

2. Ganze Zahlen

Auswertung arithmetischer Ausdrücke, Assoziativität und Präzedenz, arithmetische Operatoren, Wertebereich der Typen
`int, unsigned int`

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

15 degrees Celsius are 59 degrees Fahrenheit

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,
- enthält drei Literale, eine Variable, drei Operatorsymbole

Wie ist der Ausdruck geklammert?

Präzedenz

Punkt vor Strichrechnung

`9 * celsius / 5 + 32`

bedeutet

`(9 * celsius / 5) + 32`

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz ("binden stärker") als additive Operatoren (`+`, `-`)

Assoziativität

Von links nach rechts

$9 * \text{celsius} / 5 + 32$

bedeutet

$((9 * \text{celsius}) / 5) + 32$

Regel 2: Assoziativität

Arithmetische Operatoren ($*$, $/$, $\%$, $+$, $-$) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Stelligkeit

Regel 3: Stelligkeit

Unäre Operatoren +, - vor binären +, -.

$$-3 - 4$$

bedeutet

$$(-3) - 4$$

Klammerung

Jeder Ausdruck kann mit Hilfe der

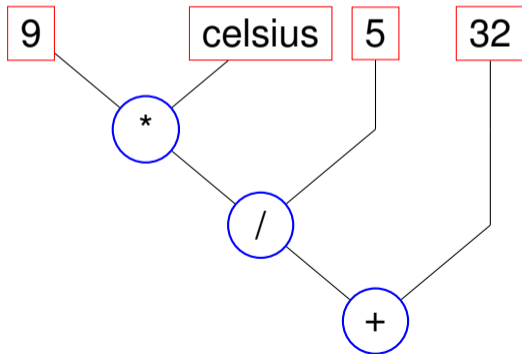
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

der beteiligten Operatoren eindeutig geklammert werden (Details im Skript).

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

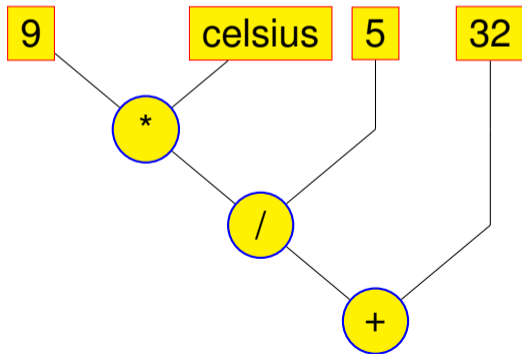
$((9 * \text{celsius}) / 5) + 32$



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

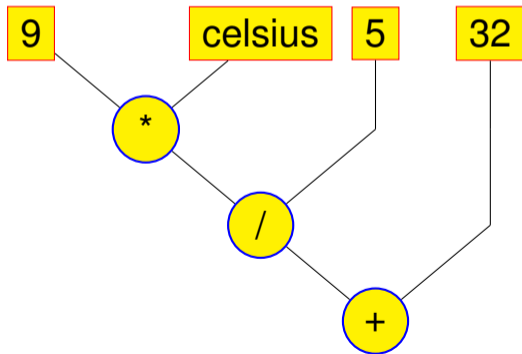
9 * celsius / 5 + 32



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

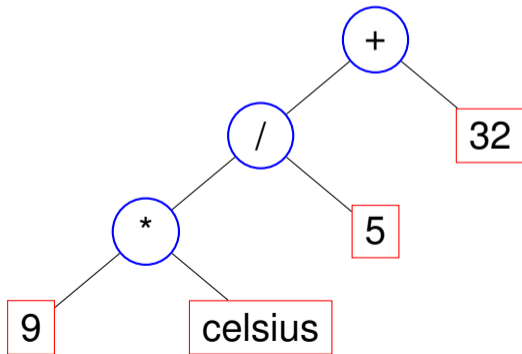
$$9 * \text{celsius} / 5 + 32$$



Ausdrucksbäume – Notation

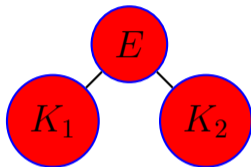
Üblichere Notation: Wurzel oben

9 * celsius / 5 + 32



Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.
- Beispiel für "schlechten Ausdruck": $(a+b)*(a++)$

Auswertungsreihenfolge

Richtlinie

Vermeide das Verändern von Variablen, welche im selben Ausdruck noch einmal verwendet werden!

