3.5 C-Datentypen/Unicode-Zeichen und Multibyte-Sequenzen

3.5.1 Die Grenzen von ASCII

Viele Benutzer kennen das Problem, dass Umlaute und andere Sonderzeichen in Dokumenten und bei der Arbeit mit Programmen nicht richtig dargestellt werden. Zur Zeit der Entstehung vieler Betriebssysteme und Programmiersprachen gab es auf jedem Computer nur einen einzigen Zeichensatz, und der war meistens 7 oder 8 Bit breit und hieß ASCII (manche benutzten auch EBCDIC). Es wurden nur Buchstaben verwendet, die im englischen Alphabet vorhanden sind. Für Benutzer, die mehr brauchten, gab es Sonderlösungen. So gab es für westeuropäische Nutzer die Möglichkeit, den Zeichensatz ISO-8859-1 (auch ISO-Latin) zu benutzen (ähnlich ANSI).

Heute dagegen sind unzählige Computer miteinander verbunden, und ihre Benutzer erwarten selbstverständlich, dass sie jederzeit in jedem Dokument beliebige Zeichen jeder Sprache verwenden können. Daher hat man in den neunziger Jahren einen universellen Zeichensatz entwickelt, der pro Zeichen 32 Bit vorsieht und damit mehr als 4 Milliarden verschiedene Zeichen ermöglicht: UCS-4.

Leider passierte während der Entwicklung etwas: Mehrere Mitglieder der Kommission, die UCS entwickelten, verloren die Geduld und bastelten aus den bisher zusammengestellten Zeichen eine abgespeckte Version, die pro Zeichen nur 16 Bit vorsah und daher maximal 65536 verschiedene Zeichen abdecken konnte. Diese wurde dann UNICODE (bzw. UCS-2) genannt. Mehrere Firmen sprangen auf den UNICODE-Zug auf (Sun mit Java, Microsoft mit verschiedenen Produkten).

Mit 65536 Zeichen konnte man allerdings immer noch nicht alle Details aller wichtigen Sprachen abdecken, besonders bei CJK (chinesisch, japanisch und koreanisch) gab es dort Lücken. Es stellte sich heraus, dass man mehr Möglichkeiten brauchte. Der erste Schritt war leicht getan: UCS-4 wurde nach UNICODE eingemeindet und hieß nun auch UNICODE.

Der zweite nötige Schritt war schwerwiegender: Man musste im 16-Bit-Zeichensatz von UCS-2 eine Erweiterung einbauen, die die restlichen Zeichen von UCS-4 per Trick ermöglicht. Man hat dazu im UCS-2 zwei Bereiche mit je 1024 Codes ausgespart. Wenn nun im Text ein Zeichen mit einer Nummer größer als 65536 gebraucht wird, wird zuerst ein 16-Bit-Zeichen aus dem ersten Bereich und danach ein 16-Bit-Zeichen aus dem ersten Bereich geschrieben. Man spricht von *surrogate pairs*. Diese Codierung hat den Namen UTF-16. Programmierer müssen bei dieser Variante des UNICODE beachten, dass es ca. 64000 Zeichen gibt, die mit einem 16-Bit-Zeichen codiert sind und bis zu einer Million Zeichen, die aus zwei 16-Bit-Zeichen bestehen.

Im WWW hat sich eine andere Variante durchgesetzt, mit der man UCS-4 bzw. UNICODE umsetzen kann, ohne allzuviel umzustellen: Bei der Codierung UTF-8 werden alle 128 ASCII-Zeichen wie bisher geschrieben. Für alle anderen Zeichen werden Folgen aus zwei bis vier Bytes (möglich wären sieben) verwendet. Diese Variante hat den Vorteil, dass keine Null-Bytes auftauchen, die uns in C das Leben schwer machen könnten. Sie hat aber wie UTF-16 den Nachteil, dass es Zeichen verschiedener Länge gibt.

