Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 6: Wertausdrücke (Expressions)

- Operatoren (infix, präfix, postfix) und Funktionen
- Prioritäten (Rangfolge), Assoziativität
- Operatoren in C/C++
- Lvalues, Zuweisungen, Seiteneffekte
- Typkorrektheit, Automatische Typumwandlungen

Inhalt

Wertausdrücke (1)

- Wertausdrücke (oder Ausrücke, engl. Expressions) sind syntaktische Konstrukte, die zu einem Element eines Datentyps ausgewertet werden können (z.B. zu einer ganzen Zahl, einer Gleitkommazahl, einer Hauptspeicheradresse).
- Beispiel: `i + 1` ist ein Wertausdruck.
- Der Wert ist von der aktuellen Belegung der Variablen abhängig (Inhalt der Variablen).
- Wenn z.B. `i` gerade den Wert 3 hat, so hat der Wertausdruck `i + 1` den Wert 4.

Wertausdrücke (2)

- In Sprachen wie Pascal spezifizieren Wertausdrücke nur die Berechnung eines Wertes, und bewirken aber keine Zustandsänderung.

  Eher unabsichtlich könnte das dort durch den Aufruf einer Funktion im Wertausdruck doch passieren, wenn die Funktion z.B. eine Zuweisung an eine globale Variable enthält. Aber das gilt dort als schlechter Stil oder ist sogar verboten.

- In C/C++ sind dagegen auch Zuweisungen Wertausdrücke, also z.B. `i = 5`.

  Diese Sprachen erlauben ja allgemein recht kompakte Formulierungen, die manchmal auch als kryptisch empfunden werden.

Wertausdrücke (3)

- Elementare Wertausdrücke sind:
  - ◇ Konstanten/Datentyp-Literale, z.B. `0.5`.
  - ◇ Namen von Variablen, z.B. `x`.
- Aus diesen ganz einfachen Wertausdrücken können komplexere zusammengesetzt werden, u.a. durch
  - ◇ Anwendung eines Operators, z.B. `x + 0.5`.
  - ◇ Aufruf einer Funktion, z.B. `sin(x)`.
- Da dies wieder Wertausdrücke sind, kann man daraus auch noch komplexere zusammensetzen.

Wertausdrücke (4)

- Semantisch (hinsichtlich der Bedeutung) besteht zwischen einem Funktionsaufruf und der Anwendung eines Operators kein Unterschied:
  - ◇ `x + 0.5` bedeutet auch nur, dass die Additionsfunktion auf den aktuellen Wert von `x` und die Zahl 0.5 angewendet wird.
  - ◇ Bei passender Deklaration einer Funktion `add` könnte man auch `add(x, 0.5)` schreiben.

    Eine solche Funktion ist nicht in C++ vordefiniert, aber man kann man sie sich leicht selbst deklarieren. Übrigens sind mindestens für Operatoren für benutzerdefinierte Datentypen in C++ tatsächlich Funktionen hinterlegt.

Wertausdrücke (5)

- Syntaktisch (hinsichtlich der Art, wie man es aufschreibt) besteht natürlich ein Unterschied:
  - ◇ Bei einem Funktionsaufruf, z.B. `sin(x)`, wird zuerst der Name der Funktion geschrieben, und dann in Klammern die Eingabewerte (durch `','` getrennt, falls es mehrere sind).
  - ◇ Ein Operator wie + wird zwischen seine Eingabewerte geschrieben, z.B. `x + 0.5`. Klammern sind nur manchmal notwendig.

    Die Notwendigkeit von Klammern ergibt sich aus der Priorität (Bindungsstärke) der Operatoren, siehe unten.

Wertausdrücke (6)

Noch mehr Begriffe (Eingabewerte):

- Die Eingabewerte heißen auch die Argumente der Funktion.
- Die Anzahl der Argumente heißt auch die Stelligkeit der Funktion, z.B. ist
  - ◇ `sin` eine einstellige Funktion.
  - ◇ die Addition eine zweistellige Funktion.
- Die Argumente eines Operators werden auch Operanden genannt.

Inhalt

Operatorsyntax (1)

- Nach der Anzahl von Operanden unterscheidet man:
  - ◇ unäre Operatoren: Ein Argument
  - ◇ binäre Operatoren: Zwei Argumente
  - ◇ ternäre Operatoren (selten): Drei Argumente
- Manchmal wird auch das gleiche Symbol für unterschiedliche Typen von Operatoren verwendet:
  - ◇ `-x`: unäres Minus
  - ◇ `x-y`: binäres Minus.

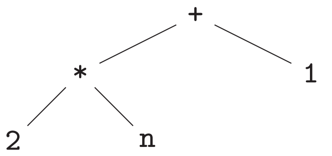
  Es werden ganz unterschiedliche Maschinenbefehle erzeugt (unterschiedliche Funktionen).

Operatorsyntax (2)

- Nach der Position des Operators (im Vergleich zu den Operanden) unterscheidet man:
  - ◇ Präfix-Operatoren: Operator (unär) steht vor dem Operand, z.B. `-x`.
  - ◇ Postfix-Operatoren: Operator (unär) steht nach dem Operand, z.B. `i++`.
  - ◇ Infix-Operatoren: Operator (binär) steht zwischen den Operanden, z.B. `x + y`.
  - ◇ Mixfix-Operatoren: Operator (beliebig) besteht aus mehreren Teilen, z.B. `b ? x : y`.