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	-	1	16	rechts
Multiplikation	*	2	14	links
Division	/	2	14	links
Modulus	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

Alle Operatoren: $[R\text{-Wert } \times] R\text{-Wert} \rightarrow R\text{-Wert}$

Zuweisungsausdruck – nun genauer

- Bereits bekannt: $a = b$ bedeutet Zuweisung von b (R-Wert) an a (L-Wert).
Rückgabe: L-Wert
- Was bedeutet $a = b = c$?
- Antwort: Zuweisung rechtsassoziativ, also

$$a = b = c \quad \iff \quad a = (b = c)$$

Beispiel Mehrfachzuweisung:

$$a = b = 0 \implies b=0; a=0$$

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent... aber nicht in C++!

```
9 / 5 * celsius + 32
```

```
15 degrees Celsius are 47 degrees Fahrenheit
```


Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

$5 / 2$ hat Wert 2, $5 \% 2$ hat Wert 1.

- Es gilt immer:

$(a / b) * b + a \% b$ hat den Wert von a .

Inkrement und Dekrement

- Erhöhen / Erniedrigen einer Zahl um 1 ist eine häufige Operation
- geht für einen L-Wert so:

```
expr = expr + 1.
```

Nachteile

- relativ lang
- expr wird zweimal ausgewertet (Effekte!)

In-/Dekrement Operatoren

Post-Inkrement

```
expr++
```

Wert von expr wird um 1 erhöht, der *alte* Wert von expr wird (als R-Wert) zurückgegeben

Prä-Inkrement

```
++expr
```

Wert von expr wird um 1 erhöht, der *neue* Wert von expr wird (als L-Wert) zurückgegeben

Post-Dekrement

```
expr--
```

Wert von expr wird um 1 verringert, der *alte* Wert von expr wird (als R-Wert) zurückgegeben

Prä-Dekrement

```
--expr
```

Wert von expr wird um 1 verringert, der *neue* Wert von expr wird (als L-Wert) zurückgegeben

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Prüz	Assoz.	L/R-Werte
Post-Inkrement	<code>expr++</code>	1	17	links	L-Wert → R-Wert
Prä-Inkrement	<code>++expr</code>	1	16	rechts	L-Wert → L-Wert
Post-Dekrement	<code>expr--</code>	1	17	links	L-Wert → R-Wert
Prä-Dekrement	<code>--expr</code>	1	16	rechts	L-Wert → L-Wert

In-/Dekrement Operatoren

Beispiel

```
int a = 7;  
std::cout << ++a << "\n"; // 8  
std::cout << a++ << "\n"; // 8  
std::cout << a << "\n"; // 9
```

In-/Dekrement Operatoren

Ist die Anweisung

`++expr;` ← wir bevorzugen dies

äquivalent zu

`expr++;`?

Ja, aber

- Prä-Inkrement ist effizienter (alter Wert muss nicht gespeichert werden)
- Post-In/Dekrement sind die einzigen linksassoziativen unären Operatoren (nicht sehr intuitiv)

C++ **vs.** ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,
- während C++ ja immer noch das alte C liefert.

