5.4 Datenstrukturen II/Unions

5.4.1 `union` – eine weitere Datenstruktur

In einem Programm zur Verwaltung elektrischer Bauelemente gibt es einen Datentyp mit dem Namen `struct bauelement_t`:

```
1  struct bauelement_t
2  {
3     double kennung;                /* z.B. 0.0001 */
4     char gehaeuse[20];             /* z.B. to-220 */
5     char hersteller[80];           /* z.B. atmel */
6     unsigned long einzelpreis; /* in ct */
7  };
```

Er hat die Komponenten `kennung`, `gehaeuse`, `hersteller` und `einzelpreis`. Bei der Komponente `kennung` gibt es aber ein Problem: Es kann sich handeln um

a) den Nennwert (z. B. bei Widerständen, Kondensatoren und Induktivitäten), das ist dann sinnvoll in einer `double`-Variablen unterzubringen

b) den Typ (z. B. bei Halbleitern), dazu braucht man eine Zeichenkette

Beides zu speichern gibt keinen Sinn. Man brauchte eigentlich ein Record, das nur entweder eine `double`-Variable oder eine Zeichenkette speichert.

5.4.2 Union als Lösung

In C gibt es eine sehr einfache Lösung für dieses Problem: Die Datenstruktur Union (Schlüsselwort: `union`) entspricht fast genau der Datenstruktur Record (Schlüsselwort: `struct`), mit der man Records vereinbart.

```
1     /* Record: */
2     struct styp
3     {
4        int a;
5        double b;
6     };
7
8     /* Union: */
9     union utyp
10    {
11       int x;
12       double y;
13    };
```

Nur: Bei einer `union` liegen alle Komponenten auf derselben Startadresse (Abbildung 1). In diesem Beispiel beginnen also `x` und `y` an derselben Adresse. Eine Variable dieses Typs ist damit nur so lang wie die längste Komponente, in diesem Fall wie `y`. Allerdings kann man eben gleichzeitig auch nur eine Komponente darin speichern, entweder `x` oder `y`. Man nennt bei der Union die Komponenten daher nicht Komponenten, sondern *Alternativen*.

5.4.3 Wie benutzt man eine Union?

Hier ist die entsprechende Union für das Problem mit den Bauelementen eingefügt:

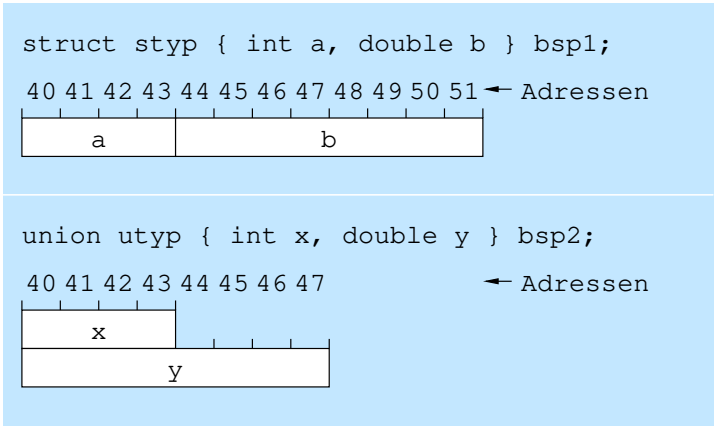

Abbildung 1: Vergleich zwischen Record und Union

```
union kennung_t
{
   double nennwert; /* falls R     */
   char typ[20];    /* falls nicht */
};
struct bauelement_t
{
   union_t kennung;             /* z.B. 0.0001 oder BC547 */
   char gehaeuse[20];           /* z.B. to-220 */
   char hersteller[80];         /* z.B. atmel */
   unsigned long einzelpreis; /* in ct */
};
```

Es gibt aber noch eine Kleinigkeit: Woher soll das Programm wissen, ob zur Zeit ein Widerstand oder ein Halbleiter in der Variablen gespeichert wurde?

Das kann man – leider – in der Union selbst nicht speichern. Es fehlt einfach jeder Speicherplatz dazu (C ist eben enorm effizient). Aber *außerhalb* kann man diese Information unterbringen: Der Datentyp, der die Union enthält, bekommt eine weitere Komponente `ist_wid`, in der man angeben kann, was gerade in der Union gespeichert wird:

```
union kennung_t
{
   double nennwert; /* falls R     */
   char typ[20];    /* falls nicht */
};
struct bauelement_t
{
   int ist_wid;                 /* 0=nein, sonst=ja */
   union_t kennung;             /* z.B. 4700 oder BC547 */
   char gehaeuse[20];           /* z.B. to-220 */
   char hersteller[80];         /* z.B. atmel */
   unsigned long einzelpreis; /* in ct */
};
```

Falls man in `ist_wid` eine 0 schreibt, soll das heißen, dass man die Variante `typ` verwenden will, ansonsten soll es heißen, dass man die Variante `nennwert` verwenden will. Das muss man allerdings selbst mit seinem Programm sicherstellen. Der Datentyp liefert nur die Möglichkeit

dafür. Hier dazu ein kurzes Programmstück:

```
    struct bauelement_t x, y;
    x.ist_wid=1;
    x.kennung.nennwert=4700.0; /* 4,7 kOhm */
    ...
    x.ist_wid=0;
    strcpy(x.kennung.typ, "BC547");
    ...
    if(x.ist_wid)
        printf("Widerstand mit %f Ohm\n", x.kennung.nennwert);
    else
        printf("Element des Typs %s\n", x.kennung.typ);
```

Die Komponente `ist_wid` dient als sogenannter *Diskriminator* für die Union.

5.4.4 Der ganze Aufwand für vier oder acht Bytes?

Bei kleinen und mittleren Datenmengen bringt es nicht viel, wenn man eine Union benutzt. Das obige Beispiel kann man genauso gut auch so verwirklichen:

```
struct bauelement_t
{
    int ist_wid;                /* 0=nein, sonst=ja */
    double nennwert;            /* falls R */
    char typ[20];               /* sonst*/
    char gehaeuse[20];          /* z.B. to-220 */
    char hersteller[80];        /* z.B. atmel */
    unsigned long einzelpreis; /* in ct */
};
```

Man belegt dann pro Bauelement ganze acht Bytes mehr. Deshalb kommt die Union auch in der Praxis eher selten vor.

Interessant wird es, wenn man sucht, wo die Union wirklich auftritt. Und das ist der Fall, wo man Hardware- oder Netzwerk-nah programmieren muss und trotzdem portabel, also unabhängig vom gerade benutzten Rechner und Betriebssystem bleiben will. So finden sich in den Header-Dateien der C-Compiler oft folgende Strukturen:

```
union ipv4adresse {
    uint8_t adresse_als_array[4];
    uin32_t adresse_als_int_zahl;
} x;
```

In diesem Fall kann man auf zwei Arten auf die Daten zugreifen: Byte-weise und als Ganzes. Man kann mit dieser Union herausfinden, in welcher Byte-Reihenfolge das eigene System Daten in einer 32-Bit-Zahl speichert. Falls es eine andere Reihenfolge als die des Netzes ist (das ist bei PC-artigen Systemen immer der Fall), dann kann man umspeichern:

```
    x=1;
    if(x.adresse_als_array[0]<>x.adresse_als_int_zahl)
    {
        swap(x.adresse_als_array[0], x.adresse_als_array[3]);
        swap(x.adresse_als_array[1], x.adresse_als_array[2]);
    }
```