Rechtlicher Hinweis

Legal notice

Informatik I (B.Sc. Physik)

Ausdrücke und Operatoren

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

Inhalt

Ausdrücke (expressions)

- Ausdrücke: syntaktische Konstrukte zur Darstellung bzw. Berechnung von Werten (ähnlich zur Mathematik: Terme)
- Literale, Konstanten und Variablen dienen als Operanden
  - Literale: 3, 128L, 4.2, .25, 2.11f, 3.5e−5, 1E3
  - Konstanten: **const int** k=1024; **const double** pi=3.1415;
  - Variablen: **int** i=2; **double** d=5.5;
- Verknüpfung erfolgt durch Operatoren
  - Unäre Operatoren: −x (Negation)
  - Binäre Operatoren: x+y (Summe)

  $\Rightarrow$ Schreibweise ähnlich der mathematischen Notation
- Weitere Unterscheidung:
  - Arithmetische Operatoren (für mathematische Operationen)
  - Relationale Operatoren (für Vergleiche)
  - Logische Operatoren (für logische Ausdrücke / Wahrheitswerte)

Einige Operatoren im Überblick

Operator	unär/binär	Bedeutung
()	unär	Klammerung
++ (Post)	unär	Inkrement
– – (Post)	unär	Dekrement
+	unär	unäres Plus
–	unär	unäres Minus
++ (Präfix)	unär	Inkrement
– – (Präfix)	unär	Dekrement
*	binär	Multiplikation
/	binär	Division
%	binär	Modulo
+	binär	Addition
–	binär	Subtraktion
=	binär	Zuweisung

Reihenfolge: absteigende Bindung (Priorität)

Inhalt

Was sind unäre Operatoren?

- Unäre Operatoren haben nur einen Operanden
- Vorzeichen:
  - Negation (negatives Vorzeichen bzw. Vorzeichenwechsel): $-$
  - Unäres Plus (positives Vorzeichen: $+$) wird in der Regel (wie in der Mathematik auch) weggelassen
- Inkrement und Dekrement
  - Inkrementieren (Wert um 1 erhöhen): $++$
  - Dekrementieren (Wert um 1 verringern): $--$
    $\Rightarrow$ Typische Anwendung: Zählen
- Präfix-Notation ist Standard-Schreibweise: erst Operator, dann Operand

  $\Rightarrow$ Beispiel Negation: -5 oder $-x$
- Ausnahme: $++$ und $--$

  $\Rightarrow$ Präfix- und Postfix-Notation möglich

  $\Rightarrow$ Achtung: unterschiedliche Bedeutung!

Präfix vs. Postfix bei ++ und − −

```
int x = 5;
x++;
cout << x << endl; // 6
--x;
cout << x << endl; // 5
```

Präfix

Schreibweise:

```
y = ++x;
```

Bedeutung:

```
x = x+1;
y = x;
```

Postfix

Schreibweise:

```
y = x++;
```

Bedeutung:

```
y = x;
x = x+1;
```

Inhalt

Was sind binäre Operatoren?

- Binäre Operatoren verknüpfen zwei Operanden
- Infix-Notation: Operator zwischen Operanden
- Beispiel Grundrechenarten: x+y, x−y, x∗y

- Division abhängig von Datentypen der Operanden
- Gleiches Zeichen für ganzzahlige Division und Gleitkomma-Division: /
- Ganzzahlige Division (Division mit Rest) bei ganzzahligen Datentypen (**int**)

```
double d1 = 5.0;
double d2 = 2.0;
int i1 = 5;
int i2 = 2;

// Gleitkomma-Division:
cout << d1 / d2 << endl; // 2.5

// Ganzzahlige Division:
cout << i1 / i2 << endl; // 2
```

Was ist mit dem Rest der ganzzahligen Division?

Der Modulo-Operator

- Operationszeichen für den Modulo-Operator: %
- Berechnet den Rest der ganzzahligen Division

- Verwendung oft in Verbindung mit ganzzahliger Division
- Beispiel: Zeitangaben umrechnen

```
int min = 225;
cout << min / 60 << "Std. und " << min % 60 << "Min." << endl;
// 3 Std. und 45 Min.
```

- z = y % x; // Was ist der Wertebereich von z?
  $\Rightarrow 0 \leq z < x$ bzw. $z \in \{0, 1, \ldots, x-1\}$

- Weitere Anwendungen:
  - Ist eine Zahl z gerade oder ungerade?
    $\Rightarrow$ Teilbarkeit durch 2: z % 2
  - Teilbarkeit allgemein: ist z durch x teilbar?
    $\Rightarrow$ Ist z % x gleich 0?

