6.10 Extras/Datum und Zeit in C

6.10.1 Echtzeit

Viele Rechnersysteme besitzen eine eingebaute Echtzeituhr. Diese Uhr läuft auch dann, wenn die Stromversorgung ausgeschaltet ist. Beim Start holt sich das Betriebssystem das Datum und die Uhrzeit aus der Echtzeituhr.

Falls es Zugriff auf ein Netzwerk hat, kann es sich möglicherweise das Datum und die Uhrzeit auch von einem Zeitserver holen.

Die Information über Datum und Uhrzeit werden benutzt, um Dateien und Verzeichnissen einen oder mehrere Zeitstempel zu geben: Wann wurde die Datei angelegt, zum letzten Mal geändert oder gelesen? Die Informationen können auch wichtig sein in Verbindung mit sicherheitsrelevanten Diensten auf dem System.

Die Sprache C bietet eine Reihe von Bibliotheksfunktionen an, mit denen Informationen über Uhrzeit und Datum aus dem System geholt und im Programm verarbeitet werden können. Um sie zu benutzen, muss man die Headerdatei `time.h` einbinden.

6.10.2 Schritt 1: Sekundenzeit

Die wichtigste Funktion ist `time()`. Mit ihr holt man die Zeitinformation aus der Echtzeituhr in eine Variable vom Typ `time_t`:

```
1 #include <stdio.h>
2 #include <time.h>
3 int main(void)
4 {
5     time_t x;
6     x=time(NULL);        /* oder: time(&x); */
7     printf("%i\n", x);
8     return 0;
9 }
```

In Zeile 6 wird die Variable x so behandelt, als sei sie eine int-Variable. Der ausgegebene Wert war in diesem Fall 1523464757. Zehn Sekunden später war er 1523464767. Was bedeutet nun diese große Zahl? Es handelt sich um die Anzahl der Sekunden seit dem 1. Januar 1970 um 0.00 Uhr. Tatsächlich ist `time_t` ein ganzzahliger Wert und kann einem `int`, aber auch einem `long int` entsprechen.

Mit der Anzahl der Sekunden kann man in Netzwerken und Prozesssteuerungen viel anfangen: Man kann einfach vergleichen, welcher Zeitpunkt früher war als ein anderer. Man kann durch einfaches Subtrahieren eine Zeitdauer bestimmen.

6.10.3 Schritt 2: Aufgespaltene Zeit

Sobald das Programm herausfinden soll, ob es morgens oder abends ist, ob ein bestimmter Wochentag ist oder ob ein Feiertag ist, reicht die Sekundenzeit in `time_t` nicht mehr aus. Hier braucht man eine Funktion, die die Sekundenzeit umrechnen kann. Dafür gibt es gleich zwei Funktionen, die sich nur in einer Kleinigkeit unterscheiden:

a) `gmtime()` gibt die Weltzeit (UTC) an, früher auch GMT genannt

b) `localtime()` gibt die Ortszeit an, z. B. die mitteleuropäische Sommerzeit

Die beiden Funktionen schreiben in eine Record-Variable vom Typ `struct tm`:

```
1 struct tm
2 {
3     int tm_sec;    /* Sekunde (0-60, wegen Schaltsek.) */
```

```
4     int tm_min;    /* Minute (0-59) */
5     int tm_hour;   /* Stunde (0-23) */
6     int tm_mday;   /* Tag im Monat (1-31) */
7     int tm_mon;    /* Anzahl Monate seit Januar */
8     int tm_year;   /* Anzahl Jahre seit 1900 */
9     int tm_wday;   /* Anzahl Tage seit Sonntag */
10    int tm_yday;   /* Anzahl Tage seit 1. Januar */
11    int tm_isdst; /* Sommerzeit (nein=0,ja=pos.,unbek.=neg.) */
12 };
```

Wichtig ist: Das Record-Variable, in die die beiden Funktionen schreiben, ist eine Modul-globale Variable. Sie ist irgendwo in der C-Bibliothek angelegt, wahrscheinlich in `time.c`. Für uns ist die Record-Variable aber unsichtbar. Und beim nächsten Aufruf irgendeiner Funktion, die in `time.h` deklariert ist, kann die Record-Variable wieder überschrieben werden.

