Teil 4: Numerische Datentypen

Gliederung

Speicherklassen

Numerische Typen

Typumwandlung

Felder und Zeichenketten

Speicherklassen

Vereinbarung = Deklaration + Definition

- **Deklaration** Bekanntmachung von Name ↔ Typ Verbindung
- **Definition** Speicherreservierung für Variablen bzw. Funktionen
  Deklaration ist automatisch eingeschlossen

```
extern int var1;          // Deklaration

int var2;                 // Deklaration + Definition
                          // = Vereinbarung

int var3 = 5;             // Vereinbarung
                          // mit Initialisierung
```

Beispiel 1: Wo liegt der Fehler?

```
int main()
{
   int a;

   a = 0;
   {
      int b;

      a = 3;
      b = 3;
   }
   b = 0;      //  Zugriff außerhalb des Gültigkeitsbereichs

   return 0;
}
```

Beispiel 2: Was gibt folgendes Programm aus?

```
int main()
{
   int a;

   a = 1;
   {
      int a;    // lokale Definition verdeckt gleichnamigen Bezeichner

      a = 3;
      printf("%d", a);
   }
   printf("%d", a);

   return 0;
}
```

Gültigkeit eines Objekts ergibt sich aus

- **Sichtbarkeit**

  - bzgl. einer bestimmten Position im Programm
  - Zugriff auf über dessen Namen möglich / nicht möglich (verdeckt)

- **Lebensdauer**

  - während seiner Lebensdauer besitzt ein Objekt einen Speicherplatz

- **Globaler** vs. **Lokaler Definition**

  - bedingt wiederum auch Sichtbarkeit und Lebensdauer

Speicherklasse bedingt:

- **Speicherort** (Arbeitsspeicher / Prozessorregister)
- **Gültigkeitsbereich** (Block / Modul / Programm)
- **Lebensdauer** (Blockeintritt bis -austritt / Programmstart bis -ende)
- **Initialisierung** (implizit / einmalig / bei jedem Blockeintritt)

4 Speicherklassen

```
auto    register    static    extern
```

- Explizite Angabe bei Definition:

```
auto int a;
```

auto

- Standard für **lokale** Definitionen
- Gültigkeit und Lebensdauer auf Funktion / Block beschränkt
- keine automatische Initialisierung

```
int funktion()
{
   int n = 3;
   ...
   return 0;
}
```

```
int funktion()
{
   auto int n = 3;
   ...
   return 0;
}
```

register (veraltet)

- Speicherung in **Prozessorregister**
- Geschwindigkeitsvorteil
- sonst wie **auto**

```
register int n;
```

static

- Gültigkeit auf Block bzw. Modul beschränkt
- Lebensdauer **gesamte Programmlaufzeit**
- implizite Initialisierung mit 0 (bzw. NULL bei Zeigern)

- Beispiel:

```
void f()
{
   int a;

   a++;
   printf(" Aufruf Nummer: %d\n", a);
}

int main()
{
   f();
   f();
   f();
   return 0;
}
```

static

- Gültigkeit auf Block bzw. Modul beschränkt
- Lebensdauer **gesamte Programmlaufzeit**
- implizite Initialisierung mit 0 (bzw. NULL bei Zeigern)

- Beispiel:

```
int a;                          // globale Variable

void f()
{
   a++;
   printf("Aufruf Nummer: %d\n", a);
}

int main()
{
   f();
   f();
   f();
   return 0;
}
```

static

- Gültigkeit auf Block bzw. Modul beschränkt
- Lebensdauer **gesamte Programmlaufzeit**
- implizite Initialisierung mit 0 (bzw. NULL bei Zeigern)

- Beispiel:

```
void f()
{
   static int a;            // statische Variable

   a++;
   printf("Aufruf Nummer: %d\n", a);
}

int main()
{
   f();
   f();
   f();
   return 0;
}
```

extern

- Standard für **globale** Definitionen
- ermöglicht Verwendung in anderen Modulen
- implizite Initialisierung mit 0 (bzw. NULL bei Zeigern)

- **Achtung**: Angabe des Schlüsselwortes **`extern`** ist nicht bei der Definition, sondern bei der Deklaration erforderlich!