Arithmetische Zuweisungen

$a \ += \ b$

\Leftrightarrow

$a \ = \ a \ + \ b$

Analog für $-$, $*$, $/$ und $\%$

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
+=	<code>expr1 += expr2</code>	<code>expr1 = expr1 + expr2</code>
-=	<code>expr1 -= expr2</code>	<code>expr1 = expr1 - expr2</code>
*=	<code>expr1 *= expr2</code>	<code>expr1 = expr1 * expr2</code>
/=	<code>expr1 /= expr2</code>	<code>expr1 = expr1 / expr2</code>
%=	<code>expr1 %= expr2</code>	<code>expr1 = expr1 % expr2</code>

Arithmetische Zuweisungen werten `expr1` nur einmal aus.
Zuweisungen haben Präzedenz 4 und sind rechtsassoziativ

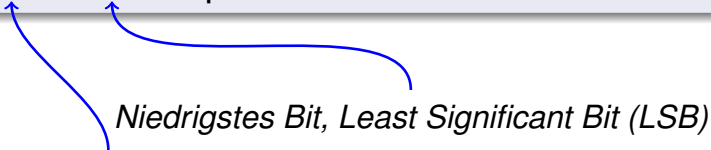
Binäre Zahlendarstellungen

Binäre Darstellung ("Bits" aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Beispiel: 101011 entspricht 43.



Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Binäre Zahlen: Zahlen der Computer?

Wahrheit: Computer rechnen mit Binärzahlen.

NEUE ZÜRCHER ZEITUNG

TECHNIK

Mittwoch, 26. August 1950 Blatt 5
Mittagsausgabe Nr. 1796 (50)

Das programmgesteuerte Recheng Gerät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika wurde in dem Artikel „Elektronische Rechenmaschine“ (vgl. Nr. 2149 der „N.Z.Z.“ vom 13. Oktober 1949) und „Die neueste elektronische Rechenmaschine“ (vgl. Nr. 871 der „N.Z.Z.“ vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K-6, Neukirchen — die Rolle sein, welches im Juli dieses Jahres am Institut für angewandte Mathematik der Eidgenössischen Technischen Hochschule in Zürich, das unter der Leitung von Prof. Dr. E. Stiefel steht, in Betrieb genommen wurde. Damit ist dieses Institut in der Lage, das in der Schweiz immer stärker werdenden Bedarf nach einer leistungsfähigen Zentralstelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden, und die Erlösung vieler anderer Aufgaben ist vorbereitet.

Merkmale des Gerätes

Das Gerät ist ein Glied in dem rasigen Entwicklungsprogramm des Ingenieurs Konrad Zuse; es wurde im Auftrag des Institutes für angewandte Mathematik der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als „Modell 2^o“ konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen in den Vereinigten Staaten. Es ist überaus interessant festzustellen, wie für die meisten wichtigen funktionalen Probleme beiderorts genau dieselbe Lösung gefunden wurde, wie aber andererseits gewisse Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Behandlung beigemessen wurde.

Eine kurze technische Charakterisierung lautet wie folgt: Elektromechanisch arbeitendes Gerät mit 2200 Relais, 21 Schritt- und einem Speicher für 64 Zahlen, welcher mit neuartigen, mechanischen Schaltkreisen arbeitet; Verwendung des Dualsystems und der halblogarithmischen Darstellung; Multiplikationszeit 2,5 Sekunden; Programmsteuerung mit Hilfe zweier Lochstreifen, auf die wahlweise umgeschaltet werden kann; Eingabe von Zahlen durch eine Tastatur oder durch einen Lochstreifen; Ausgabe der Resultate durch Lampenfeld, Lochstreifen oder Druckwerk.

Das duale Zahlensystem

Allgemein wird programmgesteuerten Rechengereäten häufig das duale Zahlensystem zugrunde gelegt, welches nur die zwei Ziffern 0 und 1 verwendet, während das bekannte Dezimalsystem

lesen wir eine Dezimalzahl von rechts nach links, so schließt sich das Gewicht von Stelle zu Stelle um den Faktor 10 durch 2 zu erweitern. Also bedeutet die (nummerale duale) Zahl siebenzig den Ausdruck:

$$a \cdot 10^7 + b \cdot 10^6 + c \cdot 10^5 + d \cdot 10^4 + e \cdot 10^3 + f \cdot 10^2 + g \cdot 10^1$$

