

Grundkonzepte imperativer Sprachen

Die folgenden Folien gehen auf die Vorlesungsfolien von Prof. Dr. R. Manthey im WS 2007/2008 zurück,
wurden jedoch von PASCAL nach C übersetzt.

Grundkonzepte imperativer Sprachen

● Beispiel

- ◆ Zerlegen eines C-Programms in die Grundlegenden Bausteine der Sprache
- ◆ Ausdrücke
 - ◊ Literale
 - ◊ Variablen
 - ◊ Operatoren
 - ◊ Zuweisungen
 - ◊ Noch mehr Zuweisungen und Operatoren
- ◆ Anweisungen
 - ◊ Ausdruck-Anweisungen
 - ◊ Deklarationen
 - ◊ Kontrollfluss

1. Beispiel eines imperativen Programms in C

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert)
    }
    return 0;
}
```



Schlüsselworte

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Die hier hervorgehobenen Zeichenreihen sind **Schlüsselworte** von C – syntaktische Einheiten mit fest vorgegebener Bedeutung, die das Programm strukturieren und in jedem Programm dieselbe Bedeutung haben.

Namen

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Andere Programmteile sind vom Programmierer jeweils frei wählbar und dienen dazu, Programmteile zu benennen, z.B. die Variablen **Namen**.

Standardnamen

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Andere Namen
gehören fest zu C,
sind aber keine
Schlüsselworte:
Standardnamen

Operatoren

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Operatoren stellen
was mit den Zahlen
bzw den Werten der
Variablen an

Zahlenwerte

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Zahlenwerte
selbsterklärend,
oder nicht?
- Literale!

Bibliotheken

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Um weitere, nicht Standard-Funktionen von C zu nutzen, können **Bibliotheken eingebunden** werden.

Kopf der Funktion

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Hier werden
Funktionsname,
Rückgabetyp und
das Interface der
Funktion definiert

Sonderzeichen

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Sonderzeichen
wie:
(), {}, ;,
strukturieren den
Code.

Blockstruktur

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Schlüsselworte mit darauf-folgender { starten einen Block, der mit } beendet wird.

(Ausnahme: if(){}
else{})

Blöcke können ineinander verschachtelt werden.

Einrücken zur Lesbarkeit des Codes.

Eingabe / Ausgabe bei C

