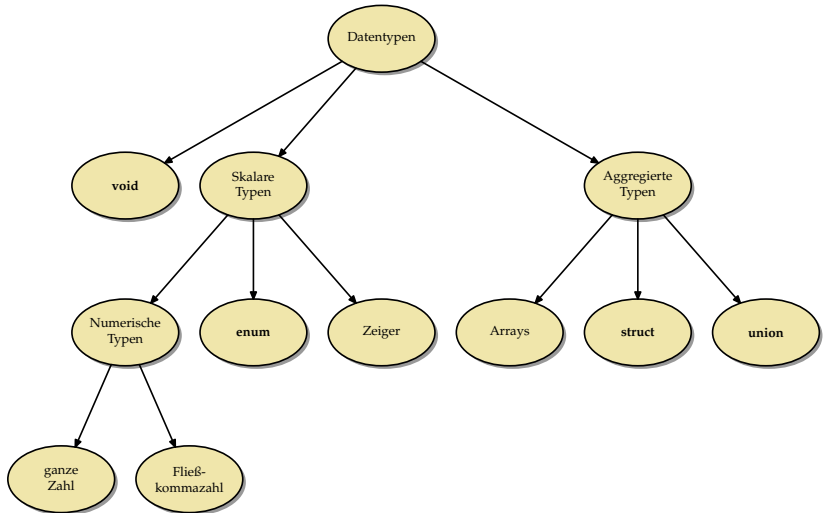


Datentypen legen

- den *Speicherbedarf*,
- die *Interpretation* des Speicherplatzes sowie
- die *erlaubten Operationen* fest.



- Im einfachsten Falle lässt sich eine Variablenvereinbarung sehr einfach zerlegen in die Angabe eines Typs und die des Variablennamens:

int *i*;

Hier ist *i* der Variablenname und **int** der Typ.

- Diese Zweiteilung entspricht soweit der Grammatik:

$\langle \text{declaration} \rangle$	\longrightarrow	$\langle \text{declaration-specifiers} \rangle$ [$\langle \text{init-declarator-list} \rangle$]
$\langle \text{declaration-specifiers} \rangle$	\longrightarrow	$\langle \text{storage-class-specifier} \rangle$ [$\langle \text{declaration-specifiers} \rangle$]
	\longrightarrow	$\langle \text{type-specifier} \rangle$ [$\langle \text{declaration-specifiers} \rangle$]
	\longrightarrow	$\langle \text{type-qualifier} \rangle$ [$\langle \text{declaration-specifiers} \rangle$]
	\longrightarrow	$\langle \text{function-specifier} \rangle$ [$\langle \text{declaration-specifiers} \rangle$]

- Leider trennt die Syntax nicht in jedem Falle sauber den Namen von dem Typ.
- Beispiel:
int* *ip*;
- Hier besteht die linke Seite, d.h. der \langle declaration-specifier \rangle nur aus **int**. Der Dereferenzierungs-Operator wird stattdessen syntaktisch der rechten Seite, dem \langle init-declarator-list \rangle zugeordnet.
- Dies hat zur Konsequenz, dass bei
int* *ip1,ip2*;
ip1 und *ip2* unterschiedliche Typen erhalten. So ist *ip1* ein Zeiger auf **int**, während *ip2* schlicht nur den Typ **int** hat.

- Zu den skalaren Datentypen gehören alle Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert oder umgekehrt ein Null-Zeiger ist äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C unterstützten Adressarithmetik begründet.

⟨integer-type-specifier⟩	→	⟨signed-type-specifier⟩
	→	⟨unsigned-type-specifier⟩
	→	⟨character-type-specifier⟩
	→	⟨bool-type-specifier⟩
⟨signed-type-specifier⟩	→	[signed] short [int]
	→	[signed] int
	→	[signed] long [int]
	→	[signed] long long [int]
⟨unsigned-type-specifier⟩	→	unsigned short [int]
	→	unsigned [int]
	→	unsigned long [int]
	→	unsigned long long [int]
⟨character-type-specifier⟩	→	char
	→	signed char
	→	unsigned char
⟨bool-type-specifier⟩	→	_Bool

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahme hiervon sind **char** und **_Bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- Für jeden der ganzzahligen Datentypen gibt es Mindestintervalle, die abgedeckt sein müssen. (Die zugehörige Übersichtstabelle folgt.)
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^n$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^n a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt.

Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C11 (und den früheren Standards) nur drei zugelassene Varianten:

► **Zweier-Komplement:**

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^{n-1}$$

Wertebereich: $[-2^{n-1}, 2^{n-1} - 1]$

Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

► **Einer-Komplement:**

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n (2^{n-1} - 1)$$

Wertebereich: $[-2^{n-1} + 1, 2^{n-1} - 1]$

Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt:

$$-a == \sim a$$

Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

► **Trennung zwischen Vorzeichen und Betrag:**

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $[-2^{n-1} + 1, 2^{n-1} - 1]$

Vorsicht: Es gibt zwei Repräsentierungen für die Null.

Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

Was passiert bei einer Addition, Subtraktion oder Multiplikation, die den Wertebereich des jeweiligen Datentyps verlässt?

- ▶ Bei vorzeichenbehafteten ganzen Zahlen ist das Resultat undefiniert. In der Praxis bedeutet dies, dass wir die repräsentierbaren niederwertigen Bits im Zweierkomplement erhalten.
- ▶ Bei ganzen Zahlen ohne Vorzeichen stellt C sicher, dass wir das korrekte Resultat modulo 2^n erhalten.

Alle gängigen Prozessorarchitekturen erkennen einen Überlauf, aber C ignoriert dieses. Das wird in Java genauso gehandhabt.

div0.c

```
int main() {  
    int i = 1; int j = 0;  
    int k = i / j;  
    return k;  
}
```

- Dies ist generell offen.
- Es kann zu einem undefinierten Resultat führen oder zu einem Abbruch der Programmausführung.
- Letzteres ist die Regel.

```
clonard$ gcc -std=gnu11 -Wall -o div0 div0.c && ./div0  
Arithmetic Exception (core dumped)  
clonard$
```

Datentyp	Bits	Intervall	Konstanten
signed char	8	$[-127, 127]$	<i>SCHAR_MIN</i> , <i>SCHAR_MAX</i>
unsigned char	8	$[0, 255]$	<i>UCHAR_MAX</i>
char	8		<i>CHAR_MIN</i> , <i>CHAR_MAX</i>
short	16	$[-32767, 32767]$	<i>SHRT_MIN</i> , <i>SHRT_MAX</i>
unsigned short	16	$[0, 65535]$	<i>USHRT_MAX</i>
int	16	$[-32767, 32767]$	<i>INT_MIN</i> , <i>INT_MAX</i>
unsigned int	16	$[0, 65535]$	<i>UINT_MAX</i>
long	32	$[-2^{31} + 1, 2^{31} - 1]$	<i>LONG_MIN</i> , <i>LONG_MAX</i>
unsigned long	32	$[0, 4294967295]$	<i>ULONG_MAX</i>
long long	64	$[-2^{63} + 1, 2^{63} - 1]$	<i>LLONG_MIN</i> , <i>LLONG_MAX</i>
unsigned long long	64	$[0, 2^{64} - 1]$	<i>ULLONG_MAX</i>

Einige typische Konstellationen haben sich durchgesetzt:

	32-Bit	64-Bit		
		LLP	LP	ILP
sizeof(short)	2	2	2	2
sizeof(int)	4	4	4	8
sizeof(long int)	4	4	8	8
sizeof(long long int)	8	8	8	8
sizeof(void*)	4	8	8	8

- Das 32-Bit-Modell lässt sich durch die Übersetzungsoption „-m32“ auswählen. Unter Solaris ist das die Voreinstellung.
- Mit „-m64“ gibt es einen 64-Bit-Adressraum. Unter Linux ist dies die Voreinstellung. Linux, Solaris und OS X unterstützen das LP-Modell, Microsoft Windows das LLP-Modell.

Um portabel programmieren zu können ohne Kenntnis des verwendeten Modells, empfiehlt es sich, folgende vordefinierten Datentypen zu verwenden, die über `<stddef.h>` zugänglich sind:

- ▶ `size_t` ist der ganzzahlige Datentyp ohne Vorzeichen, den **sizeof** zurückliefert. Dieser Typ kann auch grundsätzlich als Index für Arrays verwendet werden. (Bei LLP oder LP kann **int** dafür zu klein sein.)
- ▶ `uintptr_t` (ohne Vorzeichen) bzw. `intptr_t` (mit Vorzeichen) können einen Zeigerwert aufnehmen.
- ▶ `ptrdiff_t` (mit Vorzeichen) wird für die Differenz zweier Zeiger zurückgeliefert.

