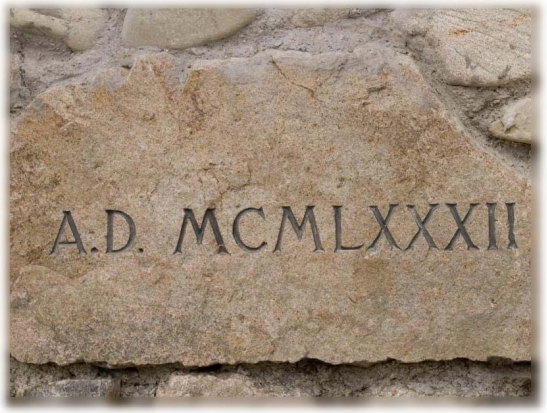
INF1
Zahlensysteme, Datentypen, Operatoren, einfache Ein-/Ausgabe

School of Engineering



- Variablen: Deklaration, Definition, Initialisierung
- Ganzzahltypen
- Fliesskommatypen
- Arithmetische Operatoren und Typumwandlung
- Mathematische Funktionen
- Strings und einfache Ein-/Ausgabe

Variablen: Deklaration, Definition, Initialisierung

Beispiel

```c
/**
 * Nach Anforderung einer Temperatur in Fahrenheit wird die Temperatur
 * in Celsius berechnet und ausgegeben.
 *
 * @author  Gerrit Burkert
 * @version 17-jul-2012
 */
#include <stdio.h>

int main (void) {
    double celsius, fahrenheit;

    // Eingabe in Fahrenheit
    printf("Temperatur in Fahrenheit eingeben: ");
    scanf("%lf", &fahrenheit);

    // Ausgabe in Celsius
    celsius = 5.0 * (fahrenheit - 32.0) / 9.0;
    printf("Temperatur in Celsius: %f\n", celsius);

    return 0;
}
```

Variablen vereinbaren

- Um eine Variable nutzen zu können, muss sie zunächst *vereinbart* werden
- Beispiel: double celsius;
- Variablenvereinbarungen stehen am Anfang eines Blocks vor den ausführbaren Anweisungen (nach der öffnenden *geschweiften Klammer*)
- Sie sind dann in diesem Block gültig:
- Ab C99 müssen Variablen nicht mehr unbedingt am Anfang eines Block stehen

Deklaration und Definition

- Eine Variablenvereinbarung hat die Form:

  Typbezeichnung Variablenname; /* Erklaerung */

- Beispiele

```
int zahl;                    /* Zaehler fuer eingegebene Werte */
double wertAlt, wertNeu;     /* Wert vor und nach Bearbeitung  */
```

- Zwei Aspekte
  - *zahl* ist eine Variable vom Typ int *Deklaration*
  - Für *zahl* wird Speicherplatz reserviert *Definition*

Definition mit Initialisierung

- Man kann einer Variablen auch gleich einen Initialwert zuweisen

  Typbezeichnung Variablenname = Wert;

- Beispiele

```
int zahl = 42;
double wertAlt = 3.14, wertNeu = 3.14;
```

Deklaration und Definition

School of Engineering

- Deklaration und Definition/Initialisierung von Variablen kommen meist zusammen vor (die Definition enthält die Deklaration)
- Nicht initialisierte Variablen haben keinen definierten Wert !
- Ausnahme: Deklaration einer Variablen, die in einem anderen Modul definiert ist (extern-Deklaration, mehr dazu später)
- Variablen können auch ausserhalb von Funktionen definiert werden
  - Globale Variablen, sie können dann von mehreren Funktionen aus angesprochen werden
  - Alle Variablen, die nur von einer Funktion gebraucht werden lokal deklarieren!

Namen von Variablen

- Sie müssen aus Buchstaben, Ziffern und Unterstrichen bestehen
- Es wird zwischen Gross- und Kleinschreibung unterschieden
- Umlaute und andere Sonderzeichen sind nicht erlaubt
- Sie dürfen nicht mit Ziffern beginnen
- Sie sollen nicht mit einem Unterstrich beginnen
  - oft beginnen von Werkzeugen generierte Variablen mit "_"
- Sie dürfen nicht Schlüsselwörtern entsprechen
- Sie sollten klein geschrieben werden

Namen von Variablen

- Richtig
  - saldo
  - ueberlauf
  - grenzwert
  - celsiusToFahrenheit

- Falsch:
  - 6namen (beginnt mit Ziffer)
  - char (Schlüsselwort)
  - Übergabe (Umlaut)
  - Berechne_a/b (Sonderzeichen)

Namenskonventionen

- Variablennamen sollen den Zweck der Variablen beschreiben (CamelCase):
  int dieErsteZahl;