Die für Programmierer einfachste Lösung, nämlich jedes UNICODE-Zeichen 1:1 mit 4 Byte darzustellen, hat einen neuen Namen bekommen: Sie heißt jetzt (statt nur UCS-4) UTF-32.

3.5.2 ASCII-Zeichen in C

Der Umgang mit Strings und Einzelzeichen in C ist relativ einfach durch den **Datentyp** `char`: Jedes Zeichen entspricht einem Byte.

```
1  char x;    // 1 Byte bzw. CHAR_BIT bits
2  x=68;      // eine Zahl (0 bis 255 bzw. -128 bis 127)
```

Die wörtlichen (=literalen) **Zeichenkonstanten** in Hochkomma entsprechen einer (`int`-) Zahl zwischen 0 und 127 und können direkt zugewiesen werden:

```
1  x='D';     // eine literale Zeichenkonstante
```

Mit Hilfe von **Ersatzdarstellungen** können auch solche Werte benutzt werden, die im Quelltext sonst nicht darstellbar sind:

```
x='\n';    // \n bedeutet Zeilenende
x='\104'; // ein bis maximal drei Oktalziffern, max. Wert 255
x='\x44'; // ein bis N Hexadezimalziffern, max. Wert 255
```

Die Ersatzdarstellungen sind nur dann wichtig, wenn man wörtliche (=literale) **Stringkonstanten** schreiben will:

```
char s[80]="Erste Zeile\nZweite Zeile.";
```

3.5.3 Nicht-ASCII-Zeichen in C

Wenn es nur um Ausgabe geht, kann man sich in C mit UTF-8 abhelfen:

```
char t[80]="Hauptstadt von NRW: D\xFCsseldorf";      // ISO-8859-1
char s[80]="Hauptstadt von NRW: D\xC3\xBCsseldorf"; // UTF-8
```

Wie man sieht, ist der Aufwand bei UTF-8 nicht größer als bei ISO-8859-1.

Schwieriger ist es, wenn man einzelne Zeichen abfragen oder verändern will:

```
int ch;
ch=getchar();
if(ch=='\xC3') // geht nicht, weil auch andere Zeichen
               // mit C3 beginnen koennten
{...}
```

Auch viele der eingebauten C-Funktionen spielen bei Sonderzeichen nicht mit:

```
char s[80]="Hauptstadt von NRW: D\xC3\xBCsseldorf";
for(lauf=0; s[lauf]!='\0'; ++lauf)
    s[lauf]=toupper(s[lauf]); // klappt nicht bei lauf==1
```

Hier sieht man gleich zwei Schwierigkeiten: Erstens kann jedes Zeichen durch ein bis vier Bytes kodiert sein. Zweitens kann die Frage, ob ein Zeichen Klein- oder Großbuchstabe oder sonst etwas ist, unter anderem auch von der gewählten Lokalisierung abhängen. Aus diesem Grund muss man z. B. vor Benutzung der C-Funktion `strcoll()` (Vergleich zweier Zeichenketten zum Zwecke der Sortierung) mit `setlocale()` die richtige Lokalisierung einstellen.

3.5.4 Lösung I: Wide Characters in C

Eine Lösung für diese Schwierigkeit sind sogenannte Wide Characters, die schon in C89 vorgesehen sind und für die in C95 und C99 weitere Sprachelemente hinzugefügt wurden. Der Datentyp heißt `wchar_t`:

```
#include <wchar.h>
wchar_t w;
w=920; // Theta
```

Seine Größe ist so festgelegt, dass sie größer ist als die von `char`.