Definition in Modul A:

```
int n;
```

Deklaration (Verwendung) in Modul B:

```
extern int n;
```

Beispiel: Programm aus mehreren Quelldateien

modul2.h

```
extern int i;

int sum(int, int);
```

modul3.h

```
void abc();
```

modul1.c

```
#include <stdio.h>
#include "modul2.h"
#include "modul3.h"

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
#include "modul2.h"

int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
#include "modul2.h"
#include "modul3.h"

void abc()
{
  i = 10;
}
```

Übersicht Speicherklassen

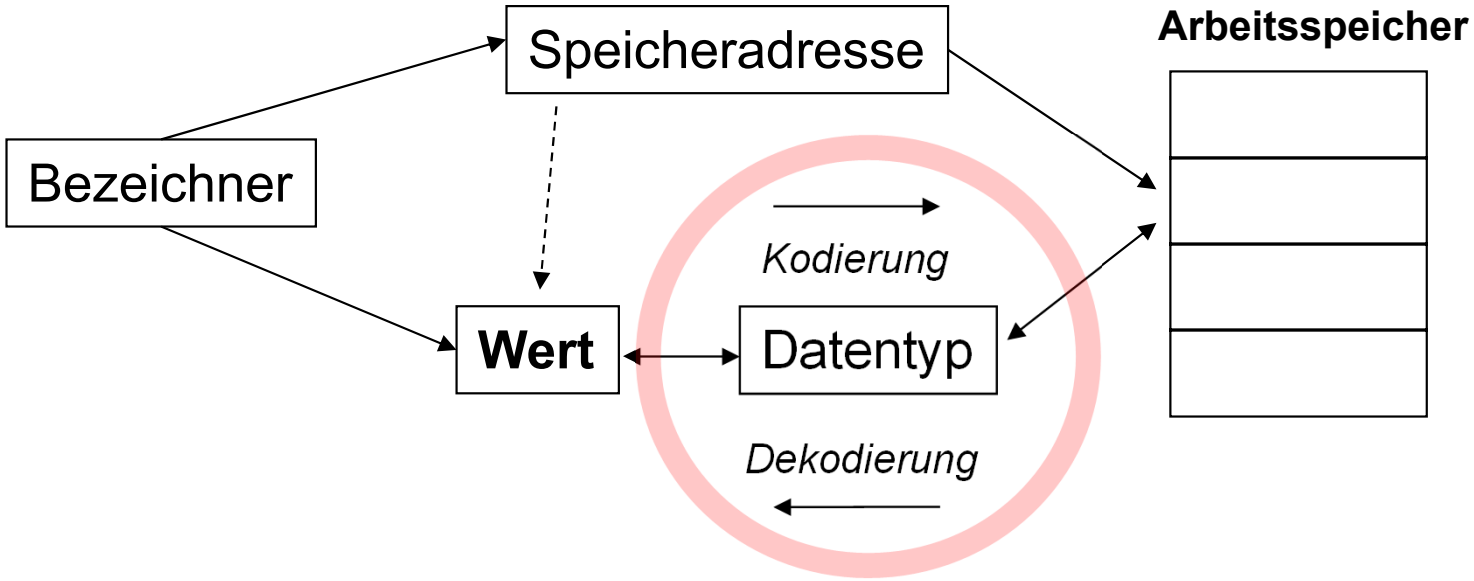
Eigenschaft	auto	register	static	extern
Definition in Bezug auf einen Block {...}	innerhalb	innerhalb	innerhalb oder außerhalb	außerhalb
Speicherort	Arbeitsspeicher	Register oder Arbeitsspeicher	Arbeitsspeicher	Arbeitsspeicher
Gültigkeitsbereich	Block	Block	Block / Modul	Modul (erweiterbar auf andere Module)
Lebensdauer	Blockeintritt bis Blockaustritt	Blockeintritt bis Blockaustritt	Programm-laufzeit	Programm-laufzeit
Initialisierung	bei jedem Blockeintritt	bei jedem Blockeintritt	implizit, einmalig beim Programmstart	implizit, einmalig beim Programmstart

Welche Ausgabe erzeugt "mystery.c"?