Inhalt

Der Zuweisungsoperator

- Bisher: mit dem Zuweisungsoperator (=) den Wert einer Variablen verändern
- Beispiel: y = 2*x;

- Zuweisung ist binäre Operation: <lvalue> = <rvalue>;
- Jede Operation hat ein Resultat (Ergebniswert)

- Resultat einer Zuweisung ist der zugewiesene Wert
- Ermöglicht folgende Verkettung:
  a = b = c = d = 0;

- Abarbeitung (Berechnung) von rechts nach links
  ⇒ Der Zuweisungsoperator ist rechts-assoziativ
- Die anderen binären Operatoren (Grundrechenarten und Modulo) sind links-assoziativ: Abarbeitung (Berechnung) von links nach rechts

Erweiterte Zuweisungen

- Kombination aus arithmetischen Operatoren und Zuweisung
- Abkürzende Schreibweise für die binäre Operation und Zuweisung des Ergebnisses an den ersten Operanden:
  - a += 2; // entspricht: a = a + 2;
  - a -= 2; // entspricht: a = a - 2;
  - a *= 2; // entspricht: a = a * 2;
  - a /= 2; // entspricht: a = a / 2;
  - a %= 2; // entspricht: a = a % 2;
- Vereinfachter Ablauf und Zeitersparnis: direktes Verändern der Speicherzelle des ersten Operanden anstatt Ergebnis der Operation temporär zu speichern und anschließend der Variablen zuweisen
- Bei "eingebauten / einfachen" Typen: Optimierung des Compilers führt zu gleicher Ausführung beider Varianten

Inhalt

Ganzzahlige Division vs. Gleitkomma-Division

```
int i = 2;
double d = 2.0;

cout << 7 / 2 << endl; // 3
cout << 7.0 / 2 << endl; // 3.5

cout << 7 / i << endl; // 3
cout << 7 / d << endl; // 3.5
```

- Automatische Konvertierung der Operanden
- Datentyp des Ergebnisses abhängig von Datentypen der Operanden

  $\Rightarrow$ Beides gilt für alle binären Operatoren, nur bei der Division kann es zu unterschiedlichen Ergebnissen kommen

Binäre Operatoren und automatische (implizite) Typkonvertierung

- Typ der Operation hängt vom Datentyp der Operanden ab
  (jeder Operator ist für jeden Datentyp definiert)
- Haben beide Operanden den gleichen Datentyp ⇒ keine Konvertierung,
  Verwenden des entsprechenden Operators
- Bei unterschiedlichen Datentypen der Operanden: Regeln gemäß
  "Rangfolge" der Datentypen (von niedrigem zu hohem Rang):

char – **short int** – **int** – **long int** – **long long int** – **float** – **double** – **long double**

- Konvertierungsregeln:
  1. Alle Typen unter **int** werden in **int** umgewandelt (integer promotion)
  2. Der Operand mit dem niedrigeren Typ wird in den Typ des Operanden mit dem höheren Typ umgewandelt
  3. Ist ein Typ **unsigned**, wird der andere Operand zu **unsigned** umgewandelt

Implizite vs. explizite Typkonvertierung

- Eben: implizite Konvertierung des Datentyps bei Operatoren
- Problem: ggf. Veränderung des Wertes ohne Fehlermeldung (bei unterschiedlicher Genauigkeit oder Verlassen des Wertebereiches)
- Zuweisung: implizite Konvertierung aller numerischen Datentypen ineinander

- Fehler vermeiden: klar ausdrücken, welcher Typ gemeint ist

  $\Rightarrow$ Literale richtig verwenden

  $\Rightarrow$ Beispiel: `2.0 / 3.0 * a` statt `2 / 3 * a`

```
int b = 2;
int c = 3;
double a = 3.15;
cout << a * b / c << endl; // 2.1
cout << b / c * a << endl; // 0.0
```

Explizite Typkonvertierung

- Explizite Typkonvertierung durch casten
- Schreibweise:
  (<ZIEL−DATENTYP>) <ZU−KONVERTIERENDER−WERT>

```
int b = 2;
int c = 3;
double a = 3.15;
cout << a * (double) b / (double) c << endl; // 2.1
cout << (double) b / (double) c * a << endl; // 2.1
```

- Andere Schreibweise (Funktionsschreibweise): Ziel-Datentyp als Funktion verwenden
  - Beispiele: **double** d = **double**(i); **int** i = **int**(d);
  - Zu konvertierender Ausdruck besser erkennbar durch Klammerung
    (Bei erster Schreibweise: Bindung der Operatoren entscheidend)

Neue Cast-Operatoren

- **static_cast**<TYP>(AUSDRUCK)
  - Häufigste Art der Umwandlung
  - Ein Wert, der in einem Typ vorliegt, soll (möglichst unverändert) in einem anderen Typ dargestellt werden
- **reinterpret_cast**<TYP>(AUSDRUCK)
  - Die vorliegende Repräsentation eines Typs soll unverändert als Repräsentation des neuen Typs verwendet werden
  - Der Compiler behandelt einfach im folgenden die Daten im Speicher so, als ob sie den neuen Typ hätten.