Weiterhin gibt es für diesen Datentyp erweiterte wörtliche Zeichenkonstanten:

```
w=L'a';     // Buchstabe a, dargestellt mit 32 (oder 16) Bit
w=L'\xFC'; // Buchstabe "u, Nr. 253, dargest.m. 32 (o.16) Bit
w=L'\463'; // Niederl. ij, UNICODE Nr. 307
w=L'\x133';// Niederl. ij, UNICODE Nr. 307
```

Die Variante aus Zeile 4 hat den Vorteil, dass beliebig viele Hexadezimalziffern angegeben werden können:

```
1    w=L'\x20AC';   // Euro-Zeichen,        UNICODE Nr.    8364
2    w=L'\x1D11E'; // Violin-Schluessel,  UNICODE Nr.  119070
```

Genauso gibt es auch erweiterte wörtliche String-Konstanten:

```
1    wchar_t ws[]=L"D\xFCsseldorf:␣\x1D11E␣statt␣\x20AC";
```

Die Funktionen `printf()` und `scanf()` bekommen besondere Platzhalter für Wide Characters:

```
1  #include <locale.h>
2  #include <wchar.h>
3  #include <stdio.h>
4  int main(void)
5  {
6      setlocale(LC_ALL,"");
7      wchar_t wc=L'\x3C6'; // griech. Kleinbuchstabe phi
8      wchar_t ws[]=L"D\xFCsseldorf:␣\x1D11E␣statt␣\x20AC";
9      printf("%lc\n", wc);
10     printf("%ls\n", ws);
11     return 0;
12 }
```

Zeile 6 Ohne `setlocale()` keine Ausgabe! Es ist sinnvoll, `setlocale()` vor der ersten Ausgabe auszuführen[1].

Zeile 9 Platzhalter `%lc` für ein `wchar_t`-Zeichen

Zeile 10 Platzhalter `%ls` für einen String aus `wchar_t`-Zeichen

3.5.5 Zeichen- und String-Funktionen für `wchar_t`

Außerdem sind Zeichen- und String-Funktionen für `wchar_t` vorhanden. Die Namen der Funktionen richten sich meistens nach den Namen der entsprechenden Funktionen für `char`-Daten:

- `iswalpha()` aus `<wctype.h>` statt `isalpha()` aus `<ctype.h>`
- `wcstol()` aus `<wchar.h>` statt `atol()` aus `<stdlib.h>`
- `wcscpy()` aus `<wchar.h>` statt `strcpy()` aus `<string.h>`

Typische Include-Dateien sind `<wctype.h>`, `<wchar.h>` und `<inttypes.h>`. Hier ein Beispiel:

```
1    char x[80];
2    setlocale(LC_ALL, "");
3    wcscpy(x, L"Hallo");
```

Es sind auch einige neue Funktionen hinzugekommen wie die Funktion `wccwidth()`, die die Anzahl der Spalten auf dem Bildschirm (oder Drucker) zurückgibt, die der Text belegen würde. Eine solche Funktion wäre bei ASCII unnötig.

[1] Bei der aktuellen Ausgabe der Standard-Bibliothek für GCC scheint es so zu sein, dass man `setlocale()` mit `LC_TYPE` oder `LC_ALL` nur ein einziges Mal in einem Prozess ausführen kann.

3.5.6 Konvertierung nach `wchar_t` und zurück

Es gibt mittlerweile eine Reihe von Funktionen für die Konvertierung von `char` nach `wchar_t` und zurück. Dabei wird bei `char` immer die Möglichkeit von UTF-8 mit beachtet. Deshalb wird bei den Funktionsnamen statt `char` immer `mbs` angegeben (*Multibyte String*). Im folgenden Beispiel sollen mehrere Zeichen von `char` nach `wchar_t` konvertiert werden:

```
#include <wchar.h>
#include <locale.h>
#include <stdlib.h> // MB_CUR_MAX
#include <stdio.h>
int main(void)
{
    setlocale(LC_ALL,"");
    wchar_t x; int erg;
    char s[]=u8"G\xC3\xBCtersloh";
    char *p=s;
    erg=mbtowc(&x, p, MB_CUR_MAX);
    while(erg>0)
    {
        printf("%lc", x);
        p+=erg;
        erg=mbtowc(&x, p, MB_CUR_MAX);
    }
    return 0;
}
```

Tabelle 1 listet Funktionen auf, die für das Konvertieren von Einzelzeichen benutzt werden. Funk-