```
#include <stdio.h>
int main() {
    int wert, zaehler, m;
    printf("bitte eine positive Zahl eingeben \n");
    scanf("%i", &m);
    if (m <= 0) {
        printf("%i ist nicht positiv \n", m);
    } else {
        zaehler = 1;
        wert = 1;
        while ( zaehler < m) {
            wert = wert * zaehler;
            zaehler++;
        }
        printf("%i ungleich %i \n", m, wert);
    }
    return 0;
}
```

Eingabe und Ausgabe mit Zeilenumbruch \n und Ausgabe der Werte von Variablen %i,%f

Ausdrücke

Anweisungen versus Ausdrücke in C

● Anweisungen

- ◆ steuern den Kontrollfluss des Programms

○ Beispiel: `if (happy) danceWith(woolf); else kill(woolf);`

● Ausdrücke

- ◆ berechnen Werte

○ Beispiel: „`3+1`“ wird ausgewertet zu „`4`“

● Mischformen

- ◆ Bedingter Ausdruck: `Bedingung ? Expr1 : Expr2`
 - Wenn `Bedingung` wahr ist, liefere als Ergebnis den Wert von `Expr1`, sonst den Wert von `Expr2`
- ◆ Sequentielle boolesche Operatoren: `Expr1 && Expr2,`
...
 - Werte nur `Expr1` aus wenn das reicht um das Endergebnis zu bestimmen

Syntax von Ausdrücken

Ein Ausdruck ist eine Kombination aus Variablen, Konstanten, Operatoren und Rückgabewerten von Funktionen. Die Auswertung eines Ausdrucks ergibt einen Wert.

- Ausdrücke setzen sich zusammen aus:
 - ◆ Literalen
 - ◆ Variablenbezeichnern
 - ◆ Funktionsaufrufen
 - ◆ Operatorsymbolen
 - ◆ Klammern

Ausdrücke: Literale

Konstanten für Zahlen, Zeichen, Zeichenketten, Wahrheitswerte und Objekte

Literale

- Literale sind eine textuelle Darstellung von Werten.
- Es gibt in C Literale für
 - ◆ elementare Datentypen
 - ◆ Zeichenketten (‘Strings’)
- Array-Werte haben ebenfalls eine textuelle Darstellung

Literale für ganze Zahlen

- **int**-Literale sind Ziffernfolgen (evtl. mit Vorzeichen)
 - ◆ Ziffernfolgen, die **nicht** mit **0** oder **0x** beginnen werden als **Dezimalzahlen** interpretiert
 - ◊ Beispiele: **2 -34 53**
 - ◆ Ziffernfolgen, die mit **0** beginnen werden als **Oktalzahlen** interpretiert
 - ◊ Beispiel: **027**
 - ◆ Ziffernfolgen, die mit **0x** beginnen sind **Hexadezimalzahlen**
 - ◊ Dürfen auch die Zeichen **a, b, c, d, e, f** für 10, 11, 12, 13, 14, 15 enthalten
 - ◊ Beispiel: **0x17 0x1b 0xffff**
 - ◆ **Achtung: $17 \neq 017 \neq 0x17 !!!$**
- **long**-Literale sind int-Literale mit nachgestelltem **l** oder **L**
 - ◆ Beispiel: **23l -3L +53L 027L**
- **64Bit** int haben den suffix: **i64**
 - ◆ Beispiel: **32i64**

Literale für reelle Zahlen

- **double**-Literale enthalten (**double** = 64Bit)
 - ◆ einen Dezimalpunkt
 - ◆ oder ein **e** bzw. **E** für Zehnerexponenten
 - ◆ oder ein nachgestelltes **d** (oder **D**)
 - ◆ Beispiele: **2.** **-3.4** **53e4** **0d** **2d** **-3.4d** **53e4d**
- **float**-Literale (32Bit) sind **double**-Literale, die statt des **d** bzw. **D** ein nachgestelltes **f** (oder **F**) haben
 - ◆ **2f** **-3.4f** **53e4f**

Literale für alphabetische Zeichen (Character)

In C stellt man einzelne Zeichen dar

- entweder als Zeichen in Einfach-Hochkommata
 - ◆ Falls die Tastatur es erlaubt und es kein Sonderzeichen ist
 - ◆ Beispiel: `char c = 'A';`
 - ◆ Escape Sequences

Escape-Sequenz	Unicode	Zeichen
\b	\u0008	backspace, BS
\t	\u0009	Tabulator, horizontal tab, HT
\n	\u000a	Zeilenvorschub, linefeed, newline, LF
\f	\u000c	Seitenvorschub, form feed, FF
\r	\u000d	Wagenrücklauf, carriage return, CR
\"	\u0022	Anführungszeichen, double quote, "
\'	\u0027	Hochkomma, single quote, '
\\	\u005c	backslash, \

Literale für Zeichenketten, Wahrheitswerte, Objekte

- String-Literale sind in doppelte Anführungszeichen eingeschlossene Zeichenketten.
 - ◆ "Welcome to the real world!"
- boolean-Literale sind in C nicht definiert.
 - ◆ Es werden integer Werte 0 (false) und 1 (true) verwendet.

Ausdrücke: Variablenbezeichner

Variablen

- Variablen dienen der Speicherung von Werten
- Die Verwendung eines Variablennamens ist (neben Literalen) die einfachste Form eines Ausdrucks.

Variabtentyp	Deklaration	Ausgabebefehl
Ganzzahl	<code>int v;</code>	<code>printf("%i\n", v);</code>
Kleine Ganzzahl	<code>signed short int v;</code>	<code>printf("%hi\n", v);</code>
Große Ganzzahl	<code>signed long int v;</code>	<code>printf("%li\n", v);</code>
Ganzzahl ≥ 0	<code>unsigned int v;</code>	<code>printf("%u\n", v);</code>
Kleine Ganzzahl ≥ 0	<code>unsigned short int v;</code>	<code>printf("%hu\n", v);</code>
Große Ganzzahl ≥ 0	<code>unsigned long int v;</code>	<code>printf("%lu\n", v);</code>
Byte (8 Bit)	<code>char c;</code>	<code>printf("%c\n", c);</code>
Kleine Fließkommazahl	<code>float f;</code>	<code>printf("%f\n", f);</code>
Große Fließkommazahl	<code>double d;</code>	<code>printf("%f\n", d);</code>

Variablen

- Jede Variable hat:

- ◆ einen **Namen**:
 - ◊ Besteht aus Buchstaben, Ziffern und Unterstrichen.
 - ◊ Darf nicht mit einer Ziffer beginnen und darf kein ANSI-C Schlüsselwort sein.
- ◆ einen **Typ**:
 - ◊ Legt fest, welche Art von Werten die Variable aufnehmen kann (z.B. nur ganze Zahlen).
 - ◊ Legt fest, welche Operationen erlaubt sind (z.B. Addition usw.).
- ◆ einen **Wert**:
 - ◊ Steht in binärer Zahlendarstellung im Hauptspeicher.
- ◆ eine **Adresse**:
 - ◊ Die Anfangsadresse des Werts im Hauptspeicher.
- ◆ einen **Platzbedarf**:
 - ◊ Anzahl Bytes, die der Wert im Hauptspeicher belegt. Hängt vom Typ ab.

Deklaration von Variablen

Vorhergename: Deklarationen kommen eigentlich erst später dran! Variablen sind aber einfach zu elementar um sie so lange zu verschweigen... ☺

● Variablendeklarationen haben *immer* die Form

[Modifikatoren] Typname Variablename = Initialisierung ;

Der graue Teil ist optional. Das Semikolon nicht!

- ◆ Obiges gilt auch für nicht-elementare Typen (d.h. für die später eingeführten Klassen und Interfaces)

● Beispiele

