Datentypen in C

Informatik Universität Hamburg

Proseminar: C-Grundlagen und Konzepte

Jan Branitzki

31.05.13

Inhalt

1. Der Aufzählungstyp „enum“
2. Bit Shifting/Bitweise Operatoren
 - a. Beispiel: Int
 - b. Nutzen
3. Speicherbedarf von Datentypen
 - a. bei MCC und GCC
 - b. beeinflussen
4. Komplexe Zahlen
5. Wann welchen Datentypen nutzen?
6. Anmerkungen
7. Quellenverzeichnis

1. Der Aufzählungstyp „enum“

Wird benutzt, um Aufzählungen darzustellen

– Syntax:

enum *name* {*Wert0*, *Wert1*, ... , *Wertn*}

– Sofern nicht anders angegeben, beginnt enum bei 0.

enum mit anderem Startwert ist möglich

– Syntax:

enum *name* {*Wert0* = 42, *Wert1*, ... *Wertn*}

– Die Aufzählung beginnt hier bei 42, das zweite Element (*Wert1*) hat also der 43, das dritte 44, usw.

1. Der Aufzählungstyp „enum“ [Fortsetzung]

Es ist auch möglich den Index nicht an den Anfang zu setzen

- Syntax:

```
enum name {Wert0, Wert1, Wert2=42, ... , Wertn}
```

- Beachte: Die Elemente vor dem indizierten Element beginnen immer noch bei 0 (Also *Wert0* = 0, *Wert1* = 1).

Interne Repräsentation von enum: Int.

2. Bit Shifting/Bitweise Operatoren

Die Operatoren für bitweises Shiften $\ll$ und $\gg$ verschieben die Bits eines Bitstrings um den, als int, angegebenen Offset.

- Syntax:

Bitstring $\ll$ *Offset*

Bitstring $\gg$ *Offset*

Überschüssige Bits gehen verloren und werden durch 0 ersetzt

- Alternative: Bitweises Rotieren (nicht implementiert in C)

Ein Left Shift ($\ll$) um n entspricht einer Multiplikation mit 2^n

Entsprechend entspricht Right Shift ($\gg$) Division mit 2^n

- Beispiel:

$$\begin{array}{rcl} 0010 \ll 0010 & = & 1000 \\ = 2 \ll 2 & = & 8 \\ 2 * 2^2 & = & 8 \end{array}$$

2. Bit Shifting/Bitweise Operatoren [Fortsetzung]

Bitweises NICHT: Negiert einen Bitstring bitweise.

- Syntax:

$\sim$ *Eingabestring*

- Beispiel:

$\sim 1001 \Rightarrow 0110$

Bitweises UND: Verknüpft zwei Bitstrings bitweise mit log. UND.

- Syntax:

Eingabestring1 & *Eingabestring2*

- Beispiel:

$1001 \& 0101 \Rightarrow 0001$, da

$1\&0 \Rightarrow 0$

$0\&1 \Rightarrow 0$

$0\&0 \Rightarrow 0$

$1\&1 \Rightarrow 1$

2.Bit Shifting/Bitweise Operatoren [Fortsetzung]

Bitweises ODER: Verknüpft zwei Bitstrings bitweise mit log. ODER.

- Syntax:

Eingabestring1 | Eingabestring2

- Beispiel:

1001 | 0101 => 1101, da

1|0 => 1

0|1 => 1

0|0 => 0

1|1 => 1

2.Bit Shifting/Bitweise Operatoren [Fortsetzung]

Bitweises XOR: verknüpft zwei Bitstrings bitweise mit XOR.

- Syntax:

Eingabestring1 ^ *Eingabestring2*

- Beispiel:

1001 ^ 0101 => 1100, da

1^0 => 1

0^1 => 1

0^0 => 0

1^1 => 0

- Sind beide Eingabestrings gleich, erhält man 0. Da $1^1 = 0 = 0^0$
Register auf 0 setzen ist so meist schneller, als 0 ins Register zu laden.
 - WertVonRegisterX=WertVonRegisterX^WertVonRegisterX
=> WertVonRegisterX = 0

2.Bit Shifting/Bitweise Operatoren

a. Beispiel: Int

Repräsentation eines Ints als Bitstring ist gleich der Zahl in Binärdarstellung.

- Ints können *signed* oder *unsigned* sein, also mit oder ohne Vorzeichen vorliegen.
 - Bei *signed* Ints ist das erste Bit das Vorzeichenbit und zeigt an, ob der Int negativ(1), oder positiv(0) ist.
 - Sofern nicht anders angegeben, sind Ints in C *signed*.

```
main()
{
    unsigned int a = 42;    /* 42 = 0010 1010 */
    unsigned int b = 23;    /* 23 = 0001 0111 */
    unsigned int c = 0;

    c = a & b;              /* 2 = 0000 0010 */
    c = a | b;              /* 63 = 0011 1111 */
    c = a ^ b;              /* 61 = 0011 1101 */
    c = a << 2;             /* 168 = 1010 1000 */
}
```

2.Bit Shifting/Bitweise Operatoren

b. Nutzen

Elementare Rechenoperationen auf Maschinenebene

- Maschine kennt nur 0, 1 und bitweise Operationen

Flags(z.B. Bei langen „Flag-Ketten“)

Kodierung(Bitmuster, -maske)