Normalerweise haben diese ganzzahligen Typen die gleiche Größe (nämlich die von **sizeof(void*)**), aber auf einigen historischen Architekturen (z.B. mit segmentierten Adressräumen) können diese voneinander abweichen.

- Der Datentyp **char** orientiert sich in seiner Größe typischerweise an dem Byte, der kleinsten adressierbaren Einheit.
- In `<limits.h>` findet sich die Konstante `CHAR_BIT`, die die Anzahl der Bits bei **char** angibt. Dieser Wert muss mindestens 8 betragen und weicht davon auch normalerweise nicht ab.
- Der Datentyp **char** gehört mit zu den ganzzahligen Datentypen und entsprechend können Zeichen wie ganze Zahlen und umgekehrt behandelt werden.
- Der C-Standard überlässt den Implementierungen die Entscheidung, ob **char** vorzeichenbehaftet ist oder nicht. Wer sicher gehen möchte, spezifiziert dies explizit mit **signed char** oder **unsigned char**.
- Für größere Zeichensätze gibt es den Datentyp `wchar_t` aus `<wchar.h>`.

Zeichenkonstanten werden in einfache Hochkommata eingeschlossen, etwa 'a' (vom Datentyp **char**) oder L'a' (vom Datentyp *wchar_t*). Für eine Reihe von nicht druckbaren Zeichen gibt es Ersatzdarstellungen:

<code>\a</code>	BEL	<i>alert</i> , Signalton
<code>\b</code>	BS	<i>backspace</i>
<code>\f</code>	FF	<i>formfeed</i>
<code>\n</code>	LF	<i>newline</i> , Zeilentrenner
<code>\r</code>	CR	<i>carriage return</i> , „Wagenrücklauf“
<code>\t</code>	HT	Horizontaler Tabulator
<code>\v</code>	VT	Vertikaler Tabulator
<code>\\</code>	<code>\</code>	„Fluchtsymbol“
<code>\'</code>	<code>'</code>	einfaches Hochkomma
<code>\0</code>	NUL	Null-Byte
<code>\ddd</code>		ASCII-Code (oktal)

rot13.c

```
#include <stdio.h>

const int letters = 'z' - 'a' + 1;
const int rotate = 13;
int main() {
    int ch;
    while ((ch = getchar()) != EOF) {
        if (ch >= 'a' && ch <= 'z') {
            ch = 'a' + (ch - 'a' + rotate) % letters;
        } else if (ch >= 'A' && ch <= 'Z') {
            ch = 'A' + (ch - 'A' + rotate) % letters;
        }
        putchar(ch);
    }
}
```

⟨floating-point-type-specifier⟩	→	float
	→	double
	→	long double
	→	⟨complex-type-specifier⟩
⟨complex-type-specifier⟩	→	float _Complex
	→	double _Complex
	→	long double _Complex

- In der Vergangenheit gab es eine Vielzahl stark abweichender Darstellungen für Gleitkommazahlen, bis 1985 mit dem Standard IEEE-754 (auch IEC 60559 genannt) eine Vereinheitlichung gelang, die sich rasch durchsetzte und von allen heute üblichen Prozessor-Architekturen unterstützt wird.
- Der C-Standard bezieht sich ausdrücklich auf IEEE-754, auch wenn die Einhaltung davon nicht für Implementierungen garantiert werden kann, bei denen die Hardware-Voraussetzungen dafür fehlen.

Bei IEEE-754 besteht die binäre Darstellung einer Gleitkommazahl aus drei Komponenten,

- ▶ dem Vorzeichen s (ein Bit),
- ▶ dem aus q Bits bestehenden Exponenten $\{e_i\}_{i=1}^q$,
- ▶ und der aus p Bits bestehenden Mantisse $\{m_i\}_{i=1}^p$.

- Für die Darstellung des Exponenten e hat sich folgende verschobene Darstellung als praktisch erwiesen:

$$e = -2^{q-1} + 1 + \sum_{i=1}^q e_i 2^{i-1}$$

- Entsprechend liegt e im Wertebereich $[-2^{q-1} + 1, 2^{q-1}]$.
- Da die beiden Extremwerte für besondere Kodierungen verwendet werden, beschränkt sich der reguläre Bereich von e auf $[e_{min}, e_{max}]$ mit $e_{min} = -2^{q-1} + 2$ und $e_{max} = 2^{q-1} - 1$.
- Bei dem aus insgesamt 32 Bits bestehenden Format für den Datentyp **float** mit $q = 8$ ergibt das den Bereich $[-126, 127]$.