```
int counter;                                // Deklaration eines Zählers
int counter = 0;                             // Deklaration mit Initialisierung
int start, end;                            // Zwei integer-Variablen
double[] prices = { 100, 200, 300, -99.99 }; // Initialisierte Deklaration eines Double-Arrays
const double π = 3.14159265;                // Konstantendeklaration („const“)
```

- ◆ Wir werden später noch weitere **Modifikatoren** kennen lernen.

Ausdrücke: Operatoren

Arithmetische Operatoren

Zuweisung und Zuweisungsoperatoren

Boolesche Operatoren

Bitweise Operatoren

Ausdrücke: Operatoren

- Operatoren sind in die Programmiersprache eingebaute Funktionen, die dadurch gegebenenfalls komfortabler geschrieben werden können
- In C verwendete Notationen
 - ◆ Präfix: vorangestellt, z.B. Preinkrement `++i`
 - ◆ Postfix: nachgestellt, z.B. Postinkrement `i++`
 - ◆ Infix: in die Mitte gestellt, z.B. `1+2`
 - ◆ Roundfix: drumherumgestellt, z.B. ein Klammerpaar als Operator aufgefaßt: `{1, 3, 4}` als Operator der ein Array konstruiert.

Präzedenzen von Operatoren

- Die besondere Schreibweise, die bei eingebauten Operatoren möglich ist, kann zu mehrdeutigen Ausdrücken führen
 - ◆ $1+2*3$ könnte syntaktisch (gemäß den Regeln 1 bis 3 für die Konstruktion von Ausdrücken) sowohl als Ausdruck $(1+2)*3$ als auch als $1+(2*3)$ gelesen werden
- Damit die Semantik eindeutig ist, ohne Ausdrücke mit Klammern pflastern zu müssen, gibt man jedem Operator eine bestimmte **Präzedenz**
 - ◆ Synonyme: Präzedenz, Bindungskraft, Vorrang
 - ◆ Wir sagen $*$ bindet stärker als $+$ und meinen damit, dass der Ausdruck als $1+(2*3)$ gelesen werden soll
 - δ Punkt vor Strich

Präzedenzen von Operatoren in C

Operator

(expr) [index] -> .

! ~ ++ -- (type) sizeof
Unary operator: + - * &

* / %

+ -

<< >>

< <= > >=

== !=

Binary operator: &

Binary operator: ^

Binary operator: |

&&

||

expr ? true_expr : false_expr

+= -= *= /= <<=

&= ^= |= %= >>= =

,

Associativity

Left ==> Right

Right <=> Left

Left ==> Right

Right <=> Left

Right <=> Left

Left ==> Right

Unary Operator

+

-

*

&

Binary Operator

&

^

|

Example

+23209

-value

*pointer

&variable

Example

