

Einführungs-Kurs



ANSI-C Programmierung

Autor: Prof. Dipl.-Math.
E. Engelhardt

Stand: 08. September 2024

Vorbemerkungen

- ◆ Diese Präsentation ist keine vollständige ANSI-C-Sprachbeschreibung
- ◆ Es werden nur Teile von ANSI-C in knapper Form vorgestellt, die für erste Programmierübungen benötigt werden
- ◆ Begleitend zu dieser Präsentation ist die Präsentation „ANSI-C Übungen“, die nicht an die Studenten verteilt wird

Inhalt

- 1 Warum ANSI-C?
- 2 C-Entwicklungsumgebungen (IDEs)
- 3 Phasen der Programmentwicklung
- 4 Erstes C-Programm, kürzestes Programm
- 5 Grundsätzlicher Aufbau eines C-Programms
- 6 Syntax-Beschreibungsmöglichkeiten
- 7 Wichtige elementare Datentypen
- 8 Operatoren
- 9 Ausdruck, Anweisung
- 10 Kontrollstrukturen
- 11 Funktionen
- 12 Rekursive Funktionen
- 13 Felder (Vektoren, Arrays)
- 14 Zeichenketten (String, Texte)
- 15 Präcompiler

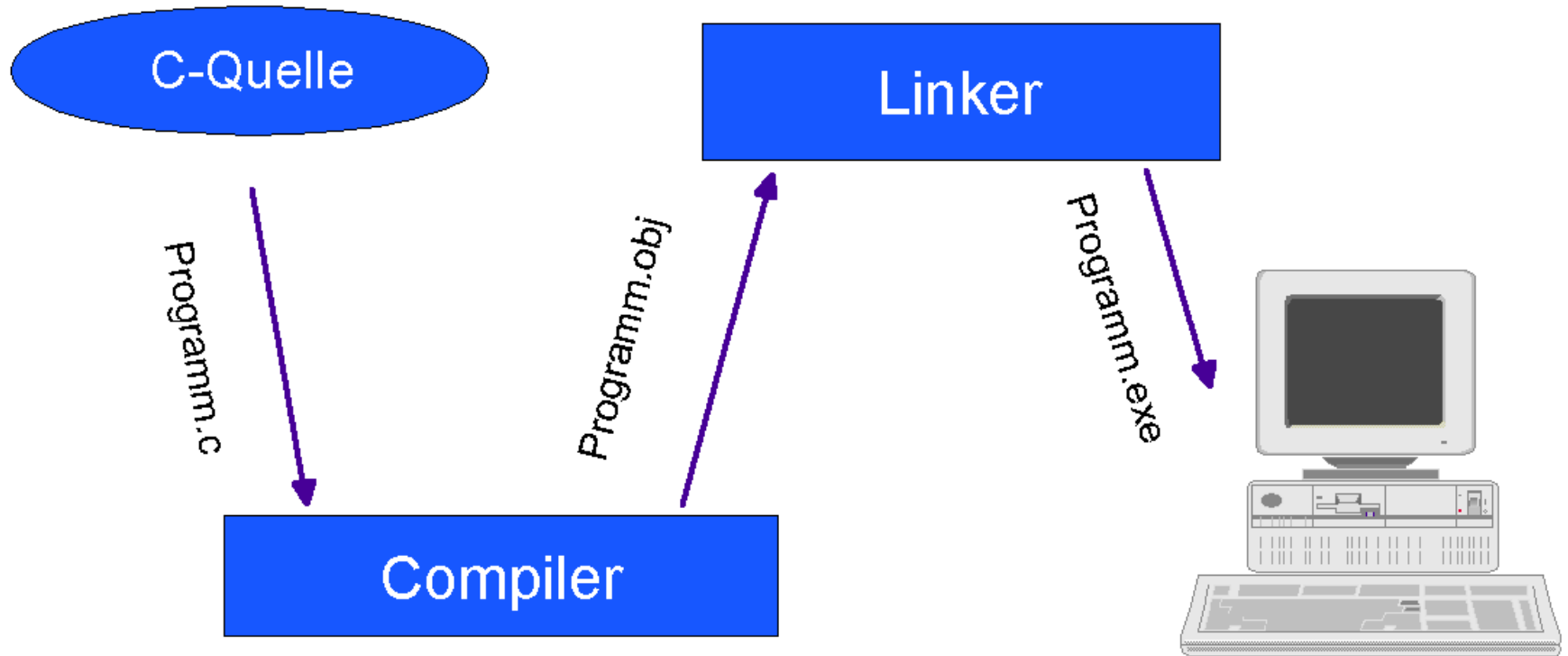
1 Warum ANSI-C?