- In speziellen Fällen, wenn der Zweck klar ist (x-Koordinate, Laufvariable i), können auch kurze Namen verwendet werden:
  int x, y;
  int i;

- Variablen beginnen mit einem Kleinbuchstaben, bei mehreren Wörtern werden die Anfangsbuchstaben ab dem zweiten Wort gross geschrieben:
  int dieErsteZahl;

Namenskonventionen

- Eine Alternative ist, die Wörter mit "_" zu verbinden
  int die_erste_zahl;
- Wichtig: Regeln für konsistente Wahl der Namen festlegen und konsequent einhalten
- Konstanten werden komplett in Grossbuchstaben geschrieben:
  const int MAXIMALGEWICHT = 100; /* kg */
- Bei grossen Projekten manchmal: Kürzel für Modul voranstellen
  - verträgt sich schlecht mit ungarischer Notation
- C++ kennt sog. Namespaces

Modul = Gruppe von zusammengehörigen Programmen (später)

```
namespace a
{
  int foo() { return 5; }
}
int main () {
  cout << a::foo() << '\n';
}
```

Namenskonventionen - Ungarische Notation

- Ungarische Notation
  - Prefix unterscheidet den Datentyp
- Findet sie oft in alten Quellen
  - erste Compiler hatten keinen Test auf Typenzuweisungskompatibiltät

```
iSize // Integer
u16Size // 16 Bit Integer
dWord // Double
pStruct // Pointer to ...
lpStruct // long Pointer to ...
szChar // String
```

Programmierstil

- Was bedeutet das?

```
int p,q,r;
```

innerhalb von Funktionen erlaubt

- Etwas besser:

```
int accountNumber;
int balanceOwed;
```

- Gut:

```
int accountNumber;    // Index for account table
int balanceOwed;      // Total owed us (in pennies)
```

Globale Variablen (ausserhalb von Funktionen) immer mit Kommentar

Programmierstil

- Ähnliche Namen nach Möglichkeit vermeiden
- Schlecht:

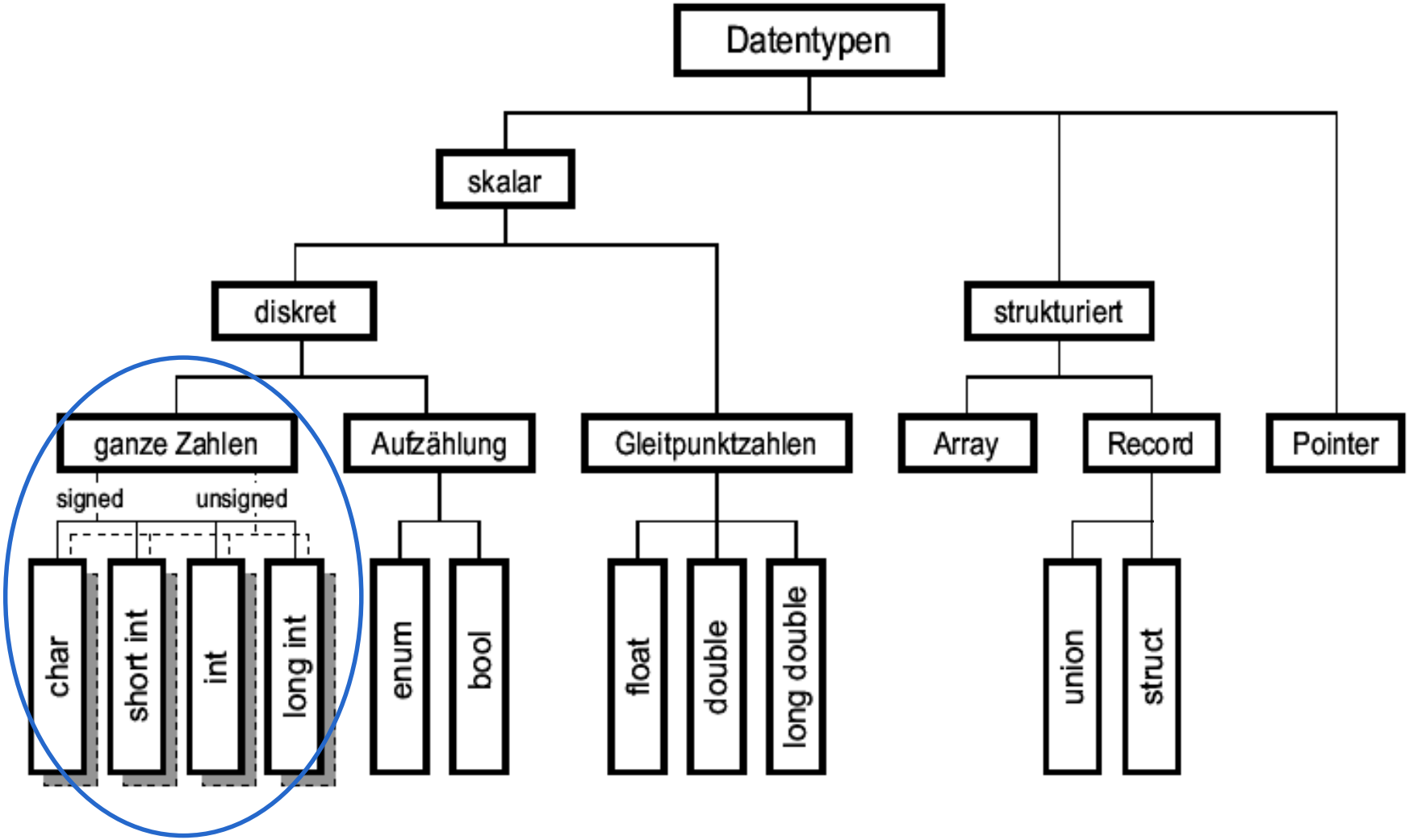
```
int total, totals;                  // ...
```

