Programmierkurs

Vorlesung 02

M.Sc. Laslo Hunhold

Department Mathematik/Informatik
Abteilung Informatik
Universität zu Köln

16. November 2020

Letzte Vorlesung

- Daten im Computer
- Binäre Kodierungen
- Typen, Variablen und Operatoren
- Kontrollstrukturen

Literale/Konstanten

Motivation

- Binäre Kodierung schlecht für Lesbarkeit
- Wollen Variablen auch mit festen Werten füllen (z.B. mit 9)
- „`byte i = 9`“ lesbarer als „`byte i = 00001001`“

Lösung: Literale

- Abstraktion der binären Kodierung im Quelltext
- Compiler wandelt Literale in binäre Kodierung um
- Kein Geschwindigkeitsverlust

```
int i = 12;
double d = 20.5;
String s = "Hallo Welt";
boolean b = false;
```

Integerliterale

- Dezimal: Einfache Dezimalzahl (mit optionalem Vorzeichen); z.B. 20, -51392, ...
- Oktal (Basis 8): Beginnt mit einer `0` gefolgt von Ziffern 0 bis 7; z.B. `0324`
- Hexadezimal (Basis 16): Beginnt mit `0x` oder `0X` gefolgt von Ziffern 0 bis 9 und a-f bzw. A-F; z.B. `0x20`, `0xDEADBEEF` (siehe „Hexspeak“)
- Binär (Basis 2): Beginnt mit `0b` oder `0B` gefolgt von Ziffern 0 und 1; z.B. `0b0101`, `0b1`
- Standardtyp ist int, `l` oder `L` am Ende hinzufügen für Typ long; z.B. `0xDEADBEEFL`
- Optionales Trennzeichen _ für Lesbarkeit; z.B. `0xDEAD_BEEF`

Fließkommazahlliterale

Bestandteile (der Reihe nach)

1. (optional) Vorzeichen + oder -
2. Eines der folgenden Zahlenformate
   - „`ziffern`“ gefolgt von einem Exponenten, Suffix oder beidem (damit unterscheidbar von einem Ganzzahlliteral)
   - „`ziffern.`“
   - „`ziffern.ziffern`“
3. (optional) Exponent der Form
   - Exponentenanzeiger `e` oder `E`
   - (optional) Vorzeichen + oder -
   - `ziffern`
4. (optional) Eines der folgenden Suffixe
   - `f` oder `F` für den `float`-Typ
   - `d` oder `D` für den `double`-Typ (Standard)

Beispiele: `-12e8`, `12.`, `-128.24e-15f`, entsprechen $-12 \cdot 10^{8}$, 12 und $-128.24 \cdot 10^{-15}$

Weitere

- Zeichenliterale (Typ `char`): Beginnen mit `'`, enden mit `'` und repräsentieren ein Zeichen; z.B. `'a'`, `'%'`, etc.
- Stringliterale (Typ `String`): Beginnen mit `"`, enden mit `"` und repräsentieren eine Zeichenkette; z.B. `"Hallo Welt"`, etc.
- Boolesche Literale (Typ `boolean`): `true` und `false`
- Nullzeiger: `null` (später)

Vertiefung Typumwandlung

Abbildung: Ariane 5 V88-Fehlschlag (float64-zu-int16-Typumwandlungsüberlauf), 4. Juni 1996, 300 Millionen Euro Schaden (Quelle: ESA 229241)

Implizite Typumwandlung

- Betrachte z.B. Zuweisungsoperator „=“
- Letzte Vorlesung: Implizite Typumwandlung findet statt, falls

  $$\text{Typ(Ausdruck)} \rightarrow \text{Typ(Variable)}$$

  „verlustfrei“ ist, sonst Fehlermeldung.

Was heißt „verlustfrei“?