- ◆ Schafft gute Voraussetzungen zum Erlernen anderer Programmiersprachen wie C++, C#, Java, Prolog, ...
 - Nach wie vor sehr weit verbreitete Programmiersprache
 - Gleiche Sprachelemente aus C übernommen
 - Möglichkeiten für strukturierte Programmierung, Rekursion, maschinennahe Programmierung, ...
- ◆ Standardisierte Programmiersprache
 - Auf vielen Plattformen quellcode-kompatibel und frei verfügbar
 - Viele Entwicklungsumgebungen von einfach/rudimentär bis hochkomplex
- ◆ Keine Durchmischung mit objektorientierter Programmierung
 - Dadurch können einfache Programme leicht und unkompliziert entwickelt werden (Theorem von Böhm/Jacobini)

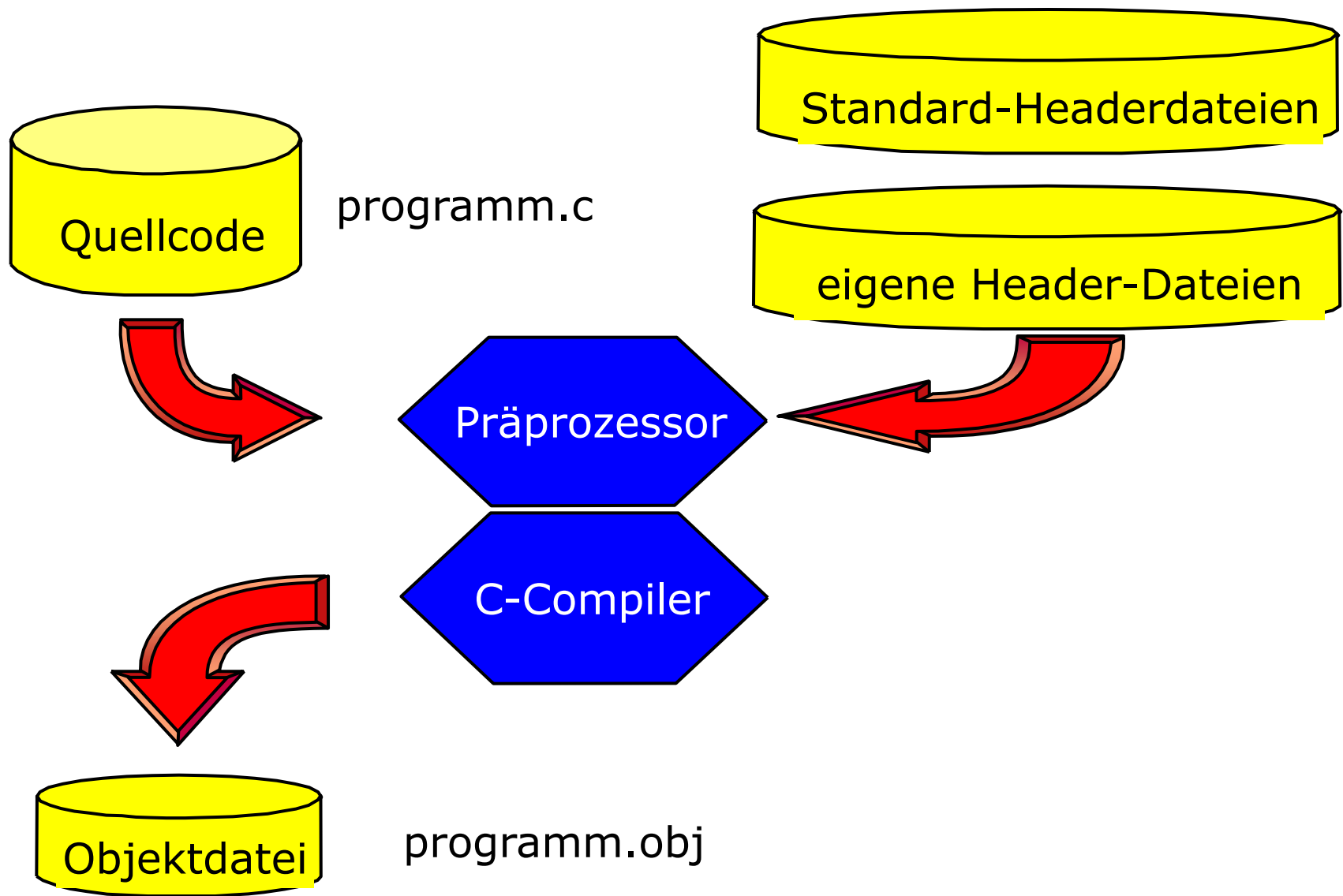
2 C-Entwicklungsumgebungen (IDEs)