- Besser:

```
int entryTotal, allTotal;     // ...
```

Ganzzahltypen

Datentypen Übersicht



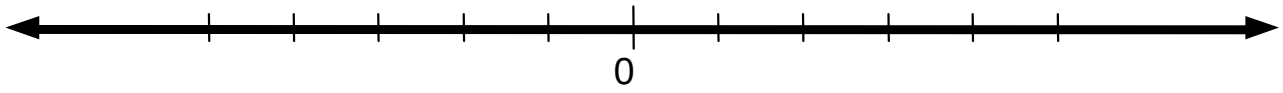
Ganzzahltypen

`signed int`	`unsigned int`
`signed short int`	`unsigned short int`
`signed long int`	`unsigned long int`

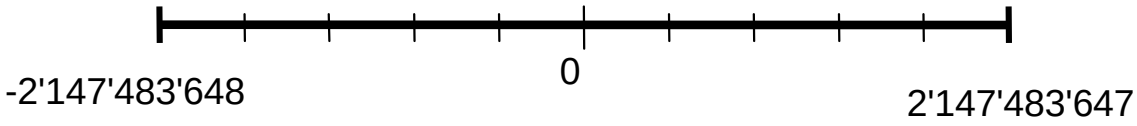
- Genaue Grössen/Wertebereiche compilerabhängig

int: eine Computer-Repräsentation von $\mathbb{Z}$

- $\mathbb{Z}$ Wertebereich: ganze Zahlen von -∞ bis +∞
  - Problem: ∞ ist schon eine "verdammt" grosse Zahl und braucht ziemlich viel Speicherplatz auf dem Computer
- Lösung: Einschränken des Wertebereichs
- **Zahlenstrahl Darstellung**
  - ganze Zahlen in der Mathematik



  - ganze Zahlen auf dem Computer: **int**



- der Computer kann alle Berechnungen innerhalb dieses Wertebereichs **exakt** durchführen: $3_{int} + 4_{int} = 7_{int}$

Datentyp: int

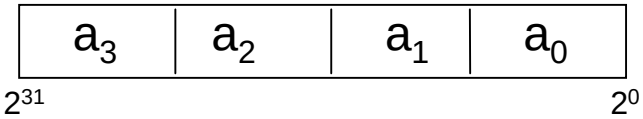
- Datentyp int

  Beispiele:
  - 5, -193, 0x6B (Darstellung als Hexadezimalzahl = 10710)

Datentyp	`int`
Anzahl Bits	`32`
Wertebereich	`-2147483648` bis `2147483647`
Integer-Berechnungen	Immer exakt

- Es gibt noch weitere Datentypen für ganze Zahlen
  - Wertebereich kann überschritten werden -> **long** hat doppelt so grossen Wertebereich, **short** den halben

Implementation von int im Computer



- $a_0 + a_1 * 256 + a_2 * 256^2 + a_3 * 256^3$
- Operationen a + b: $a_0 + b_0 + (a_1 + b_1) * 256 + (a_2 + b_2) * 256^2 + (a_3 + b_3) * 256^3$
- Problem: es können so nur positive Zahlen dargestellt werden
  - Trick: schränke den Bereich ein und nehme das vorderste Bit als Vorzeichen