Operatorsyntax (3)

- Die Struktur von komplexen Wertausdrücken, z.B. `2*n + 1`, kann man als "Baum" veranschaulichen:



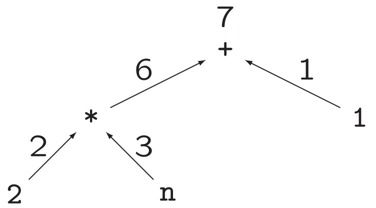
- In der Informatik wachsen Bäume verkehrt herum:
  - ◇ Der oberste Knoten ist die Wurzel des Baumes.
  - ◇ Hat Knoten X (oben) eine Verbindung zu Knoten Y (unten), so heißt Y Nachfolger/Kind von X.
  - ◇ Knoten ohne Nachfolger heißen Blätter.

Operatorsyntax (4)

- Im Operatorbaum ist jeder innere Knoten (d.h. alle Knoten außer Blätter) mit einem Operator (oder dem Namen einer Funktion) markiert.
- Blätter sind markiert mit:
  - ◇ Namen von Variablen,
  - ◇ Konstanten (Datentyp-Literalen), oder
  - ◇ Namen von nullstelligen Funktionen.
- Die Operanden eines Operators sind die Teilbäume, die mit den Kindknoten des Operators beginnen.

Operatorsyntax (5)

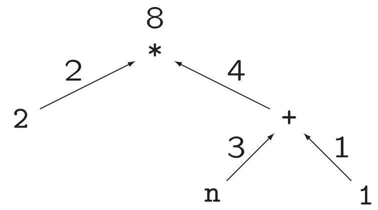
- Werte fließen im Operatorbaum von unten nach oben.
- Hat z.B. `n` gerade den Wert 3, so ergibt sich:



- Der Wert des gesamten Ausdrucks ist der Wert, der sich an der Wurzel des Baums errechnet: 7.

Operatorsyntax (6)

- Man beachte, dass der Ausdruck `2 * n + 1` nicht für folgenden Operatorbaum steht:



- Der Grund ist die bekannte Regel “Punktrechnung vor Strichrechnung”. Falls man die obige Struktur will, muß man Klammern setzen: `2 * (n + 1)`.

Operatorsyntax (7)

- Operatoren haben “Prioritäten”.

  Auch “Bindungsstärken” genannt.
- “*” hat höhere Priorität als “+”.

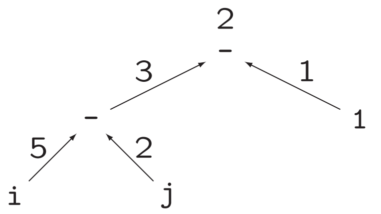
  Man kann auch sagen: “*” bindet stärker als “+”.
- Um die implizite Klammerung explizit zu machen, kann man so vorgehen, dass in der Reihenfolge der Prioritäten jeder Operator zusammen mit den jeweils kürzestmöglichen Operanden links und/oder rechts eingeklammert wird.

Operatorsyntax (8)

- Beispiel:
  - ◇ Betrachtet sei wieder `2 * n + 1`.
  - ◇ Der Operator höchster Priorität ist $*$.
  - ◇ Links steht ohnehin nur `2`, rechts bestünde die Wahl zwischen den Operanden `n` und `n + 1`.
  - ◇ Es wird der kürzere gewählt, also `n`, und der Operator mit seinen Operanden in Klammern eingeschlossen: `(2 * n) + 1`.
  - ◇ Wenn jetzt `+` drankommt, muß links der komplette geklammerte Ausdruck gewählt werden.

Operatorsyntax (9)

- Die obigen Regeln legen noch nicht die Struktur von `i - j - 1` fest.
- In diesem Fall wird von links geklammert, also `(i - j) - 1`.
- Ist $i = 5$ und $j = 2$, so ergibt sich:



Operatorsyntax (10)

- Um diese implizite Klammerung zu beschreiben, sagt man: “der Operator `-` ist linksassoziativ”.

  Das Assoziativgesetz gilt aber natürlich gerade nicht für `-`, denn $(a - b) - c$ ist im allgemeinen verschieden von $a - (b - c)$. Selbst für den Operator `+`, für den mathematisch das Assoziativgesetz gelten würde, ist es in der Berechnung möglicherweise aufgrund unterschiedlicher Rundung oder arithmetischen Überläufen verletzt. Deswegen legt C/C++ auch hier eine klar definierte implizite Klammerung fest (von links).

- Selbstverständlich darf man auch eigentlich überflüssige Klammern setzen, um die Struktur noch deutlicher zu machen, z.B. `(i - j) - 1`.

Operatorsyntax (11)

- In C/C++ sind fast alle Operatoren linksassoziativ, ausgenommen sind nur:
  - ◇ Präfixoperatoren, hier gibt es ja nur die Möglichkeit, implizit von rechts zu klammern, z.B. `- -x` bedeutet `-(-x)`.

    Das Leerzeichen zwischen den beiden `-` ist hier wichtig, sonst liefert die lexikalische Analyse den Dekrement-Operator `--`.

  - ◇ Zuweisungen, z.B. steht `a = b = c` für `a = (b = c)`.

    Hierzu muß man wissen, dass der Wert einer Zuweisung der zugewiesene Wert ist (s.u.). Der Wert von `c` wird also erst in `b` gespeichert und dann in `a`. Die Regel gilt auch für `+=` etc.