- ◆ C-Programme können auch ohne ‚IDE‘ entwickelt werden, es ist nur ein C-Compiler (mit Linker) nötig
- ◆ IDEs helfen, Programme leichter zu entwickeln
- ◆ IDEs enthalten Editor, Compiler, Linker, Debugger, ...
 - Editor ist auf die Programmiersprache angepasst durch Erkennen von Schlüsselwörtern, Kommentaren, Klammerung, Variablen, ...
 - Einstellmöglichkeiten für C-Varianten, Level für Warnungen, ...
 - Möglichkeiten von vereinfachten Abläufen (Funktions-Tasten)
 - Test-Möglichkeiten

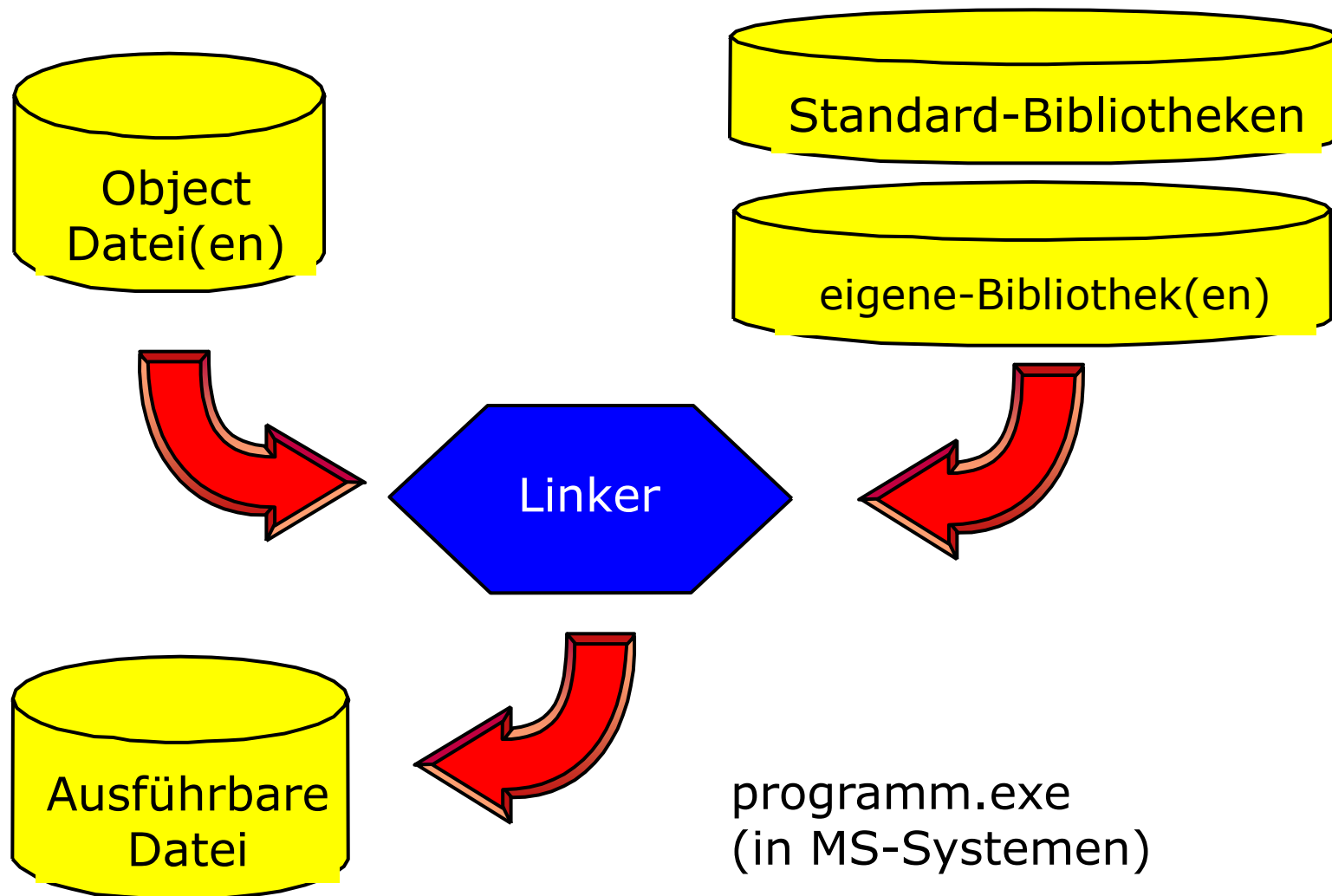
3 Phasen der Programmentwicklung



Ein C-Programm ausführen



Ein C-Programm ausführen



Hinweise für den Quellcode

- ◆ Der Dateityp ist .c, nicht .cpp
- ◆ Keine Leerzeichen, Umlaute und ß im Dateinamen und im Verzeichnis-Pfad
- ◆ Nur wenige Sonderzeichen sind erlaubt, wie _, -, u.a.
- ◆ Viel kommentieren
- ◆ Einrückungen

4 Erstes C-Programm („hallo.c“)

- ◆ Einstellen der Entwicklungsumgebung (Editor, ANSI-Quellcode und Verzeichnisse)
- ◆ Arbeitsfähigkeit herstellen

```
/* erstes C-Programm */  
  
#include <stdio.h>  
  
int main(void)  
{  
    printf("Hallo Welt!\n");  
    return 0;  
}