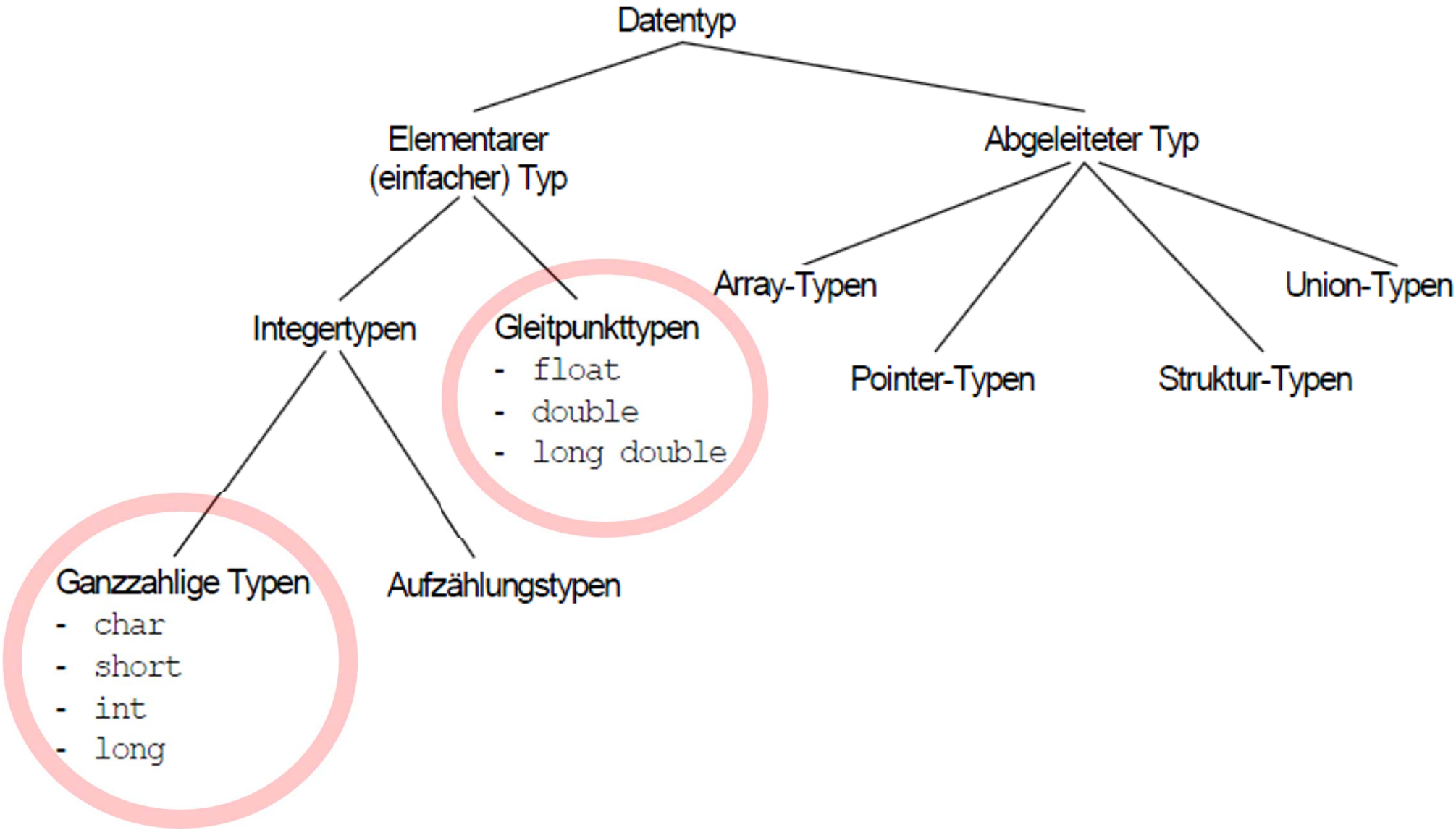
```
#include <stdio.h>

int b;

int main()
{
    int a;

    printf("%d %d\n", a, b);
    {
        int a = 6;

        printf("%d %d\n", a, b);
        a = 4;
        b = 3;
    }
    printf("%d %d\n", a, b);

    return 0;
}
```