t = 0xCC; p = 0xAA;
(t & p) == 0x88;

r = 0xF0; w = 0xCC;
(r ^ w) == 0x3C;

x = 0x99; y = 0x96;
(x | y) == 0x9F;

Assoziativität: Motivation

- Problem: Präzedenz reicht nicht!
 - ◆ In $f(1) + g(2) + h(3)$ könnten die Funktionen f , g und h Seiteneffekte auslösen (z.B. Ausgaben am Bildschirm)
 - ◆ Daher muss selbst bei gleichen Operatoren (gleiche Präzedenz, mathematisch kommutativ) die Reihenfolge der Ausführung definiert sein!
- Idee
 - ◆ Die Assoziativität regelt die Reihenfolge der Ausführung von Operatoren gleicher Präzedenz
 - ◆ Ergibt die Anwendung der Präzedenz- und Assotiativitäts-regeln nicht die gewünschte Ausführungsreihenfolge so müssen explizit Klammern gesetzt werden

Assoziativität: Definition

- **Rechts-Asssoziativität:** Auswertung beginnt rechts
 - ◆ Zuweisungsoperatoren sind in C rechts-assoziativ
 - ◆ $x=y=1$ steht somit für $x=(y=1)$
- **Links-Asssoziativität:** Auswertung beginnt links
 - ◆ Alle binären Operatoren außer den Zuweisungsoperatoren sind links-assoziativ
 - ◆ $1+2+3$ steht somit für $(1+2)+3$

Stil: Expliziter ist besser!

- Was ist das Ergebnis des Ausdrucks $a \& b == b | c$?
 - ◆ Da $==$ stärker bindet als $\&$ und $|$ entspricht Obiges
 $a \& (b == b) | c$
 - ◆ Da $\&$ stärker bindet als $|$ entspricht es ferner
 $(a \& true) | c$ und somit $a | c$
- Solche Fehler sind sehr schwer zu finden!
 - ◆ Das Programm ist syntaktisch korrekt, somit meldet der Compiler keine Fehler
 - ◆ Selbst überliest man den Fehler immer wieder
- Empfehlung: Eindeutigkeit durch Klammerung
 - ◆ Bei Verwendung von weniger gebräuchlichen Operatoren die intendierte Semantik durch Klammerung festlegen!

Ausdrücke: Zuweisung und Zuweisungsoperatoren

Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.

C. A. R. Hoare (1969)

Anweisungen versus Ausdrücke

● Anweisungen

- ◆ steuern den Kontrollfluss des Programms

◊ Beispiel: `if (...) danceWith(woolf); else kill(woolf);`

● Ausdrücke

- ◆ berechnen Werte

◊ Beispiel: „`3+1`“ wird ausgewertet zu „`4`“

- ◆ können Seiteneffekte haben!

◊ Beispiele: Funktionsaufrufe und Zuweisungen (gleich mehr dazu)

● Mischformen

- ◆ Bedingter Ausdruck: `Bedingung ? Expr1 : Expr2`

◊ Wenn Bedingung wahr ist, liefere den Wert von Expr1, sonst den Wert von Expr2