```

Kommentar

Einbindung von Header-Dateien

Haupt-Funktion

Ausgabefunktion

Rückgabewert

Kürzestes C-Programm

- ◆ Mit Warnungen

```
main() { }
```

- ◆ ANSI-gerecht

```
int main(void)
{
    return 0;
}
```

- ◆ Die Semantik der Kontrollstrukturen wird durch Programmablaufpläne (PAP nach DIN 66 001) beschrieben.

5 Grundsätzlicher Aufbau

- ◆ Sammlung von Funktionen, auch über mehrere Dateien (Module)
 - Mindestens eine Funktion, nämlich *main*
 - Genau eine Funktion *main* in einem Programm
 - Meist Aufrufe von Bibliotheksfunktionen (z.B.: für Ein- und Ausgabe) im Funktionsblock
- ◆ Funktionen bestehen aus:
 - Typ des Rückkehrwertes (*return*-Wert)
 - Name der Funktion
 - Parameterliste in runden Klammern (Argumente)
 - Funktionsblock (oder einfach Block)

Grundsätzlicher Aufbau

- ◆ außerhalb der Funktionen-Definition stehen:
 - Anweisungen für Präcompiler (`#include`, `#define`, ...)
 - Funktionen-Deklarationen (Prototypen)
 - Globale/ externe Variablen
- ◆ außerdem können dort stehen:
 - Typ-Definitionen
 - und natürlich auch Kommentare
- ◆ Kommentare
 - `/* ... */` auch über mehrere Zeilen
 - `//` ab 1999 (C99)