- 2-er - Komplement Darstellung
  - IF zahl >= 0 THEN setze z_0 bis z_{n-1} als Binärzahl
  - IF zahl < 0 THEN setze z_n und wähle z' so dass z = $(-2^n) + z'$; setze z'_0 bis z'_{n-1} als Binärzahl

```
Zahl                    VZ Z Z Z Z Z Z Z

                +47     => 0 0 1 0 1 1 1 1

-47 -> 128-47 = 81      => 1 1 0 1 0 0 0 1
```

Ganzzahlige Literale in C

- Typ int:

```
dezimal       15     20
hexadezimal   0xF    0x14
oktal         017    024
```

- Die führende Null für Oktalzahlen stellt eine häufige Fehlerquelle dar, wenn z.B. der besseren Übersicht im Programmcode halber ein Array etwa mit 001, 002, 003, ..., 999 gefüllt wird

011 ist nicht gleich 11 !!

Ganzzahlige Literale

- Typ long int :

  dezimal 15L 20L

- Kleinbuchstabe "l" ist aber möglich - aber nicht verwenden, da leicht mit 1 (eins) verwechselt

- Weitere Massnahmen um Code "garantiert" unwartbar zu machen
  - http://mindprod.com/jgloss/unmain.html

- Typ unsigned int :

  dezimal 15u 20u

- Auch der Grossbuchstabe U ist möglich

Mehr zu Ganzzahltypen

- Achtung: bei Bereichsüberschreitungen

```
unsigned int vorzeichenlos = -1;
printf("%u", vorzeichenlos);
```

- Ergibt zum Beispiel (compilerabhängig):

  4'294'967'295

Computerpanne: Der ICE nach München kommt leider erst im Jahr 6093

Typische Grössen von Ganzzahlen

Typ	Bytes	Wertebereich (signed)	Wertebereich (unsigned)
[un]signed char	1	-128..127	0..255
[un]signed short (int)	2	-32768..32767	0..65535
[un]signed int	4	-2147483648..2147483647	0..4294967295
[un]signed long (int)	8	$-2^{63}..2^{63}-1$	$0..2^{64}-1$

Char als int

- Auch char kann zu den Ganzzahltypen gezählt und gerechnet werden (s. später)

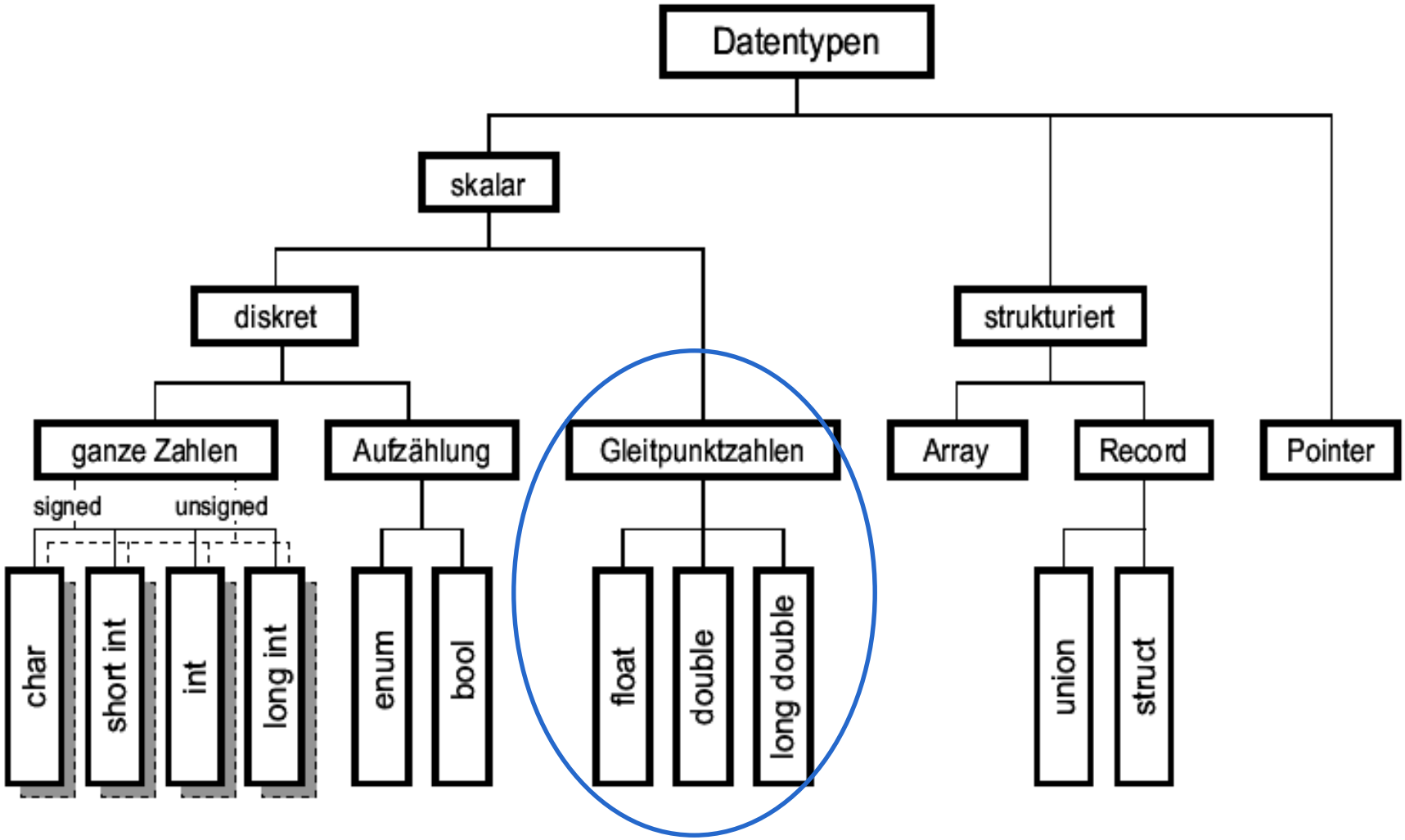
```
char c = 'a';
char c = 44;
 c = c + 1; // 'b'
```

