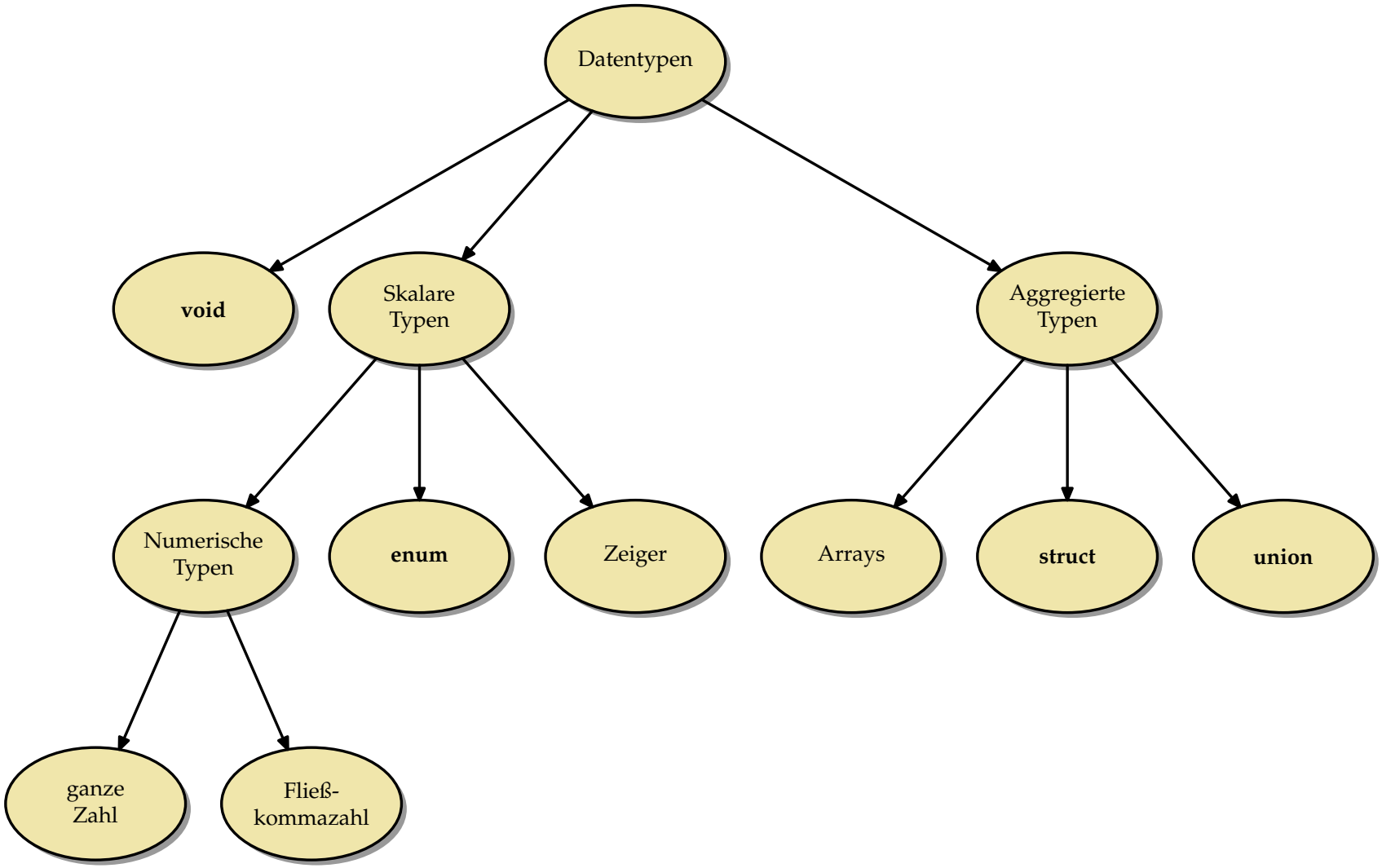
Datentypen legen

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


Datentypen
void
Skalare Typen
Aggregierte Typen
Numerische Typen
enum
Zeiger
Arrays
struct
union
ganze Zahl
Fließ-kommazahl

