2 Datentypen

Die folgenden Datentypen gibt's in C:

- kein Boolean Typ
- char: Character - Zeichen, **8 Bit** ASCII Codiert (Java, C#: 16 Bit Unicode)
- short: Ganzzahl. **16 Bit**.
- int: Integer, Ganzzahl. Die Größe hängt vom Rechner, Betriebssystem und Kompiler ab, üblich: 16, **32** auf 64 **Bit** wird üblicherweise verzichtet.
- long: Ganzzahl. **32 Bit**.
- long long: Ganzzahl. **64 Bit**.
- float: Fließkomma, einfach genau. **32 Bit** (die Gößе hängt vom Betriebssystem/Kompiler ab).
- double: Fließkomma, doppelt genau. **64 Bit** (die Gößе hängt vom Betriebssystem/Kompiler ab).

Idee der Maschinenabhängigen Länge: Berechnungen laufen unter optimaler Speicherausnutzung und sind schnell (auf der jeweilig verwendeten Hardware). Nachteil: Code läuft auf jedem Rechner anders. Die Datenbreite hängt ab von:

- Rechner
- Betriebssystem
- Compiler

in <limits.h> sind die Wertebereiche der Ganzzahl-Datentypen abgelegt (UCHAR_MAX ...), in <float.h> sind die Wertebereiche der Fließkommatypen abgelegt (FLT_... DBL...).

Um eine definierte Breite zur Verfügung zu stellen gibt's in <stdint.h> klar definierte Typen: int32_t, int16_t, int8_t ...

Für sämtliche Typen können die Vorsätze *signed* und *unsigned* verwendet werden. Ohne Vorsatz ist die Art Implementationsabhängig (meist vorzeichenbehaftet).

Speicherbedarf

Die Funktion **sizeof(Ausdruck)** liefert wieviel Speicherplatz die Variable oder der Typ *Ausdruck* liefert:

```
int a;
int groesze = sizeof(a);                // 1. Moeglichkeit: Speicherbedarf der Variable "a"
groesze = sizeof(unsigned char);         // 2. Moeglichkeit: Speicherbedarf einer "unsigned char" Variable
```

Konstante Werte ("Literale")

Für Nummern gilt: ohne Zusatz wird für Ganzzahlen, sofern es sich ausgeht der Typ *int* gewählt. Gesteuert werden kann mit den Suffix L (long) und U (unsigned). Mit vorangestellten 0 wird die Konstante als Oktale- und mit einem 0x als Hexadezimale-Zahl interpretiert. Fließkommazahlen werden als float interpretiert, ausser die Konstante ist für float zu groß.

Charakter-Literale werden durch einfache Anführungsstriche dargestellt `'a'`. Charakter-Sonderzeichen können durch sogenannte *Escape-Sequenzen* ausgedrückt. Dafür wird ein *Back-Slash* vor ein weiteres Zeichen gestellt (\n, \t ...). Unter vielen anderen gibt's: \n für Zeilenumbruch, \t für Tabulator, \" für doppeltes Anführungszeichen, \a akustisches Zeichen ... Zeichen können ebenfalls durch ihren hexadezimalen Code mit **\xhh** übergeben werden als Beispiel `'\x41'` für ein 'A' (dez. 65 = 0x41)

Zeichenketten-Literale (Strings) haben doppelte Anführungsstriche `"Hallo"`.

Typecast

Typen können von einem Typen in einen anderen Typen umgewandelt werden. Das kann automatisch (*implizit*) oder bewusst (*explizit*) erfolgen. Im englischen nennt man diese Umwandlung *Casten*.

Automatisch passiert's ganz einfach:

```
int num = 3;
char zeichen = num;
```

Hier wird die Nummer *num* automatisch in den Typen char umgewandelt (denglisch: gecastet). Unproblematisch ist der Autocast wenn ein "kleinerer" Datentyp einem "größeren" zugewiesen wird. Umgekehrt muss klar sein dass evtl. Daten verloren oder falsch interpretiert werden können. Im angeführten Beispiel kann *num* ja einen Wert haben der größer ist als in *zeichen* Platz hat.

Explizite Typumwandlung:

```
long numKurz = 0x10000000;
long long numLang = (long long)numKurz * numKurz;
```

Wird *"numKurz * numKurz"* berechnet, dann werden zwei long miteinander multipliziert, das Ergebnis wird auf einem long-Platz gespeichert und der ist zu klein also stimmt das Ergebnis nicht. Warum wird das Ergebnis auf einem long-Platz gespeichert? Weil in der gesamten Rechnung maximal long-Werte verwendet werden.

