3.6 C-Datentypen/Typkonvertierung

3.6.1 Wozu Typkonversion?

In einem C-Programm befinden sich die Zeilen:

```
1 char a=3; int b=4; float c;
2 c=a+b;
```

Der Plus-Operator verknüpft die Operanden a und b. Dieser Ausdruck hat einen Wert, der dem Ergebnis entspricht. Das rechte Gleichheitszeichen verknüpft die Variable c und diesen Wert. Es setzt c auf diesen Wert; selbst liefert der Ausdruck wiederum einen Wert, der mit `d=c=a+b` wiederum an die Variable d weitergegeben werden könnte.

Nun ist es so, dass alle drei Variablen verschiedene Datentypen aufweisen. In einigen Programmiersprachen ist es so, dass man Operatoren nur zwischen gleichen Datentypen benutzen kann. Für den Fall, dass man verschiedene Datentypen im Programm miteinander verknüpfen will, gibt es dort eigene Operatoren zur sogenannten expliziten Typumwandlung. Explizit heißt hier ausdrücklich bzw. gewollt. Statt Typumwandlung kann man übrigens auch Typkonversion sagen.

3.6.2 Explizite Typkonversion

Auch in C gibt es die Möglichkeit der expliziten Typkonversion. Dazu wird die runde Klammer benutzt (eine weitere Benutzung der runden Klammer!). Der Ziel-Typ der Konversion wird in ein Paar runder Klammern geschrieben und beides zusammen vor die Variable:

```
1 c=(int)a+b; // in C
```

In C++ ist auch eine andere Form erlaubt, die die Typkonversion so aussehen lässt wie einen Funktionsaufruf:

```
1 c=int(a)+b; // nur in C++
```

3.6.3 Grundsätze der Integer-Umwandlung

Was passiert nun bei der Umwandlung zwischen verschiedenen ganzzahligen Typen?

Falls der Zieltyp breiter als der Quelltyp ist, bleibt der dargestellt Wert erhalten: Bei einem unsigned-Typ wird der Inhalt nach links mit Nullen aufgefüllt. Bei einem Typ im Zweierkomplement wird die Variable nach links mit dem MSB aufgefüllt: 0001 wird zu 00000001, 1100 wird zu 11111100.

Falls der Zieltyp schmaler als der Quelltyp ist, wird der Inhalt links abgeschnitten [1]: 00000001 wird zu 0001. Hier kann es passieren, dass der dargestellte Wert sich ändert: 00110000 wird zu 0000.

Auch bei der Umwandlung zwischen signed- und unsigned-Typen kann es passieren, dass der dargestellte Wert sich ändert, wenn er im Zieltyp nicht dargestellt werden kann.

Hier ein paar Beispiele für Umwandlungen zwischen ganzzahligen Datentypen (`umwandlung_int.c`):

```
Umwandlung long int li nach short int hi:
li=12345, hi=12345     # klappt
li=2147483617, hi=-31  # Ueberlauf
Umwandlung int i nach unsigned int u:
i=12345, u=12345       # klappt
i=-3, u=4294967293     # nicht darstellbar
Umwandlung unsigned int u nach int i:
u=12345, i=12345       # klappt
u=4294967255, i=-41    # Ueberlauf
```

[1] Zumindest bei unsigned-Typen oder beim Zweierkomplement. Hätte man eine Darstellung mit Betrag und Vorzeichen, wäre es komplizierter. Dieser Fall ist aber selten.

3.6.4 Grundsätze der Umwandlung innerhalb Gleitpunkttypen

Bei der Umwandlung zwischen verschiedenen Gleitpunkttypen ist es ähnlich.

Falls der Zieltyp breiter als der Quelltyp ist, bleibt der dargestellt Wert auch hier erhalten.

