6.7 Extras/Typdefinitionen mit typedef

6.7.1 Komplizierte Deklarationen und `typedef`

Eine englischsprachige Manual-Seite gibt für die `signal()`-Funktion den folgenden Prototypen an:

```
1  typedef void (*sighandler_t)(int);
2  sighandler_t signal(int signum, sighandler_t handler);
```

Die etwas ältere deutschsprachige Manual-Seite derselben Funktion nennt dagegen den bekannten Prototypen:

```
1  void (*signal(int signum, void (*handler)(int)))(int);
```

Man fragt sich, warum der eine Funktionsprototyp so viel einfacher ist als der andere? Kann man das für andere Vereinbarungen in C auch hinkriegen? Und wer oder was ist `typedef`?

Offenbar sind die beiden Prototypen identisch, und der Unterschied liegt in der Verwendung von `typedef`. Mit `typedef` kann man nämlich neue Typen definieren, ähnlich wie mit `#define` oder mit `struct`.

6.7.2 Neue Typen und `#define`

Schon mit dem `#define`-Makro des C-Präprozessors kann man neue Namen für bestehende Typen definieren. Nach den folgenden beiden Zeilen hat man zwei neue Namen für `long int` und für `int*`, die man ab sofort für den Rest der Quelltextdatei benutzen kann.

```
1  #define int32 long int
2  #define intpointer int *
3     int32 meine_ip_adresse; /* gibt: long int meine_ip_adresse */
4     intpointer px;          /* gibt: int * px;                 */
```

Schwieriger wird es, wenn man einen neuen Namen für `char string[80]` definieren will. Denn es steht die eine Hälfte des Typs vor dem Variablennamen, die andere Hälfte dahinter. `#define` kann aber nur eine komplette Kette ersetzen (also etwa `stringtyp` durch `char[80]`. Es gibt aber eine Abhilfe, denn die `#define`-Makros dürfen auch Parameter haben:

```
1  #define string80(A) char A[80]
2     string80(s);                /* gibt: char s[80]; */
3  #define string(A,B) char A[B]
4     string(s,123);              /* gibt: char s[123]; */
```

Leider weiß der Präprozessor nichts von Blöcken, Funktionen und Sichtbarkeit; um die Gültigkeit dieser neuen Namen muss man sich selbst kümmern.

Umgekehrt weiß der C-Compiler nichts mehr von den Ersetzungen, die der Präprozessor vorgenommen hat. Wer im Programm überall die Zahl `PI` durch `3.14159` ersetzt hat, kann mit den C-Werkzeugen (Linker, Profiler, Debugger) nicht mehr nach `PI` nachsehen, denn im C-Text gibt es nun kein `PI` mehr.

6.7.3 `typedef`

Diese Nachteile haben nun dazu geführt, dass man in ANSI-C eine echte Typdefinition eingebaut hat. Leider war zu diesem Zeitpunkt die Sprache schon fertig, so dass die Typdefinition mit einem Trick in die Sprache eingebaut wurde. — Zunächst einmal sieht die Typdefinition mit `typedef` sehr ähnlich aus wie die Lösung mit `#define` (Zeile 1). Und wieder kann sofort eine Variable (hier `ipaddr`) mit diesem neuen Typ angelegt werden (Zeile 2).

```
1  typedef long int int32_t;
2  int32_t ipaddr;
```

Aber im Gegensatz zu `#define` muss man hier um eine Ecke denken: **Die Typdefinition mit `typedef` sieht genauso aus wie sonst die Vereinbarung einer Variablen, nur steht hier am Anfang das Wort `typedef`. Durch dieses eine Schlüsselwort erhält man statt einer neuen Variablen einen neuen Typ.**[1]

Wie legt man nun in der Praxis einen neuen Typ mit `typedef` an? Das kann man in mehreren Schritten machen:[2]

a) Schritt 1: eine beliebige `static`-Variable dieses Typs anlegen:
`static long int x;`

b) Schritt 2: `static` durch `typedef` ersetzen:
`typedef long int x;`

c) Schritt 3: den Variablennamen durch den gewünschten Typnamen ersetzen:
`typedef long int int32_t;`

Das obige Beispiel mit dem 80 Zeichen langen String ergibt nun:

```
1  typedef char string80_t[80];
2  string80_t meinstring;
```

Man kann auch die Typen von Funktionsprototypen mit `typedef` vereinbaren:

```
1  /* statt: #include <math.h>, aber gcc -lm nicht vergessen! */
2  typedef double trigfunk(double); /* der Typ fuer die Prototypen */
3  trigfunk cos, sin, tan;          /* meine 3 Prototypen */
```

Auch Funktionszeiger kann man definieren, z.B.
`int (*vergleichsfunktion)(int *, int *);`

```
1  typedef int (*vergleichsfunktion_t)(int *, int *);
2  vergleichsfunktion_t intvergleich;
```

Mit `typedef` kann man in C Records benutzen, ohne das Wort `struct` zu verwenden (wird oft so gemacht):

```
1  typedef struct retyp
2  {
3     int x;
4     int y;
5  } rechteck_t;
6  rechteck_t fenster; /* statt: struct retyp fenster; */
```

Mit `typedef` steht nun also endlich eine universelle Möglichkeit zur Definition von Typen zur Verfügung, die in C sonst vermisst wurde: Recordtypen konnte man schon immer mit `struct` vereinbaren, Arraytypen und Zeigertypen dagegen nicht. Und diese Möglichkeit hält sich an alle Spielregeln von C: Die Typdefinition ist nur bis zum Ende des Blocks gültig, in dem sie vereinbart wurde. Soll sie im ganzen Quelltext gelten, wird sie eben global vorgenommen; im Gegensatz zu globalen Variablen sind global gültige Typen nämlich in der Regel nicht gefährlich für die Programmstabilität.

[1] Dadurch brauchte man die *Struktur* der Sprache C in nichts zu verändern. Nur das Schlüsselwort `typedef` kam hinzu. Nur der *Sinn* einer solchen Vereinbarung ist natürlich ganz anders, wenn ein Typ anstelle einer Variablen vereinbart wird, das muss der Compiler unterscheiden können.

[2] Hier wird einmal angenommen, dass man das Schlüsselwort `static` kennt, mit dem man für Variablen oder Funktionen Angaben über die Lebensdauer oder die Sichtbarkeit macht. Es dient hier aber nur als Platzhalter.