- ◆ Sequentiellen boolesche Operatoren: `Expr1 && Expr2`,

◊ Werte nur `Expr1` aus wenn es reicht um das Endergebnis zu bestimmen

Zuweisung: Pascal versus Java, C, C++

Pascal

- Der Gleichheitstests ist
=
- Der Zuweisungsoperator
ist :=
- Die Zuweisung ist eine
Anweisung
 - ◆ Sie liefert kein Ergebnis!

Java, C, C++

- Der Gleichheitstests ist
==
- Der Zuweisungsoperator ist
=
- Die Zuweisung ist ein
Ausdruck!
 - ◆ Sie liefert ihren rechten Teil
als Ergebnis!
 - ◆ Die eigentliche Zuweisung
geschieht als Seiteneffekt
des Ausdrucks!

Zuweisung: Beispiele

Pascal

- $x := y := 1$ ist illegal. Ersatz:
 - ◆ $y := 1;$
 - ◆ $x := y;$
 - ◆ Letztere Zeile könnte auch $x := 1;$ lauten.

- $x := (y := 1) + 5$ ist illegal.
Ersatz:
 - ◆ $y := 1;$
 - ◆ $x := y+5;$

Java, C, C++

- $x = y = 1$
 - ◆ bezeichnet $x = (y = 1)$
 - ◆ weist y den Wert 1 zu
 - ◆ der an x zugewiesene Wert von $y = 1$ ist 1

- $x = (y = 1) + 5$
 - ◆ hat den Wert 6
 - ◆ weist y den Wert 1 zu
 - ◆ weist x den Wert 6 zu

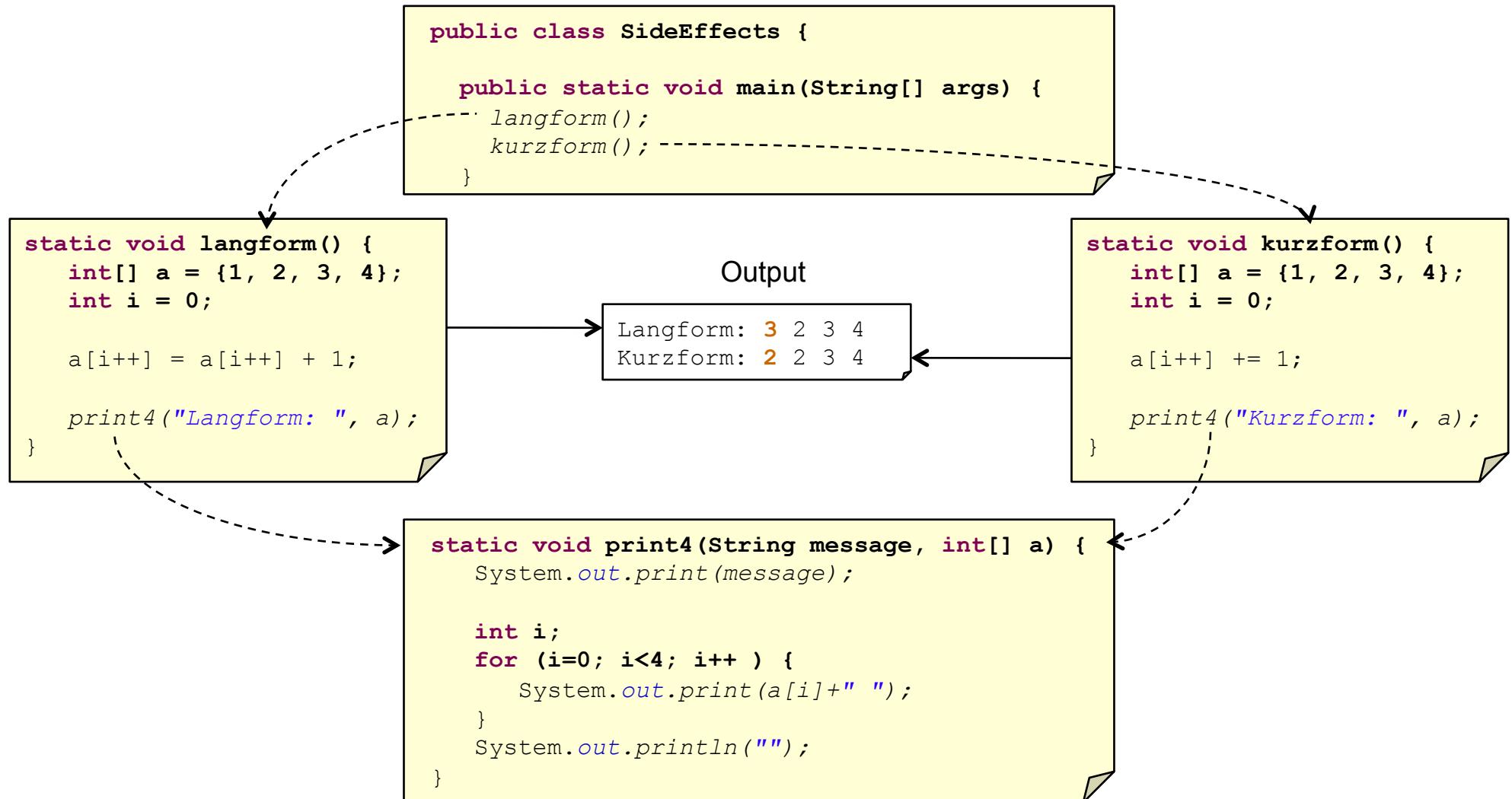
Kombinierte Zuweisungsoperatoren in C

- Zuweisungsausdrücke der Form $v \text{ += expr ;}$ sind eine Schreibabkürzung für $v = v + expr ;$
 - ◆ Solche Zuweisungen kommen häufig vor
- Für jeden binären Operator \boxed{W} gibt es einen entsprechenden kombinierten Zuweisungsoperator $\boxed{W}=$ in C.
 - ◆ Beispiele: $-= *=/=%>>=$ usw.

Kombinierte Zuweisungsoperatoren versus Langform

- Ist Wert von $v = v \text{ } \square \text{expr}$ und $v \text{ } \square = \text{expr}$ immer gleich?
 - ◆ Dabei steht \square für einen der binären Operatoren
- Falls v keine Seiteneffekte hat: „Ja!“
- Falls v Seiteneffekte hat: „Nein!“
 - ◆ Beispiel: Falls v ein Arrayzugriff mit Seiteneffekt ist:
 - ◊ $a[i++] = a[i++] + 1;$
 - ◊ $a[i++] += 1;$
 - ◆ Bei $v \text{ } \square = \text{expr}$ wird Wert von v nur ein einziges Mal ausgewertet, im Gegensatz zu $v = v \text{ } \square \text{expr}$
 - ◆ Eher pathologische Fälle
 - ◊ Sollten bei „gutem Programmierstil“ nicht vorkommen

Beispiel zu Kurzform versus Langform (Java)



Beispiel zu Kurzform versus Langform – Erklärung des Ergebnisses

5. $a[0]$ wird das Ergebnis von $2+1$ zugewiesen