Grundsätzlicher Aufbau

◆ ein Block besteht aus:

- Öffnender und schließender geschweiffter Klammer
- Vereinbarungen (Definition von Variablen und Feldern); diese müssen am Anfang des Blockes stehen
- Anweisungen bzw. Kontrollstrukturen
- Weiteren Blöcken (Blöcke können geschachtelt werden)

◆ Ab C99 können Vereinbarungen und Anweisungen gemischt untereinander stehen

- Nur für for-Schleife: `for (int i= 0; ...)` erlaubt

Syntax eines Blockes:

```
{  
    { Vereinbarung }  
    { Anweisung | Block }  
}
```

6 Erweiterte Backus-Naur-Form (EBNF)

- ◆ $[\dots]$ was zwischen den eckigen Klammern steht kann, muss aber nicht stehen.
- ◆ $\dots \mid \dots$ es gilt eine der Alternativen (die vor oder die nach dem Senkrechtrich)
- ◆ $< \dots >$ was zwischen den spitzen Klammern steht ist ein Platzhalter für einen Bezeichner
- ◆ $,$ Komma ist Trenner bei Aufzählungen
- ◆ $\{ \dots \}$ beliebige Anzahl des in Klammern stehenden Syntax-Elementes (auch null-mal)
- ◆ $\{ \dots \}_1$ wie oben, mindestens einmal
- ◆ $::=$ wird definiert durch (rechts vom Gleichheitszeichen)

7 Wichtige elementare Datentypen

◆ ganzzahlige Datentypen

- int 2 Byte, 4 Byte oder 8 Byte
(implementationsabhängig)
- long (long int) 4 Byte $-2^{31} \dots +2^{31} - 1$
(-2 147 483 648 ... 2 147 483 647)
- char 1 Byte -128 ... +127 (als Zahl)
0 ... +255 (als Zeichen)

◆ Gleitkommazahlen

- float 4 Bytes -3.4E+38 ... 3.47E+38
mindestens 6 Stellen Genauigkeit (dezimal)
- double 8 Bytes -1.7E+308 ... 1.7E+308
mindestens 15 Stellen Genauigkeit (dezimal)

- ◆ Es gibt keine elementaren Datentypen *String* (Zeichenkette, Text) und *byte*!

Vereinbarungen

◆ Vereinbarung elementarer Datentypen

- Müssen am Anfang eines Blockes stehen (bis C99)
- Reservieren Speicherplatz entsprechend der Größe der Datentypen (mit *sizeof* ermittelbar)
- Verschiedene Variablen gleichen Typs können durch Komma getrennt aufgezählt werden
- Jede Vereinbarung wird durch Semikolon abgeschlossen
- Variablen sind zufällig belegt, können aber eine Anfangsbelegung (Konstante) erhalten!
- Beispiele:

```
int i, j, k;  
float op1= 0.0f, op2= 0.0f;  
char zeichen= 'E';
```

8 Operatoren – Arithmetische Operatoren

- ◆ Für numerische Datentypen (auch Zeichen)
- ◆ `+` – Addition, `-` – Subtraktion, `*` – Multiplikation
- ◆ `/` – Division (bei ganzen Zahlen wird abgerundet, abgeschnitten)
- ◆ zusätzlich bei ganzzahligen Datentypen
- ◆ `%` – Restdivision (modulo)
- ◆ `++` – Inkrement (Bsp.: `i++`, `++i`)
- ◆ `--` – Dekrement (Bsp.: `j--`, `--j`)

Bem.: `i++` und `++i` ist nicht dasselbe

Zuweisungen, Bedingungsoperator

- ◆ `,=` – Zuweisung: der links vom Gleichheitszeichen stehenden Variable wird der rechts vom Gleichheitszeichen stehende Ausdruck zugewiesen
- ◆ `„+=“`, `„-=“`, `„*=“`, `„/=“` und `„%=“` verkürzte Schreibweise, wenn sich der Wert der Variablen aus dem alten Wert dieser Variablen `+`, `-`, `*`, `/`, und `%` einem Ausdruck ergibt (diese Schreibweise gilt auch für die Bitoperatoren)
- ◆ Bedingungsoperator (bedingter Ausdruck) ist der einzige dreistellige Operator

