2

Elementare Datentypen

2.1 Liste der Datentypen

Bitanzahl und Wertbereich sind nachfolgend für typische Intel-Plattformen gezeigt (<limits.h> <float.h>):

Typ	Breite	Wertbereich / Signifikanz	STD
char	8	-128...+127	
signed char	8	-128...+127	C89
unsigned char	8	0...255	C89
short	16	-32768...+32767	
unsigned short	16	0...65535	C89
int	32	-2147483648...+2147483647	
unsigned	32	0...4294967295	
long	32	siehe int	
unsigned long	32	siehe unsigned	C89
long long	64	-9223372036854775808... +9223372036854775807	C99
unsigned long long	64	0...18446744073709551615	C99
int	16	DOS: siehe short	
unsigned	16	DOS: siehe unsigned short	
float	32	7 Stellen	
double	64	15 Stellen	
long double	80	19 Stellen (Intel iX86)	C89
long double	128	33 Stellen (z. B. HP-UX)	C89
sizeof(long double)	10	DOS	C89
sizeof(long double)	12	32 Bit (10+2 FüllByte)	C89
sizeof(long double)	16	HP-UX	C89

Die Integer-Typen wurden in Kurzschreibweise angegeben; Hinter short, unsigned und long kann jeweils noch int stehen: z. B.: unsigned long int

H. O.B. Schellong, *Moderne C-Programmierung,* Xpert.press,
DOI 10.1007/978-3-642-54437-8_2, © Springer-Verlag Berlin Heidelberg 2014

2.2 Erklärungen zu den Datentypen

`signed` kann auch mit anderen Integer-Typen als `char` verwendet werden, was aber redundant wäre. Eine Vorzeichenbehaftung des Typs `char` ist implementationsabhängig: `char` ist entweder `signed char` oder `unsigned char`. Genau betrachtet sind `char`, `signed char` und `unsigned char` drei verschiedene Typen.

Die meisten Compiler haben eine sehr sinnvolle Option, die global umfassend `char` als `unsigned char` bewertet. Diese Option erzeugt oft etwas kleinere und schnellere Programme und beseitigt eine Reihe von potentiellen Problemen. Auch Zeichenkonstanten `'c'` sind dabei `(int)(unsigned char)`. Wurde diese Option gegeben, hat `signed char` einen Sinn. Allerdings will man höchst selten im Zahlenbereich eines `char` arithmetische Berechnungen mit Vorzeichen betreiben. Andererseits können beispielsweise Temperaturwerte platzsparend gespeichert werden, die dann bei Verwendung automatisch (implizit) vorzeichenerhaltend auf `int` erweitert werden (▶ 95).

1er-Komplement-Prozessoren können in 8 Bit nur `-127..+127` darstellen, nicht `-128..+127`. Dafür haben sie eine negative und eine positive `0`. Der Standard fordert `-127..+127` als Mindestwertebereich für einen vorzeichenbehafteten `char`.

Die Wertebereiche in der obenstehenden Tabelle entsprechen den Standard-Mindestforderungen, mit der Ausnahme, daß laut Standard die negativen Werte um 1 positiver sind und daß für `int` der Wertbereich eines `short` ausreicht, was auch für deren `unsigned`-Varianten gilt.

`long long` ist neu seit dem Standard C99. Der Compiler `gcc` kannte schon zuvor `long long`, Borland Builder kennt `__int64` und `-2345i64`, `2345ui64`. Der neue Standard enthält: `long long`, <stdint.h>: `int64_t`, etc.