Mit dem Klammerausdruck (long long) wird der erste Ausdruck numKurz als long long verwendet, dadurch wird auch das Ergebnis in ein long long gespeichert (groß genug). Damit wird das richtige Ergebnis an numLang zugewiesen.

Explitziter Typcast: `(neuer-Typ)Variable` damit wird der Wert von Variable als neuer-Typ verwendet.

Konstante

Mit dem Schlüsselwort *const* können Konstante gekennzeichnet werden. Sie können während der Programmlaufzeit nicht (direkt) verändert werden:

```
const int MehrwertSteuer = 20;
const float PI = 3.14f;
```

Char Typ

Ein char ist 1 Byte (8 Bit) groß. char steht für *character* also ein *Zeichen*. Im Speicher ist dieser eine ganz normale Zahl zwischen -128 und +127. Wird eine char-Variable ausgegeben wird der Zahlenwert mittels ASCII-Tabelle in ein Zeichen übersetzt und ausgegeben. Intern ist und bleibt ein char eine Zahl. Entsprechend kann mit char-Variablen ganz normal gerechnet werden.

```
char c = 'A';
c++;
```

Hier wird der char-Variable c das Zeichen 'A' zugewiesen. In der Variable steht die numerische Entsprechung aus der ASCII-Tabelle ('A' -> 65). In der folgenden Zeile wird die Variable um 1 erhöht. Wird das Zeichen anschließend ausgegeben, dann wird ein 'B' angezeigt ('B' -> 66).

Auf diese Art können auch zum Beispiel Groß- in Kleinbuchstaben umgewandelt werden:

```
char c = 'C';       // in c steht 67 (='C')
c = c - 'A';        // in c steht 2 (3ter Buchstabe im Alphabeth)
c = c + 'a';        // in c steht 2 + 97 (97 = 'a') und damit 99 ( = 'c')
```

Genauso können auch Zahlen verwendet werden, die müßten der ASCII-Tabelle entnommen werden. Die ASCII-Tabelle definiert Zeichen für die Nummern von 0 bis 127 (eigentlich nur 7 Bit). Je nachdem wo das C-Programm die char-Zeichen ausgibt werden die verbleibenden 128 Zeichen anders kodiert. Um sämtliche 8 Bits (256 Zeichen) auszunutzen (und auch um dringend gewünschte Zeichen darzustellen) wird in eigenen Codierungen definiert, wie die verbleibenden 128 Zeichen codiert werden. In der Windows-Eingabeaufforderung kann die eingestellte Codepage mit `chcp` angezeigt werden (oder auch verändert werden), häufig bei uns *Codepage 850*. In UNIX kann das mit `cat /etc/default/locale` angezeigt werden, häufig bei uns *utf-8*.

Makros

Makros können auch so etwas Ähnliches wie Konstante definieren:

```
#define MAXDAYS 31

...

if (days < MAXDAYS) {
   ...
```

Makrodefinitionen gehören zu den sogenannten Präprozessorabweisungen. Während des Kompiliervorgangs wird vor der eigentlichen Komilierung der sogenannte Präprozessor über das Programm laufen lassen. Dieser beschäftigt sich nur mit Zeilen die mit dem #-Zeichen beginnen ("Präprozessoranweisungen").

Für #define werden dabei immer die folgenden Bezeichner (MAXDAYS) durch die als zweites folgenden Begriffe (31) ersetzt. Der Präprozessor läuft den ganzen Code durch und ersetzt überall wo er MAXDAYS findet durch 31. Er prüft nicht ob das Sinn haben kann oder vieleicht auch die Syntax verletzt wird.

An den Präprozessor anschliessend läuft der Kompiler über das Programm und der finden an allen Stellen an denen MAXDAYS geschrieben war nur mehr 31.

Zum Kenntlich machen, dass MAXDAYS ein Makro ist, wird es gesperrt groß geschrieben.

Enumerator

Wenn Aufzählungen erforderlich sind, können Enumeratoren hilfreich sein, etwa für Wochentage, Monatsnamen, Verbuchungstypen o.ä. Der Vorteil der Enums sind die einfache Erweiterbarkeit und die mögliche Sicherheit, dass die Werte unterschiedlich sind.