Operatorsyntax (12)

- Aufgabe:
  - ◇ Zeichnen Sie einen Operatorbaum für

    `a + 2 * (b - c + 1)`

    (die Operatoren + und - haben gleiche Priorität).
  - ◇ Was ist der Wert dieses Ausdrucks für `a=5`, `b=8`, `c=3`?
- Bemerkung: Die in der Mathematik übliche Notation $2n + 1$ ist in C/C++ (und fast allen anderen Programmiersprachen) nicht erlaubt: Man muß den Multiplikationsoperator explizit schreiben.

Prioritätsstufen

18	`::`	Gültigkeitsbereich
17	`++` (Postfix), `.`, `->`, `[]`, `f()`, ...	Postfix-Operatoren
16	`-` (unär), `!`, `*` (deref), `++` (Präfix), ...	Präfix-Operatoren
15	`.*`, `->*`	Zeiger auf Komp.
14	`*`, `/`, `%`	Multiplikation etc.
13	`+`, `-`	Addition, Subtraktion
12	`<<`, `>>`	Shift etc.
11	`<`, `<=`, `>`, `>=`	kleiner etc.
10	`==`,`!=`	gleich, verschieden
9	`&`	Bit-und
8	`^`	Bit-xor
7	`|`	Bit-oder
6	`&&`	und
5	`||`	oder
4	`?:`	Bedingter Ausdruck
3	`=`, `+=`, `-=`, `*=`, `/=`, ...	Zuweisungen
2	`throw`	Exception auslösen
1	`,`	Sequenz

Arithmetische Operatoren (1)

- `+`: Addition
- `-`: Subtraktion
- `*`: Multiplikation
- `/`: Division

  Wenn man / auf zwei ganze Zahlen anwendet, erhält man eine ganze Zahl, und zwar wird für positive Eingabewerte immer abgerundet, z.B. ist `5/3=1`. Sind einer oder beide der Eingabewerte negativ, ist die Auf/Abrundung implementierungsabhängig. Es ist aber immer `(a/b)*b + a%b = a` garantiert (außer für `b=0`, das gibt einen Fehler). Ist einer der beiden Operanden eine Gleitkommazahl, erhält man die normale Division (mit einer Gleitkommazahl als Ergebnis).

Arithmetische Operatoren (2)

- `%`: Divisionsrest (Modulo)

  Z.B. `8 % 3 = 2`.
- `-` (unär): Negation/Komplement
- `+` (unär): Identität
- Es gibt drei Prioritätsstufen:
  - ◇ Die unären Operatoren binden am stärksten,
  - ◇ dann die Punktrechnung (`*`, `/`, `%`),
  - ◇ und zuletzt die Strichrechnung (binäres `+`, `-`).

Inhalt

Typ-Korrektheit (1)

- Operatoren und Funktionen können nur auf Werte der korrekten Datentypen angewendet werden.
- Wenn z.B. `x` den Typ `double` hat, wird `x % 2` einen Typfehler liefern:

```
'%' : illegal, left operand has type 'double'
```

- Jede Funktion/Jeder Operator ist nur für Eingabewerte von bestimmten Datentypen definiert, und produziert dann jeweils einen Ausgabewert eines definierten Datentyps.

Typ-Korrektheit (2)

- Der Modulo-Operator `%` ist z.B. für Eingabewerte vom Typ `int` definiert, und produziert dann einen Wert vom Typ `int` als Ergebnis.
- Man kann dies in folgender Form aufschreiben:

$$\%\colon \texttt{int} \times \texttt{int} \to \texttt{int}$$

Dies ist die in der Mathematik übliche Notation, kein C/C++. Funktionsdeklarationen in C/C++ behandeln wir später.

- Die Spezifikation konkreter Eingabetypen und des zugehörigen Resultattyps nennt man auch die Signatur des Operators/der Funktion.

Typ-Korrektheit (3)

- Der gleiche Operator kann mit mehreren verschiedenen Signaturen definiert sein, er könnte dann auch völlig unterschiedliche Funktionen berechnen.
- Man nennt solche Operatoren “überladen”.
- Z.B. liefert `7.0 / 2.0` das Ergebnis `3.5`, degegen liefert `7 / 2` den Wert `3`.
- Es gibt also zwei verschiedene Varianten der Division (eigentlich noch mehr):
  - ◇ $/\colon \texttt{int} \times \texttt{int} \to \texttt{int}$
  - ◇ $/\colon \texttt{double} \times \texttt{double} \to \texttt{double}$

Typ-Korrektheit (4)

- Auch die anderen arithmetischen Operatoren sind mit verschiedenen Typvarianten überladen, selbst wenn es da nicht so offensichtlich ist.
- Z.B. muß der Compiler für eine Integer-Addition und eine Gleitkomma-Addition ganz unterschiedliche Maschinenbefehle erzeugen.

  Die Zahlen sind ja ganz verschieden in Bitmustern codiert.
- Auch `unsigned`-Werte und Werte unterschiedlicher Längen (etwa `double` und `long double`) müssen anders behandelt werden.

Typ-Korrektheit (5)

- Die arithmetischen Operatoren `+`, `-`, `*`, `/` haben insgesamt 7 Varianten der Form $T \times T \to T$
  - ◇ `int` × `int` → `int`
  - ◇ `unsigned int` × `unsigned int` → `unsigned int`
  - ◇ `long` × `long` → `long`
  - ◇ `unsigned long` × `unsigned long` → `unsigned long`
  - ◇ `float` × `float` → `float`
  - ◇ `double` × `double` → `double`
  - ◇ `long double` × `long double` → `long double`

Typ-Korrektheit (6)

- Man kann aber z.B. auch `s + 1` schreiben, wenn `s` den Typ `short` hat.
- Hierzu gibt es “integral promotions” (automatische Typ-Vergrößerungen, “Ganzzahl-Erweiterungen”):
  - ◇ Alle Typen, die kleiner als `int` sind, nämlich `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, werden automatisch in `int` umgewandelt.