Syntax:

Vergleichsausdruck ? Ausdruck_1 : Ausdruck_2

Weitere Operatoren

◆ Vergleichsoperatoren

- | | | | |
|------|----------|----|----------------|
| • < | kleiner, | <= | kleiner gleich |
| • > | größer, | >= | größer gleich |
| • == | gleich, | != | ungleich |

◆ logische Operatoren

- && logisches ,und' (and)
- || logisches ,oder' (or)
- ! logisches ,nicht' (not)

◆ logische Werte ab C99

- Datentyp *bool* (als Makro in <stdbool.h>, _Bool)
- *true* und *false*

9 Ausdruck, Anweisung

◆ Ausdruck

- Beliebiger arithmetischer oder logischer Ausdruck
- Zuweisung oder Funktionsaufruf (jeweils ohne Semikolon als Abschluss)

Ausdrücke mit Wert ungleich 0 werden als ‚wahr‘ (true) gewertet. Gleich 0 ergibt ‚falsch‘ (false)!

◆ Anweisung

- Ausdruck oder Funktionsaufruf mit Semikolon als Abschluss
- Kontrollstruktur
- Ein Semikolon allein ist die leere Anweisung!

10 Kontrollstrukturen – Böhm-Jacopini-Theorem

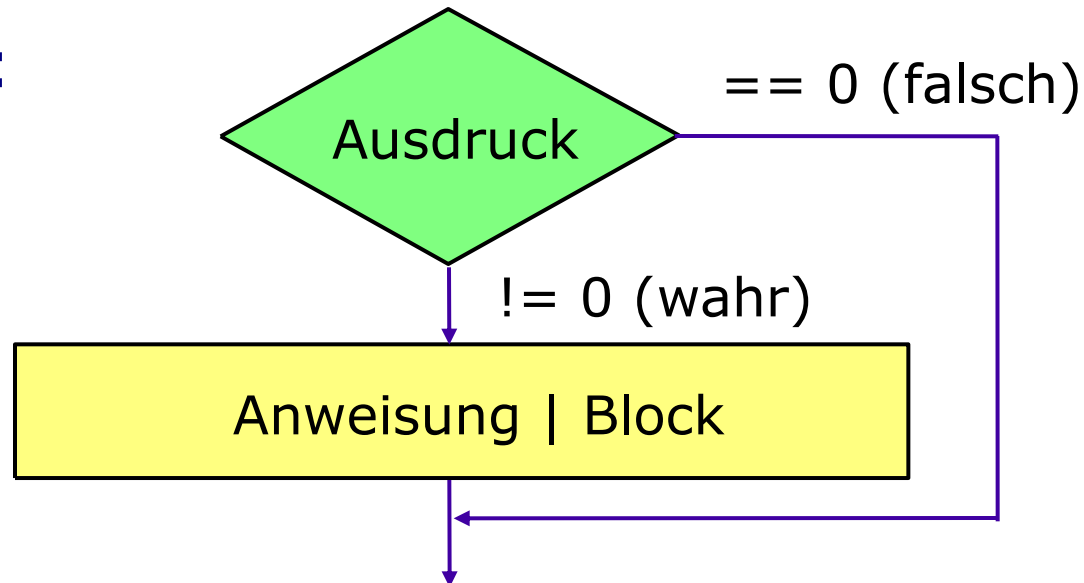
- ◆ Theorem von Corrado Böhm und Guiseppe Jacopini (1966) sinngemäß:
 - Gibt es zur Lösung eines Problems einen Algorithmus, so lässt sich dieser unter alleiniger Verwendung der drei Grundstrukturen Sequenz (Folge), Selektion (Verzweigung) und Zyklus (Iteration, Schleife) darstellen.
 - Grundlage für ‚strukturierte Programmierung‘ (goto-frei)
- ◆ Sequenz (Folge)
 - Folge von Aktionen, die genau einmal abgearbeitet werden
 - Algorithmen, die nur aus einer Sequenz bestehen heißen lineare Algorithmen (z. B.: Vertauschen der Inhalte zweier Variablen)
- ◆ Selektion (Auswahl, Verzweigung)
 - in Abhängigkeit einer Bedingung werden Aktionen ausgeführt oder nicht ausgeführt
- ◆ Zyklus/Iteration (Repetition, Wiederholung, Schleife)
 - Grundstrukturen werden wiederholt (mehrfach) abgearbeitet