Fall der Zieltyp dagegen schmaler als der Quelltyp ist, können zwei Fälle auftreten:

a) Der Wert ist entweder so groß (positiv oder negativ), dass er im schmaleren Typ nicht mehr dargestellt werden kann. Dann ist das Ergebnis undefiniert, das heißt beliebig.

```
d=22.875, f=22.875
```

b) Der Wert liegt innerhalb des Bereichs, der im schmaleren Typ auch dargestellt werden kann. Dann ist das Ergebnis, falls möglich, natürlich der gleiche Wert, falls nicht, der nächsthöhere oder nächstniedrigere Wert.

Hier ein paar Beispiele für die Umwandlung von double nach float (`umwandlung_gleit.c`):

```
d=1.79769313486e+308, f=inf # ausserhalb des Bereichs
d=22.875, f=22.875          # gleicher Wert
d=2.22507385851e-308, f=0   # naechstniedrigerer Wert
d=123456789, f=123456792    # naechsthoeherer Wert
```

3.6.5 Grundsätze der Umwandlung zwischen Integer- und Gleitpunkttypen

Die Umwandlung von ganzer Zahl in eine Gleitpunktzahl ist nahezu intuitiv: Falls der Wert innerhalb des Bereichs liegt, wird er, falls möglich, in denselben Wert umgewandelt, andernfalls in den nächsthöheren oder den nächstniedrigerer Wert. Falls der Wert außerhalb des Bereichs des Zieltyps liegt, ist das Ergebnis undefiniert.

Die Umwandlung einer Gleitpunktzahl in eine ganze Zahl funktioniert so: Der Nachkomma-Anteil wird abgeschnitten. Falls das Ergebnis nicht im ganzzahligen Typ dargestellt werden kann, ist das Ergebnis wiederum undefiniert. Hier die entsprechenden Beispiele (`umwandlung_gemischt.c`):

```
Umwandlung float f nach int i:
f=123.9, i=123
f=-123.9, i=-123
f=1.239e+17, i=-2147483648
Umwandlung int i nach float f:
i=24000000, f=24000000.000000
i=24000001, f=24000000.000000
```

3.6.6 Implizite Typkonversion

Falls der Operand nicht zum Operator passt oder ein binärer Operator (wie plus) zwei verschiedene Datentypen verknüpft, dann tritt die implizite (hier: eingeschlossene, verdeckte) Typumwandlung in Kraft.

Genauso ist es, wenn man einen Parameter falschen Typs an eine Funktion übergibt.

Die implizite Typkonversion benutzt die oben genannten Regeln, damit man weiß, *wie* konvertiert wird. Es gibt aber einige zusätzliche Regeln, damit man weiß, *wer* und *wohin* konvertiert wird:

```
1  unsigned int u=3;
2  int i=-2;
3  unsigned long int ul;
4  ul=u+i;
```

Im folgenden werden zwei wichtige Schemata der impliziten Typkonversion vorgestellt.

3.6.7 Schema 1: Integer-Erweiterung

Manche Operatoren, wie zum Beispiel << und >>, verlangen einen oder zwei Operanden, die mindestens die Breite des Typs `int` haben. Wenn jetzt aber nur ein Operand vom Typ `char` zur Verfügung steht, wird er in den Typ `int` umgewandelt. Das nennt man Integer-Erweiterung.

Betroffen sind die Typen `char` und `unsigned char`, `short` und `unsigned short`, Integer-Bit-Felder (gehören thematisch zu den Records) und Aufzählungstypen.

Falls alle Werte des Quelltyps mit `int` darstellbar sind, wird nach `int` gewandelt (also fast immer), sonst nach `unsigned int`.

3.6.8 Schema 2: Übliche arithmetische Umwandlung

Die übliche arithmetische Umwandlung (so wird sie im C-Standardwerk genannt) findet unter anderem statt bei Rechenoperatoren wie plus, minus, mal und geteilt.

Hier ein Beispiel:

```
1 long double a=20.0;
2 int b=4;
3 printf("%lf", a-b);
```

Wass passiert hier implizit? Da a vom Typ `long double` ist, wird b nach `long double` konvertiert, das Ergebnis der Subtraktion wird berechnet und ist auch vom Typ `long double`.

Bei der üblichen arithmetische Umwandlung

a) wird der kleinere zum größeren Typ hin konvertiert.

b) hat das Ergebnis den größeren Typ.