Numerische Typen

Variablenmodell



Typ-Klassifikation



ANSI C89: Elementare Datentypen

Typ	Wertebereich (Genauigkeit)	Größe	E / A
int	- 32 768 ... 32 767 bei 16 Bit Maschinen - 2 147 483 648 ... 2 147 483 647 32 Bit	2 Byte 4 Byte	%d
unsigned int	0 ... 65 535 bei 16 Bit Maschinen 0 ... 4 294 967 295 bei 32 Bit Maschinen	2 Byte 4 Byte	%u
short int	- 32 768 ... 32 767	2 Byte	%d
unsigned short int	0 ... 65 535	2 Byte	%u
long int	- 2 147 483 648 ... 2 147 483 647	4 Byte	%ld
unsigned long int	0... 4 294 967 295	4 Byte	%lu
char	alle Zeichen im ASCII Code	1 Byte	%c
char	-128 ... 127	1 Byte	%d
unsigned char	0 ... 255	1 Byte	%u
float	1,2 E-38 ... 3,4 E+38 (6 Stellen)	4 Byte	%f
double	2,3 E-308 ... 1,7 E+ 308 (15 Stellen)	8 Byte	%lf
long double	3.4 E-4932 ... 1.1 E+4932 (19 Stellen)	10 Byte	%lf
void	leerer Typ	0 Byte	

ANSI C99: Erweiterungen

Typ	Wertebereich	Größe
_Bool	0 und 1	1 Byte
long long	–9223372036854775808 bis 9223372036854775807	8 Byte
unsigned long long	0 bis 18446744073709551615	8 Byte

Regeln für die Compiler-Konformität:

short int	≤	int
long int	≥	int
int	≥	2 Byte
long long int	=	8 Byte

Bestimmung der Typgröße zur Laufzeit

- Ermittlung zur Laufzeit, wieviel Speicher (in Bytes) ein Variable/Konstante bzw. ein Datentyp tatsächlich belegt:

```
int sizeof(Variable);
int sizeof(Konstante);
int sizeof(Datentyp);
```

- Beispiel:

```
#include <stdio.h>

int main()
{
   long int x;

   printf("Anzahl Bytes des Typs long int: %d\n", sizeof(long int));
   printf("Anzahl Bytes der Variablen x: %d\n", sizeof(x));
   return 0;
}
```

Ganzzahltypen: Interne Repräsentation

unsigned		signed	
0	0 0 0 0	-8	1 0 0 0
1	0 0 0 1	-7	1 0 0 1
2	0 0 1 0	-6	1 0 1 0
3	0 0 1 1	-5	1 0 1 1
4	0 1 0 0	-4	1 1 0 0
5	0 1 0 1	-3	1 1 0 1
6	0 1 1 0	-2	1 1 1 0
7	0 1 1 1	-1	1 1 1 1
8	1 0 0 0	0	0 0 0 0
9	1 0 0 1	1	0 0 0 1
10	1 0 1 0	2	0 0 1 0
11	1 0 1 1	3	0 0 1 1
12	1 1 0 0	4	0 1 0 0
13	1 1 0 1	5	0 1 0 1
14	1 1 1 0	6	0 1 1 0
15	1 1 1 1	7	0 1 1 1

Vorzeichenbit

Ganzzahltypen: Zweierkomplement

- A ↔ B: Invertierung aller Bits und Addition von 1

	$_1$0000	$_1$0000	$_1$0000		$_1$0000
B	− 0001	− 0010	− 0011		− 0111
	———	———	———	.....	———
A	1111	1110	1101		1001

- zu jedem Bit-Tupel gibt es ein Bit-Tupel so dass gilt:

B	=	{0 B_{n-1} … B_1}
A	=	{1 A_{n-1} … A_1}
A + B	=	1{0 … 0}

- wenn man den Überlauf ignoriert, ist A = −B
- A heißt "Zweier-Komplement" zu B

Ganzzahltypen: Beispiel

```
#include <stdio.h>

int main()
{
   short int a = 50;
   short int b = 1000;
   short int c;

   c = a * b;
   printf("%d * %d = %d\n", a, b, c);

   return 0;
}
```

Mögliche Fehler bei Rechenoperationen: **Überlauf**

Reelle Zahlen

- Gebrochene Zahlen - wie Speichern?
- Bestandteile: **Vorzeichen**
  Ziffernfolge
  Komma (bzw. Position)
- In Exponentialschreibweise: $\pm\, \mathbf{y} \bullet 10^{\mathbf{z}}$
  - **±** Vorzeichen
  - **y** Stellenzahl bestimmt Genauigkeit
  - **z** Stellenzahl bestimmt Größe
- Mantisse **y** = Festkommazahl durch Normierung auf $1 \leq \mathbf{y} < 10$
- Exponent **z** = Position des Kommas