    Unter der Voraussetzung, dass sie wirklich kleiner als `int` sind, also `int` eine Obermenge $\supseteq$ der Werte des jeweiligen Typs enthält. So ist garantiert, dass bei der Umwandlung kein Wert zerstört wird. Auf 16-Bit Maschinen wäre z.B. `unsigned short` nicht eine Teilmenge von `int`, dann wird nach `unsigned int` umgewandelt.

Typ-Korrektheit (7)

- “integral promotions”, Forts.:
  - ◇ Der Typ `bool` wird ebenso in ein `int` umgewandelt: `false` wird zu 0, und `true` wird zu 1.
  - ◇ Aufzählungstypen und `wchar_t` werden in den ersten Typ folgender Liste umgewandelt, der alle Werte des Quelltyps aufnehmen kann: `int`, `unsigned int`, `long`, `unsigned long`.

    Fast immer dürfte das `int` sein. Aufzählungstypen werden später behandelt, sie sind einfach eine Liste von möglichen Werten, z.B. `montag`, `dienstag`, ..., `sonntag`. Man kann dort Zahlwerte zur Repräsentation explizit festlegen, und die könnten groß sein (z.B. um Mengen dieser Werte mit Bitmustern zu repräsentieren).

Typ-Korrektheit (8)

- Der Zweck der “integral promotions” ist es, dem Programmierer des Compilers das Leben etwas zu erleichtern: Er muß für die arithmetischen Operatoren nicht ganz so viele Fälle zu behandeln.

  Eventuell war das historisch auch in C wichtig, weil früher Argumenttypen beim Funktionsaufruf nicht unbedingt bekannt waren. Es wurde durch die “integral promotions” z.B. niemals ein `char`-Wert übergeben, sondern immer mindestens ein `int`. So konnte es weniger Typfehler geben. In C++ sind die Argumenttypen beim Funktionsaufruf aber immer bekannt, und nun können auch Werte der kleinen arithmetischen Typen ohne Umwandlung übergeben werden.
  Falls der Compiler-Entwickler möchte, könnte er natürlich z.B. einen Maschinenbefehl für die Addition einzelner Bytes einsetzen, wenn der Benutzer keinen Unterschied bemerken kann. (Also die Summe zweier `char`-Werte wieder in ein `char` gespeichert wird.)

Typ-Korrektheit (9)

- Es gibt noch mehr Typ-Umwandlungen: Man darf z.B. auch `x + 1` schreiben, wenn `x` ein `float` ist.
- In diesem Fall würde `1` zunächst in ein `float` umgewandelt, und dann

  $$\texttt{+}\colon \texttt{float} \times \texttt{float} \to \texttt{float}$$

  angewendet.
- Allgemein werden die beiden Operanden eines arithmetischen Operators (und anderer Operatoren) zuerst in einen gemeinsamen Typ umgewandelt.
- Dies ist der “größere” der beiden Typen.

Typ-Korrektheit (10)

- Die Regeln sind:
  - ◇ Ist `long double` der Typ von einem der beiden Operanden, wird der andere Operand in diesen Typ konvertiert.
  - ◇ Ist andernfalls einer vom Typ `double`, wird der andere in diesen Typ konvertiert.
  - ◇ Trifft auch das nicht zu, aber ist einer vom Typ `float`, wird der andere in diesen Typ konvertiert.
  - ◇ Ansonsten werden zunächst die “integral promotions” auf beide Operanden angewendet.

Typ-Korrektheit (11)

- Regeln für gemeinsamen Typ beider Operanden, Forts.:
  - ◇ Ist nun `unsigned long` der Typ eines der beiden Operanden, so ist das der gemeinsame Typ.
  - ◇ Ist einer vom Typ `long int`, und der andere vom Typ `unsigned int`, und hat `long int` eine Obermenge der Werte von `unsigned int` (16-Bit Maschine), so wird `long int` gewählt. Gilt dies nicht (32-Bit Maschine), ist `unsigned long` der gemeinsame Zieltyp.

Typ-Korrektheit (12)

- Regeln für gemeinsamen Typ beider Operanden, Forts.:
  - ◇ Ist sonst einer von beiden Operandentypen `long`, wird der andere Operand in diesen Typ umgewandelt.
  - ◇ Gilt auch das nicht, und ist einer der Operanden vom Typ `unsigned int`, so ist das der gemeinsame Typ.
  - ◇ Ansonsten können die Operanden nur noch beide vom Typ `int` sein.

Inhalt

Vergleichsoperatoren (1)

- `==`: gleich
- `!=`: verschieden
- `<`: kleiner
- `<=`: kleiner oder gleich (kleinergleich)
- `>`: größer
- `>=`: größer oder gleich (größergleich)
- Diese Operatoren haben geringere Priorität als die arithmetischen Operatoren, z.B. wird `a - b > 10` als `(a - b) > 10` verstanden.

Vergleichsoperatoren (2)

- Die Vergleichsoperatoren haben die Signatur(en)

  $$T \times T \to \texttt{bool}$$

  wobei T ein Zahl-Datentyp (`int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`) oder ein Zeiger-Typ (Pointer-Typ) sein kann.