- $\texttt{byte} \leq \texttt{short} = \texttt{char} \leq \texttt{int} \leq \texttt{long} \leq \texttt{float} \leq \texttt{double}$
- Umwandlung in einen „größeren“ oder „gleichgroßen“ Typ ist „verlustfrei“
- Zwischen Ganzzahltypen immer exakt
- Zwischen Ganzzahl- und Fließkommazahltypen häufig nicht exakt!

```
long z = 123456789L;
float f = z;
System.out.println(f); // 1.23456792E8
```

Explizite Typumwandlung

- Explizite Typumwandlung mit Cast-Operator „`(typ)`“; z.B.

```
double z = 5.7;
int d = (int)z;
```

- `byte` $\leq$ `short` $=$ `char` $\leq$ `int` $\leq$ `long` $\leq$ `float` $\leq$ `double`
- Erzwingt (eventuell verlustbehaftete) Umwandlung in einen „kleineren“ Typ

Operationen auf Variablen

Übersicht

Anzahl der Operanden

- Ein Operand: „Unärer“ Operator (z.B. - bei -1 („unäres Minus“))
- Zwei Operanden: „Binärer“ Operator (z.B. +, -, etc.)
- Drei Operanden: „Ternärer“ Operator (?:)

Assoziativität

- Linksassoziativ: Auswertungsrichtung $\rightarrow$ (z.B. -, /) (Standard)
- Rechtsassoziativ: Auswertungsrichtung $\leftarrow$ (z.B. =, !)
- Nicht assoziativ: Keine feste Auswertungsrichtung (Klammerung entscheidet)

```
int a = 3 - 4 - 5;
int a = ((3 - 4) - 5); /* Linksassoziative Auswertung */
int a = (3 - (4 - 5)); /* Rechtsassoziative Auswertung */
```

Präzedenz (später)

Unäre Operatoren

- Prädekrement/Präinkrement (--/++)
  Verringert/Erhöht Zahloperanden (Variable) um 1 und gibt seinen Wert zurück, rechtsassoziativ

```
int n = 10;
int p = ++n;
/* p = 11, n = 11 */
```

- Postdekrement/Postinkrement (--/++)
  Gibt Wert des Zahloperanden (Variable) zurück und verringert/erhöht ihn um 1, nicht assoziativ

```
int n = 10;
int p = n++;
/* p = 10, n = 11 */
```

- Negation (-)
  gibt negierten Operandenwert zurück, rechtsassoziativ

```
int n = 10;
n = -n;
/* n = -10 */
```

- Cast-Operator ((T))
  gibt Operandenwert auf Typ T gecastet zurück, rechtsassoziativ

Binäre Operatoren

- „+“: Addition
- „−“: Subtraktion
- „*“: Multiplikation
- „/“: Division
- „%“: Modulo (Rest der Division)

```
int n = 20;
int p = n % 6;
/* p = 2 */
```

- Achtung: Operatorergebnis ist ganzzahlig, wenn beide Operanden ganzzahlig sind

```
int a = 1, b = 2;
double d = a / b;
/* d = 0.0, da a/b = 0 */
double d = (double)a / b;
/* d = 0.5 */
```

- „=“: Zuweisungsoperator, rechtsassoziativ

Boolesche Ausdrücke

Vergleichsoperatoren

- Binäre Operatoren, geben `true` oder `false` zurück
- Definiert für Ganzzahlen und Fließkommazahlen
- „`>`“: Echt größer
- „`>=`“: Größer oder gleich
- „`==`“: Gleich, definiert für alle Typen
- „`!=`“: Ungleich, definiert für alle Typen
- „`<=`“: Kleiner oder gleich
- „`<`“: Echt kleiner

Boolesche Operatoren

- Operieren auf `boolean` und geben `boolean` zurück
- „`!`“: NOT, logische Negation, unärer Operator, rechtsassoziativ
- „`&&`“: AND, logisches Und, binärer Operator
- „`||`“: OR, logisches Oder, binärer Operator

```
boolean b = (2 > 5) || (1 != 2) || (true && !false);
```

Ternärer Operator „?:“

- Allgemeine Form

```
bedingung ? wert1 : wert2
```

- Wenn `bedingung` wahr ist, gebe `wert1` zurück, sonst `wert2`
- Beispiel:

```
x = bedingung ? 2 : 3;
```

ist also eine Kurzschreibweise für

```
if (bedingung) {
        x = 2;
} else {
        x = 3;
}
```