```
public class SideEffects {  
  
    public static void main(String[] args) {  
        langform();  
        kurzform();  
    }  
}
```

4. $a[0]$ wird das Ergebnis von $1+1$ zugewiesen

```
static void langform() {  
    int[] a = {1, 2, 3, 4};  
    int i = 0;  
  
    a[i++] = a[i++] + 1;  
    ↑↑↑↑  
    print4("Langform: ", a);  
}  
}
```

Langform: 3 2 3 4
Kurzform: 2 2 3 4

```
static void kurzform() {  
    int[] a = {1, 2, 3, 4};  
    int i = 0;  
  
    a[i++] += 1;  
    ↑↑↑↑  
    print4("Kurzform: ", a);  
}  
}
```

4. $a[1] == 2$ wird gelesen

3. $i==1$ wird gelesen, danach i auf 2 gesetzt

1. $i==0$ wird gelesen, danach i auf 1 gesetzt

2. $a[0]$ wird als Ziel der Zuweisung bestimmt

3. zu $a[0] == 1$ wird 1 addiert

1. $i==0$ wird gelesen, danach i auf 1 gesetzt

2. $a[0]$ wird als Ziel der Zuweisung bestimmt

Zur Erklärung der Reihenfolge der Operationen, siehe die Folien über Präzedenzregeln!

Ausdrücke: Weitere Operatoren

Boolesche Operatoren

Bitmuster-Operatoren

Arithmetische Operatoren

Boolesche Operatoren

- In C wird der Typ **boolean** mit den Integer Literalen **1** und **0** zur Verfügung gestellt
- Boolesche Operatoren sind

&	logisches UND (\wedge)
	logisches ODER (\vee)
\wedge	logisches EXKLUSIVES ODER (XOR)
!	logische Negation (\neg)
$\&\&$	bedingtes logisches UND (\wedge)
$\ $	bedingtes logisches ODER (\vee)

Bedingte und symmetrische boolesche Operatoren

- Die bedingten booleschen Operatoren `&&` und `||` werten ihren rechten Operanden nur dann aus, falls das Ergebnis nicht schon aus dem Linken herleitbar ist
 - ◆ Beispiel: `((b = false) && (c = true))`
 - ◊ Die Zuweisung `c = true` wird nicht ausgeführt, da der Wert der Zuweisung `b = false` wieder `false` ist und damit der Wert des Gesamtausdrucks schon als `false` feststeht
 - ◆ Synonyme: Faule Auswertung, lazy evaluation
- Die symmetrischen binären Operatoren `&`, `|` und `^` werten hingegen immer *beide* Operanden aus
 - ◆ Beachte, dass diese Operatorsymbole auch Operationen auf Bitmustern bezeichnen können

Vergleichsoperatoren

- In C gibt es die üblichen Vergleichsoperatoren für die gängigen Datentypen

- > größer
- \geq größer oder gleich
- < kleiner
- \leq kleiner oder gleich
- \equiv gleich
- \neq ungleich

- Beispiele

- ◆ $(x \% 2 \equiv 0)$ liefert true falls x eine gerade ganze Zahl ist
- ◆ $((a \geq b) \text{ || } (a \leq b)) \text{ && } (a \neq b)$ liefert false

Vergleichsoperatoren: Warnung!

- In Java, C und C++ führt die Verwechslung des **Gleichheitsoperators** `==` mit dem **Zuweisungsoperator** `=` häufig zu schwer zu findenden Fehlern
 - ◆ Beispiel: Obwohl `a == false` evaluiert `(a = true)` zu `true`, da dies der Wert der Zuweisung ist
- Es hat sich daher eingebürgert, keine Vergleiche auf boolesche Werte anzuwenden
 - ◆ Man schreibt `(a && !b)` statt `(a == true && b == false)`

Bitmuster und bitweise logische Operatoren

- Jeder Wert eines Ganzzahltyps kann auch als Bitmuster angesehen werden
- Die **bitweisen (‘bitwise’) logischen Operatoren** wenden die jeweilige Operation simultan auf jedes Bit der Operanden an und liefern ein neues Bitmuster als Wert

$a \& b$ bitweises UND

$a | b$ bitweises ODER

$a ^ b$ bitweises EXKLUSIV-ODER

$\sim a$ bitweises KOMPLEMENT

- Beispiel

x	y	x & y	x y	x ^ y	$\sim x$	$\sim y$
1010	1100	1000	1110	0110	0101	0011

Bitmuster: Schiebeoperatoren

- Die **Schiebe-Operatoren** (‘shift operators’) verschieben das Bitmuster innerhalb eines Ganzahlwertes
 - $x << n$ schiebt die Bits in x um n Stellen nach links und füllt rechts Nullen auf (Beispiel: unsigned Datentyp):


00101010000 42 → 84 → 168
 - $x >> n$ schiebt die Bits in x um n Stellen nach rechts und füllt links mit dem Vorzeichenbit auf (**arithmetic shift**) (wenn x signed datentyp):


11101010000 -88 → -44 → -22
00001010000 +40 → +20 → +10
 - $x >> n$ schiebt die Bits in x um n Stellen nach rechts und füllt links mit Nullen auf (**logical shift**) (wenn x unsigned datentyp)


00101010000 168 → 84 → 42
- Die jeweils herausgeschobenen Bits werden einfach fallengelassen

Bitmuster: Schiebeoperatoren

- $x << n$ und $x >> n$ entsprechen einer Multiplikation mit 2^n (bzw. Division durch 2^n)
 - ◆ Der arithmetische Shift dient zur Bewahrung des Vorzeichens bei Divisionen einer negativen Zahl
 - ◊ Man beachte, dass $(-1 >> 1) == -1$, wogegen $(-1/2) == 0$
- Der Ergebnistyp einer Schiebeoperation ist der Typ des linken (zu verschiebenden) Operanden
- Der rechte Operand, der ganzzahlig sein muss, ist der **Schiebezähler** (shift count)
 - ◆ Für den Schiebezähler ist nur ein nicht-negativer Wert sinnvoll, der kleiner ist als die Anzahl der Bits im Typ des zu verschiebenden Wertes
 - ◊ Um dies sicherzustellen wird in Java der Wert des Schiebezählers automatisch entsprechend maskiert

Arithmetische Operatoren

- Für die elementaren Zahltypen sind jeweils die folgenden Operationen definiert