- Wenn e im Intervall $[e_{min}, e_{max}]$ liegt, dann wird die Mantisse m so interpretiert:

$$m = 1 + \sum_{i=1}^p m_i 2^{i-p-1}$$

- Wie sich dieser sogenannten normalisierten Darstellung entnehmen lässt, gibt es ein implizites auf 1 gesetztes Bit, d.h. m entspricht der im Zweier-System notierten Zahl $1, m_p m_{p-1} \dots m_2 m_1$.
- Der gesamte Wert ergibt sich dann aus $x = (-1)^s \times 2^e \times m$.
- Um die 0 darzustellen, gilt der Sonderfall, dass $m = 0$, wenn alle Bits des Exponenten gleich 0 sind, d.h. $e = -2^{q-1} + 1$, und zusätzlich auch alle Bits der Mantisse gleich 0 sind. Da das Vorzeichenbit unabhängig davon gesetzt sein kann oder nicht, gibt es zwei Darstellungen für die Null: -0 und $+0$.

- IEEE-754 unterstützt auch die sogenannte denormalisierte Darstellung, bei der alle Bits des Exponenten gleich 0 sind, es aber in der Mantisse mindestens ein Bit mit $m_i = 1$ gibt. In diesem Falle ergibt sich folgende Interpretation:

$$m = \sum_{i=1}^p m_i 2^{i-p-1}$$
$$x = (-1)^s \times 2^{e_{min}} \times m$$

- Der Fall $e = e_{max} + 1$ erlaubt es, ∞ , $-\infty$ und *NaN* (*not a number*) mit in den Wertebereich der Gleitkommazahlen aufzunehmen. ∞ und $-\infty$ werden bei Überläufen verwendet und NaN bei undefinierten Resultaten (Beispiel: Wurzel aus einer negativen Zahl).

overflow.c

```
#include <stdio.h>
#include <math.h>

int main() {
    float ov = 1.0, uv = 1.0;
    for (int i = 0; isfinite(ov) || uv > 0; ov *= 2, uv /= 2, ++i) {
        printf("%3d %20g %20g\n", i, uv, ov);
    }
}
```

- Mit *isfinite* aus *<math.h>* kann getestet werden, ob eine Gleitkommazahl einen endlichen Wert besitzt, d.h. bei *NaN*, ∞ und $-\infty$ liefert *isfinite false*.

```
clonmel$ overflow | sed -n '1p; 128,129p; 148,$p'
0          1          1
127        5.87747e-39    1.70141e+38
128        2.93874e-39    Inf
147        5.60519e-45    Inf
148        2.8026e-45     Inf
149        1.4013e-45     Inf
clonmel$
```

```
#include <math.h>
#include <stdio.h>

const char* get_class(float f) {
    switch (fpclassify(f)) {
        case FP_INFINITE: return "infinite";
        case FP_NAN:      return "NaN";
        case FP_NORMAL:   return "normal";
        case FP_SUBNORMAL: return "subnormal";
        case FP_ZERO:     return "zero";
        default:          return "?";
    }
}

int main() {
    for (int exp = 126; exp <= 128; ++exp) {
        float f1 = exp2f(exp);
        printf("2^%d: %s\n", exp, get_class(f1));
        float f2 = exp2f(-exp);
        printf("2^-%d: %s\n", exp, get_class(f2));
    }
    printf("sqrt(-1): %s\n", get_class(sqrt(-1)));
}
```

```
clonmel$ fclass
2^126: normal
2^-126: normal
2^127: normal
2^-127: subnormal
2^128: infinite
2^-128: subnormal
sqrt(-1): NaN
clonmel$
```

- Der Bereich der normalisierten Exponenten erstreckt sich bei **float** auf $[-126, 127]$. Entsprechend lässt sich 2^{127} noch darstellen, während bei 2^{-127} bereits auf die denormalisierte Darstellung zurückgegriffen werden muss. 2^{128} ist nicht mehr darstellbar, während bei der denormalisierten Darstellung das Ende erst bei 2^{-149} erreicht wird, d.h. 2^{-150} würde zur Null werden.