Die Funktionen `gmtime()` und `localtime()` geben nun einen Zeiger auf die beschriebene interne Record-Variable zurück. Mit diesem Zeiger kann man nun an ihre Komponenten heran:

```
1  #include <stdio.h>
2  #include <time.h>
3  int main(void)
4  {
5     time_t x;
6     struct tm *pxeinzel;
7     x=time(NULL);
8     pxeinzel=localtime(&x);
9     printf("Jahreszahl: %i\n", (*pxeinzel).tm_year);
10 /* printf("Jahreszahl: %i\n", pxeinzel->tm_year); */
11    return 0;
12 }
```

6 Zeiger auf ein Record vom Typ `struct tm` bereitgestellt

8 `pxeinzel` bekommt die Adresse der internen Record-Variablen

9 `*pxeinzel` meint das Record, `(*pxeinzel).tm_year` eine einzelne Komponente; genauso kann man auf jede andere Komponente zugreifen; da der Punkt-Operator Vorrang vor dem Inhalts-Operator hat, müssen die Klammern sein

10 Gleiche Bedeutung wie Zeile 9, aber andere Schreibweise mit dem Pfeil-Operator

Einfacher ist es, selbst eine Variable mit einem Record bereitzustellen. Beim Aufruf wird über den Inhalts-Operator der Inhalt der Variablen gefüllt. Anschließend kann man direkt auf den Inhalt zugreifen:

```
1  #include <stdio.h>
2  #include <time.h>
3  int main(void)
4  {
5     time_t x;
6     struct tm xeinzel;
7     x=time(NULL);
8     xeinzel=*localtime(&x);
9     printf("Jahreszahl: %i\n", xeinzel.tm_year);
10    return 0;
11 }
```

6 Ein Record vom Typ `struct tm` wird bereitgestellt

8 `xeinzel` bekommt den Inhalt der internen Record-Variablen zugewiesen, auf die der Rückgabewert zeigt

9 `xeinzel` ist das Record, `xeinzel.tm_year` eine einzelne Komponente

Damit ist die Variable `xeinzel` sicher vor Veränderungen durch spätere Aufrufe von Funktionen aus `time.h`.

6.10.4 Schritt 3: Anzeige der Zeit

Manchmal möchte man einen Zeitpunkt auf dem Bildschirm anzeigen oder in eine Datei schreiben. Dazu kann man einfach die aufgespaltene Zeit aus dem Record nehmen und `printf()` benutzen:

```
1    printf("Es ist %02:%02:%02 Uhr\n",
2           xeinzel.tm_hour, xeinzel.tm_min, xeinzel.tm_sec);
```

Bei einigen Komponenten muss man die auszugebende Zahl anpassen (z. B. bei `tm_year`). Bei anderen muss man zur Ausgabe ein Hilfs-Array benutzen:

```
1    char tagname[7][3]={"so","mo","di","mi","do","fr","sa"};
2    printf("Tag: %s\n", tagname[xeinzel.wday]);
```

Eine ganz einfache Art der Ausgabe bieten die Funktionen `asctime()` und `ctime()`.

`asctime()` wertet den Inhalt einer Record-Variablen aus. Die Ausgabe erfolgt in einem String, von dem nur die Adresse zurückgegeben wird. Auch dieser String liegt in einer Modul-globalen Variablen, die irgendwo in der C-Bibliothek angelegt wurde. Für uns ist auch diese String-Variable unsichtbar; und bei einem weiteren Aufruf irgendeiner Funktion aus `time.h` kann auch diese Variable überschrieben werden. Falls man das nicht möchte, muss man sie mit `strcpy()` an einen Ort im eigenen Programm kopieren.

```
1  #include <stdio.h>
2  #include <time.h>
3  int main(void)
4  {
5      time_t x;
6      struct tm xeinzel;
7      char *p;
8      x=time(NULL);
9      xeinzel=*localtime(&x);
10     p=asctime(&xeinzel);
11     printf("Aktuell: %s\n", p);
12     return 0;
13 }
```