Die Zahl 1 wird in beiden Systemen gleich dargestellt. Um jedoch diese von denselben Zahlen deutlich zu trennen, schreiben wir die duale 1 als 1₁₀. Dagegen wirkt schon die 2 ab, indem sie dual 10 lautet; denn das bedeutet ja 1₁₀ · 2₁₀ = 2₁₀. Wenn einer Zahl (ohne Stellen nach dem Komma) rechts eine Null zugefügt wird, so vergrößert sie sich um den Faktor 2 (und so, wie im Dezimalsystem, um den Faktor 10). Auf diese Weise kann aus 10₁₀ = 2₁₀ auf einstufige Weise gebildet werden: 100 = 4, 1000 = 8, 10000 = 16, usw.

Die Dualzahl 10101₁₀ bedeutet nun also:

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Ganz analog sind etwaige Stellen nach dem Komma zu interpretieren; so wird 1,611₁₀ wie folgt übersetzt:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + \frac{1}{4} + \frac{1}{8} = 1,375$$

Der große Vorteil, der das Dualsystem für Rechenoperationen so geeignet macht, nämlich die Reduktion der Anzahl der versetzten Symbole auf nur zwei, wird allerdings durch einen Nachteil erkauft: Es braucht mehr Stellen, um eine bestimmte Zahl darzustellen. Die einstufige Zahl 8

Änderung des Maßstabes durchgerechnet werden können.

Die beschriebene Darstellung bringt eine gewisse Komplikation der Rechenoperationen mit sich. So müssen vor einer Addition die beiden Summanden zunächst so verschoben werden, daß ihre Kommata untereinander zu liegen kommen, was an Hand eines Beispiels erläutert werden soll. Damit der Leser nicht durch das ungewöhnliche duale Zahlensystem verwirrt wird, ist das Beispiel im Dezimalsystem durchgeführt; doch wird daran erinnert, daß das Gerät in Wirklichkeit mit dualen Zahlen rechnet.

Es soll also etwa addiert werden: 2,345078 × 10³ + 9,876543 × 10⁴ (Man beachte, daß die eigentliche Zahl stets zwischen 1 und 10 liegt, also das Komma nach der ersten Stelle hat.) Nun müssen die beiden Summanden „ausgerichtet“ werden, d. h. die beiden Exponenten sind einander gleich gemacht, und zwar erhält der kleinere Exponent den Wert des größeren also 2. Die Zahlen lauten nun, richtig untereinander geschrieben und skizziert, wie folgt:

$$\begin{array}{r} 2,345078 \times 10^3 \\ 9,876543 \times 10^4 \\ \hline 2,355554 \times 10^4 \end{array}$$

Es ist ersichtlich, daß bei der kleineren der beiden Zahlen rechts einige Stellen abgeschrieben werden müssen; denn wenn die Summanden scheinbar richtig gegeben waren, so soll auch das Resultat nicht mehr als sieben Stellen enthalten.



Abb. 2. Der Schaltapparat bei der Fertigung eines Rechenplans. Die Abtaster für den Lochstreifen sind deutlich sichtbar.

Defekte können „beiläufig“ gegeben werden, d. h. ihre Ausfüllung wird von der Natur eines errechneten Resultates abhängig gemacht. Erst dadurch werden die unvollständigen Verfahren-

Hexadezimale Zahlen

Zahlen zur Basis 16. Darstellung

$$h_n h_{n-1} \dots h_1 h_0$$

entspricht der Zahl

$$h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0.$$

Schreibweise in C++: vorangestelltes 0x

Beispiel: 0xff entspricht 255.

Hex Nibbles

hex	bin	dec
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits. Die Zahlen 1, 2, 4 und 8 repräsentieren Bits 0, 1, 2 und 3.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen bestehen aus acht Hex-Nibbles:

`0x00000000` -- `0xffffffff` .

`0x400` = *1Ki*.

`0x100000` = *1Mi*.

`0x40000000` = *1Gi*.


`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.

