeichenkette mit Umlauten: Ä
Ihre Eingabe: Ä
Die Eingabe war 2 Byte lang.
$ ./einfachezeichenketten
Zeichenkette mit Umlauten: äöüÄÖÜ und ß
Bitte noch eine Zeichenkette mit Umlauten: 文
Ihre Eingabe: 文
Die Eingabe war 3 Byte lang.
```

Das Wesentliche, was wir jedoch erkennen können, ist: Die Eingabezeichen sind offensichtlich unterschiedlich groß. Während die Buchstaben des Grundalphabets lediglich ein Byte je Zeichen benötigen, benötigt das Ä (Umlaut) schon zwei Bytes. Das japanische Schriftsymbol benötigt gar drei Bytes. Das ist, so kann man wohl sagen, UTF-8 in der Praxis. Am Bildschirm ist die unterschiedliche Byte-Größe nicht erkennbar.

Vorläufige Erkenntnis: In einer normalen `String`-Variablen oder einen Zeichen-Array (C/C++) können UTF-8-Zeichen(ketten) gespeichert werden. Weder die Ein- noch die Ausgabe bereiten Schwierigkeiten. Obwohl `String` intern aus `Char`-Elementen besteht, können UTF-8-Zeichen gespeichert werden. Wie wir aufgrund der Längen haben erkennen können, belegt ein UTF-8-Zeichen gegebenenfalls einfach mehrere Bytes, sprich `Char`-Elemente eines Strings.

2.1.2 Byte- und Zeichenlänge von UTF-8-Zeichenketten

Wir haben soeben gesehen, das Sonderzeichen mehr als ein Byte benötigen. Die Funktion `Length` ermittelt, wenn wir ihr eine (unbehandelte) UTF-8-Zeichenkette geben, die Anzahl der Bytes, die die Zeichenkette beansprucht, nicht die Anzahl der Zeichen. Das ist zumindest das Ergebnis, denn tatsächlich zählt `Length` die Anzahl der belegten `Char`-Elemente in der

Zeichenkette. Da ein `Char` ein Byte groß ist, ist das Ergebnis mit der Anzahl der belegten Bytes identisch.

Nun stellt sich die Frage, wie wir die Anzahl der Zeichen in einer Zeichenkette ermitteln können. Wir können uns sicherlich an die Arbeit machen, jedes Byte der Zeichenkette einzeln zu analysieren, ob es ein Zeichen oder nur Teil eines Zeichens ist. Unabhängig davon, wie das zu bewerkstelligen ist, wäre es doch ein ziemlich mühseliges Verfahren. Praktischer wäre es, wenn wir auf eine Bibliotheksroutine zurückgreifen könnten, also jemand anderes schon die erforderliche Arbeit getan hätte. Und in der Tat werden wir, wenn wir uns die Dokumentation zu Free Pascal ansehen, fündig. Die Routine heißt `UTF8decode`. In anderen Programmiersprachen existieren überlicherweise entsprechende Pendants.

`UTF8decode` nimmt als Parameter einen UTF-8-String entgegen. Als Rückgabe erhalten wir einen `UnicodeString`. Diese sind laut Dokumentation aus `WideChar`-Elementen zusammengesetzt und UTF-16 kodiert. Zudem verrät uns die Dokumentation, das `Lenght` von derart aufgebauten Zeichenketten deren Zeichenanzahl ermitteln kann.

Demnach können wir `Length` verwenden, um anhand der unbehandelten Zeichenkette die Anzahl der Bytes zu ermitteln. Für die, mittels `UTF8decode`, konvertierte Zeichenkette liefert uns `Length` die Anzahl der Zeichen. Um `UTF8decode` verwenden zu können, müssen wir den WideString-Manager aktivieren. Unter Windows kann dieser Schritt entfallen. Bei Unix-Betriebssystemen greift Free Pascal auf die C-Bibliotheken zu, weshalb in der `uses`-Klausel die Unit `CWString` einzubinden ist.

Prüfen wir das mit einem kurzen Programm.

```
1  program BytesZeichenZaehlen;
2  uses
3    CWString;  // WideChar-Manager
4  var
5    s: String;
6  begin
7    write('Bitte eine Zeichenkette: '); readLn(s);
8    writeLn('Anzahl Bytes  : ', Length(s));
9    writeLn('Anzahl Zeichen: ', Length(UTF8decode(s)))
10 end.
```

Listing 2.2: byteszeichenzaehlen.pas

```
$ ./byteszeichenzaehlen
Bitte eine Zeichenkette: abcdefghijklMNOPQRSTUVWXYZ
Anzahl Bytes  : 26
Anzahl Zeichen: 26
$ ./byteszeichenzaehlen
Bitte eine Zeichenkette: äöüÄÖÜß
Anzahl Bytes  : 14
Anzahl Zeichen: 7
$ ./byteszeichenzaehlen
Bitte eine Zeichenkette: oooh文
Anzahl Bytes  : 7
Anzahl Zeichen: 5
```

Das war augenscheinlich einfach. Zur Demonstration habe ich das obige Programm auch einmal schnell und ziemlich schmutzig in C formuliert. Das sieht dann so aus:

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <string.h>
#include <stdlib.h>

int main() {
  char string[100];
  wchar_t wstring[100];

  setlocale(LC_ALL, "");
  printf ("Bitte eine Zeichenkette: ");
  scanf("%s",string);
  printf("Anzahl Bytes: %lu\n", strlen(string));
  mbstowcs(wstring, string, 100);
  printf("Anzahl Zeichen: %lu\n", wcslen(wstring));
  return 0;
}
```

Listing 2.3: byteszeichenzaehlen_c.c

Wie wir leicht erkennen, sind die Parallelen der Implementierung unverkennbar, auch wenn die verwendeten Funktionen ein wenig anders heißen.