3. Größe von Datentypen

a. bei MCC und GCC

Name	Anderer Name	Größe* bei MCC**	Größe* bei GCC***	Zahlenbereich
unsigned char	Unsigned_int8	1	1	0...255
signed char	Signed_int8	1	1	-128...127
wchar_t	_wchar_t	2	2	0...65.535
unsigned short int	Unsigned_int16	2	2	0...65.535
signed short int	Signed_int16	2	2	-32.768...+32.767
unsigned int	Unsigned_int32	4	4	0...4.294.967.295
signed int	Signed_int32	4	4	-2.147.83.648...2.147.483.647
unsigned long int	Unsigned_int64	8	8	0...18.446.744.073.709.551.615
signed long int	Signed_int64	8	8	-9.223.372.036.854.775.808...9.223.372.036.854.775.807
(signed) float	-	4	4	3,4E-38...3,4E38
(signed) double	-	8	8	1,7E-308...1,7E308
(signed) long double	-	8	12/16(Prozessorarchitektur)	MCC: s. Double GCC: 3,65E-4951...1,18E4932
(signed) float complex	-	8	8	Reell: s. Float; Imaginär: s. Float
(signed) double complex	-	16	16	Reell: s. Double; Imaginär: s. Double
(signed) long double complex	-	16	24/32(Prozessorarchitektur)	Reell: s. Long double; Imaginär: s. Long double

* in Byte auf 32/64Bit-Maschinen

**Microsoft Visual C-Compiler

***GNU C-Compiler

Welche Datentypen sind immer gleich groß?

KEINE!!

Dies ist von Compiler zu Compiler und von Maschine zu Maschine verschieden.

3. Größe von Datentypen b. beeinflussen

Es gilt: 0Byte < Char <= short <= int <= long

- Keine Aussage über feste Bitgrößen
- Für feste Bitgrößen gibt es <stdint.h>
→ bietet Typedefs für beliebig breite ints:
- Syntax:

intN_t x oder **uintN_t** x; für einen unsigned int x der Breite *N*

int_leastN_t x oder **uint_leastN_t** x; für einen unsigned int x mit Mindestbreite *N*

int_fastN_t x oder **uint_fastN_t** x; für einen int x vom schnellsten int-Typ mit Mindestbreite *N*

intmax_t x oder **uintmax_t** x; für einen größtmöglichen int x

intptr_t x oder **uintptr_t** x; für einen int x, der numerische Werte von Pointern aufnehmen kann

4. Komplexe Zahlen

Zur Darstellung von komplexen Zahlen gibt es die Headerdatei `<complex.h>`.

- komplexe Zahlen bestehen aus einem reellen und einem imaginären Teil.

Beide Teile werden als floats oder doubles repräsentiert.

- Syntax:

float _Complex *c* = 2.0 + 3.0*I; oder

double _Complex *c* = 2.0 + 3.0*I; zur Erstellung von komplexen Zahlen (statt **_Complex** geht auch **complex**)

4. Komplexe Zahlen

[Fortsetzung]

Mit `creal(c)` und `cimag(c)` können die reellen und imaginären Teile einer komplexen Zahl ausgegeben werden.

- Größe eines `complex'` = Addition des reellen und imaginären Teils.
Also 8 Bytes bei einem `float complex` und 16 bei einem `double complex`.
 - Long double möglich, aber Speicherbedarf und Präzision nur größer/höher, wenn long double auch größer als double

5. Wann welchen Datentypen nutzen?

Bei Iterationen Short benutzen, da Wertebereich meist ≤ 255

Bei Gleitkommarechnungen überlegen, wie präzise das Ergebnis sein soll bzw. die Werte sind

Bei int-Variablen, die auf Speicherzellen zeigen sollen, macht es Sinn, sie unsigned zu setzen

Char nicht als Short Int benutzen

intmax_t ist performance-technisch gesehen mit Vorsicht zu genießen!

IMMER den Wertebereich bedenken! Nehmt den kleinstmöglichen Datentypen, aber achtet darauf, dass es keine Überläufe geben kann

6. Anmerkungen

Weitere Details sind in den jeweiligen Headerdateien einzusehen(float.h, limits.h, stdint.h, etc).

Zu finden:

Visual Studio: *Installationsverzeichnis/VC/include*

7. Quellen

http://pronix.linuxdelta.de/C/standard_C/c_programmierung_8_2.shtml
(abgerufen: 11.05.13)

[http://msdn.microsoft.com/en-US/library/17zwb64t\(v=vs.80\).aspx](http://msdn.microsoft.com/en-US/library/17zwb64t(v=vs.80).aspx)
(abgerufen 12.05.13)

http://openbook.galileocomputing.de/c_von_a_bis_z/006_c_operatoren_005.htm
(abgerufen 15.05.13)

http://de.wikipedia.org/wiki/Bitweiser_Operator
(abgerufen 22.05.13)

<http://home.fhtw-berlin.de/~junghans/cref/CONCEPT/bitwise.html>
(abgerufen 22.05.13)

http://en.wikipedia.org/wiki/Long_double
(abgerufen 23.05.13)

http://openbook.galileocomputing.de/c_von_a_bis_z/005_c_basisdatentypen_010.htm
(abgerufen 23.05.13)

<http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html>
(abgerufen 23.05.13)