 - ◆ unäres + und -
 - ◆ binäres +, -, *, / und % (Rest bei Division)
- Besonderheiten
 - ◆ Auf den Ganzzahltypen ist / die Ganzzahldivision
 - ◆ Auf Floating-Point Werten ist / die Division reeller Zahlen
 - ◆ Die *modulo* Operation % existiert nicht auf den Gleitkommatypen
 - ◊ Beispiel: 7.0 % 2.5 ergibt nicht 2.0 sondern ist nicht zulässig.

Inkrement und Dekrement-Operatoren

- C kennt die unären Operatoren **++** und **--**
- Sie können in Postfix und Präfixform verwendet werden
 - ◆ **i--** (bzw. **i++**) liefert den Wert der Variablen **i**
 - ◊ als **Seiteneffekt** wird der Wert von **i** um Eins erniedrigt (bzw. erhöht)
 - ◆ **--i** (bzw. **++i**) liefert den Wert von **i-1** (bzw. **i+1**)
 - ◊ als **Seiteneffekt** wird der Wert von **i** um Eins erniedrigt (bzw. erhöht)
 - ◆ Der Seiteneffekt der **Präfixform** findet **vor** der Rückgabe des Variablenwertes statt, der der **Postfixform** **danach!**

Inkrement und Dekrement-Operatoren

- Die Operatoren wie `+=` und `++` wurden in C eingeführt, um spezielle Instruktionen von CISC Prozessoren ausnützen zu können
 - ◆ Und damit effizienteren Code erzeugen zu können
- Ob sie zu einem Geschwindigkeitsgewinn führen, hängt vom Prozessor und dem Übersetzer ab
 - ◆ Bei heutigen Übersetzern, die den erzeugten Code hoch optimieren können, wird es so gut wie nie zu einem Geschwindigkeitsvorteil kommen

Inkrement und Dekrement-Operatoren

- In jedem Fall ist der Gebrauch der Kurzformen inzwischen z. T. **idiomatisch** geworden
 - ◆ d. h. Ausdrücke wie **i++** sind **Teil gewisser Programmiermuster** (oder **-Idiome**), die ein **C, C++ oder Java** Programmierer **automatisch anwendet und versteht**
- Es hat sich z. B. **eingebürgert**, die Postfixform **i++** als **kanonische Inkrementform** zu benutzen; man schreibt also **immer i++**; statt der gleichwertigen Anweisungen **i += 1;** oder **i = i+1;**
- Wir werden weitere solche Idiome im Zusammenhang mit Schleifenkonstrukten und Reihungen kennen lernen

Ganzzahlarithmetik

- Ganzzahlarithmetik in C ist Zweierkomplement-Arithmetik modulo dem Darstellungsbereich
 - ◆ Es werden keine Überläufe erzeugt sondern nicht darstellbare Bits einfach abgeschnitten
 - ◆ Dadurch wird der Darstellungsbereich zu einem „Ring“ geschlossen
 - ◆ Zählt man über das Ende des positiven Bereichs hinaus, kommt man zur kleinsten negativen Zahl
- Siehe Vorlesung „Technische Informatik“

Ganzzahlarithmetik: Ganzzahldivision

- Ganzzahldivision wird in C mit `/` bezeichnet
- Der Rest bei Ganzzahldivision (modulo) mit `%`
- C runden immer zur Null hin, um Ganzzahlen zu erhalten
 - ◆ Die Ganzzahl-Division $5/2$ ergibt 2
 - ◆ Die Ganzzahl-Division $-5/2$ ergibt -2
 - ◆ Der Rest $5\%2$ bei Ganzzahldivision ergibt 1
 - ◆ Der Rest $(-5)\%2$ bei Ganzzahldivision ergibt -1
- Für Ganzzahlen gilt $(x/y)*y + x \% y == x$

Gleitkommaarithmetik nach IEEE 754-1985 Standard

- Arithmetik kann nach $+\infty$ und $-\infty$ überlaufen (**overflow**)
 - ◆ Der Zahlenbetrag wird zu groß für die gewählte Repräsentation
 - ◆ Es gibt Werte für **POSITIVE INFINITY** und **NEGATIVE INFINITY**
- Arithmetik kann nach $+0.0$ oder -0.0 unterlaufen (**underflow**)
 - ◆ Der Zahlenbetrag wird zu klein für die gewählte Repräsentation
 - ◆ Es gibt eine positive und eine negative Null, wobei $+0.0 == -0.0$

Gleitkommaarithmetik: NaN

- C-Arithmtik ist ‚non-stop‘
 - ◆ Rechnung liefert immer einen Wert, evtl. **NaN**
- Der Wert **NaN** („not a number“)
 - ◆ Repräsentiert Rechnungen, denen kein sinnvoller Wert zugewiesen werden kann
 - ◆ Beispiele
 - ◊ $0 * (+\infty)$
 - ◊ $0/0$
 - ◊ $+\infty + -\infty$
 - ◆ Beachte aber, dass etwa $+\infty + 5$ den Wert $+\infty$ hat
 - ◆ Ausdrücke, in denen **NaN** vorkommt, liefern immer **NaN** als Ergebnis

Anweisungen

Anweisungen in C

- Eine Anweisung (statement) weist den Computer an
 - ◆ eine Berechnung vorzunehmen (expression statement)
 - ◆ eine Variable einzurichten und zu initialisieren (declaration statement)
 - ◆ einen Schritt im Programmfluss vorzunehmen (control flow statement)
- Eine Anweisung wird durch ein Semikolon ; abgeschlossen
- Die leere Anweisung besteht nur aus dem abschließenden Semikolon ; oder aus einem Paar geschweifter Klammern { }

Ausdrucks-Anweisung (expression statement)

- Eine Ausdrucks-Anweisung
 - ◆ nimmt eine Berechnung vor
 - ◆ besteht aus einem Ausdruck, der durch ein Semikolon abgeschlossen ist
- Es sind aber nur bestimmte Ausdrücke zugelassen
 - ◆ Zuweisungsausdrücke (mit `=`, `+=` etc.),
 - ◊ `int i=5; x += 1;`
 - ◆ Präfix- oder Postfixformen von `++` oder `-`
 - ◊ `i++;`
 - ◆ Methodenaufrufe
 - ◊ `printf("hello, world"); sin(3.14d);`

Deklarations-Anweisung (declaration statement)

- Deklarations-Anweisungen sind mit ; abgeschlossene Variablen-deklarationen

[Modifikatoren] Typname Variablenname = Initialisierung ;

- Beispiele