2.1.3 Zeichen in Zeichenketten ersetzen

Eines oder mehrere Zeichen in einer Zeichenkette ersetzen zu wollen, ist sicherlich eine häufig vorkommende Aufgabe. In der Unit `StrUtils` finden wir zwei Routinen, die Hilfe versprechen. Es sind `replaceStr` und `replaceText`. Letztere lässt Groß- und Kleinschreibung unberücksichtigt. Laut Dokumentation erwarten die beiden eine Ansi-Zeichenkette, die (ebenfalls laut Dokumentation) auch UTF-8 kodiert sein kann. Besondere Vorbereitungen sind demnach nicht zu treffen.

```
program ZeichenErsetzen;
uses
  StrUtils;
var
  s: String;   // Zeichenkette, in der ersetzt werden soll
  m: String;   // Muster, welches ersetzt werden soll
  e: String;   // Ersatzzeichen/-text
begin
  write('Bitte eine Zeichenkette: '); readLn(s);
  write('zu ersetzen: '); readLn(m);
  write('Ersatztext : '); readln(e);
  s:=replaceStr(s, m, e);
  writeLn('Ergebnis : ', s)
end.
```

Listing 2.4: zeichenersetzen.pas

Probieren wir es aus:

```
$ ./zeichenersetzen
Bitte eine Zeichenkette: Ich bin der Weihnachtsmann.
zu ersetzen: Weihnachtsmann
Ersatztext : Nikolaus
Ergebnis : Ich bin der Nikolaus.
$ ./zeichenersetzen
Bitte eine Zeichenkette: In dieses Haus wurde Mineralöl verbaut.
zu ersetzen: öl
Ersatztext : wolle
Ergebnis : In dieses Haus wurde Mineralwolle verbaut.
```

Das war, wie zu erwarten, auch kein Problem. Wer selbst mal versucht hat, eine Routine zum Suchen und Ersetzen zu schreiben weiß, dass es ziemlich egal ist, wie etwas in einer Zeichenkette gespeichert wird. Man vergleicht einfach Byte für Byte beziehungsweise in diesem konkreten Fall `Char` für `Char`. Das was dort jeweils gefunden wird, wird nicht auf dessen darstellbaren Inhalt geprüft. Es werden einfach Bitmuster verglichen.

Wie sieht es mit anderen Standardfunktionen zur Manipulation von Zeichenketten aus? - Wir wollen hier nicht alle Funktionen durchhecheln. Wichtig ist es nur, das Prinzip zu verstehen, wie diese Funktionen in Free Pascal (oder einer anderen Sprache) implementiert sind und anhand der Dokumentation zu erkennen, ob vielleicht Besonderheiten bei deren Verwendung mit UTF-8-Zeichen(ketten) zu berücksichtigen sind.

Schauen wir uns exemplarisch noch die Funktionen `UpCase` und `LowerCase` an. Diese verwandeln Klein- in Großbuchstaben und umgekehrt. Hier kommt es also auf die konkrete Erkennung eines Zeichens an, um dessen groß- beziehungsweise kleingeschriebenes Pendant zu ermitteln.

```
1 program GrossKleinSchreibung;
2 var
3  s: String;
4 begin
5   write('Bitte eine Zeichenkette: ');
6   write('-> '); readLn(s);
7   writeLn('UpperCase: ', UpCase(s));
8   writeLn('LowerCase: ', LowerCase(s))
9 end.
```

Listing 2.5: grosskleinschreibung1.pas

```
Bitte eine Zeichenkette:
-> Groß- und Kleinschreibung mit Umlauten: äöüÄÖÜ - Ok?
UpperCase: GROß- UND KLEINSCHREIBUNG MIT UMLAUTEN: äöüÄÖÜ - OK?
LowerCase: groß- und kleinschreibung mit umlauten: äöüÄÖÜ - ok?
```

Ganz offensichtlich wurden die Umlaute, also Zeichen mit mehr als einem Byte nicht korrekt behandelt. Für Free Pascal, welches hier nur einfache `Char`-Zeichen erkennt, sind die Elemente

der Multi-Byte-Zeichen keine Zeichen, die es transformieren kann. Also passiert auch nichts damit. Sie bleiben, wie sie sind.

Schauen wir nochmals in die Dokumentationen und suchen eine Alternative. Wir finden die Funktionen `AnsiUpperCase` und `AnsiLowerCase` in der Unit `SysUtils`. Nicht wirklich offensichtlich, die beiden Funktionen machen aber genau das, was wir beabsichtigen. Es wäre allerdings der Hinweis nützlich, die Unit `CWString` einbinden zu müssen. Ohne die funktioniert es nämlich nicht. Allerdings sollte man sich unter Linux oder macOS grundsätzlich angewöhnen, `CWString` einzubinden. Es bleibt einem dann die eine oder andere Überraschung mit Zeichen erspart.

Modifizieren wir also unseren Programmcode:

```
1  program GrossKleinSchreibung;
2  uses
3    CWString, // WideChar-Manager (unter Windows nicht erforderlich)
4    SysUtils;
5  var
6   s: String;
7  begin
8    writeLn('Bitte eine Zeichenkette: ');
9    write('-> '); readLn(s);
10   writeLn('UpperCase: ', AnsiUpperCase(s));
11   writeLn('LowerCase: ', AnsiLowerCase(s))
12 end.
```

Listing 2.6: grosskleinschreibung2.pas

```
$ ./grosskleinschreibung
Bitte eine Zeichenkette:
-> Groß- und Kleinschreibung mit Umlauten: äöüÄÖÜ  Ok?
UpperCase: GROß- UND KLEINSCHREIBUNG MIT UMLAUTEN: ÄÖÜÄÖÜ  OK?
LowerCase: groß- und kleinschreibung mit umlauten: äöüäöü  ok?
$ _
```

Alles ok. - Schauen wir uns dennoch nach einer Alternativen um. In der Unit `SysUtils` verbergen sich zwei auf `WideChar`-Zeichenketten spezialisierte Funktionen. Die Funktionen heißen `WideUpperCase` und `WideLowerCase`.

```
1  program GrossKleinSchreibung;
2  uses
3    CWString, SysUtils;
4  var
5   s: String;
6  begin
7    writeLn('Bitte eine Zeichenkette: ');
8    write('-> '); readLn(s);
9    writeLn('UpperCase: ', UTF8encode(WideUpperCase(UTF8decode(s))));
10   writeLn('LowerCase: ', UTF8encode(WideLowerCase(UTF8decode(s))))
11 end.
```

Listing 2.7: grosskleinschreibung3.pas

```
$ ./grosskleinschreibung
Bitte eine Zeichenkette:
-> Groß- und Kleinschreibung mit Umlauten: äöüÄÖÜ  Ok?
UpperCase: GROß- UND KLEINSCHREIBUNG MIT UMLAUTEN: ÄÖÜÄÖÜ  OK?
LowerCase: groß- und kleinschreibung mit umlauten: äöüäöü  ok?
$ _
```

Der Programmcode ist umständlicher und deutlich weniger schön als die vorige Lösung, aber das Ergebnis ist dennoch in Ordnung.