Funktion	seit	Bemerkung
wctomb()	C89	konvertiert ein Langzeichen nach ASCII oder MB
wcrtomb()	C99	reentrant
wctomb_s()	C11	sichere Version, nicht reentrant
wcrtomb_s()	C11	sichere Version, reentrant
mbtowc()	C89	konvertiert ein ASCII- oder MB-Zeichen nach wchar_t
mbrtowc()	C99	reentrant
mblen()	C89	Länge eines Zeichen
mbrlen()	C99	reentrant

Tabelle 1: Funktionen für breite Einzelzeichen

tionen mit r im Namen sind wiedereintrittsfähige (=reentrante) Versionen, Funktionen mit angehängtem s haben Ergänzungen für die Sicherheit. Man fragt sich, warum eine Funktion wie `mblen()`, die nur die Länge eines Multibyte-Zeichens ausgibt oder eine Funktion wie `wctomb()` extra eine wiedereintrittsfähige Zusatzversion bekommen haben. Das hat folgenden Grund: Bei anderen Lokalisierungen als UTF-8 wird ein so genannter *shift state* gespeichert. Dort kann es nämlich so sein, dass die Bedeutung nachfolgender Multibyte-Sequenzen von der Dekodierung des bisherigen Multibyte-Strings abhängt. Betroffen sind wohl die Kodierungen ISO-2022 (JIS) und UTF-7.

Tabelle 2 listet Funktionen auf, die für das Konvertieren von Strings benutzt werden.

Außerdem gibt es noch zwei Zahlenwerte, die für den Speicherbedarf bei der Konvertierung von Strings benutzt werden können:

- `MB_LEN_MAX` gibt es seit C99, diese Konstante gibt die maximal mögliche Anzahl von Bytes pro Multibyte-Zeichen an. Sie wird in `<limits.h>` bereitgestellt.

Funktion	seit	Bemerkung
mbstowcs()	C95?	konvertiert MB-String nach WC-String
mbsrtowcs()	C99	reentrant
mbsrtowcs_s()	C11	sicher+reentrant
wcstombs()	C95?	konvertiert WC-String nach MB-String
wcsrtombs()	C99	reentrant
wcsrtombs_s()	C11	sicher+reentrant

Tabelle 2: Funktionen für breite Strings

- `MB_CUR_MAX` gibt es seit C11, es handelt sich um eine Makro-Variable(!), die die Anzahl Bytes pro Multibyte-Zeichen in der momentanen Locale-Einstellung ermittelt. Sie wird in `<stdlib.h>` bereitgestellt.

3.5.7 Spezielle Ein- und Ausgabe-Funktionen für `wchar_t`

Zu den Funktionen `getchar()` und `putchar()` gibt es nun (seit C99) außerdem die Funktionen `getwchar()` und `putwchar()`. Zu `printf()` und `scanf()` gibt es nun außerdem noch `wprintf()` und `wscanf()`. Zum Dateiende EOF gibt es jetzt noch WEOF.

3.5.7.1 Datenstrombreite Die Ausgabe mit diesen neuen Funktionen hat leider eine dunkle (=umständliche) Seite: Sie können nur aufgerufen werden, wenn der Ein- bzw. Ausgabestrom entsprechend eingestellt ist.

Es ist so, dass man einmal im Prozess einen Datenstrom (z. B. die Standardausgabe) auf `char` oder auf `wchar_t` einstellen kann, entweder implizit durch Ein-/Ausgabebefehle oder explizit durch die Funktion `fwide()`.

Wenn man nichts weiter macht, wird der Ausgabestrom durch den ersten Ausgabebefehl implizit eingestellt: Ist der erste Befehl ein `putchar()` oder ein `printf()`, so wird er als `char`-Datenstrom eingestellt. So war es in allen bisher vorgestellten Programmen.

Ist der erste Befehl dagegen ein `putwchar()` oder `wprintf()` (das ist die `wchar_t`-Version von `printf()`), so wirde der Ausgabestrom als `wchar_t`-Datenstrom eingestellt.