Fazit Ganzzahltypen

- Wertebereiche compilerabhängig
- Beim Rechnen genaue Ergebnisse
- Um mögliche Wertebereichsüberläufe muss man sich selber kümmern
  - Kein Compiler-Fehler
  - Kein Laufzeitfehler
  - Nur falsches Ergebnis !

Fliesskommatypen

Datentypen Übersicht



Fliesskommatypen

```
float          double          long double
```

- Genaue Grössen/Wertebereiche compilerabhängig
- Normalerweise in der Datei *float.h* beschrieben
- Achtung: Ungenauigkeit von Gleitpunktberechnungen!
  - Rundungsfehler, die sich im Laufe von Berechnungen verstärken können
- Test ob x == 1:

```
if (fabs(x-1) < EPS)
```

Z.B.1E-10

float: eine Computer-Repräsentation von $\mathbb{R}$

- $\mathbb{R}$ Wertebereich: rationale Zahlen beliebiger Genauigkeit von -∞ bis +∞
- Wertebereich einschränken wie bei Ganzzahlen
- Gewisse Zahlen haben unendlich viele Stellen
  - z.B. 7 / 3 = 2.33333333333333333333333333333…, 3.1415926535 8979323846 2643383279 ...
- Lösung: nur begrenzte Anzahl Ziffern speichern
  - z.B. 7 / 3 = 2.333333

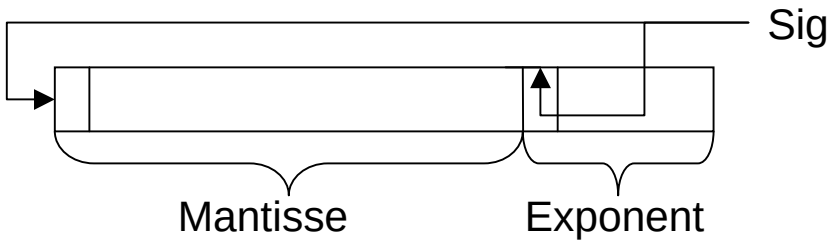
Zahlenstrahl Darstellung

- reelle Zahlen in der Mathematik

- reelle Zahlen im Computer: **float**

Floating Point Darstellung (Fliesskomma)

- Aufteilung in Mantisse und Exponent

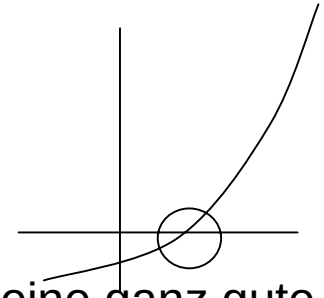


- Definition, Berechnungen: $a_{mant} \cdot 10^{a_{\exp}}$
  - Definition: ***a: (a_{mant},a_{exp})*** $a_{mant} \cdot b_{mant} \cdot 10^{a_{\exp}+b_{\exp}}$
  - Berechnung ***a* b:***

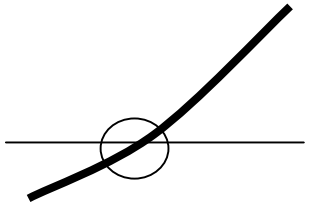
- Floating Point Darstellung (Namen "float")
  - Beispiel: 2.34223E+23f
  - bei grossen Zahlen mehr Stellen vor dem Komma
  - bei kleinen Zahlen mehr Stellen nach dem Komma
  - immer gleiche Anzahl Dezimalstellen: 6
    - *Vorteil: kann in immer gleich grosse Speicherzelle gespeichert werden (float 4 Byte)*