Nun, ich würde mir wünschen, in der Dokumentation gäbe es mehr Hinweise auf das konkrete Verhalten der Routinen in Bezug auf UTF-8. Ich persönlich empfinde manche Hinweise auch als missverständlich. Im Zweifelsfall muss man es eben kurz ausprobieren. Aber, und das ist die Erkenntnis aus dieser Übung, es gibt definitiv immer einen Weg. - Man muss allerdings manchmal etwas suchen. Die Ausführungen zu `Strings` und wie der Compiler in welchen Modi wie mit ihnen umgeht, sie automatisch wandelt oder nicht, sind zu diesem Thema sicherlich hilfreich und ich empfehle, diese Abschnitte unbedingt zu lesen. Die Dokumentation hierzu findet sich im *Free Pascal Reference Guide* unter der Überschrift „Character Types“. Analoges gilt natürlich für Programmierer anderer Programmiersprachen.

2.1.4 Zeichenketten zeichenweise bearbeiten

Jetzt wollen wir uns weiter damit befassen, wirklich zeichenweise auf eine Zeichenkette zuzugreifen. Bislang haben wir Zeichenketten letztlich immer als Ganzes betrachtet. Beim Suchen und Ersetzen haben wir auf vorgefertigte Routinen zurückgreifen können. Wie können wir es anstellen, eine Zeichenkette zeichenweise auszugeben. Wenn wir diese Frage beantworten können, ist das auch gleich die Antwort auf alle anderen Fragen, die eine zeichenweise Bearbeitung verlangen. Schauen wir einmal.

Einen Ansi-String in einer `for`-Schleife zu durchlaufen und dabei von `Char`-Element zu `Char`-Element zu hüpfen, ist mit Sicherheit keine Lösung. Sehen wir uns das dennoch als abschreckendes Beispiel an:

```
1  program Zeichenweise;
2  var
3    s  : String;  // Zeichenkette
4    i,j: Integer; // Laufvariable
5  begin
6    write('Bitte eine Zeichenkette: '); readLn(s);
7    for i:=1 to Length(s) do begin
8      for j:=1 to i do
9        write(s[j]);
10     writeLn
11   end
12 end.
```

Listing 2.8: zeichenweise1.pas

```
$ ./zeichenweise1
Bitte eine Zeichenkette: Eine Zeichenkette
E
Ei
Ein
Eine
Eine
Eine Z
Eine Ze
Eine Zei
Eine Zeic
Eine Zeich
Eine Zeiche
Eine Zeichen
Eine Zeichenk
Eine Zeichenke
Eine Zeichenket
Eine Zeichenkett
Eine Zeichenkette
$ ./zeichenweise1
Bitte eine Zeichenkette: Noch eine, aber ääh ...
N
No
Noc
Noch
Noch
Noch e
Noch ei
Noch ein
Noch eine
Noch eine,
Noch eine,
Noch eine, a
Noch eine, ab
Noch eine, abe
Noch eine, aber
Noch eine, aber
Noch eine, aber ?
Noch eine, aber ä
Noch eine, aber ä?
Noch eine, aber ää
Noch eine, aber ääh
Noch eine, aber ääh
Noch eine, aber ääh .
Noch eine, aber ääh ..
Noch eine, aber ääh ...
$ _
```

Ja, ja, ich weiß, das Ergebnis war voraussehbar. Dennoch ist das Resultat interessant. Solange das darzustellende Zeichen nicht als solches identifiziert werden kann, also noch kein vollständiges UTF-8-Zeichen vorliegt, wird ein ?-Zeichen ausgegeben. Aber Achtung: Das dargestellte Fragezeichen ist nur der Hinweis auf einen Fehler in der Zeichenkodierung. Es bedeutet nicht, dass das Bitmuster, welches wir in die Ausgabe geschickt haben, ein Fragezeichen war.

Das bestätigt übrigens nochmals das schon erlangte Wissen über deutsche Umlaute. Sie bestehen aus zwei Byte. Testen Sie das mal mit einem 3-Byte-Zeichen. Da kommen dann zwei Fragezeichen bevor das eigentliche Zeichen erscheint - logisch.

```
1  program Zeichenweise;
2  var
3    s  : String;              // Zeichenkette
4    i,j: Integer;             // Laufvariable
5    a  : array[0..3] of Byte; // Hilfsarray für Zeichenelemente
6    k  : Byte;                // Array-Index
7    z  : UTF8String;          // fertiges Zeichen (1-Zeichen-String)
8  begin
9    write('Bitte eine Zeichenkette: '); readLn(s);
10   if Length(s) >0 then begin
11     i:=1;
12     while i<=Length(s) do begin      // für  alle Elemente
13       j:=1;
14       while j<=i do begin             // für alle Bytes j=1 bis i
15         fillByte(a[0], SizeOf(a), 0); // Hilfsarray nullen
16         a[0]:=Byte(s[j]);
17         if ord(a[0])<=$7f then begin
18           write(s[j]); // normales Zeichen gefunden (Bit 7 war 0)
19           inc(j)
20         end
21         else begin
22           // Multi-Byte-Zeichen gefunden. Bit 7 des ersten Bytes
23           // gesetzt. Jetzt folgen bis zu 3 Bits, die die Anzahl
24           // der Bytes darstellen.
25           k:=1;
26           while (k<=3) and ByteBool((a[0] shl k) and %10000000) do
27           begin
28             // jetzt alle Bytes lesen, wie Bits gesetzt sind
29             a[k]:=Byte(s[j+k]);
30             inc(k)
31           end; // alle Bytes gelesen,
32           j:=j+k; // Laufvariable für Zeichenkette aktualisieren
33           setLength(z, k); move(a[0], z[1],k);
34           write(z)
35         end
36       end;
37       i:=j; // Zeichenzähler für Gesamtzeichenkette aktualisieren
38       writeLn
39     end
40   end
41 end.
```

Listing 2.9: zeichenweise2.pas

Was also tun? - Nun, wir müssen uns wohl oder übel mit der Situation abfinden, die Zeichen einzeln durch „scharfes Hinsehen“ aus der Zeichenkette herauszumeißeln. Dazu ist die obige Idee, die Zeichenkette Byte-weise durchzugehen, durchaus nicht schlecht. Wir müssen lediglich schauen, ob wir ein komplettes Zeichen zusammenhaben oder hierzu noch weitere Bytes lesen müssen. Hierzu müssen wir uns die eingangs beschriebene UTF-8-Kodierung und deren Ausfbau erinnern.