7 Zeiger `p` wird bereitgestellt

10 Zeiger `p` zeigt auf das richtige Ergebnis

11 Ausgabe

Man sieht, dass das Ergebnis auf englisch angezeigt wird. Daran könnte man auch mit `setlocale()` nichts ändern.

`ctime()` dagegen nimmt direkt eine Sekundenzeit, ruft intern `localtime()` auf und gibt dessen Ergebnis aus.

```c
#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t x;
    char *p;
    x=time(NULL);
    p=ctime(&x);
    printf("Aktuell: %s\n", p);
    return 0;
}
```

Auch hier wird das Ergebnis auf englisch angezeigt. Allerdings kann man hier nicht mehr zwischen Weltzeit und örtlicher Zeit wählen; es wird immer die örtliche Zeit angezeigt.

Wesentlich komfortabler ist die Funktion `strftime()`. Man übergibt ihr unter anderem die Adresse der Record-Variablen, einen Format-String mit besonderen Platzhaltern und einen freien Speicherplatz zum Hineinschreiben. Die Vorgehensweise mit Platzhaltern ähnelt der Benutzung von `printf()`, es sind aber andere Platzhalter, und sie beziehen sich alle auf die Record-Variable. Hier das Beispiel:

```c
#include <stdio.h>
#include <time.h>
#include <locale.h>
int main(void)
{
    time_t x;
    struct tm xeinzel;
    char buf[80];
    x=time(NULL);
    xeinzel=*localtime(&x);
    setlocale(LC_ALL, "");
    strftime(buf, 80, "%A, %d. %B %Y", &xeinzel);
    printf("Aktuell: %s\n", buf);
    return 0;
}
```

11 `strftime()` gibt in jeder Sprache aus, für die das System eingerichtet ist; deshalb ist `setlocale()` hier sinnvoll

12 `buf` ist der Speicherplatz zum Hineinschreiben und 80 seine Länge; es folgen der Formatstring und die Adresse der Record-Variablen

13 `printf()` benutzt auch wieder Platzhalter; das Wort `Aktuell:` und den Zeilentrenner `\n` hätte man schon in Zeile 12 einsetzen können; das Ergebnis ist gleich

Tabelle 1 zeigt die möglichen Platzhalter für `strftime()`.

6.10.5 Zurück zur Sekundenzeit

Wie oben beschrieben, hat die Sekundenzeit enorme Vorteile beim Rechnen mit Zeiten und beim Vergleichen von Zeitpunkten. Wer möchte schon gerne bei einer Record-Variablen zur aktuellen Zeit 100000 Stunden hinzurechnen und dabei unterschiedliche Monatslängen und Schaltjahre beachten, ganz zu schweigen von Schaltsekunden und Zeitumstellung?

PH	Bedeutung
%a	Wochentag, kurz
%A	Wochentag, lang
%b	Monatsname, kurz
%B	Monatsname, lang
%c	Datum+Uhrzeit
%d	Tag im Monat
%F	Datum nach ISO 8601 (seit C99)
%H	Stunde 00-23
%I	Stunde 01-12
%j	Tag im Jahr
%m	Monat 01-12
%M	Minute 00-59
%p	AM/PM
%S	Sekunde
%U	Wochennummer 00-53 ab erstem Sonntag
%w	Wochentagsnummer 0-6
%W	Wochennummer 00-53 ab erstem Montag
%x	Datum in lokaler Darstellung
%X	Uhrzeit in lokaler Darstellung
%y	Jahr, zweistellig
%Y	Jahr, vierstellig
%Z	Zeitzone
%%	Prozentzeichen

Tabelle 1: Platzhalter für `strftime()`

Die C-Bibliothek bietet zu diesem Zweck die Funktion `mktime()` an. Man füttert die Funktion mit einer Record-Variablen und erhält danach die zugehörige Sekundenzeit. `mktime()` ist damit die Umkehrung zu `localtime()`.