  Zeiger-Typen werden später erklärt (Hauptspeicher-Adressen).

- Vergleiche zwischen `unsigned`-Werten und negativen `int`-Werten funktionieren nicht wie erwartet. Ist z.B. `int i = -1` und `unsigned u = 0`, so gilt `i > u`!

  Das Problem ist, dass der `int`-Wert in ein `unsigned` umgewandelt wird, und dabei eine sehr große positive Zahl wird.

Vergleichsoperatoren (3)

- Die aus der Mathematik bekannte Schreibweise $1 \le n \le 100$ funktioniert in allen mir bekannten Programmiersprachen nicht.
- C/C++ sind aber besonders gefährlich, weil hier

  `1 <= i <= 100`

  ein legaler Ausdruck ist. Er ist aber auch dann wahr, wenn `i` z.B. die Werte -1 oder 200 hat.

  Implizite Klammerung von links: `(1 <= i) <= 100`. Nun liefert `1 <= i` einen boolschen Wert, der für den zweiten Vergleich in eine ganze Zahl umgewandelt wird, und zwar 0 oder 1. Beide sind kleiner als 100. Die meisten anderen Sprachen liefern einen Typfehler, weil man dort boolesche Werte nicht mit Zahlen vergleichen kann.

Logische Operatoren (1)

- `&&`: Logisches "und" (Konjunktion).

P	Q	P `&&` Q
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

  P `&&` Q ist dann und nur dann wahr, wenn P und Q beide wahr sind.
- `&&` hat eine geringere Priorität als die Vergleiche.
- Z.B. funktioniert `1 <= i && i <= 100` wie erwartet.

Logische Operatoren (2)

- `||`: Logisches "oder" (Disjunktion).

P	Q	P `||` Q
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

  P `||` Q ist wahr genau dann, wenn mindestens einer der Operanden P und Q wahr ist.
- `||` hat eine geringere Priorität als `&&`.

Logische Operatoren (3)

- `!`: Logisches "nicht" (Negation).

P	`!` P
`false`	`true`
`true`	`false`

  `!` P ist wahr genau dann, wenn P falsch ist.
- `!` hat eine sehr hohe Priorität.

  Wie alle Präfixoperatoren.
- Z.B. sind bei `!(n>100)` die Klammern nötig.

  Natürlich könnte man auch einfach `n <= 100` schreiben.

Logische Operatoren (4)

- C/C++ garantieren, dass bei P `&&` Q der zweite Operand (Q) nur dann ausgewertet wird, wenn der erste (P) wahr ist ("short-circuit-evaluation").

  Z.B. in Pascal wird das nicht garantiert.

- Z.B. kann `n != 0 && a/n > 2` keinen Fehler geben.

  In Pascal müßte man dagegen explizit ein `if` verwenden, wenn man sichergehen möchte, dass auch ein anderer Compiler (bzw. eine neue Compilerversion) niemals die Division ausführt, wenn `n` gleich `0` ist.

- Entsprechend wird bei `||` der zweite Operand nur dann ausgewertet, wenn der erste falsch ist.

Bit-Operatoren (1)

- Man kann ganze Zahlen auch als Folgen von Bits auffassen.
- Z.B. wird 5 intern (im Dualsystem) als `0...00101` dargestellt.
- Die üblichen logischen Verknüpfungen können jetzt auf jede Bitposition angewendet werden, wobei `1` "`true`" entspricht und `0` "`false`".
- Wenn `int` 32 Bit groß ist, können mit einem Maschinenbefehl z.B. 32 "und"-Verknüpfungen durchgeführt werden.

Bit-Operatoren (2)

- Beispiel:

1	0	0	1	1	···	1	0	0	1	0
|	|	|	|	|	&	|	|	|	|	|
1	0	1	1	0	···	0	1	0	1	1
↓	↓	↓	↓	↓		↓	↓	↓	↓	↓
1	0	0	1	0	···	0	0	0	1	0

- Z.B. können Mengen (mit bis zu 32 Elementen in der Grundmenge) so effizient repräsentiert werden.

  Jede Bitposition steht für ein Element der Grundmenge. Ist das Bit 1, so ist das Element enthalten, ist es 0, so ist es nicht enthalten. `&` wäre dann der Mengenschnitt $\cap$, und `|` die Vereinigung $\cup$.

Bit-Operatoren (3)

- Die bitweisen logischen Verknüpfungen sind:
  - ◇ `&`: Bit-und
  - ◇ `|`: Bit-oder
  - ◇ `^`: Bit–exklusiv-oder (XOR)
  - ◇ `~`: Bit-Komplement (Negation)

A	B	A & B	A | B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bit-Operatoren (4)

- Alle Bit-Operatoren haben die Signatur(en)

  $$T \times T \rightarrow T,$$

  wobei T einer der folgenden Typen ist:

  `int`, `unsigned int`, `long`, `unsigned long`.

  Kleinere ganzzahlige Typen, z.B. `char` und `short` werden (zumindest konzeptuell) zuerst in `int` umgewandelt (mit den "integral promotions", s.o.).

- Z.B. würde die Anwendung eines Bit-Operators auf einen `float`-Wert einen Typfehler geben.

Bit-Operatoren (5)

- Z.B. kann man mit

  ```
  if(s & 0x8) { ... }
  ```

  testen, ob das Bit an Position 4 gesetzt ist.