Die Rangfolge dabei ist:

a) long double, double, float,

b) dann Integer-Erweiterung,

c) dann unsigned long int, long int, unsigned int, int.

d) Hier gibt es eine Sonderregel bei long int und unsigned int: Falls long int Obermenge von unsigned int ist, falls also long int mehr Bits hat als unsigned int, dann wird nach long int gewandelt, ansonsten nach unsigned long int.

Bei C99 und C11 ist die Aufstellung ergänzt um die Datentypen long long int und unsigned long long int.

3.6.9 Implizite Typkonversion bei Parametern

Die erste Version von C wurde beschrieben im Standardwerk von Kernighan und Ritchie. In dieser Version enthalten Prototyp und der Funktionskopf der Funktionsdefinition keine Information über den Typ der formalen Parameter:

```
1  double xquadrat(); /* Prototyp ohne Parameter, nur Rueckg.-Wert */
2  int main()
3  {
4      double a;
5      a = xquadrat(40.2); /* Woher weiss C den Typ des Parameters? */
6      printf("%f\n", a);
7  }
8  /******************/
9  double xquadrat(x) /* x ohne Typ! */
10 double x;
```

```
11 {
12     double y;
13     y=x*x;
14     return y;
15 }
```

Wenn man in dieser C-Version einen aktuellen Parameter übergibt, kann der Compiler nicht herausfinden, ob der Typ stimmt. Falls er nicht stimmt, kann der Compiler ihn auch nicht implizit umwandeln. Deshalb befolgt er nur zwei Regeln:

a) Aktueller Parameter ist ganzzahlig: Integer-Erweiterung

b) Aktueller Parameter ist vom Typ `float`: Umwandlung nach `double`

In den Versionen seit C89 stehen andere Möglichkeiten zur Verfügung: Man kann den Typ des aktuellen Parameters mit dem Typ des formalen Parameters vergleichen. Bei Gleichheit ist nichts zu tun. Bei Verschiedenheit wird der aktuelle Parameter in den Typ des formalen Parameters umgewandelt.

Wozu muss man jetzt auch über die alte Version Bescheid wissen?

a) Vergisst man bei Bibliotheksfunktionen das Einbinden der Header-Datei, so findet automatisch das Verhalten der alten Version statt (außerdem ist dann der Rückgabetyp `int`!)

b) Funktionen wie `printf()` und `scanf()` benutzen eine variable Parameterliste. Nur der erste Parameter ist vom Typ her angegeben (eine Zeichenkette, die den Formatstring enthält), der Rest ist unbekannt. Für diese unbekannten Parameter findet das Verhalten der alten Version statt

3.6.10 Implizite Typkonversion bei Operatoren

Bei den Operatoren wird es etwas unübersichtlich. Tabelle 1 versucht eine Einordnung. Dabei bedeutet IE Integer-Erweiterung und ÜAU übliche artihmetische Umwandlung.

Operator	Art der Konversion
`+x`	IE
`-x`	IE, Besonderheit s. u.
`~x`	IE, x muss ganzzahlig sein
`x<<y`	IE, x und y müssen ganzzahlig sein
`x>>y`	IE, x und y müssen ganzzahlig sein
`x<y`	ÜAU, ebenso `<, >=, <=, ==, !=`
`x+y`	ÜAU, ebenso `-, *, /`
`x%y`	ÜAU, y muss ganzzahlig sein
`x&y`	ÜAU, x und y müssen ganzzahlig sein
`x|y`	ÜAU, x und y müssen ganzzahlig sein
`x^y`	ÜAU, x und y müssen ganzzahlig sein
`x&&y`	ÜAU, Ergebnis ganzzahlig
`x&&y`	ÜAU, Ergebnis ganzzahlig
`!x`	ÜAU, Ergebnis ganzzahlig
`x?y:z`	für y und z ÜAU
`x=y`	Umwandlung y in Typ von x
`x,y`	keine
`return x`	Umwandlung in Rückgabetyp der Funktion

Tabelle 1: Typkonversion bei Operatoren