Fliesskommazahlen ~ $\mathbb{R}$

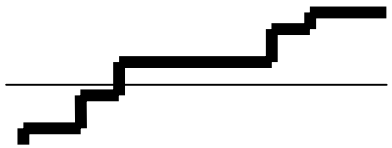
- Fliesskommazahlen als eine Näherung von $\mathbb{R}$



- … eigentlich eine ganz gute Näherung



- aber schliesslich nur endliche Genauigkeit: Treppenlinie;
- Effekt: **u.U. nie einen bestimmten Wert erreicht**



```
float d = 16777217;
if ((int)d == 16777217){...
```

float Datentyp

- Datentyp `float`
- Beispiele:

  `1.34f, -234.837F, 1E-3F, 3f, 0.75f, .001f`
- Achtung: Ohne `f` am Schluss wird automatisch der Datentyp `double` angenommen

Datentyp	**`float`**
Anzahl Bits	**32**
Wertebereich	$-3.4 \cdot 10^{38}$ bis $3.4 \cdot 10^{38}$
relative Genauigkeit	**6–7** Dezimalstellen
kleinste Zahl	$\pm 1.4 \cdot 10^{-45}$

double Datentypen

- Datentyp `double`
- Beispiele: `1.34 -10.3 1.e5 5d 0.3D .001`
- Wann soll der Datentyp `float`, wann `double` gewählt werden?
- Bei Rechnungen mit reellen Zahlen sollte **immer** der Datentyp `double` verwendet werden !

Datentyp	**double**
Anzahl Bits	**64**
Wertebereich	$-1.79 \cdot 10^{308}$ bis $1.79 \cdot 10^{308}$
relative Genauigkeit	**14–15** Dezimalstellen
kleinste Zahl	$\pm 4.9 \cdot 10^{-324}$

Falsche Wahl des Variablen-Typs kann teuere Konsequenzen haben,
z.B. Crash der Ariane-Rakete →http://en.wikipedia.org/wiki/Ariane_5_Flight_501

Heute in der Regel nur noch double verwenden

Gleitpunktliterale

- Typ **double**:

```
dezimal        203.5
exponential    2.035e2   (auch E möglich)
```

- Typ **float**:

```
dezimal        1.414f
exponential    2.035e2f  (auch F möglich)
```

- Typ **long double**:

```
dezimal        203.5L
```

Typische Grössen

Typ	Bytes	Wertebereich ca.
float	4	$+/- 3.4 * 10^{+/-38}$
double	8	$+/- 1.7 * 10^{+/-308}$
long double	10	$+/- 1.7 * 10^{+/-4932}$

Fazit Fliesskommatypen

- Rechnen mit Fliesskommatypen ist mit Ungenauigkeiten verbunden
- Viele einfache rationale Zahlen wie 0.1 lassen sich mit Fliesskommatypen nicht genau darstellen, nur annähern
- Wenn Nachkommastellen und exakte Ergebnisse benötigt werden, kann ein Ganzzahltyp mit entsprechender Interpretation der Werte verwendet werden
- Beispiel: 3.45 Fr. könnte zum Beispiel anstatt mit 3.45f auch als Integer 3450 (zehntel Rappen) dargestellt werden

Grössenbestimmung von Zahlentypen

sizeof-Funktion

- Mit Hilfe des sizeof-Funktion kann man die Grösse eines Datentyps oder eines Datenobjekts **in Bytes** ermitteln
- Beispiel (Werte für bestimmte Plattform/Compiler):

```
printf("%d", sizeof(long));             // 8
printf("%d", sizeof(unsigned int));     // 4
double gewicht;
printf("%d", sizeof(gewicht));          // 8
printf("%d", sizeof(5.25f));            // 4
printf("%d", sizeof('A'));              // 1
```

Grössen der Basistypen

- Es gilt

  sizeof(short int) <= sizeof(int) <= sizeof(long int)

  sizeof(float) <= sizeof(double) <= sizeof(long double)

- Die tatsächlichen Grössen sind compilerabhängig

- Ausser mit sizeof können die aktuellen Grössen auch über die include Dateien *limits.h* und *float.h* festgestellt werden, e.g.

```
INT_MIN         -32768          Defines the minimum value for an int.
INT_MAX         +32767          Defines the maximum value for an int.
UINT_MAX        65535           Defines the maximum value for an unsigned int.
LONG_MIN        -2147483648     Defines the minimum value for a long int.
```