- **const_cast** und **dynamic_cast** mit Bedeutung in anderem Kontext

- Umwandlung von ganzzahligen zu Gleitkommawerten: ohne Genauigkeitsverlust
- Casten von Gleitkommawerten zu ganzzahligem Datentyp: Abschneiden der Stellen nach dem Komma (Runden “zur Null hin”)

Umwandlungsfunktionen

- **double** floor(**double**)
  - Ermitteln der nächstkleineren ganzen Zahl
  - Ergebnis bleibt vom Typ **double**, keine Konvertierung
  - Anschl. Konvertierung zu **int** liefert richtige Zahl, falls im Wertebereich
  - **double** d = floor(3.14); cout << d << endl; // 3
- **double** ceil(**double**)
  - Ermitteln der nächstgrößeren ganzen Zahl
  - Ergebnis bleibt vom Typ **double**, keine Konvertierung
  - Anschl. Konvertierung zu **int** liefert richtige Zahl, falls im Wertebereich
  - **double** d = ceil(3.14); cout << d << endl; // 4
- **double** round(**double**)
  - Rundet auf die nächste ganze Zahl
  - Ergebnis bleibt vom Typ **double**, keine Konvertierung
  - Anschl. Konvertierung zu **int** liefert richtige Zahl, falls im Wertebereich
  - **double** d = round(3.14); cout << d << endl; // 3

⇒ Erfordern **#include** <cmath>

Inhalt

Logische Ausdrücke (Boolesche Ausdrücke)

- Wichtig für Steueranweisungen (Kontrollstrukturen): Alternativen und Schleifen
- Ergebnis eines logischen Ausdrucks ist Wahrheitswert (**true**/**false**)

  $\Rightarrow$ Datentyp **bool**

- Beispiel:

```
int a = 5;
int b = 7;
bool isGreater;
isGreater = a > b;
```

Operatoren für logische Ausdrücke

Operator	unär/binär	Bedeutung
!	unär	logische Negation
<	binär	kleiner als
<=	binär	kleiner gleich
>	binär	größer als
>=	binär	größer gleich
==	binär	Test auf Gleichheit
! =	binär	Test auf Ungleichheit
&&	binär	logisches UND
||	binär	logisches ODER

Reihenfolge: absteigende Bindung (Priorität)

- 6 Operatoren für den Vergleich zweier (Zahlen-) Werte: $<, <=, >, >=, ==, !=$
  $\Rightarrow$ relationale Operatoren
- Verknüpfung logischer Ausdrücke durch &&, || und Negation durch !
  $\Rightarrow$ logische Operatoren

Beispiele für logische Ausdrücke

```
int a = 5;
int b = 7;

bool isGreater = a > b; // false
bool isGreater2 = !(a <= b); // false


bool isEqual = a == b; // false
bool isDifferent = a != b; // true


bool bothPositive = (a > 0) && (b > 0); // true
bool sumBetween0and10 = (a+b >= 0 && a+b <= 10); // false


bool multipleOfFive = (a % 5 == 0) || (b % 5 == 0); // true
bool bothOdd = (a % 2 == 1) && (b % 2 == 1); // true
```

Längere Verkettungen sind möglich!

Vorsicht bei Test auf Gleichheit

```
double d1 = 0.1/0.3;
double d2 = 20.0/60.0;
double d3 = d1-d2;

cout << (d3 == 0.0) << endl;
cout << "d3: " << d3 << endl;
// 0
// d3: 5.55112e-17
```

Besser:

```
cout << (abs(d3) <= 1e-12) << endl;
cout << "d3: " << d3 << endl;
// 1
// d3: 5.55112e-17
```

Funktion abs für Absolutbetrag erfordert **#include** <cmath>

Weitere logische Ausdrücke

- Numerische Werte (u.a. Ergebnisse von Berechnungen / Operatoren) können als logischer Ausdruck verwendet werden
  - Der Wert 0 entspricht **false**
  - Alle anderen Werte entsprechen **true**

