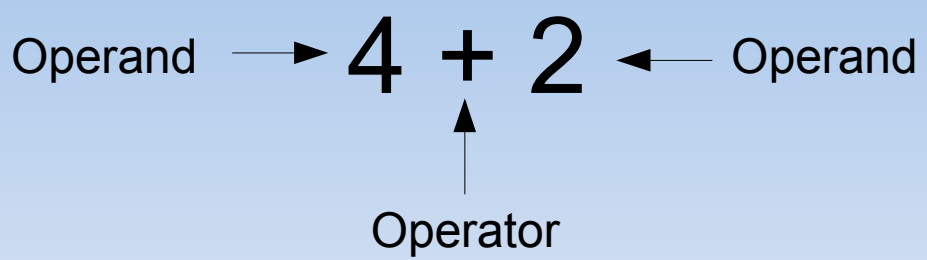
Operatoren in C

Gastvorlesung - 11.01.10

Andreas Textor
andreas.textor@hs-rm.de

Terminologie



- Unterscheidung nach Anzahl Operanden
  - Unäre, Binäre und Ternäre Operatoren
  - Z.B.: `!x; 4 + 2; x ? 7 : 10`
- Unterscheidung nach Position
  - Vor (Präfix), zwischen (Infix) oder hinter (Postfix) den Operanden
  - Z.B.: `-1; 4 + 2; x++`

Terminologie

- Notwendig, weil einige Zeichen mehrfach als Operator belegt
  - Unäres Minus: -1; Binäres Minus: 1-2
  - Unäre Pointerdereferenzierung: *x;
    Binäre Multiplikation: x*y
  - etc.

Arithmetische Operatoren

- +,-,*,/,%,==: Bereits bekannt
- Präzendenz nach ”Punkt vor Strich”
- Achtung bei Verwendung von float/double:
  - int <op> int → int; double <op> double → double
    double <op> int → double
    Z.B: double a = 1.0; int b = 2; int c = a * b;
  - Keine Gleichheit bei Fließkommazahlen verwenden

Falsch

```
for(double a = 1.0; a != 2.0; a = a + 0.1) {
…
}
```

Richtig

```
for(double a = 1.0; a < 2.0; a = a + 0.1) {
…
}
```

Zuweisungsoperatoren

- Verkürzende Schreibweisen
  - `a += b` gleichbedeutend mit `a = a + b`
  - Analog: -=, *=, /=, %=
- Zuweisungsoperator ”=” - mit Rückgabewert!
  - Rückgabewert für x = y ist y
  - Häufig zu lesen:
    `while((x = f(...) != 5) { … }`
  - Achtung mit unterschiedlichen Typen:

```
char ch; int i; i = 321; ch = i; // ch==65
int i; double pi = 3.14159; i = pi; // i==3
```

  - Achtung mit = und == !!!

Inkrement/Dekrement Operatoren

- `a = a + 1` kann verkürzt werden zu `a++` oder `++a`
- Unterschied: `a++` liefert alten Wert von a, `++a` liefert neuen Wert
- Beispiel:

```
int x = 3;
printf("%d\n", x++); // 3
printf("%d\n", x); // 4
printf("%d\n", ++x); // 5
```

- Analog: `a--; --a`

Vergleich und Logik

- Vergleichsoperatoren
  - `a < b; a <= b;`
    `a > b; a >= b;`
    `a != b; a == b`
- Logische Operatoren
  - `!a; a && b; a || b`
  - a==false bei a&&b: b wird nicht ausgewertet
  - a==true bei a||b: b wird nicht ausgewertet

Bitoperatoren

- Bekannt aus Digitaltechnik, Einf. Inf.:
  & (AND), | (OR), ~ (NOT), ^ (XOR)
- Bitoperatoren arbeiten Bitweise auf der Binärdarstellung einer Zahl
- Beispiel: 11 & 5 11 | 5 11 ^ 5 ~11

1011	1011	1011	1011
0101	0101	0101	0100
0001	1111	1110	
1	15	14	4

Bitoperatoren

- ”Shift”-Operatoren schieben die Bits der Zahl, aufgefüllt wird mit Nullen
- Left shift: 5 << 1 Right shift: 16 >> 2

```
  101          10000
 1010          00100

   10             4
```

- Vorsicht bei **`signed`** Typen

Bitoperatoren

- Left shift effektiv: *2, Right shift /2
  - Kann sehr viel schneller als numerisch sein
  - Nur bei Ganzzahlen!
  - Moderne Compiler optimieren ausreichend
- Abkürzende Zuweisung auch für die Bit- und Shift-Operationen:

```
a <<= b; a >>= b;
a &= b; a |= b; a ^= b
```