Bei einem Multi-Byte-Zeichen in UTF-8-Kodierung ist im ersten Byte das Bit 7 gesetzt, also 1. Es wird im Gegensatz zum erweiterten ASCII-Code nicht für die eigentliche Kodierung des Zeichens verwendet, sondern fungiert als Flag. Ist das Bit 0, haben wir es mit einem „normalen“ Zeichen zu tun. Ist das Bit dagegen gesetzt, folgen danach bis zu drei weitere Bits, die gesetzt sein können. Jedes dieser Bit steht für ein Folge-Byte. Danach kommt eine 0 und der Rest des ersten Bytes gehört dann schon zur eigentlichen Zeichenkodierung. - Das war jetzt eine Wiederholung.

Um für alle Fälle gerüstet zu sein, spendieren wir uns ein Array aus vier Byte, in denen wir ein extrahiertes Zeichen aufnehmen können. Zur späteren Ausgabe kodieren wir es wieder in übliches UTF-8-Zeichen um, das genau so groß ist, wie es eben sein muss.

Schauen wir uns genauer an, was das Programm macht. `s` ist die Variable, in die die zu bearbeitende Zeichenkette eingelesen wird. Die Variablen `i` und `j` sind Laufvariablen. Dabei iterieren sie natürlich über jedes `Char`-Element der Zeichenkette. `i` steuert die äußere Schleife, `j` iteriert über die Zeichenkette für die aktuelle Ausgabezeile. Ist diese ausgegeben, wird `i` auf den Wert von `j` gesetzt. Das gewährleistet, dass `i` wirklich zeichenweise, im Sinne darstellbarer Zeichen, zählt.

Nehmen wir an, die Zeichenkette beginnt mit einem Multi-Byte-Sonderzeichen. `i` steht zu Beginn auf 1. Hat dieses Sonderzeichen drei Byte, so wird unsere Laufvariable `j` bis vier gezählt, zeigt also auf den Beginn des nächsten Zeichens der Zeichenfolge. `i` überspringt also die `Char`-Elemente (Bytes) einer Zeichenkette die zu einem Multi-Byte-Zeichen gehören. Die wurden, so es denn ein solches Zeichen war, was wir unter dem Messer hatten, in der innersten Schleife bereits abgearbeitet.

Die innere `while`-Schleife (Zeile 26 bis 31)ist denn auch das eigentliche Kernstück des Programms. Wenn die `if`-Bedingung (Zeile 17) einen Wert kleiner als 127 ($7f) feststellt, ist das Flag für ein Multi-Byte-Zeichen nicht gesetzt. Wir haben ein 7-Bit-ASCII-Zeichen gefunden. Andernfalls haben wir ein Multi-Byte-Zeichen in den Händen. Das heißt, neben dem aktuellen Byte, welches wir in den Händen halten, können bis zu drei weitere Byte folgen, die das Zeichen kodieren, mindestens aber eins. Wie viele Bytes folgen, sagen uns die auf das Flag-Bit folgenden drei Bits. Jedes gesetzte Bit steht für ein Folge-Byte. Das ergibt die Bedingung für unsere `while`-Schleife. Wir prüfen anhand von `k`. Dazu verschieben wir die im aktuellen Byte vorhandenen Bits temporär um `k` Bits nach links und maskieren sie 128 - im Code „bildlich“ durch die Dualzahl `%10000000` notiert. Die `and`-Verknüpfung bewirkt, dass das Ergebnis entweder 128 oder 0 ist. `ByteBool` gibt für jeden Wert ungleich Null den Wahrheitswert `true` zurück. Ist also ein Bit gesetzt, wird `true` geliefert und weiter geforscht. Die Versuche werden durch den Zähler `k` auf drei begrenzt. Sie brechen aber auch dann ab, wenn `ByteBool` den Wahrheitswert `false` geliefert hat. Innerhalb der Schleife werden dann die zu einem Zeichen gehörenden Bytes unserem Hilfsarray zugewiesen.

Ist die Schleife beendet, wird die Laufvariable `j` um die gezählten Bytes des analysierten Zeichen inkrementiert. Sie zählt also tatsächlich die Anzahl darstellbarer Zeichen.

Übrig bleibt, unser Hilfsarray in eine Zeichenkette zu wandeln. Um die Bit-Verschiebungen durchführen zu können, brauchten wir Bytes, daher eine Feld aus Bytes. Die Umwandlung in eine Zeichenkette ist aber unproblematisch. Wir benutzen einfach eine Speicherverschiebung, indem wir die Daten aus dem Array in einen Speicherbereich schieben, der für unseren Compiler als `UTF88tring` deklariert ist. Das ist die Variable `z`. Hierhin schieben wir so viele Bytes, wie wir bei unserer Analyse des Zeichens gefunden haben. Unser letztlich anzuzeigendes Zeichen ist also im Sinne von Pascal eigentlich eine Zeichenkette. Das verwundert jetzt aber wohl niemanden mehr.