Eine Umkehrung zu `gmtime()` gibt es leider nicht in der Standard-C-Bibliothek. Die GNU-C-Bibliothek bietet jedoch die Funktion `timegm()`, die diese Lücke schließt.

Hier ein Beispiel für die gewünschte Anwendung von `mktime()`:

```
1  #include <stdio.h>
2  #include <time.h>
3  int main(void)
4  {
5      time_t x;
6      struct tm xeinzel={40,30,8,24,11,118,0,0,-1};
7      x=mktime(&xeinzel);
8      printf("Ergebnis: %i\n", x);
9      return 0;
10 }
```

5 Variable für das Ergebnis

6 Variable für den zu berechnenden Zeitpunkt (24. Dezember 2018, 8:30:40 Uhr). Wochentag und Tag im Jahr können leer bleiben. Ob zum Zeitpunkt Sommerzeit ist oder nicht, das soll die Funktion selbst herausfinden. Auch die Uhrzeit muss richtig gesetzt sein, sonst stimmt der errechnete Zeitpunkt nicht!

7 Berechnung des Zeitpunkts

6.10.6 Berechnung eines Wochentags

Manchmal muss man wissen, welcher Wochentag zu einem bestimmten Datum gehört. Beim aktuellen Datum hilft uns `localtime()`. Was aber, falls es sich um ein Datum in der Zukunft handelt, etwa bei einem Terminplaner-Programm?

In diesem Fall kann `mktime()` helfen. Die erste Idee ist, mit `mktime()` die Sekundenzeit des zukünftigen Datums auszurechnen. Aus dem Ergebnis könnte dann `localtime()` wieder eine Record-Variable bauen, die das richtige Datum enthält.

Aber es geht auch einfacher: `mktime()` berechnet nämlich nicht nur die Sekundenzeit. `mktime()` ändert auch einzelne Felder in der Record-Variablen, die es quasi als Eingabe bekommt. Man nennt dies den *Seiteneffekt* einer Funktion. Im Fall von `mktime()` sieht der Seiteneffekt so aus, das `tm_wday`, `tm_yday` und `tm_isdst` auf die passenden Werte gesetzt werden:

```
1 #include <stdio.h>
2 #include <time.h>
3 int main(void)
4 {
5     struct tm xeinzel={40,30,8,24,11,118,0,0,-1};
6     mktime(&xeinzel);
7     printf("Wochentag: %i\n", xeinzel.tm_wday);
8     return 0;
9 }
```

Hier ist es besonders wichtig, dass man vor dem Aufruf die Felder `tm_sec`, `tm_min` und `tm_hour` auf die richtigen Werte setzt, ansonsten landet man auf *vollkommen* falschen Zeitpunkten (um Jahrzehnte verschoben)!

6.10.7 Was fehlt

Einige Zeit-Funktionen sind in der C-Standardbibliothek nicht vorhanden, und man muss sie sich an anderer Stelle suchen:

- Nutzung einer genaueren Systemzeit; die Standard-C-Bibliothek von GNU bietet dazu die Erweiterungen `gettimeofday()` und `settimeofday()`, die in Mikrosekunden messen
- Schreiben der Systemzeit bzw. der Echtzeituhr; dies ist systemabhängig, bei Linux gibt es dazu die Funktion `adjtime()`
- Berechnungen zu Feiertagen wie Ostern
- Funktionen zu Zeitpunkten vor dem 01.01.1970; die Bibliothek GLib bietet Funktionen an, die das leisten (allerdings nur für die Zeit seit der Einführung des gregorianischen Kalenders am betreffenden Ort)
- Umrechnungen zu anderen Kalendern (z. B. julianischer Kalender)