- 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, wobei ⟨declaration-specifiers⟩ zu „**int**“ expandiert und „*i*“ zu ⟨init-declarator-list⟩:

$$\begin{aligned} \langle \text{declaration} \rangle &\longrightarrow \langle \text{declaration-specifiers} \rangle \\ &\quad [\ \langle \text{init-declarator-list} \rangle\ ] \\ \langle \text{declaration-specifiers} \rangle &\longrightarrow \langle \text{declaration-specifier} \rangle \\ &\quad [\ \langle \text{declaration-specifiers} \rangle\ ] \end{aligned}$$

- 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 ⟨declaration-specifier⟩ nur aus **int**. Der Dereferenzierungs-Operator wird stattdessen syntaktisch der rechten Seite, der ⟨init-declarator-list⟩ 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.
- Das schließt Datentypen wie **int**, **double** und auch beliebige Zeigertypen ein.
- Ein Wert eines skalaren Datentyps kann ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert. (Ein Schlüsselwort wie **null** oder **nullptr** wird von C nicht unterstützt.)
- Und umgekehrt ist ein Null-Zeiger 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 Ausnahmen 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: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
  Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- Vorsicht: Im Wertebereich ist $-2^{n-1}$ enthalten, jedoch nicht $2^{n-1}$, somit liefert das unäre Minus angewendet auf die niedrigste Zahl nicht das erwartete Resultat.

- **Einer-Komplement**:

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

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
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: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

C unterstützt einige Bit-Operationen für ganzzahlige Datentypen:

| | |
|---|---|
| $\sim x$ | bitweises Komplement; bei ganzen Zahlen ohne Vorzeichen entspricht dies dem maximalen Wert des Typs abzüglich von $x$. |
| $x \mid y$ | bitweise Oder-Operation |
| $x \,\&\, y$ | bitweise Und-Operation |
| $x \,\hat{}\, y$ | bitweise Exklusiv-Oder-Operation |
| $x << i$ | Verschiebeoperation mit dem Wert $x \times 2^i$, wobei $i \geq 0$ gelten und das Ergebnis repräsentierbar sein muss, damit es wohldefiniert ist |
| $x >> i$ | Verschiebeoperation mit dem Wert $\lfloor \frac{x}{2^i} \rfloor$. Bei vorzeichenbehafteten ganzen Zahlen lässt der Standard den Effekt undefiniert, falls $x < 0$. |

Mit Hilfe der Bit-Operationen lassen sich für Mengen mit sehr kleinen Kardinalitäten ganze Zahlen ohne Vorzeichen verwenden. Seien $A$ und $B$ Variablen des Typs **unsigned int**, die zwei Mengen repräsentieren. Die Elemente der Menge werden durch ganze Zahlen repräsentiert von 0 bis maximal $WORD\_BIT-1$ (aus **#include** <limits.h>).

| Mengenoperation | Bit-Operation in C |
|---|---|
| $A \cup B$ | $A \mid B$ |
| $A \cap B$ | $A \,\&\, B$ |
| $A \setminus B$ | $A \,\&\, \sim B$ |
| $\{i\}$ | $1 << i$ |

# Was passiert bei Überläufen?



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, dies wird aber von C nicht unterstützt. Das wird in Java genauso gehandhabt.

# Was passiert, wenn durch 0 geteilt wird? 76

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 | $[-127, 127]$ | *SCHAR_MIN*, *SCHAR_MAX* |
| **unsigned char** | 8 | $[0, 255]$ | *UCHAR_MAX* |
| **char** | 8 | | *CHAR_MIN*, *CHAR_MAX* |
| **short** | 16 | $[-32767, 32767]$ | *SHRT_MIN*, *SHRT_MAX* |
| **unsigned short** | 16 | $[0, 65535]$ | *USHRT_MAX* |
| **int** | 16 | $[-32767, 32767]$ | *INT_MIN*, *INT_MAX* |
| **unsigned int** | 16 | $[0, 65535]$ | *UINT_MAX* |
| **long** | 32 | $\left[-2^{31}+1, 2^{31}-1\right]$ | *LONG_MIN*, *LONG_MAX* |
| **unsigned long** | 32 | $[0, 4294967295]$ | *ULONG_MAX* |
| **long long** | 64 | $\left[-2^{63}+1, 2^{63}-1\right]$ | *LLONG_MIN*, *LLONG_MAX* |
| **unsigned long long** | 64 | $\left[0, 2^{64}-1\right]$ | *ULLONG_MAX* |