```
int counter;                                // Deklaration eines Zählers
int counter = 0;                            // Deklaration mit Initialisierung
int start, end;                            // Zwei integer-Variablen
double[] prices = { 100, 200, 300, -99.99 }; // Initialisierte Deklaration eines Double-Arrays
const float pi = 3.14159265;                // Konstantendeklaration (const)
```

Kontrollfluss-Anweisungen: Blöcke

Blöcke

Lokale Deklarationen / Lokale Variablen

Speicherverwaltung auf dem Stapel

Kontrollfluss-Anweisungen (control flow statements)

- **Blöcke (blocks)**
 - ◆ Blöcke fassen durch Klammerung **{ }** Anweisungssequenzen zu einer einzigen Anweisung zusammen
- **Verzweigungen (branches)**
 - ◆ Verzweigungen veranlassen Übergänge zu anderen Anweisungen im Kontrollfluss
 - ◊ bedingte Übergänge: **if**, **switch**
 - ◊ unbedingte Übergänge: **break**, **continue**
- **Iterationen (loops)**
 - ◆ Iterationen organisieren strukturierte Wiederholungen im Kontrollfluss: **while**, **do while**, **for**

- Ein **Block** (**block**) ist eine Anweisung, die aus einer Folge von Anweisungen besteht, die durch **Begrenzer** zusammengefasst werden
 - ◆ In **C**, **C++** und **Java** sind die Begrenzer geschweifte Klammern { ... }
 - ◆ In der **ALGOL-Familie** (incl. **Pascal**) sind die Begrenzer ein Paar **begin** ... **end**
- Der Block { } stellt die leere Anweisung dar
- **Geschachtelte Blöcke** (**nested blocks**) sind möglich, also z. B.
{ ... { ... } ... }
 - ◆ Wir sprechen von **äußeren** und **inneren Blöcken** (**outer / inner blocks**) und von der **Schachtelungstiefe** eines Blocks (**nesting depth**)

Blöcke mit lokalen Deklarationen

- In einem Block enthaltene Deklarationen werden als **lokale Deklarationen** dieses Blocks bezeichnet
 - ◆ Jeder Block kann lokale Deklarationen (von **Variablen** oder **Typen**) enthalten
 - ◆ In einem Block deklarierte Variablen sind **lokale Variablen** des Blocks.
- In einem Block deklarierte Namen sind **nur** in diesem Block und seinen geschachtelten Unterblöcken sichtbar
 - ◆ Dies dient vor allem der **Kapselung von Information**
 - ð Deklaration und Gebrauch von Namen sollen nahe beisammen liegen, und die Deklaration eines Namens soll nicht den ganzen globalen Namensraum belasten
 - ◆ Somit können **verschiedene Blöcke** jeweils ihre **eigene Variable i oder x** haben, ohne sich gegenseitig zu stören