  ◇ Falls es gesetzt ist, ist das Ergebnis `0x8`. C/C++ behandeln jeden Wert $\neq 0$ wie "`true`" (wahr).

  ◇ Falls es nicht gesetzt ist, ist das Ergebnis `0` (wird in C/C++ wie "`false`" behandelt).

  Man könnte auch `(s & 0x8) == 0x8` schreiben, oder `(s & 0x8) != 0`.

  Die Hexadezimalnotation ist natürlich nicht nötig, aber so braucht man keine großen Zweierpotenzen auswendig zu lernen.

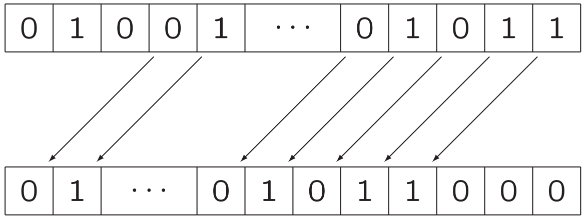
Bit-Operatoren (6)

- Es gibt noch zwei weitere Bit-Operationen:

  ◇ `<<`: Links-Shift.

  ◇ `>>`: Rechts-Shift.

- Z.B. liefert `n << 3` den Wert der Zahl `n` um drei Bit-Positionen nach links verschoben.



Bit-Operatoren (7)

- Ein Shift um drei Bit-Positionen nach links entspricht einer Multiplikation mit $8 = 2^3$.

  Üblicherweise sind Shift-Operationen schnell, während Multiplikationen möglicherweise einige Taktzyklen länger dauern. Wenn der Compiler eine Multiplikation mit einer Zweierpotenz entdeckt, wird er selbständig einen Shift-Befehl benutzen.

- Beim Rechts-Shift von negativen Zahlen ist nicht definiert, ob das Vorzeichenbit beim Shiften erhalten bleibt, also von links 1-Bits hereingeschoben werden, oder nicht.

  Ein Shift mit einem negativen Wert für die Anzahl der Bit-Positionen (rechter Operand) ist nicht erlaubt bzw. undefiniert.

Inhalt

Zuweisungen (1)

- = speichert den Wert des rechten Operanden in die Variable auf der linken Seite.
- Tatsächlich kann auf der linken Seite nicht nur eine Variable stehen, sondern auch ein Ausdruck, der zu einer Variablen ausgewertet werden kann.

  Z.B. ein Array- oder Strukturzugriff, oder die Dereferenzierung eines Zeigers (alles Dinge, die in dieser Vorlesung erst später kommen).
- Ein "Lvalue" ist ein Wert, der links von einer Zuweisung stehen darf, also eine eindeutig zu bestimmende Adresse hat (wie eine Variable).
- Z.B. ist `(a+1) = 5` ein Fehler: `a+1` ist kein Lvalue.

Zuweisungen (2)

- Man kann die Typkorrektheit durch Einführung von neuen Typen $Lvalue(T)$ beschreiben, die der Typ einer Variablen vom Typ T sind.
- Wie bei anderen automatischen Typ-Anpassungen kann ein Operand vom Typ $Lvalue(T)$ bei Bedarf in einen Wert vom Typ T umgewandelt werden.

  Dies geschieht einfach durch Zugriff auf den Wert, der in der Variablen gespeichert ist. Natürlich ist es dazu wichtig, dass irgendwann vorher tatsächlich ein Wert in die Variable gespeichert wurde (mit einer Zuweisung): Die Variable muß initialisiert sein.

Zuweisungen (3)

- Der Zuweisungsoperator hat dann die Signatur

$$Lvalue(T) \times S \rightarrow T$$

  wobei einer der folgenden Fälle gilt:
  - S und T sind beides arithmetische Typen (möglicherweise unterschiedlich).

    Arithmetische typen sind: `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `bool`, und Aufzählungs-Typen (s.u.).
  - T ist ein Zeiger-Typ, und $S = T$ oder $T =$ `void*`,
  - T ist ein Struktur/Union/Objekt-Typ, und $S = T$.

Zuweisungen (4)

- Bei der Zuweisung werden also zwischen beliebigen arithmetischen (Zahl-) Datentypen implizite Umwandlungen vorgenommen.
- Wenn man einen Gleitkomma-Wert einer ganzen Zahl zuweist, wird der Teil hinter dem Komma abgeschnitten. Z.B. wird `1.9` in `1` umgerechnet.
- Umgekehrt bleibt der Wert meistens erhalten, bei großen Werten muß aber gerundet werden.

  Wenn man z.B. `123456789` in eine `float`-Variable speichert, und dann zurück in ein `long int`, so erhält man `123456792`.

Zuweisungen (5)

- Bei Umwandlungen zwischen ganzzahligen Typen bleibt das Bitmuster meist erhalten, ggf. werden
  - vorne Bits abgeschnitten
    (falls der neue Typ kleiner ist)
  - vorne 0-Bits aufgefüllt
    (falls der neue Typ `unsigned` ist oder der zugewiesene Wert nicht negativ ist)
  - vorne 1-Bits aufgefüllt
    (falls der neue Typ `signed` ist und der Wert negativ ist: auf diese Art bleibt der Wert erhalten).

Zuweisungen (6)

- Achtung:
  - Negative Werte, die `unsigned`-Größen zugewiesen werden, werden zerstört.
  - Große Werte, die in kleine Variablen gespeichert werden, werden zerstört.

  Man sollte solche Zuweisungen vermeiden, oder wenigstens entsprechende Tests vorsehen.