- IEEE-754 gibt Konfigurationen für einfache, doppelte und erweiterte Genauigkeiten vor, die auch so von C übernommen wurden.
- Allerdings steht nicht auf jeder Architektur **long double** zur Verfügung, so dass in solchen Fällen ersatzweise nur eine doppelte Genauigkeit verwendet wird.
- Umgekehrt rechnen einige Architekturen grundsätzlich mit einer höheren Genauigkeit und runden dann, wenn eine Zuweisung an eine Variable des Typs **float** oder **double** erfolgt. Dies alles ist entsprechend IEEE-754 zulässig – auch wenn dies zur Konsequenz hat, dass Ergebnisse selbst bei elementaren Operationen auf verschiedenen konformen Architekturen voneinander abweichen können.
- Hier ist die Übersicht:

Datentyp	Bits	q	p
float	32	8	23
double	64	11	52
long double		≥ 15	≥ 63

- Rundungsfehler beim Umgang mit Gleitkomma-Zahlen sind unvermeidlich.
- Sie entstehen in erster Linie, wenn Werte nicht exakt darstellbar sind. So gibt es beispielsweise keine Repräsentierung für $0,1$. Stattdessen kann nur eine der „Nachbarn“ verwendet werden.
- Bedauerlicherweise können selbst kleine Rundungsfehler katastrophale Ausmaße nehmen.
- Dies passiert beispielsweise, wenn Werte völlig unterschiedlicher Größenordnungen zueinander addiert oder voneinander subtrahiert werden. Dies kann dann zur Auslöschung wesentlicher Bits der kleineren Größenordnung führen.

- Gegeben seien die Längen a , b , c eines Dreiecks. Zu berechnen ist die Fläche A des Dreiecks.
- Dazu bietet sich folgende Berechnungsformel an:

$$s = \frac{a + b + c}{2}$$

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

triangle.c

```
double triangle_area1(double a, double b, double c) {  
    double s = (a + b + c) / 2;  
    return sqrt(s*(s-a)*(s-b)*(s-c));  
}
```

- Bei der Addition von $a + b + c$ kann bei einem schmalen Dreieck die kleine Seitelänge verschwinden, wenn die Größenordnungen weit genug auseinander liegen.
- Wenn dann später die Differenz zwischen s und der kleinen Seitelänge gebildet wird, kann der Fehler katastrophal werden.
- William Kahan hat folgende alternative Formel vorgeschlagen, die diese Problematik vermeidet:

$$A = \frac{\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}}{4}$$

Wobei hier die Werte a , b und c so zu vertauschen sind, dass gilt:
 $a > b > c$.

triangle.c

```
#define SWAP(a,b) {int tmp; tmp = a; a = b; b = tmp;}
double triangle_area2(double a, double b, double c) {
    /* sort a, b, and c in descending order,
       applying a bubble-sort */
    if (a < b) SWAP(a, b);
    if (b < c) SWAP(b, c);
    if (a < b) SWAP(a, b);
    /* formula by W. Kahan */
    return sqrt((a + (b + c)) * (c - (a - b)) *
                (c + (a - b)) * (a + (b - c)))/4;
}
```

triangle.c

```
int main() {
    double a, b, c;
    printf("triangle side lengths a b c: ");
    if (scanf("%lf %lf %lf", &a, &b, &c) != 3) {
        printf("Unable to read three floats!\n");
        return 1;
    }
    double a1 = triangle_area1(a, b, c);
    double a2 = triangle_area2(a, b, c);
    printf("Formula #1 delivers %.16lf\n", a1);
    printf("Formula #2 delivers %.16lf\n", a2);
    printf("Difference: %lg\n", fabs(a1-a2));
    return 0;
}
```

```
dublin$ gcc -Wall -std=c99 triangle.c -lm
dublin$ a.out
triangle side lengths a b c: 1e10 1e10 1e-10
Formula #1 delivers 0.000000000000000000
Formula #2 delivers 0.500000000000000000
Difference: 0.5
dublin$
```

- Wann können zwei Gleitkommazahlen als gleich betrachtet werden?
- Oder wann kann das gleiche Resultat erwartet werden?
- Gilt beispielsweise $(x/y)*y == x$?
- Interessanterweise garantiert hier IEEE-754 die Gleichheit, falls n und m beide ganze Zahlen sind, die sich in doppelter Genauigkeit repräsentieren lassen (also **double**), $|m| < 2^{52}$ und $n = 2^i + 2^j$ für natürliche Zahlen i, j . (siehe Theorem 7 aus dem Aufsatz von Goldberg).
- Aber beliebig verallgemeinern lässt sich dies nicht.