Ein Probelauf unseres Programms könnte folgendermaßen aussehen:

```
$ ./zeichenweise
Bitte eine Zeichenkette: Zeichenkette mit öäü und 文
Z
Ze
Zei
Zeic
Zeich
Zeiche
Zeichen
Zeichenk
Zeichenke
Zeichenket
Zeichenkett
Zeichenkette
Zeichenkette
Zeichenkette m
Zeichenkette mi
Zeichenkette mit
Zeichenkette mit
Zeichenkette mit ö
Zeichenkette mit öä
Zeichenkette mit öäü
Zeichenkette mit öäü
Zeichenkette mit öäü u
Zeichenkette mit öäü un
Zeichenkette mit öäü und
Zeichenkette mit öäü und
Zeichenkette mit öäü und 文
$ _
```

Aus algorithmischer Sicht, könnte das Programm natürlich optimiert werden. Mein Anliegen war es lediglich, zu zeigen, wie wir einzelne Zeichen aus einer Zeichenkette extrahieren können. Das ist hoffentlich deutlich geworden.

Eine Anmerkung noch zum Schluss: Wir haben hier `while`-Schleifen verwendet, weil wir die genauen Anzahlen der Durchläufe nicht im Vorfeld haben wissen können. `for`-Schleifen wären, zumindest in Pascal, somit nicht geeignet gewesen.

2.1.5 readKey-Funktion

Das vorangegangene Problem führt uns direkt zu einem anderen. Nehmen wir an, wir wollen die Tastatur direkt auslesen, Weil wir auf die Eingabe eines bestimmten Zeichens durch den Anwender warten. Die Standard-Funktion von Free Pascal, `readKey`, versagt, weil sie uns auf Zeichen, mit einer Kodierung kleiner oder gleich 127 einschränkt, also auf Zeichen aus dem 7-Bit-ASCII-Code. Das gilt jedenfalls dann, wenn wir uns in einer Multi-Byte-Umgebung bewegen. Mit normalen 8-Bit-Zeichen kommt die Funktion klar. Wir wollen aber eventuell in einer Multi-Byte-Umgebung auch Sonderzeichen direkt von der Tastatur auslesen.

Nun, wir können quasi direkt auf dem aufbauen, was wir uns bis hierher erarbeitet haben. Dazu verwenden wir die Standard-Funktion `readKey` von Free Pascal und den obigen Code zur Analyse von UTF-8-Zeichen. Die Funktion gibt einen `UTF8String` mit bis zu vier Byte Länge zurück, die ein UTF-8-Zeichen eben haben kann.

Zur Demonstration soll eine Schleife solange durchlaufen werden, bis unser schönes japanisches Symbol 文 eingegeben wird.

```
1  program ReadKeyForUTF8;
2  uses
3    CWString,
4    CRT;  // für readKey-Funktion
5
6  function readKeyUTF8: String;
7  var
8    a: array[0..3] of Byte;  // Hilfsarray für Zeichenelemente/-bytes
9    e: String;               // Ergebnis
10   i: Byte;                 // Zähler für Zeichenelemente/-bytes
11 begin
12   a[0]:=Byte(readKey);          // benutze Standard-Funktion
13   if ord(a[0])<=$7f then begin  // normales Zeichen gefunden
14     setLength(e,1);
15     move(a[0],e[1],1)
16   end else begin                // UTF-8-Zeichen gefunden
17     i:=1;
18     while (i<=3) and ByteBool((a[0] shl i) and %10000000) do begin
19       a[i]:=Byte(readkey);      // nächstes Byte holen
20       inc(i);
21     end;
22     setLength(e,i);
23     move(a[0],e[1],i);
24   end;
25   readKeyUTF8:=String(UTF8decode(e));
26 end;
27
28 var
29   key, quitkey: String;
```

```
30 begin
31   quitkey:=UTF8encode(#$6587);
32   writeLn('Bitte ein Zeichen (Ende mit ', quitkey, '):');
33   repeat
34     write('-> '); key:=readKeyUTF8;
35     writeLn('Einabezeichen: ', key, '   Anzahl Byte: ',
36              Length(key));
37   until key=quitkey;
38 end.
```

Listing 2.10: readKey-Funktion mit UTF-8-Unterstützung

Eine Beispielsitzung könnte so aussehen:

```
$ ./readkeyutf8
Bitte ein Zeichen (Ende mit 文):
-> Einabezeichen: ä   Anzahl Byte: 2
-> Einabezeichen: #   Anzahl Byte: 1
-> Einabezeichen: k   Anzahl Byte: 1
-> Einabezeichen: k   Anzahl Byte: 1
-> Einabezeichen: k   Anzahl Byte: 1
-> Einabezeichen: q   Anzahl Byte: 1
-> Einabezeichen: 文   Anzahl Byte: 3
$ _
```

Es sieht zwar so aus, als würde `writeLn` ein zusätzliches Leerzeichen bei der Ausgabe produzieren, das ist jedoch eine Täuschung. Tatsächlich befinden sich immer nur drei Leerzeichen zwischen dem angezeigten Zeichen und dem Hinweis auf die Anzahl der Bytes. Die Verschiebung ist dem Zeichenfont geschuldet, der für die spezielle Glyphe zwei Zeichen Raum benötigt.

2.1.6 Ausgabebreite von Sonderzeichen

Das vorige Beispiel zeigt ein typisches Problem von verschiedenen Sonderzeichen. Die Glyphe 文 benötigt zwei Ausgabespalten. Auch die Formatierungsfunktionen haben mit Multi-Byte-Zeichen ihre Probleme. So beziehen sich die Formatierungsangaben von `write` und `writeLn` immer auf die Länge eines Zeichens respektive einer Zeichenkette in Bezug auf ihre jeweilige Byte-Anzahl. Um Texte mit Multi-Byte-Zeichen vernünftig formatieren zu können, benötigen wir also eine Routine, die uns die Ausgabebreite eines Zeichens beziehungsweise einer Zeichenkette ermittelt. Free Pascal bietet uns dazu standardmäßig nichts an. C-Programmierer haben es leichter. In der C-Bibliothek gibt es eine Funktion `wcwidth`, welche die Breite eines Ausgabezeichens ermittelt und zurückliefert. Unter Microsoft-Windows fehlt diese Funktion allerdings - jedenfalls habe ich sie bislang nicht gefunden. Unter macOS und Gnu/Linux ist sie auf jeden Fall vorhanden und wir können sie nutzen.