➡ Fließkommazahl = **Vorzeichenbit + Mantisse + Exponent**

Fließkommatypen

- IEEE 754-1985

Typ	Bits	Zahlenbereich	Genauigkeit
float	32	$+/-3.4*10^{-38}$ ... $+/-3.4*10^{38}$	7
double	64	$+/-1.7*10^{-308}$ ... $+/-1.7*10^{308}$	15
long double	80	$+/-3.4*10^{-4932}$ ... $+/-3.4*10^{4932}$	19

- beliebige Genauigkeit nicht (für alle Zahlen) möglich
- mit 32 Bits (float) existieren nur 2^{32} = 4.294.967.296 Möglichkeiten, eine Zahl darzustellen

Fließkommatypen

	Mant. (Bit) inkl. ±	Exp. (Bit)	kleinste darstellbare positive Zahl größte darstellbare Zahl prozentuale Genauigkeit
float	24	8	$1.175494351 \cdot 10^{-38}$ $3.402823466 \cdot 10^{+38}$ $1.192092896 \cdot 10^{-07}$
double	53	11	$2.2250738585072014 \cdot 10^{-308}$ $1.7976931348623157 \cdot 10^{+308}$ $2.220446049250313 \cdot 10^{-16}$
long double 80 Bit (Intel-Architektur)	65	15	$3.362103143112093506262 \cdot 10^{-4932}$ $1.189731495357231765021 \cdot 10^{+4932}$ $1.084202172485504434007 \cdot 10^{-19}$
long double 128 Bit	113	15	$3.362103143112093506262677817321752602 \cdot 10^{-4932}$ $1.189731495357231765085759326628007016 \cdot 10^{+4932}$ $1.925929944387235853055977942584927319 \cdot 10^{-34}$

Fließkommatypen: Programmbeispiel

```
#include <stdio.h>
int main()
{
   float i;
   for (i = 0.0; i <= 1.0; i = i + 0.1)
      printf("%f\n", i);
   return 0;
}
```

Fließkommatypen: Codierung der Mantisse y

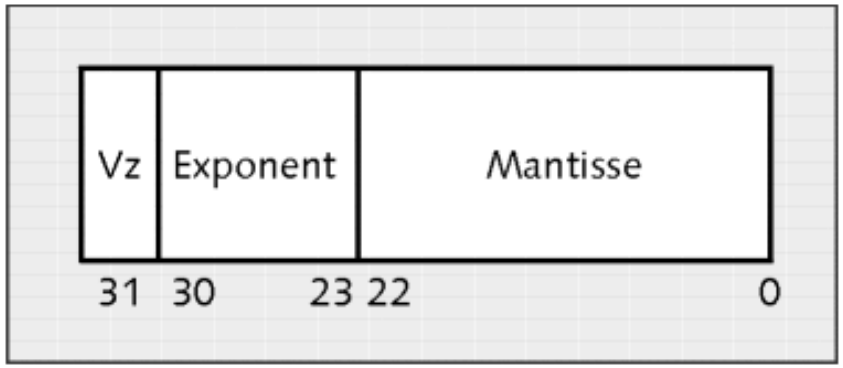
$$\pm y \cdot 10^{z}$$

dez 2^x	binär	dezimal
2^5	100000	32
2^4	10000	16
2^3	1000	8
2^2	100	4
2^1	10	2
2^0	1	1
2^{-1}	0.1	0.5
2^{-2}	0.01	0.25
2^{-3}	0.001	0.125
2^{-4}	0.0001	0.0625
2^{-5}	0.00001	0.03125