Kontrollstrukturen (1/10)

Syntax der if-Verzweigung:

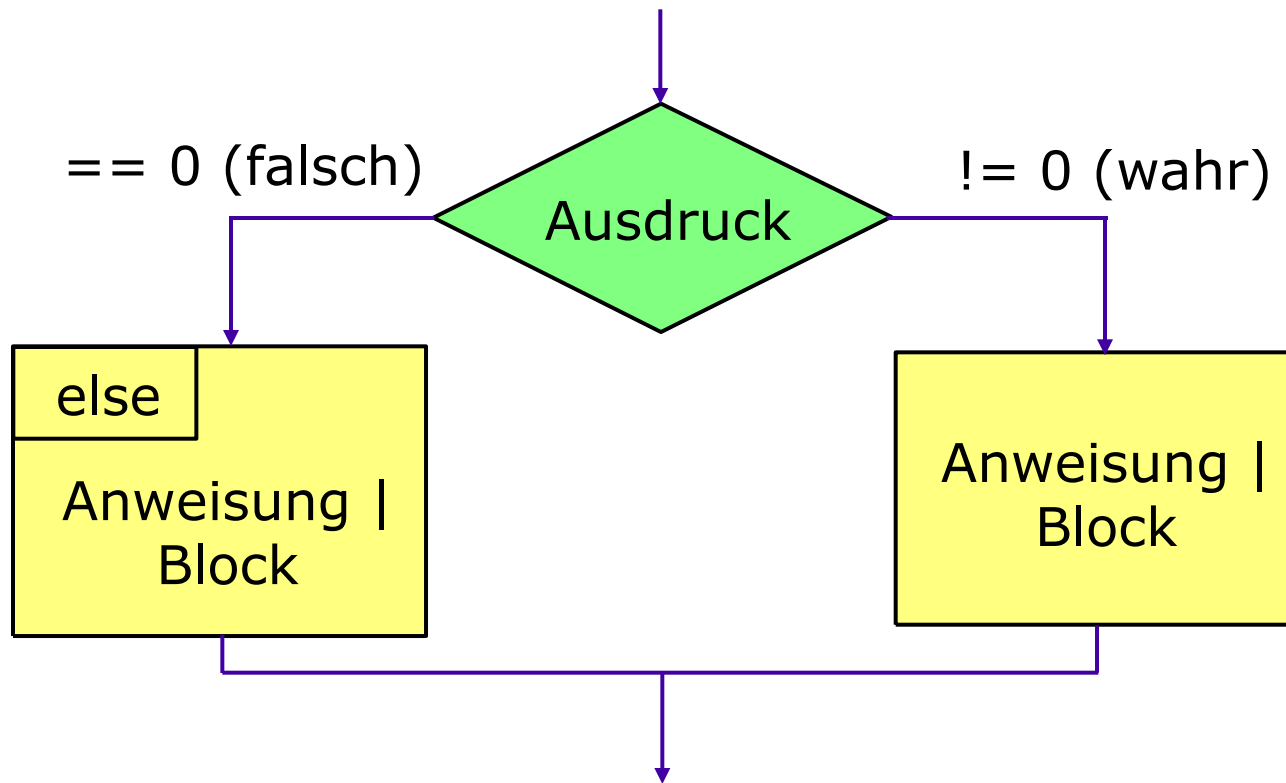
```
if (beliebiger Ausdruck)
    Anweisung | Block
[ else
    Anweisung | Block ]
```

Semantik: (1. Fall)



Kontrollstrukturen (2/10)

Semantik der if-Verzweigung (2. Fall mit *else*):



Kontrollstrukturen (3/10)