Im Folgenden stelle ich Ihnen eine prototypische Implementierung vor, die Ihnen zeigt, wie Sie C-Routinen in Ihre Free-Pascal-Programme einbauen und nutzen können.

```
(**
 * kbwidecharhelper.pas
 *
 * Unterstützungsfunktionen zum Umgang mit
 * Widechar und Widestring
 *
 * Bekannte Fehler:
 * ----------------
 * Die Funktionen zur Bestimmung der Ausgabebreite von Zeichen
 * nutzen die glibc, um Unicodezeichen zu identifizieren und
 * deren Breite zu bestimmen. Aktuell werden nicht alle Zeichen
 * von wcwidth(), der Kernroutine, erkannt. Für nicht erkannte
 * Zeichen wird eine Zeichenbreite von -1 geliefert.
 *
 * Karsten Brodmann, 2017-04-02 (Version 1)
 *)
unit KBWideCharHelper;

interface


(* Ausgabebreite eines (Multi-Byte-)Zeichens bestimmen
 *
 * ch: (Multi-Byte-)Zeichen(kette)
 *
 * Hinweis: enthält die Zeichenkette mehr als ein echtes
 *          Zeichen, werden diese zusätzlichen Zeichen
 *          nicht beachtet.
 *)
function getCharDisplayWidth(const ch: AnsiString): LongInt;

(* Ausgabebreite einer (Multi-Byte-)Zeichenkette bestimmen
 *
 * str: (Multi-Byte-)Zeichenkette
 *   n: Zeichenanzahl, für die die Breite bestimmt wird
 *)
function getStringDisplayWidth(const str: AnsiString;
                                     n  : LongInt): LongInt;

(*****************************************************************)
implementation

uses
  CLocale, CWString, BaseUnix;

// libc-Funktion: betsimmt die Ausgabebreite eines Zeichens
function wcwidth(__s: WideChar): longint;
    cdecl; external 'c' name 'wcwidth';


// Ausgabebreite eines Zeichens bestimmen
function getCharDisplayWidth(const ch: AnsiString): LongInt;
var p: PWideChar;
```

```
54 begin
55   // AnsiString konvertieren
56   p:=PWideChar(WideString(ch));
57   // libc-Funktion nutzen
58   getCharDisplayWidth := wcwidth(p^);
59 end;
60
61
62 // Ausgabebreite einer Zeichenkette bestimmen
63 function getStringDisplayWidth(const str:AnsiString;
64                                      n   : LongInt): LongInt;
65 var
66   len,              // Gesamtausgabebreite
67   chlen: Longint;   // Zeichenbreite
68   p    : PWideChar;
69 begin
70   // wenn n>Zeichenanzahl, dann n auf Zeichenanzahl begrenzen
71   if length(str) < n then n := length(str);
72   // AnsiString konvertieren
73   p:=PWideChar(WideString(str));
74   len:=0;
75   chlen:=0;
76   while (n>0)do begin       // für alle Zeichen ...
77     chlen := wcwidth(p^);   // Breite des aktuellen Zeichens
78     len := len+chlen;       // Gesamtbreite aktualisieren
79     dec(n);                 // ein Zeichen geschafft
80     inc(p)                  // nächstes Zeichen
81   end;
82   getStringDisplayWidth:=len;
83 end;
84
85 (*************************************************************************)
86 begin
87 end.
```

Listing 2.11: kbwidecharhelper.pas

Ich habe dazu eine Unit geschrieben, die mit Hilfe der C-Funktion `wcwidth` zwei Pascal-Funktionen implementiert. Die Funktion `getCharDisplayWidth` liefert die Ausgabebreite eines einzelnen Zeichens. Da ein Zeichen bis zu vier Byte haben kann, ist es als `AnsiString` an die Funktion zu übergeben. `getStringDisplayWidth` bestimmt die Ausgabebreite einer Zeichenkette. Neben der Zeichenkette ist der Funktion die Anzahl der Zeichenbytes zu übergeben, deren Ausgabebreite bestimmt werden soll. Dadurch kann die Breite einer Teilzeichenkette bestimmt werden. Es liegt in der Verantwortung des Programmierers, die Anzahl der Bytes so zu wählen, dass dadurch auch tatsächlich vollständige Zeichen analysiert werden.

Als Anwendungsbeispiel mag das folgende Programm dienen. Es zeigt die interne Speicherung des Datentyps `String` in Verbindung mit Sonderzeichen. Das Programm zeigt das Längenbyte sowie alle in einer Zeichenkette gespeicherten sichtbaren Zeichen und deren Kodierung. Zur Formatierung wird die soeben vorgestellt Unit genutzt.

```
(**
 * analysestring.pas
 *
 * Analyse des Standarddatentyps String
 *)
program AnalyseString;

{$MODE objfpc}        // wird fuer out-Parameter benoetigt
uses
  KBWideCharHelper;
type
  MBChar = String[4];

(* Fuer Multi-Byte-Zeichen werden die einzelnen Byte
 * gesammelt, bis das Zeichen vollständig ist. Danach
 * folgt dessen Ausgabe (mbchout). Normale Zeichen werden direkt
 * ausgegeben.
 *)
function isChar(ch: Char; out mbchout: MBChar): Boolean;
const
  len        : Byte = 0;        // Anzahl Byte
  isFirstByte: Boolean = true;  // Flagge: erstes Byte
  mbch       : MBChar = '';
begin
  isChar := false;
  // Erstes Byte ansehen und Anzahl Folgebyte ermitteln
  if isFirstByte then begin
    mbch := '';
    if Byte(ch) >= 240 then
      len := 4
    else if Byte(ch) >= 224 then
      len := 3
    else if Byte(ch) >= 192 then
      len := 2;
    isFirstByte := false;
  end;
  // Multi-Byte-Zeichen
  if Byte(ch) > 128 then begin
    mbch := mbch + ch;
    dec(len);
  // normales Zeichen
  end else begin
    mbch := ch;
  end;
  // Ausgabe + Re-Initialisierung
  if len = 0 then begin
    isChar := true;
    mbchout := mbch;
    isFirstByte := true;
  end
end;
const
  CH_PER_LINE = 7;      FORMATWIDTH = 9;
```

```
54 var
55   str  : String;
56   line : LongInt;
57   i, j,
58   len,
59   upper: Longint;
60   outch: MBChar;
61 begin
62   write('Bitte eine Zeichenkette: '); readLn(str);
63   len  := length(str);
64   i    := 0;
65   line := 0; upper:= 0;
66
67   // iteriere ueber alle Elemente der Zeichenkette
68   while i < len do begin
69
70     // bis zu welchem Index soll die Zeichenkette in der
71     // aktuellen Zeilel ausgegeben werden?
72     if i + CH_PER_LINE < len then upper := i + CH_PER_LINE
73                              else upper := len;
74
75     write(line: 5, ' ':2);
76     // Zeichen-Zeile ausgeben
77     for j := i to upper do           // Zeichen ausgeben
78       // 1. Element, 1. Zeile ist Längenbyte
79       if j = 0 then                  // Laengenbyte
80         write('Länge':10)            // ueberspringen
81       else
82         // pruefe, ob darstellbares Zeichen
83         if isChar(str[j], outch) then
84           write(outch:
85             FORMATWIDTH+length(outch)-getCharDisplayWidth(outch))
86         else
87           write(' ':FORMATWIDTH);
88     writeLn;
89     //Dezimalcode-Zeile ausgeben
90     write(line: 5, ' ':2);
91     for j := i to upper do
92       write(Byte(str[j]):FORMATWIDTH);
93     writeLn;
94     // Dualcode-Zeile ausgeben
95     write(line: 5, ' ':2);
96     for j := i to upper do           // Bin.-Code
97       write(BinStr(Byte(str[j]),8):FORMATWIDTH);
98     writeLn;
99
100    inc(i, CH_PER_LINE+1);                  // Index weiterzahlen
101    inc(line, CH_PER_LINE)
102  end;
103 end.
```