Eine Aufzählung hat einen Aufzählungstyp und Aufzählungselemente:

```
enum wochentag {MON, TUE, WED, THU, FRI, SAT, SUN} day;
day = FRI;
```

> **wochentag** wird als Aufzählungstyp bezeichnet
>
> **MON**, **TUE** ... sind die Elemente des Aufzählungstyps.
>
> **day** ist eine Variable des Aufzählungstyps.

Alternativ kann der Aufzählungstyp weggelassen werden (hier wochentag), wenn später keine weiteren Variablen des Typs wochentag verwendet werden müssen:

```
enum {MON, TUE, WED, THU, FRI, SAT, SUN} day;   // damit kanns nur die day-Variable als Wochentag geben
day = FRI;
```

Auch sinnvoll kann die Verwendung ohne Typ und ohne Variable sein:

```
enum {FALSE, TRUE};
```

Damit sind im Folgenden die Ausdrücke TRUE und FALSE verfügbar und sicher unterschiedlich. Mit einem Makro (#define) könnte TRUE und FALSE auf die gleichen Werte definiert werden (zum Beispiel beide auf 0).

Die Enum-Elemente sind immer vom Typ Integer (4 Bytes) und bekommen automatisch eine numerische Entsprechung. Der erste Wert entspricht 0, der zweite 1 und so fort. Die Werte können auch kontroliert gesetzt werden:

```
enum {FALSE=3, TRUE}
```

Dadurch wird FALSE drei und TRUE vier. Grundsätzlich ist es aber auch möglich absichtlich gleiche Werte zu setzen:

```
enum {YES, JA=0, NO, NEIN=1, N=1} response;     // Variable response kann JA, YES, NO, NEIN oder N sein
```

Weitere Variable werden so definiert:

```
enum wochentag day;
```

Die Bezeichnung enum muss immer mit angeführt werden (auch wenn enum als Funktionsparameter verwendet werden ...).

Bei vielfacher Verwendung kann ein typdef sinnvoll sein:

```
typedef enum wochentag TWOCHENTAG;
TWOCHENTAG day;
```

Typedef

Mit typedef können eigene Datentypen definiert werden:

```
typedef unsigned char TBYTE;        // Typ BYTE wird als vorzeichenlose 8-Bit-Groesze definiert
TBYTE b1;                          // Variable als BYTE (vorzeichenlose 8-Bit-Groesze)
```

Damit wird der neue Typ *TBYTE* als *unsigned char* definiert.

Häufig wird typedef verwendet um Definitionen abzukürzen, siehe Beispiel bei Enumerator:

```
typedef enum wochentag TWOCHENTAG;
```

Der Typ TWOCHENTAG wird definiert als *enum wochentag*. Damit können wochentag-Variablen ohne den Vorsatz enum definiert werden:

```
enum wochentag tag;
TWOCHENTAG day;         // genau gleich wie tag, besser lesbar ...
```

Zur Kenntlichmachung, dass TWOCHENTAG ein neu definierter Typ ist, wir der Name mit T begonnen und gesperrt groß geschrieben.

"Schöne" Programmierung setzt typedef durchaus sehr stark ein:

```
#define OFFSET 32.0
#define FAKTOR (9.0 / 5.0)
typedef double Celsius;
typedef double Fahrenheit;

Celsius tempC;          // Temperatur vom Typ Grad Celsius
Fahrenheit tempF;       // Temperatur vom Typ Grad Fahrenheit
tempF = 100.0;
tempC = (tempF - OFFSET) / FAKTOR;
```

Damit wird der Code sehr gut lesbar. Ein weiterer großer Vorteil ist, dass nachträglich der Datentyp sehr einfach verändert werden kann. Wenn zukünftig alle Grad-Celsius Werte nur mehr float sein sollen muss lediglich der typedef verändert werden.

Grundsätzlich werden mit typedef keine echten neuen Datentypen erstellt, sie ermöglichen lediglich alternative Bezeichnungen.

Fragen

- Wieviel Speicher (Bytes) wird durch `int x;` reserviert?
- Was bedeutet das Literal `4UL`?
- Verwenden Sie ein *enum* um die Boolschen Begriffe TRUE und FALSE zu definieren.
- Führen Sie ein Beispiel für die Verwendung von *typedef* an. Die Variable *color* kann die Werte *ROT*, *BLAU* und *GRUEN* annehmen.
- Wie kann der Speicherbedarf (Anzahl Bytes) der Variable `var1` in C ermittelt werden?
- Welchen Wertbereich kann der Typ `unsigned char` abbilden?
- Können in einem Aufzählungstypen mehrere Elemente den gleichen Wert haben?