Bitweise Operatoren/Verschiebungen

- Operieren logisch auf Bits eines `int`, im Folgenden nur Pseudocode
- „`~`“: Bitweises NOT, unärer Operator, rechtsassoziativ
  `~0100 = 1011, ~1111 = 0000`
- „`&`“: Bitweises AND, binärer Operator
  `1110 & 1011 = 1010, 0001 & 0111 = 0001`
- „`|`“: Bitweises OR, binärer Operator
  `1110 | 1011 = 1111, 0011 | 0001 = 0011`
- „`^`“: Bitweises XOR, binärer Operator
  `1110 ^ 1011 = 0101, 0011 ^ 0001 = 0010`
- „`<<`“: Bitshift Links, rechts auffüllen mit Nullen, binärer Operator
  `0011 << 1 = 0110, 0011 << 3 = 1000`
- „`>>>`“: Bitshift Rechts, links auffüllen mit Nullen, binärer Operator
  `0011 >>> 1 = 0001, 1110 >>> 4 = 0000`
- „`>>`“: Nicht empfohlen, da beim Auffüllen vorzeichenabhängig

```
int b = 0b01011101;
boolean thirdbitset = ((b & (1 << 2)) != 0);
```

Operatorzuweisung

- Ausdrücke der Form

```
x = x + 2;
```

kann man auch schreiben als

```
x += 2;
```

- Kombination von Operation und Zuweisung
- Funktioniert mit +, -, *, /, %, &, |, ^, >>>, >>, <<

Vertiefung Kontrollstrukturen

Übersicht

Bedingtes Überspringen/Ausführen von Anweisungen

- `if`
- `switch`

Bedingtes Wiederholen von Anweisungen (Schleifen)

- `while`
- `for`
- `do...while`

Bedingung wird als BOOLEscher Ausdruck angegeben (außer bei `switch`).

if

```
1  int x = 4;
2
3  if (x > 0) {
4          ...
5  } else if (x == 0) {
6          ...
7  } else { /* x < 0 */
8          ...
9  }
```

- Abarbeitung von oben nach unten
- `else if` und `else` sind optional
- Trifft eine Bedingung zu, werden nur die enthaltenen Ausdrücke ausgeführt und keine weiteren Bedingungen geprüft
- `else if` kann mehrmals vorkommen
- `else` fängt alles auf, was keine vorherige Bedingung erfüllte

switch

```
 1  int x = 4;
 2
 3  switch (x) {
 4  case 1:
 5          ... /* x == 1 */
 6          break;
 7  case 2:
 8  case 3:
 9  case 4:
10          ... /* x = 2 oder x = 3 oder x = 4 */
11          break;
12  default:
13          ... /* x ist nicht 1, 2, 3 oder 4 */
14          break;
15  }
```

- `break` beachten (historische Fehlerquelle)!
- Ähnlich zu `if`, aber übersichtlicher bei vielen Bedingungen

while

```
1 while (bedingung) {
2         ...
3 }
```

- Solange `bedingung` wahr ist, werden die enthaltenen Ausdrücke ausgeführt
- Prüfung immer zu Beginn der Schleife

for

```
1  for (vorausdruck; bedingung; innenausdruck) {
2          ...
3  }
```

ist eine Kurzschreibweise für

```
1  vorausdruck; /* Vorbereitung */
2  while (bedingung) {
3          ...
4          innenausdruck; /* Iteration */
5  }
```

Beispiel:

```
1  int x;
2
3  for (x = 0; x <= 4; x++) {
4          ...
5  }
```

do...while

```
1 do {
2         ...
3 } while (bedingung)
```

- Ob `bedingung` wahr ist wird erst am Ende geprüft (nicht am Anfang)
- Die enthaltenen Ausdrücke werden deshalb immer mindestens einmal ausgeführt

Abbildung: Sojus-U/Fregat 11A511U, 16. Juli 2000 (Quelle: ESA 195530)