equality.c

```
#include <stdio.h>
int main() {
    double x, y;
    printf("x y = ");
    if (scanf("%lf %lf", &x, &y) != 2) {
        printf("Unable to read two floats!\n");
        return 1;
    }
    if ((x/y)*y == x) {
        printf("equal\n");
    } else {
        printf("not equal\n");
    }
    return 0;
}
```

```
dublin$ gcc -Wall -std=c99 equality.c
dublin$ a.out
x y = 3 10
equal
dublin$ a.out
x y = 2 0.7777777777777777
not equal
dublin$
```

- Gelegentlich wird nahegelegt, statt dem $==$ -Operator auf die Nähe zu testen, d.h. $x \sim y \Leftrightarrow |x - y| < \epsilon$, wobei ϵ für eine angenommene Genauigkeit steht.
- Dies lässt jedoch folgende Fragen offen:
 - ▶ Wie sollte ϵ gewählt werden?
 - ▶ Ist der Wegfall der (bei $==$ selbstverständlichen) Äquivalenzrelation zu verschmerzen? (Schließlich lässt sich aus $x \sim y$ und $y \sim z$ nicht mehr $x \sim z$ folgern.)
 - ▶ Soll auch dann $x \sim y$ gelten, wenn beide genügend nahe an der 0 sind, aber die Vorzeichen sich voneinander unterscheiden.
- Die Frage nach einem Äquivalenztest lässt sich nicht allgemein beantworten, sondern hängt von dem konkreten Fall ab.

⟨enumeration-type-specifier⟩	→	⟨enumeration-type-definition⟩
	→	⟨enumeration-type-reference⟩
⟨enumeration-type-definition⟩	→	enum [⟨enumeration-tag⟩] „{“ ⟨enumeration-definition-list⟩ [„,“] „}“
⟨enumeration-tag⟩	→	⟨identifier⟩
⟨enumeration-definition-list⟩	→	⟨enumeration-constant-definition⟩ → ⟨enumeration-definition-list⟩ „,“ ⟨enumeration-constant-definition⟩
⟨enumeration-constant-definition⟩	→	⟨enumeration-constant⟩ → ⟨enumeration-constant⟩ „=“ ⟨expression⟩
⟨enumeration-constant⟩	→	⟨identifier⟩
⟨enumeration-type-reference⟩	→	enum ⟨enumeration-tag⟩

- Aufzählungsdatentypen sind grundsätzlich ganzzahlig und entsprechend auch kompatibel mit anderen ganzzahligen Datentypen.
- Welcher vorzeichenbehaftete ganzzahlige Datentyp als Grundtyp für Aufzählungen dient (etwa **int** oder **short**) ist nicht festgelegt.
- Steht zwischen **enum** und der Aufzählung ein Bezeichner (`<identifier>`), so kann dieser Name bei späteren Deklarationen (bei einer `<enumeration-type-reference>`) wieder verwendet werden.
- Sofern nichts anderes angegeben ist, erhält das erste Aufzählungselement den Wert 0.
- Bei den übrigen Aufzählungselementen wird jeweils der Wert des Vorgängers genommen und 1 dazuaddiert.
- Diese standardmäßig vergebenen Werte können durch die Angabe einer Konstante verändert werden. Damit wird dann auch implizit der Wert der nächsten Konstante verändert, sofern die nicht ebenfalls explizit gesetzt wird.

- Gegeben sei folgendes (nicht nachahmenswertes) Beispiel:

```
enum msglevel {
    notice, warning, error = 10,
    alert = error + 10, crit, emerg = crit * 2,
    debug = -1, debug0
};
```

- Dann ergeben sich daraus folgende Werte: *notice* = 0, *warning* = 1, *error* = 10, *alert* = 20, *crit* = 21, *emerg* = 42, *debug* = -1 und *debug0* = 0. C stört es dabei nicht, dass zwei Konstanten (*notice* und *debug0*) den gleichen Wert haben.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>

enum days { Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday, Sunday };
char* dayname[] = { "Monday", "Tuesday", "Wednesday",
                   "Thursday", "Friday", "Saturday", "Sunday"
};