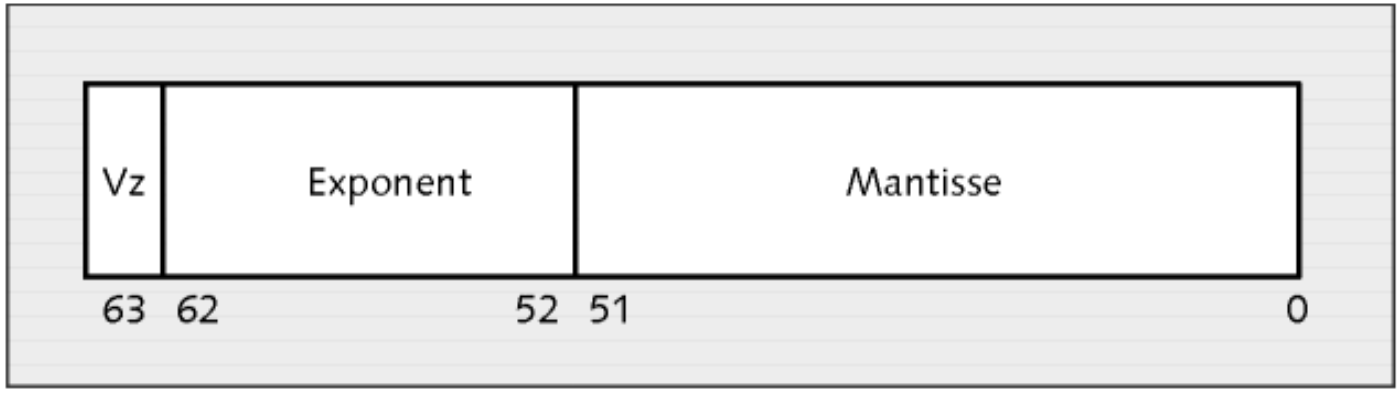
0.1 ?

Kodierungsschema für float und double

float



double



float Kodierung (nach IEEE 754)

```
SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
|                                  |
MSB                                LSB
```

S	Sign (Vorzeichenbit): 0 = Plus, 1 = Minus
EEEEEEEE	Exponent (8 Bit), Bereich von 0...255
MM...MMM	Normalisierte Mantisse (23 Bit): Vorkommabit ist immer 1 **1 ≤ Mantisse < 2** Wertigkeit der Bits: ½, ¼, usw.

$$\mathbf{Wert}_{\mathbf{dez}} = (-1)^{S} * 1,MM...MMM * 2^{EEEEEEEE-127}$$

float Kodierung (nach IEEE 754)

SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM

$$\text{Wert}_{dez} = (-1)^{S} * 1,MM...MMM * 2^{EEEEEEEE-127}$$

Reelle Zahlen: Schleifenbedingungen

```
float i;

for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);
```

Reelle Zahlen: Schleifenbedingungen

```
float i;

for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);


// besser

float i;

for (i = 0.0; i <= 1.01; i = i + 0.1)
  printf("%f\n", i);
```

Reelle Zahlen: Schleifenbedingungen

```
float i;

for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);


// besser

float i;

for (i = 0.0; i <= 1.01; i = i + 0.1)
  printf("%f\n", i);


// richtig

int i;

for (i = 0; i <= 10; i = i + 1)
  printf("%f\n", i / 10.0);
```

Reelle Zahlen: Bedingungsabfragen

```
if (d == 7.123)
{
   ...
}
else
{
   ...
}
```

Frage: Wird das gewünschte Ergebnis erreicht?

Reelle Zahlen: Numerische Auslöschung

```
#include <stdio.h>

int main()
{
   float a = 1.234567E-9;
   float b = 1.0;
   float c = -1.0;
   float s1, s2;

   s1 = a + b;
   s1 = s1 + c;

   s2 = b + c;
   s2 = s2 + a;

   printf("%.15lf\n", s1);
   printf("%.15lf\n", s2);
   return 0;
}
```

Reelle Zahlen: Zusammenfassung Rechenfehler

Problem 1: **Rundungsfehler**

Problem 2: **Numerische Auslöschung**
Subtraktion 2er fast gleich großer Werte
→ signifikante Stellen heben sich auf

Problem 3: **Reihenfolge der Operationen relevant**

Problem 4: **Überlauf** (Overflow): Division durch betragsmäßig zu kleinen Wert

Standardbibliothek <math.h>

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

Trigonometrische, Hyperbel-, Logarithmische, Exponential-, Potenz-, Rundungsfunktionen, u.a.:

```
double sin(double x);
double log(double x);
double pow(double base, double exponent);
double round(double x);
...
```

Typumwandlung

Typumwandlung (cast)

- Compiler nimmt Typumwandlung in Ausdrücken vor → **impliziter cast**
- Typumwandlung wird vom Programmierer erzwungen → **expliziter cast**
- Konvertierung führt unter Umständen zu Datenverlust!