    $\Rightarrow$ Typische Repräsentation von **true** und **false** durch 1 und 0:

    bool b1 = 1; statt **bool** b1=**true**; und

    bool b2 = 0; statt **bool** b2=**false**;
  - Beispiel: **bool** b = 2−4; cout << b << endl; // 1
- Zuweisungsoperator =
  - Wertzuweisung einer Variablen ist Operation
  - Ergebnis ist der zugewiesene Wert
  - Ergebniswert wird als Wahrheitswert interpretiert
  - Verwechselungsgefahr: Zuweisung (=) vs. Test auf Gleichheit (==)

Typische Fehlerquelle: Test auf Gleichheit

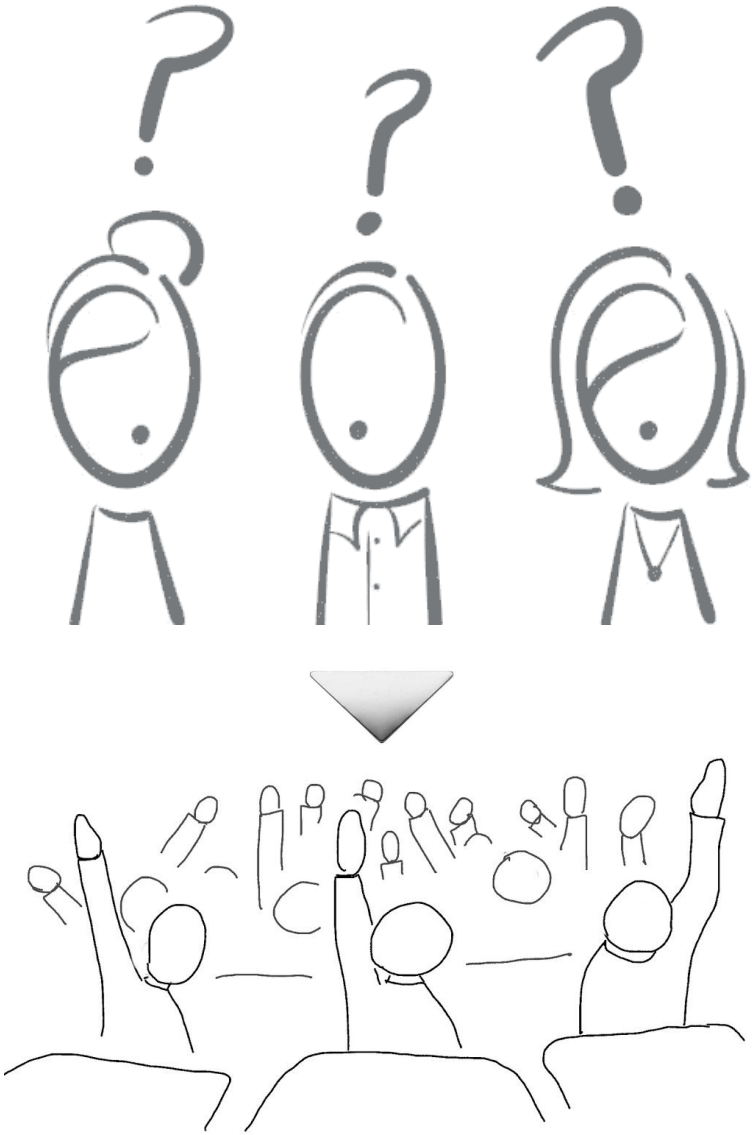
```
int x = 5;
int y = 7;

bool b1 = x == 2; // false
bool b2 = x = 2; // true, Wert von x ist nun gleich 2
```

- Keine Fehlermeldung des Compilers, weil:
  1. Zuweisung ist Operator mit Ergebniswert
  2. Als logischer Wert kann auch ein numerischer Wert verwendet werden

  $\Rightarrow$ Kein syntaktischer Fehler
- Abhilfe durch folgenden Ratschlag: Vergleich umgekehrt schreiben
  bool b3 = 2 == x; // Abarbeitung von rechts nach links
- Tippfehler führt zu ungültiger Zuweisung und wird vom Compiler als Fehler erkannt: **bool** b4 = 2 = x; // 2 = x ist ungueltig
- Funktioniert jedoch nicht beim Vergleich von Variablen:
  bool b4 = y = x; // true, keine Fehlermeldung, y hat Wert 5

Gibt es Fragen? (Es gibt keine dummen Fragen!)



"Excuse me, is this the Society for Asking Stupid Questions?"