  Siehe Abschnitt über Zusicherungen (Assertions).
- Bei Zuweisung an `bool` zählt alles von `0` verschiedene als `true`. Umgekehrt wird `true` in `1` umgerechnet und `false` in `0`.

Zuweisungen (7)

- Weil eine Zuweisung selbst ein Ausdruck ist, kann man sie in größere Ausdrücke einbauen.
- Z.B. sieht man in C-Programmen häufig

```
while((c = getc(stdin)) != EOF) { ... }
```

  `getc(stdin)` liefert das nächste Eingabezeichen (oder `EOF`: end-of-file). In C++ würde man eher schreiben `while(cin.get(c)) { ... }`
- Die Klammern um die Zuweisung sind nötig, da der Zuweisungs-Operator eine sehr niedrige Priorität hat (noch niedriger als das logische Oder).

Zuweisungen (8)

- Der Wert einer Zuweisung ist der Wert der Variablen nach der Zuweisung (d.h. der zugewiesene Wert, aber nach eventuellen Typanpassungen).
- Daher kann man z.B. mit

```
i = j = 0;
```

beide Variablen auf 0 setzen.

Es ist eine Stilfrage, ob man solche Mehrfachzuweisungen tatsächlich nutzt. Eventuell sind einzelne Zuweisungen klarer:

```
i = 0;
j = 0;
```


Zuweisungen (9)

- Ein häufiger Fehler ist, = statt == zu schreiben, z.B.

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

- Das Problem ist, dass dies in C/C++ legal ist:
  - ◇ Der Wert von `j` wird der Variablen `i` zugewiesen.
  - ◇ Falls dieser zugewiesene Wert verschieden von 0 ist, wird der von `if` gesteuerte Anweisungsblock ausgeführt.
- Dennoch werden viele Compiler in diesem Fall eine Warnung ausgeben.

Notfalls schreibe man: `if((i = j) != 0) { ... }`.

Seiteneffekte

- Normalerweise ist der offizielle Hauptzweck eines Wertausdrucks die Berechnung eines Wertes.
- Bewirkt er darüber hinaus eine Zustandsänderung (durch eine Zuweisung oder Ein-/Ausgabe), so sagt man, er habe einen Seiteneffekt.

Von einer expliziten Zuweisung, die nicht Teil eines größeren Ausdrucks ist, würden die meisten Leute wohl nicht sagen, dass sie einen Seiteneffekt hat. Hier ist die Zustandsänderung ja der Hauptzweck. Die Sprechweise stammt aus Sprachen, bei denen Zuweisungen nicht selbst ein Wertausdruck sind.

Abkürzung: Zuweisung

- Zuweisungen der Form

$$\mathtt{X} = \mathtt{X}\ \theta\ \mathtt{Y}$$

mit $\theta \in \{$+, -, *, /, %, <<, >>, &, |, ^$\}$ kommen häufig vor.

- In C/C++ gibt es dafür die Abkürzung

$$\mathtt{X}\ \theta\mathtt{=}\ \mathtt{Y}$$

- Z.B. kann man `X += 2` statt

```
X = X + 2
```

schreiben.

Eine Feinheit ist, dass `X` bei der Abkürzung nur einmal ausgewertet wird. Das ist wichtig, falls `X` selbst Zuweisungen/Seiteneffekte enthält.

Inkrement/Dekrement (1)

- Die Erhöhung einer Variablen um 1 ist besonders häufig, daher kann `i += 1` noch weiter abgekürzt werden zu `i++` oder `++i`.
- Der Unterschied ist, dass
  - ◇ `i++` den alten Wert der Variablen `i` liefert und dann 1 aufaddiert ("postincrement").

    Daher entspricht dies nicht genau `i += 1`. Das würde den neuen Wert liefern (wie `++i`).
  - ◇ `++i` erst 1 aufaddiert, und dann den neuen Wert liefert ("preincrement").

    Weiterer Unterschied: `++i` ist ein Lvalue, aber ist `i++` kein Lvalue.

Inkrement/Dekrement (2)

- Hat z.B. `i` vorher den Wert `3`, so würde
  - ◇ `i++` den Wert `3` liefern, aber
  - ◇ `++i` den Wert `4`.

  In beiden Fällen hätte die Variable hinterher den Wert `4`.
- Das Erhöhen einer Variable um 1 nennt man auch das Inkrementieren der Variablen.
- Entsprechend kann man mit `i--` und `--i` den Wert von `i` um `1` verringern (dekrementieren).

Auswertungsreihenfolge (1)

- Der Compiler darf bei fast allen Operatoren die Auswertungsreihenfolge wählen, d.h. ob linker oder rechter Operand zuerst.
- Es ist schlecht, wenn bei einem Ausdruck die Auswertungsreihenfolge eine Rolle spielt:

  ```
  i + (i++)
  ```
- Angenommen, `i` hat vorher den Wert `3`.
- Je nach Compiler kann als Wert des Ausdrucks `6` oder `7` herauskommen (Ergebnis ist nicht definiert).

  `7` ergibt sich, wenn die rechte Hälfte zuerst ausgewertet wird.

Auswertungsreihenfolge (2)

- Klammern bestimmen nur die Baumstruktur des Ausdrucks, nicht die Auswertungsreihenfolge.