Bitoperatoren

- Diverse Tricks möglich:
  - `x & 1` - Ist x ungerade?
  - `x |= (1<<n)` - n.tes Bit in x auf 1 setzen
  - `!(x & (1<<n) == 0)` - Ist n.tes Bit in x==1?
  - Vertauschen von a und b ohne Temp-Variable:

```
a ^= b;
b ^= a;
a ^= b;
```

  - Berechnung der Inversen Quadratwurzel einer float-Zahl durch direkte Bitmanipulation: http://en.wikipedia.org/wiki/Fast_inverse_square_root

Pointeroperatoren

- Schon einmal gesehen: `a[b]; *a; &a; a->b; a.b`
  - Wenn a ein Pointer ist, gibt *a den Wert an auf den a zeigt
  - `a[b]` ist effektiv `*(a + b)`
  - `a->b` ist effektiv `(*a).b`
  - Wenn `c` eine normale Variable ist, ist `&c` seine Adresse (ein Pointer darauf)

Sonstige Operatoren

- () - Funktionsaufruf ebenfalls ein Operator
- () - Cast-Operator zur Typumwandlung
- Ternärer Bedingungsoperator: **`x ? a : b`**
  - x ist ein Boole'scher Ausdruck (Wahrheitswert)
  - a, b sind Ausdrücke vom gleichen Typ
- Kommaoperator: Rückgabewert von a, b ist b
  - ```
    int x;
    x = 1, 2, 3, 4;
    if (x > 5, x < 10) {
    …
    }
    ```
    ```

Operatorpräzedenz

Präzedenz	Operator
1	++ -- (Postfix) () (Funktionsaufuf) . ->
2	++ -- (Präfix) + - ! ~ * & sizeof (Alle Unär)
3	* / %
4	+ - (Binär)
5	<< >>
6	< <=
7	> >=
8	== !=
9	&
10	^
11	|
12	&&
13	||
14	?:
15	= += -= *= /= %= <<= >>= &= ^= |=
16	,

Operatorpräzedenz

- Im Zweifelsfall immer Klammern
  - `a & b == 7` wird geparst als `a & (b == 7)`
  - `a + b == 7` wird geparst als `(a + b) == 7`
- Nicht zu viel auf einmal machen:
  - OK:
    ```
    for(int i = 1; i < 10; i++)
    ```
  - Geht das lesbarer?
    ```
    while(*++i[j])
    return x^42?y(4,x[*i]<<3)&u->t_i>=n_3:x;
    ```
  - International Obfuscated C Code Contest
    http://ioccc.org

sizeof

- sizeof gibt Auskunft über die Größe eines Datentyps (in Bytes)
- sizeof funktioniert mit Typen, Werten und Variablen

```
sizeof(char)        // 1
sizeof(int)         // 4
sizeof(int*)        // 4 oder 8
sizeof(42)          // 4
sizeof(”Hallo”)     // 6
sizeof((char)'A')   // 1
sizeof('A')         // 4
int x; sizeof x     // 4
```

sizeof

- sizeof gibt auch die Größe von Arrays zurück
  - Achtung: Größe in Bytes, nicht Anzahl Elemente!
  - Achtung: Arrays als Funktionsparameter werden zu Pointer!

```
char a[] = "Hallo";
printf("%d\n", sizeof(a));    // 6

char* b = "Hallo";
printf("%d\n, sizeof(b));     // 4

int c[10];
printf("%d\n", sizeof(c));    // 40
```

sizeof

```
void printSizePointer(char* s) {
   printf("%d\n", sizeof(s));
}

void printSizeArray(char s[]) {
   printf("%d\n", sizeof(s));
}

int main() {
   char a[10];
   printf("%d\n", sizeof(s)); // 10
   printSizePointer(a);       // 4
   printSizeArray(a);         // 4

   return 0;
}
```

sizeof

```
void printSizeArray(char s[], int numElements) {
   printf(”%d\n”, numElements);
}

int main() {
   const int elements = 10;
   char a[elements];
   printSizeArray(a, elements);    // 10

   // Oder:
   //char* b = malloc(sizeof(char) * elements);

   return 0;
}
```

sizeof

- sizeof nicht verwenden um Array-Größe zu bestimmen
- Stattdessen:
  - Bei statischen Arrays: Konstanten für Anzahl der Elemente definieren
  - Bei dynamischem Speicher Größe sowieso bekannt
  - Bei Funktionen separaten Größe-Parameter übergeben
  - Bei Strings strlen() verwenden, nicht sizeof!!!