`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.

„`0x8a20aaf0` ist eine Adresse in den oberen 2G des 32-bit Adressraumes“

Beispiel: Hex-Farben

#00FF00



r g b

Wozu Hexadezimalzahlen?

“Für Programmierer und Techniker” (Auszug aus der Bedienungsanleitung des Schachcomputers *Mephisto II*, 1981)



a) Anzeige **8200**
MEPHISTO ist mit genau 2 Bauern-Einheiten im Vorteil.



b) Anzeige **7F00**
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in **hexadezimaler Schreibweise**. Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F ($A = 10, B = 11, \dots, F = 15$).

Für mathematisch Vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (A \times 16^3) + (B \times 16^2) + (C \times 16^1) + (D \times 16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.

Eine Bauereinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten. Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

Beispiele:



c) Anzeige **805E**
(E = -14) Umrechnung nach folgendem Verfahren:
 $(14 \times 16^0) + (5 \times 16^1) + (0 \times 16^2) + (0 \times 16^3) = 14 + 80 + 0 + 0 =$
 $= +94 \text{ Punkte.}$



d) Anzeige **7F80**
(7 = -1; F = 15) Umrechnung wie folgt:
 $(0 \times 16^0) + (8 \times 16^1) + (15 \times 16^2) - (1 \times 16^3) = 0 + 128 + 3840 - 4096 =$

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich umfasst die 2^B ganzen Zahlen:

$$\{-2^{B-1}, -2^{B-1} + 1, \dots, -1, 0, 1, \dots, 2^{B-1} - 2, 2^{B-1} - 1\}$$

- Auf den meisten Plattformen $B = 32$
- Für den Typ `int` garantiert C++ $B \geq 16$
- Hintergrund: Abschnitt 2.2.8 (Binary Representation) im Skript

Wertebereich des Typs int

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

int main()
{
    std::cout << "Minimum int value is "
              << std::numeric_limits<int>::min() << ".\n"
              << "Maximum int value is "
              << std::numeric_limits<int>::max() << ".\n";
    return 0;
}
```

Zum Beispiel

Minimum int value is -2147483648.

Maximum int value is 2147483647.

Überlauf und Unterlauf

- Arithmetische Operationen (+, -, *) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

```
power8.cpp:  $15^8 = -1732076671$ 
```

```
power20.cpp:  $3^{20} = -808182895$ 
```

- Es gibt *keine Fehlermeldung!*

Der Typ `unsigned int`

- Wertebereich

$$\{0, 1, \dots, 2^B - 1\}$$

- Alle arithmetischen Operationen gibt es auch für `unsigned int`.
- Literale: `1u`, `17u` ...

Gemischte Ausdrücke

- Operatoren können Operanden verschiedener Typen haben (z.B. `int` und `unsigned int`).

```
17 + 17u
```

- Solche gemischten Ausdrücke sind vom „allgemeineren“ Typ `unsigned int`.
- `int`-Operanden werden *konvertiert* nach `unsigned int`.

Konversion

int Wert	Vorzeichen	unsigned int Wert
----------	------------	-------------------

x


≥ 0

x

x

< 0

$x + 2^B$



Bei Zweierkomplementdarstellung passiert dabei intern gar nichts

Konversion “andersherum”

Die Deklaration

```
int a = 3u;
```

konvertiert `3u` nach `int`.

Der Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig.

Vorzeichenbehaftete Zahlendarstellung

- Soweit klar (hoffentlich): Binäre Zahlendarstellung ohne Vorzeichen, z.B.

$$[b_{31}b_{30} \dots b_0]_u \hat{=} b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \dots + b_0$$

- Nun offensichtlich notwendig: Verwende ein Bit für das Vorzeichen.
- Suche möglichst konsistente Lösung

Die Darstellung mit Vorzeichen sollte möglichst viel mit der vorzeichenlosen Lösung „gemein haben“. Positive Zahlen sollten sich in beiden Systemen algorithmisch möglichst gleich verhalten.