Listing 2.12: analysestring.pas

Eine Beispielsitzung könnte so ausschauen:

```
Bitte eine Zeichenkette: Ich bin eine Zeichenkette mit Umlauten äöü und 北
 0      Länge         I         c         h                   b         i         n
 0         53        73        99       104        32        98       105       110
 0   00110101  01001001  01100011  01101000  00100000  01100010  01101001  01101110
 7                   e         i         n         e                   Z         e
 7         32       101       105       110       101        32        90       101
 7   00100000  01100101  01101001  01101110  01100101  00100000  01011010  01100101
14          i         c         h         e         n         k         e         t
14        105        99       104       101       110       107       101       116
14   01101001  01100011  01101000  01100101  01101110  01101011  01100101  01110100
21          t         e                   m         i         t                   U
21        116       101        32       109       105       116        32        85
21   01110100  01100101  00100000  01101101  01101001  01110100  00100000  01010101
28          m         l         a         u         t         e         n
28        109       108        97       117       116       101       110        32
28   01101101  01101100  01100001  01110101  01110100  01100101  01101110  00100000
35                    ä                   ö                   ü                   u
35        195       164       195       182       195       188        32       117
35   11000011  10100100  11000011  10110110  11000011  10111100  00100000  01110101
42          n         d                                       北
42        110       100        32       230       137       149
42   01101110  01100100  00100000  11100110  10001001  10010101
```

2.2 Zeichenketten sortieren

Eine Sortierung erfolgt mit dem Computer grundsätzlich so, dass ein Zeichen mit der größeren Codenummer als das größere Zeichen angesehen wird. Das bedeutet, dass wir es eigentlich mit einer numerischen Sortierung der Zeichencodes zu tun haben. Nun ist das aber nicht immer das, was wir von einer Zeichensortierung erwarten. Das war in vergangenen Zeiten, Zeiten mit lediglich 8-Bit-Zeichensätzen, aber auch nicht anders. Betrachten wir kurz ein Beispiel.

Wir wollen ein Array mit Zeichenketten sortieren. Groß- und Kleinschreibung soll ignoriert werden, so dass alles, was beispielsweise mit `a` oder `A` beginnt zusammenhängend sortiert wird. Umlaute sollen gemäß der Telefonbuchsortierung als zusammengesetzt betrachtet werden. `ä` wird also als `ae` gewertet und sortiert (siehe DIN 5007-1). Das `ß` wird wie `ss` behandelt. Die Wörterbuchsortierung sieht anders aus. Dort werden die Umlaute anders behandelt: `Ä` wird `A` und so weiter (siehe DIN 5007-2).

Wir wollen gemäß DIN 5007-1 sortieren. Als Sortieralgorithmus soll ein „billiger“ Bubblesort herhalten. Auf die Effizienz des Algorithmus kommt es hier nicht an. Entscheidend ist die Vergleichsfunktion, die ermittelt, ob eine Zeichenkette `a` kleiner, gleich oder größer als eine Zeichenkette `b` ist. In Free Pascal sind die Vergleichsoperatoren zwar für Zeichenkettenvergleiche überladen, orientieren sich aber an den Zeichencodes, was uns nichts bringt. Wir brauchen also eine eigene Vergleichsfunktion. Das folgende Programm demonstriert, wie wir das umsetzen können.

```
1 program Bubble1;
2 uses
3   CWString, StrUtils, SysUtils;
```

```
4
5  function compare(a, b: String): SmallInt;
6    { Wenn a<b dann ==> -1
7           a=b dann ==>  0
8           a>b dann ==>  1 (OHNE Groß-/Kleinschreibung)
9    }
10 begin
11   a:=AnsiLowerCase(a); b:=AnsiLowerCase(b);
12   a:=replaceStr(a,'ä', 'ae'); b:=replaceStr(b,'ä', 'ae');
13   a:=replaceStr(a,'ö', 'oe'); b:=replaceStr(b,'ö', 'oe');
14   a:=replaceStr(a,'ü', 'ue'); b:=replaceStr(b,'ü', 'ue');
15   a:=replaceStr(a,'ß', 'ss'); b:=replaceStr(b,'ß', 'ss');
16   if a<b then
17     compare:=-1
18   else if a=b then
19     compare:=0
20   else
21     compare:=1
22 end;
23
24 procedure sortBubble(var a: array of String);
25 var
26   i, j: SmallInt;
27   temp: String;
28 begin
29   for i:=Low(a) to High(a)-1 do
30     for j:=Low(a)+1 to High(a) do
31       if compare(a[j-1],a[j])=1 then begin
32         temp:=a[j]; a[j]:=a[j-1]; a[j-1]:=temp
33       end
34 end;
35
36 var
37   s: array[1..20] of String =
38        ( 'Zacharias', 'Saubermann', 'Sauerbruch', 'Meyer', 'Röllinger',
39          'Roller', 'Gunter', 'Brodmann', 'Öttinger', 'Maier', 'Meier',
40          'Mayer', 'Meyer', 'sauer', 'Krieger', 'Kröger', 'säuerlich',
41          'Kroog', 'Günter', 'Güngör' );
42   i: SmallInt;
43 begin
44   for i:=Low(s) to High(s) do
45     write(s[i], ' ');
46   writeLn; writeLn;
47   sortBubble(s);
48   for i:=Low(s) to High(s) do
49     write(s[i], ' ');
50   writeLn
51 end.
```