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.
- Mit „-m64“ gibt es einen 64-Bit-Adressraum. Dies ist inzwischen überwiegend 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 (von historischen Architekturen abgesehen) auch 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*> bzw. ab C11 über <*uchar.h*> auch *char16_t* und *char32_t*.

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:

| | | |
|---|---|---|
| \a | BEL | *alert*, Signalton |
| \b | BS | *backspace* |
| \f | FF | *formfeed* |
| \n | LF | *newline*, Zeilentrenner |
| \r | CR | *carriage return*, „Wagenrücklauf“ |
| \t | HT | Horizontaler Tabulator |
| \v | VT | Vertikaler Tabulator |
| \\ | \ | „Fluchtsymbol“ |
| \' | ' | einfaches Hochkomma |
| \0 | NUL | Null-Byte |
| \ddd | | 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);
   }
}
```

$$
\begin{aligned}
\langle \text{floating-point-type-specifier} \rangle &\longrightarrow \textbf{float} \\
&\longrightarrow \textbf{double} \\
&\longrightarrow \textbf{long double} \\
&\longrightarrow \langle \text{complex-type-specifier} \rangle \\
\langle \text{complex-type-specifier} \rangle &\longrightarrow \textbf{float \_Complex} \\
&\longrightarrow \textbf{double \_Complex} \\
&\longrightarrow \textbf{long double \_Complex}
\end{aligned}
$$

- 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 $\left[-2^{q-1}+1, 2^{q-1}\right]$.
- 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 $[-126, 127]$.
- Bei **double** mit insgesamt 64 Bit, davon $q = 11$ für den Exponenten, erhalten wir den Bereich $[-1022, 1023]$.

- 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 implizit auf 1 gesetztes Bit, d.h. $m$ entspricht der im Zweier-System notierten Zahl $1, m_p m_{p-1} \ldots m_2 m_1$.
- 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. $e = -2^{q-1} + 1 = e_{min} - 1$, 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 (also $e = e_{min} - 1$), es aber in der Mantisse mindestens ein Bit mit $m_i = 1$ gibt. In diesem Falle ergibt sich folgende Interpretation:

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

- Der Fall $e = e_{max} + 1$ erlaubt es, $\infty$, $-\infty$ und *NaN* (*not a number*) mit in den Wertebereich der Gleitkommazahlen aufzunehmen. $\infty$ 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*, $\infty$ und $-\infty$ liefert *isfinite false*.

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

fclass.c

```
#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** | | $\geq 15$ | $\geq 63$ |

- 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.

# Flächenberechnung eines Dreiecks

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

$$
\begin{aligned}
s &= \frac{a+b+c}{2} \\
A &= \sqrt{s(s-a)(s-b)(s-c)}
\end{aligned}
$$

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: $a \geq b \geq c$.

# Flächenberechnung eines Dreiecks



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 lenghts 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));
}
```

```
dublin$ gcc -Wall -std=c99 triangle.c -lm
dublin$ a.out
triangle side lenghts a b c: 1e10 1e10 1e-10
Formula #1 delivers 0.0000000000000000
Formula #2 delivers 0.5000000000000000
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 0.77777777777777
not equal
dublin$
```

- Gelegentlich wird nahegelegt, statt dem ==-Operator auf die Nähe zu testen, d.h. $x \sim y \Leftrightarrow |x - y| < \epsilon$, wobei $\epsilon$ für eine angenommene Genauigkeit steht.
- Dies lässt jedoch folgende Fragen offen:
  - Wie sollte $\epsilon$ 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.

days.c

```
#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]);
}
```