Ruft man anschließend eine „falsche“ Funktion auf, also eine Funktion mit der falschen Breite, so bekommt man eine falsche (meistens gar keine) Ausgabe.

3.5.7.2 Ein- und Ausgabe von Einzelzeichen Falls der Ausgabestrom für `wchar_t` eingerichtet ist, kann man mit `getchar()` und `putchar()` so Zeichen ein- und ausgeben:

```
1  wint_t a; // wint_t ist breiter als wchar_t, wegen WEOF
2  setlocale(LC_ALL,"");
3  a=getwchar();
4  while(a!=WEOF)
5  {
6      putwchar(a);
7      a=getwchar();
8  }
```

3.5.7.3 Ausgabe mit `wprintf()` statt `printf()` Es gibt (wie schon beschrieben) für `wchar_t` neue Platzhalter:

- `"%lc"`: Ein `wchar_t`
- `"%ls"`: String mit `wchar_t`

Falls der Ausgabestrom für `char` eingerichtet ist, kann man mit `printf()` wie gewohnt ausgeben:

```
    wchar_t a;
    wchar_t x[23];
    setlocale(LC_ALL,"");
    printf("%lc", a); /* Konversion nach MBS und Ausgabe*/
    printf("%ls", x); /* -- " --     mit wcrtomb()       */
    printf("%10.22ls", x); /* --> max. 22 Bytes werden  */
                           /* ausgegeben (z.B. 11 Umlaute)  */
                           /* 10 ist minimale Ausgabebreite */
```

Die Funktion `wprintf()` hat einen Formatstring vom Typ `wchar_t`. Die weiteren Parameter dürfen von beliebigem Typ sein. Falls der Ausgabestrom für `wchar_t` eingerichtet ist, kann man Text mit `wprintf()` ausgeben:

```
setlocale(LC_ALL,"");
wprintf(L"%lc", a);       /* direkte Ausgabe von wint_t                 */
wprintf(L"%ls", x);       /* direkte Ausgabe von const *wchar_t         */
wprintf(L"%c", 'B');      /* Konversion in wchar_t und Ausg., btowc()   */
wprintf(L"%s", "Hallo"); /* -- " -- mbrtowc()                           */
wprintf(L"%10.22ls", x); /* --> max. 22 wchar_t-Zeichen                 */
                          /* --> 10 ist minimale Ausgabebreite */
```

3.5.7.4 Einrichtung der Datenstrombreite Die erste Ausgabe auf stdout bestimmt die Einrichtung (`wchar_t` oder `Byte`). Sie kann aber auch vor der ersten Ausgabe einmalig explizit festgelegt werden durch einen Aufruf von `fwide()`:

```
fwide(stdout,  1); /* stdout ist ab jetzt fuer wchar_t eingerichtet */
fwide(stdout, -1); /* stdout ist ab jetzt fuer Byte eingerichtet */
```

Mit `fwide()` kann man die Einrichtung auch herausfinden:

```
x=fwide(stdout, 0); /* Abfrage, ob stdout f.wchar_t eingerichtet ist */
                    /* 1=ja, -1=nein, 0=unbestimmt */
```

Ist die Einrichtung des Datenstroms einmal vorgenommen worden, kann sie auch mit `fwide()` nicht mehr geändert werden[2].

3.5.7.5 Eingabe mit scanf() und wscanf() Bei einem für Byte eingerichtetem Eingabestrom nimmt man wie bisher `scanf()`:

```
scanf("%lc", &a);  /* Eingabe     Byte              Ausgabe wchar_t  */
scanf("%ls", x);   /* Eingabe ist Byte-orientiert, Ausgabe *wchar_t */
```

Bei einem für `wchar_t` eingerichtetem Eingabestrom nimmt man `wscanf()`:

```
wscanf("%lc", &a);
wscanf("%ls", x);
```

[2] Man kann allerdings den Datenstrom mit `freopen()` schließen und sofort wieder öffnen; dann hat man wieder die Möglichkeit einer neuen Einrichtung. Das funktioniert aber nur bei Datei-Ein- und Ausgabe, nicht bei der Konsole, weil `freopen()` einen Dateinamen als ersten Parameter braucht

3.5.8 Problem: Portabilität bei `wchar_t`

Leider gibt es an dieser Stelle ein Portabilitäts-Problem:

- Der MS-C-Compiler nimmt für `wchar_t` den 16 Bit breiten Datentyp `uint16_t`.
- Viele andere C-Compiler nehmen für `wchar_t` den 16 Bit breiten Datentyp `uint32_t`.