  Bei `i + (i++)` ist nicht garantiert, das `i++` zuerst ausgewertet wird. Die Klammern sind in diesem Fall überflüssig, weil `++` sowieso höhere Priorität als `+` hat. Wenn man sie weglässt, ändert das nichts an der Bedeutung des Ausdrucks.
- Es ist nicht einmal sicher, dass

  ```
  (i++) * (i++)
  ```

  den Wert `12` hat, wenn `i` vorher den Wert `3` hatte.
- Der Wert ist undefiniert, und bei manchen Compilern kommt `9` heraus.

Auswertungsreihenfolge (3)

- Allgemein definieren C/C++ sogenannte "Sequence Points": An dieser Stelle müssen alle vorherigen Zuweisungen ausgeführt sein.
- Das Ende eines vollständigen Ausdrucks ist ein "Sequence Point", also z.B. beim ";".

  `&&`, `||`, `?`, `,` (Sequenz-Operator) sind auch ein "Sequence Points".
- Ein Funktionsaufruf findet erst nach vollständiger Auswertung der Parameter statt.

  Die Reihenfolge der Parameterauswertung ist wieder beliebig (das Komma hier ist nicht der Sequenz-Operator).

Auswertungsreihenfolge (4)

- In C/C++ ist das Ergebnis eines Ausdrucks undefiniert, wenn es zwischen zwei "Sequence Points"
  - ◇ zwei Zuweisungen an die gleiche Variable gibt, oder
  - ◇ einen lesenden und einen schreibenden Zugriff auf die gleiche Variable gibt, wobei der lesende Zugriff zur Bestimmung des geschriebenen Wertes nicht nötig ist (also nicht sicher vorher erfolgt).

    Z.B. ist `i = i+1;` natürlich zulässig, obwohl hier in einem Ausdruck ein lesender und ein schreibender Zugriff auf `i` erfolgen.

Inhalt

Bedingter Ausdruck (1)

- Der bedingte Ausdruck hat die Form `A ? B : C`.
- Falls `A` von `0` verschieden ist, wird `B` geliefert, sonst `C`.

  Natürlich ist in diesem Fall die Auswertungsreihenfolge definiert: Es wird zuerst `A` ausgewertet, und dann, je nach Ergebnis, `B` oder `C`. Der andere Teil wird nicht ausgewertet.
- Z.B. kann man das Minimum von `i` und `j` auf folgende Weise berechnen:

```
(i < j)? i : j
```

Bedingter Ausdruck (2)

- In den meisten anderen Sprachen würde man ein `if`-Statement benutzen.
- Das kann man in C/C++ natürlich auch machen, aber mit dem bedingten Ausdruck sind manchmal kompaktere Formulierungen möglich.

  Vermutlich spielte für die Einführung von bedingten Ausdrücken auch die Nutzung des C-Präprozessors eine Rolle: Mit Macros kann man sehr effiziente Implementierungen von kleineren Funktionen programmieren. Heute würde man dafür aber Inline-Funktionen benutzen.
- Der bedingte Ausdruck ist rechtsassoziativ: `A?B:C?D:E` bedeutet `A?B:(C?D:E)`.

Bedingter Ausdruck (3)

- Falls `B` und `C` beide vom Typ $Lvalue(T)$ sind, hat auch der bedingte Ausdruck `A?B:C` diesen Typ:

  $$\_?\_:\_ : \texttt{bool} \times Lvalue(T) \times Lvalue(T) \to Lvalue(T)$$

  D.h. man kann einen bedingten Ausdruck auch auf der linken Seite einer Zuweisung verwenden, um die Variable auszuwählen.
- Sind S_1 und S_2 arithmetische Typen, so findet die übliche Umwandlung in gemeinsamen Typ S statt:

  $$\_?\_:\_ : \texttt{bool} \times S_1 \times S_2 \to S$$
- Sonst sollten die Typen der Alternativen gleich sein:

  $$\_?\_:\_ : \texttt{bool} \times T \times T \to T$$

Sequenz

- `A, B` ist ein Ausdruck, bei dem zuerst `A` ausgewertet wird, und dann `B`.
- Der Wert von `A, B` ist immer der Wert von `B`.
- `A` wird also nur wegen seines Seiteneffektes ausgewertet.
- In den meisten anderen Sprachen würde man Statements `A; B;` hintereinanderhängen.

  Das ist in C/C++ natürlich auch möglich. Aber wieder erlaubt der bedingte Ausdruck eventuell kompaktere Formulierungen und vergrößert die Möglichkeiten von Makros.

Sizeof

- `sizeof(`T`)` liefert den für Variablen des Typs T nötigen Speicherplatz (in `char`-Einheiten, d.h. Byte).
- `sizeof` E liefert entsprechend die Speichergröße des Ergebnistyps des Wertausdrucks E.

  Der `sizeof`-Operator hat die gleiche (hohe) Priorität wie alle Präfixoperatoren. Wenn der Wertausdruck etwas Komplizierteres ist als eine Variable, werden also meistens Klammern nötig sein. Der Ausdruck wird übrigens nicht ausgewertet, der Compiler bestimmt nur seinen Typ.
- Der Ergebnistyp von `sizeof`-Ausdrücken ist `size_t`, definiert in `<cstddef>`.

  Normalerweise ist `size_t` definiert als `unsigned int` oder `unsigned long`.

Aufgabe

```
...
int main()
{
        int n;
        bool b;

        n = (5 << 2) + 4 * 5 % 2;
        b = n;
        b = b && n < 50;
        n += b ? 100 : 200 + 5;
        cout << ++n; // Was wird hier gedruckt?
        return 0;
}
```