Arithmetische Operatoren und Typumwandlung

Arithmetische Operatoren

Operator	Int	Float	Operation	Beispiel	Ergebnis
+	X	X	Vorzeichen (unäres Plus)	+7	7
-	X	X	Vorzeichen (unäres Minus)	-8	-8
+	X	X	Addition (binäres Plus)	5+6	11
-	X	X	Subtraktion (binäres Minus)	9-3	6
*	X	X	Multiplikation	8*4	32
/	X	X	Division (bei Integer-Zahlen abgerundet!)	1/2	0
/	X	X	Division (sobald Gleitkommazahlen dabei, als Gleitkommadivision durchgeführt)	1.0/2	0.5
%	X		Modulo (Rest bei Integer-Division, Vorzeichen des Dividenden bleibt)	-7%3	-1
=	X	X	Zuweisung (linker Seite wird der Wert der rechten zugewiesen)	a=8	8

Arithmetische Operatoren

- Übliche arithmetische Operatoren: + – * / %
- Punkt- vor Strichrechnung:
  4 + 3 * 2 liefert 10 (nicht 14)

Microsoft Taschenrechner?

- Integer-Division schneidet einen möglichen Rest ab !
  Beispiel: 3 / 2 liefert 1
- Verschiedene Typen können kombiniert werden
  Beispiel: 3 / 2.0 liefert 1.5
  (mehr dazu später)

Operatoren und Ausdrücke

- Der Zuweisungsoperator gibt als Rückgabewert den zugewiesenen Wert zurück

```
a=b=c=5;    // Danach sind alle drei Variablen gleich 5
```

- Vorsicht Fehler (später):

```
if (i = 1) { ... }
```

- = steht für die Zuweisung, == ist der Vergleichsoperator (in einer späteren Lektion behandelt)

Zusammengesetzte Operatoren

- Häufig soll eine Variable verändert werden:

```
i = i + 1;
```

- Eine arithmetische Operation kann auch direkt mit einer Zuweisung kombiniert werden:

```
i += 1;
```

- Zusammengesetzte Operatoren:

```
+=   -=   *=   /=   %=
```

Inkrement und Dekrement

- Für das Erhöhen oder Vermindern um 1 gibt es spezielle Operatoren:

```
k++     // inkrementiert k und liefert vorherigen Wert
k--     // dekrementiert k und liefert vorherigen Wert
++k     // inkrementiert k und liefert neuen Wert
--k     // dekrementiert k und liefert neuen Wert
```

- Fehleranfällig, wenn in Ausdrücken verwendet, daher am besten ganz vermeiden!

```
k=1; k = k++ + k--; // ???
```

Präzedenzen & Assoziationen

Operatoren	Präzedenz	Assoziativität
++ -- ~ ! +(Vorzeichen) -(Vorzeichen)	15	rechts
* / %	13	links
+(Addition) -(Subtraktion)	12	links
<< >>	11	links
< <= > >=	10	links
== !=	9	links
&	8	links
^	7	links
|	6	links
&&	5	links
||	4	links
?:	3	links
= += -= *= /= %= <<= >>= &= |= ^=	2	rechts

Für Klarheit möglichst Klammern verwenden

Typumwandlung (type casting)

- Falls in einem Ausdruck verschiedene Typen vorkommen, werden diese vereinheitlicht (Implizite Typenumwandlung)
- Alle Operanden der Typen char, short oder enum werden in int gewandelt
- Falls mit int nicht darstellbar, wird unsigned int verwendet
- Falls danach noch unterschiedliche Typen vorkommen, wird in die höheren Typen umgewandelt:

  int < unsigned int < long < unsigned long < float < double < long double

Typumwandlung (type casting)

- Auch bei Zuweisungen erfolgt implizite Typumwandlung (ggf. auch zu kleineren Typen)
- Beispiele:

```
double r;
r = 3 / 2;      // ergibt 1.0
r = 3 / 2.0;    // ergibt 1.5
int a;
double PI = 3.1415927;
a = PI;         // ergibt 3
```

Typumwandlung (type casting)

- Explizite Typenumwandlung

```
( typname ) ausdruck
```

Beispiele zur Typumwandlung

```
int i = 7, j = 3, k;
double x;
k = i / j;              // 7/3 = 2 ==> k = 2
x = i / j;              // 7/3 = 2 ==> x = 2.0   !!!
x = (double) i / j;     // 7.0 / 3.0 = 2.3333 ==> x = 2.333
x = 1.0 / 3             // 1.0 / 3.0 = 0.3333 ==> x = 0.333
x = 2.9438;
k = x + i               // 2.9438 + 7.0 = 9.9438 ==> k = 9
k = (int)x + i          // 2 + 7 = 9 ==> k = 9 gleiches Ergebnis
```