Achtung, es gibt Plattformen, auf denen ein Byte 32 Bit hat! Und die Typen `char`, `int`, `long`, ... haben dort alle 32 Bit!, und `sizeof(typ)` für all diese Typen ist `1`, was korrekt ist.

void

```
void funktion(void)  { /*...*/  return; }
```

Die Funktion `funktion` hat *keinen* Rückgabewert und erhält *keine* Argumente beim Aufruf: `funktion();`

```
(void)printf("...");
```

Der Rückgabewert der Funktion `printf` (die einen solchen hat: `int`) soll explizit *ignoriert* werden. Dies wird aber meist implizit belassen, indem ein Rückgabewert einfach gänzlich unbenutzt bleibt.

void

```
# define uchar  unsigned char

void *memset_F(void *d0, int i, register unsigned n)
{
   register uchar *d= (uchar*)d0;
   register uchar  c= (uchar )i;
   while (n > 0)  *d++ = c, --n;
   return (d0);
}

# undef uchar
```

Der Funktion `memset_F` kann die Adresse von Zielobjekten *beliebigen Typs* übergeben werden, weil hier mittels `void*` ein unbestimmter Adressentyp vereinbart wurde. Innerhalb der Funktion wird in eine `uchar`-Adresse umgewandelt. Bei Variablen, die eine `void`-Adresse enthalten (oben: `d0`), können weder `*d0` noch `d0++` durchgeführt werden, weil der Compiler nicht weiß, mit welcher Breite er auf das Objekt zugreifen soll und um wieviel er die Adresse erhöhen soll. Der Compiler kennt nur `sizeof(d0)`, nicht jedoch `sizeof(*d0)`! Es können damit nur Zuweisungen und Vergleiche auf Gleichheit oder Ungleichheit vorgenommen werden.

```
int iarr[64];
struct dirent dea[100];

memset_F(iarr, 0, sizeof(iarr));
memset_F(dea , 0, sizeof(dea));
memset_F(&dea[8], -1, sizeof(*dea));
memset_F(&dea[8], -1, sizeof(struct dirent));

// sizeof(dea)/sizeof(*dea) ist gleich 100
```

`memset_F()` schreibt ggf. 0-Bits und löscht somit nur bedingt typgerecht (▶ 168).

Der Ausdruck `(void)v;` hat *keinen* Wert, sondern es werden eventuell nur Seiteneffekte bewirkt. Dies bei diesem Ausdruck, falls `v` mit `volatile` qualifiziert ist. In diesem Fall erfolgte ein Blind-Lesevorgang von `v`. Ohne `volatile` würde der Compiler diesen Ausdruck weg-optimieren (▶ 143).

Hingegen der Ausdruck `(void*)&v;` hat einen Wert, nämlich einen Adressenwert, der vom Originaltyp her unverändert sein muß. Der Typ `void*` ist ein zentraler, temporärer Träger für *alle* Adressentypen.

Jede Adresse `void*` kann ohne explizite Typumwandlung mit beliebigen Adressentypen verknüpft werden, im Rahmen der erlaubten Operationen. Der obenstehende Typ-Cast `(uchar*)d0` ist redundant.

```
typedef unsigned       uint;
typedef unsigned char  byte;

void *memcpy_F(void *d0, const void *s0, register uint n)
{
   register byte *d=d0;
   register const byte *s=s0;
   if (n)  do  *d = *s, ++d,++s;  while (--n);
   return d0;
}

int memcmp_F(const void *d0, const void *s0, uint n)
{
   register const byte *d=d0, *s=s0;
   if (n && d!=s)  {
     do  if (*d!=*s)  return *d - *s;
     while (--n&&(++d,++s,1));
   }
   return 0;
}

uint strlen_F(const byte *s0)
{
   register const byte *s=s0;
   while (*s)  ++s;
   return (s-s0);
}

int strcmp_F(const byte *d0, const byte *s0)
{
   register const byte *d=d0, *s=s0;
   if (d!=s)  {
     for (;  1;  ++d,++s)  {
        if (*d!=*s)  return *d - *s;
        if (!*s)  break;
     }
   }
   return 0;
}
```

Es ist zu beachten, daß `*d - *s` wegen der `int`-Promotion (▶ 95) zu falschen Ergebnissen führte, sollten vorzeichenbehaftete Byte-Werte verglichen werden. Bei Gleichheit (`d!=s`) würde ein und dieselbe Byte-Folge verglichen.

http://www.springer.com/978-3-642-54436-1

Moderne C-Programmierung
Kompendium und Referenz
Schellong, H.
2014, XVIII, 335 S. 20 Abb. in Farbe., Hardcover
ISBN: 978-3-642-54436-1