Listing 2.13: bubble1.pas

Für eine Wörterbuchsortierung hätten wir

```
a:=AnsiLowerCase(a); b:=AnsiLowerCase(b);
a:=replaceStr(a,'ä', 'a');  b:=replaceStr(b,'ä', 'a');
a:=replaceStr(a,'ö', 'o');  b:=replaceStr(b,'ö', 'o');
a:=replaceStr(a,'ü', 'u');  b:=replaceStr(b,'ü', 'u');
a:=replaceStr(a,'ß', 'ss'); b:=replaceStr(b,'ß', 'ss');
```

in unserer Vergleichsfunktion notieren müssen.

Mit Ausnahme dessen, dass wir in der Vergleichsfunktion `compare` ein wenig Rücksicht auf den Zeichensatz nehmen müssen, um die Zeichenketten hinsichtlich der Groß-/Kleinschreibung zu transformieren, passiert nichts, was wir nicht auch in vergangenen Tagen hätten tun müssen. UTF-8 stellt uns also vor keine neuen Herausforderungen, was die Sortierung angeht.

Da hier keine neuen Erkenntnisse erworben wurden, hätte ich mir diesen Abschnitt möglicherweise schenken können. Allerdings habe ich in verschiedenen Foren schon diesbezügliche Fragen gelesen, weshalb ich es dann doch für nötig erachtet habe, das Thema Sortierung kurz aufzugreifen. Grundsätzlich sind im Unicode und somit der UTF-8-Kodierung alle Zeichen in einer vernünftigen Reihenfolge angeordnet. Allerdings, und das ist tatsächlich neu, sind nationale Besonderheiten für alle gleich geregelt, so dass sich der Begriff der nationalen Besonderheit relativiert. Da für alle die Codepunkte einzelner Zeichen verbindlich geregelt sind, gibt es keine Individualismen mehr hinsichtlich ihrer Anordnung und damit auch Sortierung. Das mag man als Nachteil empfinden. Auf der anderen Seite ist so aber auch gewährleistet, dass alle Zeichen so angezeigt werden, wie es sich der Autor eines Textes gedacht hat, als er seinen Text verfasst hat. Das ist eine große Verbesserung gegenüber Früher.

3 Fazit

Es gibt sicherlich noch viele Fälle, die mit Zeichensatzproblemen zu tun haben und hier nicht behandelt wurden. Ich denke aber, sie alle lassen sich auf die hier behandelten Sachverhalte zurückführen und entsprechend lösen. Mein Anliegen war es auch nicht, für alles und jedes eine Lösung zu präsentieren. Mir war es wichtig, und das habe ich hoffentlich vermitteln können, wie man mögliche Probleme analysiert und in den Griff bekommt. Die vorgestellten Lösungen erheben auch nicht den Anspruch, den Stein der Weisen gefunden zu haben, sie sollen lediglich als Anregung dienen, eigene Lösungen zu finden, die der jeweiligen Situation angepasst und angemessen, vor allem aber erfolgreich, sind.

Für die Praxis können wir Folgendes festhalten:

1. In Free Pascal sollte die Unit `CWString` immer eingebunden werden, wenn mit Multi-Byte-Zeichenketten gearbeitet wird und es darauf ankommt, Zeichen als solche erkennen zu müssen. Für ander Programmiersprachen gelten analoge Vorkehrungen, wie beispielsweise der Aufruf von `setlocal` in C.

2. Es gibt in den mir bekannten Programmiersprachen noch nicht für alle Situationen passende Standard-Routinen. Es ist jedoch kein Problem, selbst eine passende Routine zu bauen. Multi-Byte-Zeichen in einer Zeichenkette zu identifizieren ist nicht schwierig, wenn man deren Kodierungssystematik kennt.

3. Unicode und insbesondere UTF-8 stellen uns vor nicht allzu große Herausforderungen in der Programmierung. Die Komplexität der Zeichenverarbeitung ist nur vordergründig gestiegen, weil die Zeichenkodierung komplexer geworden ist. Auf der anderen Seite haben wir einen international übergreifenden Standard, der jedem Zeichen eine verbindliche Kodierung zuweist. Damit wird der Austausch von Texten ganz wesentlich vereinfacht. Dieser Vorteil überwiegt die Nachteile, so meine ich jedenfalls, bei weitem.

Ich hoffe, ich habe mit diesem kurzen Beitrag den Schrecken vor den „vielen Zeichen“ ein wenig lindern können. Für Programmieranfänger ist die Situation tatsächlich ein wenig schwieriger geworden. Gerade im Bereich der Internationalisierung von Programmen ist das Leben aber auch extrem vereinfacht worden. Jeder, der es mit der Programmierung ernst meint, wird irgendwann mit der Internationalisierung (Mehrsprachigkeit) von Programmen in Berührung kommen. Dann ist es doch beruhigend, sich keine Sorgen mehr um irgendwelche Codepages und darin möglicherweise nicht kodierte Zeichen machen zu müssen. Das Einzige was noch fehlt, ist, dass alle Betriebssysteme durchgehend UTF-8 verwenden. Insbesondere Microsoft Windows ist da leider noch ein Stück hinter der aktuellen Entwicklung hinterher.

Karsten Brodmann