int main() {
    enum days day;
    for (day = Monday; day <= Sunday; ++day) {
        printf("Day %d = %s\n", day, dayname[day]);
    }
    /* seed the pseudo-random generator */
    unsigned int seed = time(0); srand(seed);
    /* select and print a pseudo-random day */
    enum days favorite_day = rand() % 7;
    printf("My favorite day: %s\n", dayname[favorite_day]);
}
```

⟨declaration⟩	→	⟨declaration-specifiers⟩ [⟨init-declarator-list⟩]
⟨declaration-specifiers⟩	→	⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
	→	⟨type-specifier⟩ [⟨declaration-specifiers⟩]
	→	⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
	→	⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨init-declarator-list⟩	→	⟨init-declarator⟩
	→	⟨init-declarator-list⟩ „,“ ⟨init-declarator⟩
⟨init-declarator⟩	→	⟨declarator⟩
	→	⟨declarator⟩ „=“ ⟨initializer⟩
⟨declarator⟩	→	[⟨pointer⟩] ⟨direct-declarator⟩
⟨pointer⟩	→	„*“ [⟨type-qualifier-list⟩]
	→	„*“ [⟨type-qualifier-list⟩] ⟨pointer⟩

zeiger.c

```
#include <stdio.h>

int main() {
    int i = 13;
    int* p = &i; /* Zeiger p zeigt auf i; &i = Adresse von i */

    printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);

    ++i;
    printf("i=%d, *p=%d\n", i, *p);

    ++*p; /* *p ist ein Links-Wert */
    printf("i=%d, *p=%d\n", i, *p);
}
```

- Es ist zulässig, ganze Zahlen zu einem Zeiger zu addieren oder davon zu subtrahieren.
- Dabei wird jedoch der zu addierende oder zu subtrahierende Wert implizit mit der Größe des Typs multipliziert, auf den der Zeiger zeigt.
- Weiter ist es zulässig, Zeiger des gleichen Typs voneinander zu subtrahieren. Das Resultat wird dann implizit durch die Größe des referenzierten Typs geteilt.

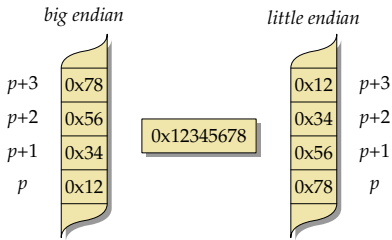
zeiger1.c

```
#include <stdio.h>

int main() {
    unsigned int value = 0x12345678;
    unsigned char* p = (unsigned char*) &value;