Die Lösung besteht darin, für diese beiden Varianten zwei Versionen zu programmieren und zwischen diesen dann mit Hilfe bedingter Compilierung umzuschalten (siehe C6.9).

3.5.9 Lösung II: UNICODE-Zeichen in C

Das Nebeneinander der 16-Bit- und 32-Bit-Varianten von `wchar_t` führt zu umständlichen Programmen. Deshalb hat man mit dem Standard C11 zwei neue Datentypen eingeführt:

- `char16_t` ist 16 Bit breit und nimmt entweder ein komplettes UTF-16-Zeichen auf oder einen Teil eines Surrogate Pairs.
- `char32_t` ist 32 Bit breit und nimmt immer ein komplettes UTF-32-Zeichen auf.

Der Datentyp `unsigned char` bleibt weiterhin geeignet zur Aufnahme von UTF-8-Inhalten.
Entsprechend gibt es neue literale Zeichenkonstanten:

- `u8'x'` meint eine 8-Bit-Konstante zwischen 0 und 255 (`unsigned char`)
- `u'x'` meint eine 16-Bit-Konstante zwischen 0 und 65535 (`char16_t`)
- `U'x'` meint ein 32-Bit-Konstante (`char32_t`)

Interessanter sind die entsprechenden Stringkonstanten:

- `u8"Lala:\x1D11E"` meint einen UTF-8-String `unsigned char[]`
- `u"Lala:\x1D11E"` meint einen UTF-16-String mit Surrogate Pairs `unsigned char16_t[]`
- `U"Lala:\x1D11E"` meint einen UTF-32-String `unsigned char32_t[]`

Die Benutzung der Ersatzdarstellung (Escape-Sequenz) `\x` für ein einzelnes Zeichen ist so in Ordnung: Sobald die erste Nicht-Hexadezimal-Ziffer erscheint, ist die Hexadezimalzahl zu Ende. Damit kann man also beliebig große Codenummern (so genannte Codepunkte) in UNICODE darstellen. Aber es ist unpraktisch, wenn hinter der Codenummer weitere Ziffern oder Buchstaben A bis F folgen. Deshalb gibt es seit C11 zwei neue Ersatzdarstellungen, hier am Beispiel des Euro-Zeichens:

- `\u20AC` – genau vier Hexadezimalziffern
- `\U000020AC` – genau acht Hexadezimalziffern

Auch für die beiden UNICODE-Datentypen gibt es eigene Funktionen. Man kann sie in Multibyte-Strings umwandeln und wieder zurück. Tabelle 3 zeigt eine Übersicht. Alle anderen Funktionen (Ein-/Ausgabe, Umwandlung usw.) sind leider nicht in C11 festgeschrieben.

Funktion	seit	Bemerkung
mbrtoc32()	C11	konvertiert ein Zeichen
mbrtoc16()	C11	konvertiert ein Zeichen
c32rtomb()	C11	konvertiert ein Zeichen
c16rtomb()	C11	konvertiert ein Zeichen

Tabelle 3: Funktionen für breite Einzelzeichen

3.5.10 Fazit

Aufgrund der weiten Verbreitung von C auf den verschiedensten Systemen ist der Umgang mit erweiterten Zeichensätzen immer noch umständlich. Wenn Portabilität kein Problem ist, kann man den Datentyp `wchar_t` mit allen seinen Funktionen benutzen. Andernfalls muss man die neuen Datentypen aus C11 verwenden.