Explizite Umwandlung in Java (später) zwingend in C empfohlen

Mathematische Funktionen

C Standard Library

- Beim Programmieren in C kann man auf zahlreiche bestehende Funktionen zurückgreifen

- Die C Standard Library (definiert für ANSI-C, erweitert für C99) stellt Funktionen (und Makros, Typdefinitionen, s. später) zu verschiedenen Bereichen zur Verfügung

- Um sie nutzen zu können, muss die entsprechende Header-Datei geladen werden, für Standard-Ein-/Ausgabe-Funktionen z.B.:

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

- Dies ist eine sog. Präprozessor-Direktive, keine C-Anweisung:
  Kein Semikolon am Ende

C Standard Library

- http://de.wikibooks.org/wiki/C-Programmierung:_Standard_Header
- http://openbook.galileocomputing.de/c_von_a_bis_z/030_c_anhang_b_001.htm

Einige mathematische Funktionen

- Bibliothek laden:
  `#include <math.h>`
- Auszug:

`sqrt(x)`	Berechnet die Quadratwurzel
`sin(x)`	Berechnet den Sinus
`exp(x)`	Berechnet die Exponentialfunktion bzgl. e
`log(x)`	Berechnet den natürlichen Logarithmus
`abs(x)`	Berechnet den Absolutbetrag (nur für Integerzahlen)
`fabs(x)`	Berechnet den Absolutbetrag (als double)

Strings und einfache Ein-/Ausgabe

Zeichentypen

char

- Kann verwendet werden wie Ganzzahltypen
- Ein Byte
- ASCII-Zeichensatz (<= 127)
- Beispiel

```
char zeichen = 'A';
int i = zeichen;
```

Zeichenliterale

- Zeichen werden in einzelne Anführungsstriche (Apostrophe) eingeschlossen: 'a'
- Dies entspricht dem ASCII-Code des betreffenden Zeichens
- Diese beiden Definitionen sind gleichbedeutend:

```
char zeichen = 100;
char zeichen = 'd';
```

- Umwandlung Klein- in Grossbuchstaben:

```
zeichen = 'd';
zeichen = zeichen + ('A' - 'a');  // zeichen ist 'D'
```

Stringliterale

- Stringliterale werden in Anführungszeichen eingeschlossen:
  `"Text"`
- Hinweis: Stringvariablen werden später besprochen

- Strings in C sind Arrays von Zeichen mit einem Null-Byte am Ende (in einer späteren Lektion behandelt)

- Steuerzeichen werden durch Backslash-Sequenzen dargestellt:
  `\n` `Linefeed`
  `\t` `Tab`
  `\0` `NUL`

Standard Eingaben: stdio.h

I/O-Funktionen aus stdio

- `printf()` - zum formatierten Ausgaben
- `scanf()` - zum formatierten Einlesen
- `putchar()` - zum Ausgaben eines Zeichens
- `getchar()` - zum Einlesen eines Zeichens

- Dazu muss die Bibliothek *stdio* eingebunden werden:

  #include <stdio.h>

Beispiele zu printf()


```
#include <stdio.h>

int main() {
  printf("Integer: %d\n", 42);
  printf("Double: %f\n", 3.141);
  printf("Zeichen: %c\n", 'z');
  printf("Zeichenkette: %s\n", "abc");
  printf("43 Dezimal ist in Oktal: %o\n", 43);
  printf("43 Dezimal ist in Hexadezimal: %x\n", 43);
  printf("Und zum Schluss geben wir noch das Prozentzeichen aus: %%\n");
  return 0;
}
```

printf()

- Erstes Argument: String, der Umwandlungszeichen für die weiteren Argumente enthalten kann
  - %d (oder %i) Integer
  - %s String
  - usw.
- Ausserdem möglich: Flags für Ausrichtung, Vorzeichen, Füllzeichen, sowie Feldbreite, Nachkommastellen
  - %5d Integer, rechtsbündig, 5 Zeichen
  - %05d Integer, rechtsbündig, 5 Zeichen, mit 0 aufgefüllt
  - usw.

scanf()


```
int x;
scanf("%d",&x);
double d
scanf("%lf",&d);
```

- scanf() liest einen Wert ein und speichert diesen in den angegebenen Variablen
- Mit dem Adressoperator & erhält man die Adresse einer Variablen

getchar() und putchar()

```
int c;
c = getchar();
putchar(c);
```

- *getchar()* liefert das nächste Zeichen vom Standard-Eingabestrom
  - kann auch verwendet werden, um auf eine Benutzerreaktion zu warten
- *putchar()* gibt ein einzelnes Zeichen c auf der Standardausgabe aus

Noch Fragen?