    for (int i = 0; i < sizeof(unsigned int); ++i) {
        printf("p+%d --> 0x%02hhx\n", i, *(p+i));
    }
}
```

- Hier wird der Speicher byteweise „durchleuchtet“.
- Hierbei fällt auf, dass die interne Speicherung einer ganzen Zahl bei unterschiedlichen Architekturen (SPARC vs. Intel x86) verschieden ist: *big endian* vs. *little endian*.



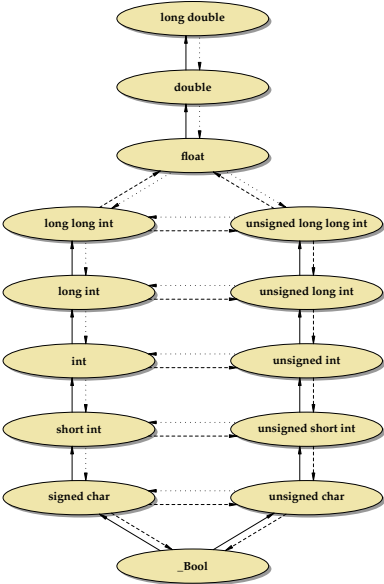
- Bei *little endian* wird das niedrigstwertige Byte an der niedrigsten Adresse abgelegt, während bei
- *big endian* das niedrigstwertige Byte sich bei der höchsten Adresse befindet.

- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schliesst auch die monadischen Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch den sogenannten Cast-Operator.

Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- ▶ Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- ▶ Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

- ▶ Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- ▶ Bei Konvertierungen zu `_Bool` ist das Resultat 0 (*false*), falls der Ausgangswert 0 ist, ansonsten immer 1 (*true*).
- ▶ Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- ▶ Umgekehrt (beispielsweise auf dem Wege von **long long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.



- Jeder Aufzählungsdattentyp ist einem der ganzzahligen Datentypen implizit und implementierungsabhängig zugeordnet. Eine Konvertierung hängt von dieser (normalerweise nicht bekannten) Zuordnung ab.
- Zeiger lassen sich in C grundsätzlich als ganzzahlige Werte betrachten. Allerdings garantiert C nicht, dass es einen ganzzahligen Datentyp gibt, der den Wert eines Zeigers ohne Verlust aufnehmen kann.
- C99 hat hier die Datentypen *intptr_t* und *uintptr_t* in `<stdint.h>` eingeführt, die für die Repräsentierung von Zeigern als ganze Zahlen den geeignetsten Typ liefern.
- Selbst wenn diese groß genug sind, um Zeiger ohne Verlust aufnehmen zu können, so lässt der Standard dennoch offen, wie sich die beiden Typen *intptr_t* und *uintptr_t* innerhalb der Hierarchie der ganzzahligen Datentypen einordnen. Aber die weiteren Konvertierungsschritte und die damit verbundenen Konsequenzen ergeben sich aus dieser Einordnung.
- Die Zahl 0 wird bei einer Konvertierung in einen Zeigertyp immer in den Null-Zeiger konvertiert.

- Bei Zuweisungen wird der Rechts-Wert in den Datentyp des Links-Wertes konvertiert.
- Dies geschieht analog bei Funktionsaufrufen, wenn eine vollständige Deklaration der Funktion mit allen Parametern vorliegt.
- Wenn diese fehlt oder (wie beispielsweise bei *printf*) nicht vollständig ist, dann wird **float** implizit zu **double** konvertiert.

Die monadischen Operatoren `!`, `-`, `+`, `~` und `*` konvertieren implizit ihren Operanden:

- ▶ Ein vorzeichenbehafteter ganzzahliger Datentyp mit einem Rang niedriger als **int** wird zu **int** konvertiert,
- ▶ Ganzzahlige Datentypen ohne Vorzeichen werden ebenfalls zu **int** konvertiert, falls sie einen Rang niedriger als **int** haben und ihre Werte in jedem Falle von **int** darstellbar sind. Ist nur letzteres nicht der Fall, so erfolgt eine implizite Konvertierung zu **unsigned int**.
- ▶ Ranghöhere ganzzahlige Datentypen werden nicht konvertiert.

Die gleichen Regeln werden auch getrennt für die beiden Operanden der Schiebe-Operatoren `<<` und `>>` angewendet.

Bei dyadischen Operatoren mit numerischen Operanden werden folgende implizite Konvertierungen angewendet:

- ▶ Sind die Typen beider Operanden vorzeichenbehaftet oder beide ohne Vorzeichen, so findet eine implizite Konvertierung zu dem Datentyp mit dem höheren Rang statt. So wird beispielsweise bei einer Addition eines Werts des Typs **short int** zu einem Wert des Typs **long int** der erstere in den Datentyp des zweiten Operanden konvertiert, bevor die Addition durchgeführt wird.
- ▶ Ist bei einem gemischten Fall (**signed** vs. **unsigned**) in jedem Falle eine Repräsentierung eines Werts des vorzeichenlosen Typs in dem vorzeichenbehafteten Typ möglich (wie etwa typischerweise bei **unsigned short** und **long int**), so wird der Operand des vorzeichenlosen Typs in den vorzeichenbehafteten Typ des anderen Operanden konvertiert.
- ▶ Bei den anderen gemischten Fällen werden beide Operanden in die vorzeichenlose Variante des höherrangigen Operandentyps konvertiert. So wird beispielsweise eine Addition bei **unsigned int** und **int** in **unsigned int** durchgeführt.

C sieht einige spezielle Attribute bei Typ-Deklarationen vor. Darunter ist auch **const**:

⟨declaration-specifiers⟩	→	⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
	→	⟨type-specifier⟩ [⟨declaration-specifiers⟩]
	→	⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
	→	⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨type-qualifier⟩	→	const
	→	volatile
	→	restrict
	→	<u>_Atomic</u>

Die Verwendung des **const**-Attributs hat zwei Vorteile:

- ▶ Der Programmierer wird davor bewahrt, eine Konstante versehentlich zu verändern. (Dies funktioniert aber nur beschränkt.)
- ▶ Besondere Optimierungen sind für den Übersetzer möglich, wenn bekannt ist, dass sich bestimmte Variablen nicht verändern dürfen.

const.c

```
#include <stdio.h>

int main() {
    const int i = 1;

    i++;          /* das geht doch nicht, oder?! */
    printf("i=%d\n", i);
}
```

- Der gcc beschränkt sich selbst dann nur auf Warnungen, wenn Konstanten offensichtlich verändert werden.