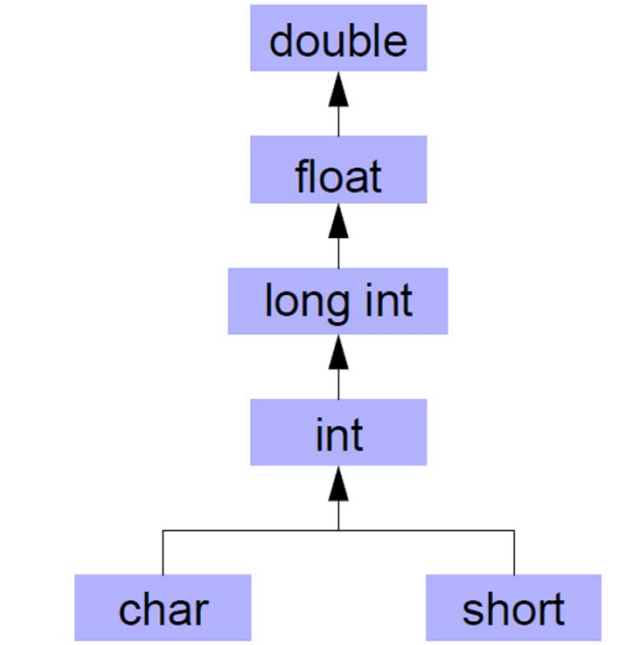
Beispiel: Welche Typumwandlung liegt vor? (implizit / explizit / keine)

```
int a;
int b = 77;
double e;
double d = 1.23;
char f = 'A';

a = f;          /* keine */
e = b;          /* implizit */
f = b;          /* keine */
a = d;          /* implizit */
a = (int) d;    /* explizit */
```

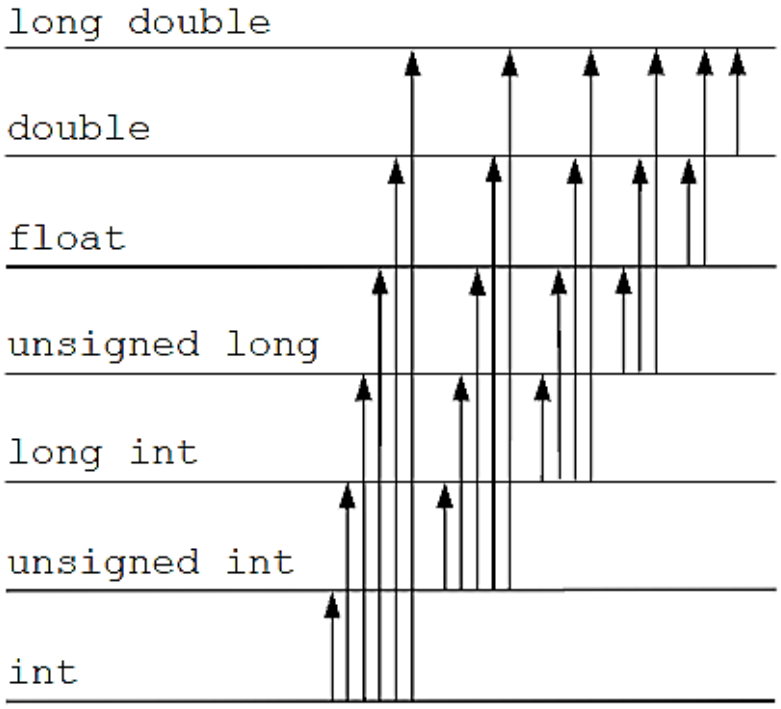
Implizite Typumwandlung

- Operanden eines binären Operators haben unterschiedliche Datentypen
  → ein gemeinsamer Datentyp wird **implizit** gebildet

- Bildung des gemeinsamen Typs erfolgt typübergreifend



Hierarchie der Typen (Auszug)



Typumwandlung: Schleifen

Programmbeispiel: Wie oft wird die folgende Schleife durchlaufen?

```
int main()
{
    int i;
    unsigned int grenze = 10;

    for (i = -1; i < grenze; i++)
        printf("%d\n", i);

    return 0;
}
```

Typumwandlung: Ausdrücke

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größtem" Typ
- Ausnahmefälle erfordern expliziten cast

```
int zaehler = 8;
int nenner = 16;
double ergebnis;
double faktor = 1.5;

ergebnis = zaehler / nenner * faktor;

ergebnis = (double) zaehler / (double) nenner * faktor;

ergebnis = (double) zaehler / nenner * faktor;

ergebnis = (double) (zaehler / nenner) * faktor;

ergebnis = faktor * zaehler / nenner;
```

Felder und Zeichenketten

Felder (Arrays)

- Verbundtypen
- variable Anzahl von **Elementen** eines Datentyps
- sequenzielle Ablage im Speicher
- ein- oder mehrdimensional
- Anwendung: Vektoren,
  Matrizen,
  Tabellen,
  Zeichenketten (**Strings**)
- Definition

```
datentyp bezeichner[anzahl];
```

Beispiele

```
char c;          // 1 Zeichen
char s[20];      // 20 Zeichen

int i;           // 1 ganze Zahl
int j[100];      // 100 ganze Zahlen (= 400 Bytes)

double f;        // 1 Fließkommazahl
double g[100]; // 100 Fließkommazahlen (= 800 Bytes)
```

Beispielanwendung

```
/* Feld mit der Summe der natuerlichen Zahlen von 0 bis 100
   belegen */

#define MAX 100


int main()
{
   int sum[MAX];
   sum[0] = 0;

   for (int i = 1; i < MAX; i++)
      sum[i] = sum[i – 1] + i;


   return 0;
}
```

Besonderheiten von Feldern

- Anzahl der Elemente muss eine Konstante sein:

```
int feld[10];
```

- Variable Feldgrenzen sind ab C99 für lokale Feldvariablen möglich:

```
{
 int x = 7;
 ...
 int feld[x];
 ...
}
```

- Zugriff über Index, gültiger Bereich von **0** bis **anzahl-1**:

```
feld[0] = 123;
...
feld[9] = 456;
feld[10] = 789;
```

Initialisierung von Feldern

- Initialisierung:

```
int feldA[4] = {0};              // Auto-Initialisierung
                                 // setzt alle Elemente auf 0
int feldB[4] = {0, 1, 3, 5};     // Voll-Initialisierung
int feldC[] = {0, 1, 3, 5};      // Compiler zaehlt selbst
```

- Vergleiche zwischen Arrays `if (feld1 == feld2)` sind **nicht** möglich!
- Zuweisungen der Form `feld2 = feld1;` sind ebenfalls **nicht** möglich!

→ Lösung: 1) elementweise vergleichen bzw. kopieren

2) gesamten Speicherbereich kopieren

Speicherabbild

```
int alpha[5];
```

alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int
?	?	?	?	?

```
for (int i = 0; i < 5; i++)
    alpha[i] = i * i;
```

alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int	
0	1	4	9	16	723

alpha[5] = 723

Beispiele: Arbeiten mit Feldern

Beispiel 1: Elemente des Arrays `alpha` ausgeben:

```
for (int i = 0; i < 5; i++)
{
    printf ("%d\n", alpha[i]);
}
```

Beispiel 2: Mittelwert aller Werte des Arrays berechnen

```
float summe = 0.0;

for (int i = 0; i < 5; i++)
{
    summe += alpha[i];
}
printf ("\nDer Mittwelwert ist: %f", summe / (i + 1));
```

Strings

- sind Arrays vom Typ **`char[]`**
- Besonderheit: Null-Terminierung durch '\0'-Byte
- Initialisierung:

```
char wochentag[] = "Samstag";
```

ist gleichbedeutend mit

```
char wochentag[] = {'S','a','m','s','t','a','g','\0'};
```

oder auch

```
char wochentag[8];

wochentag[0] = 'S';
wochentag[1] = 'a';
...
wochentag[7] = '\0';
```

Auswahl von Bibliotheksfunktionen in <string.h>

- Länge eines Strings ermitteln

```
a = strlen(s1);              // ergibt Länge, ohne '\0'-Byte
```

- String kopieren

```
strcpy(ziel_string, quell_string);
```

- String an einen anderen String anhängen

```
strcat(ziel_string, quell_string);
```

- Strings vergleichen

```
int a;
a = strcmp(s1, s2)

-1 wenn   s1 < s
 0 wenn   s1 == s2
 1 wenn   s1 > s2
```