Syntax der for-Schleife:

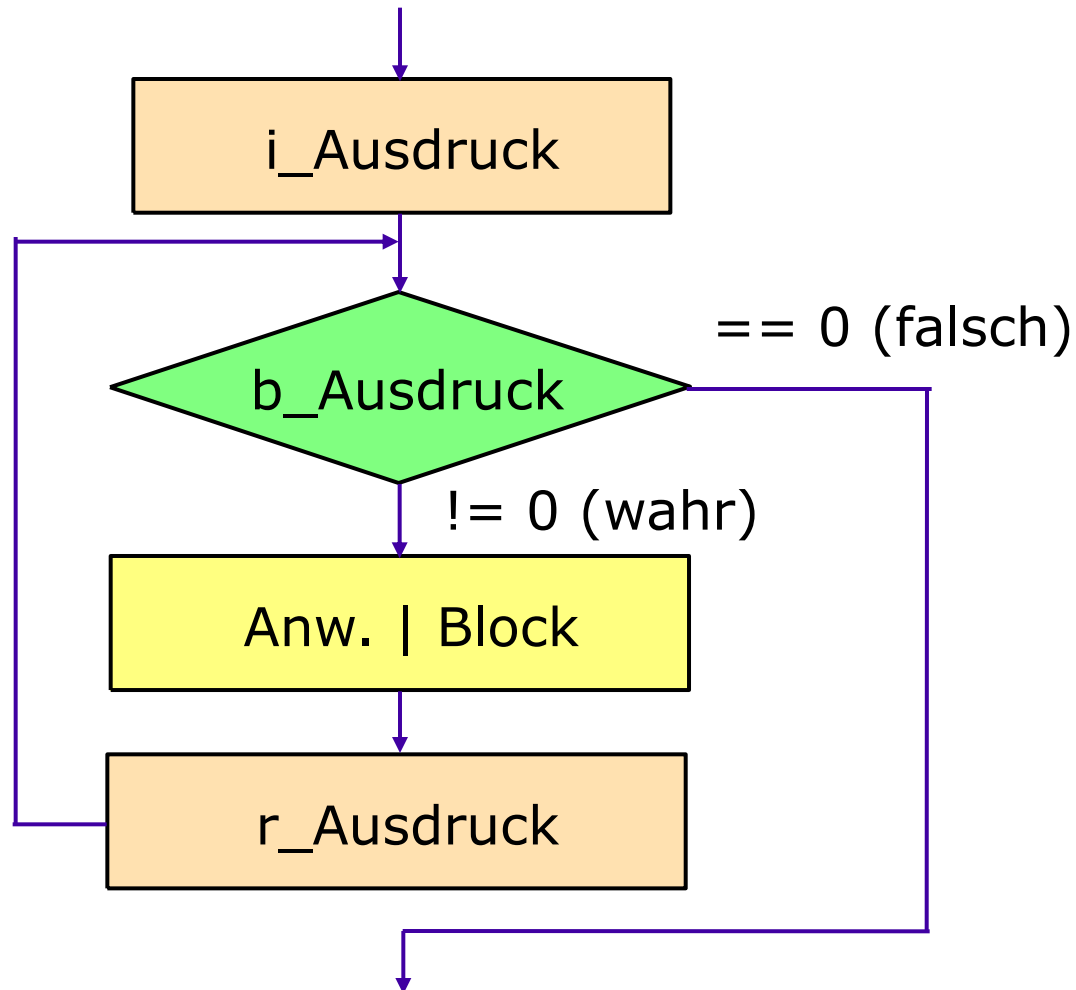
```
for ( [i_Ausdruck] ; [b_Ausdruck] ; [r_Ausdruck] )  
    Anweisung | Block
```

i_Ausdruck:	Initialisierung	(beliebiger Ausdruck)
b_Ausdruck:	Bedingung	(beliebiger Ausdruck)
r_Ausdruck:	Reinitialisierung	(beliebiger Ausdruck)

Beispiel: **for (;;);** ist zulässig!

Kontrollstrukturen (4/10)

Semantik der for-Schleife:

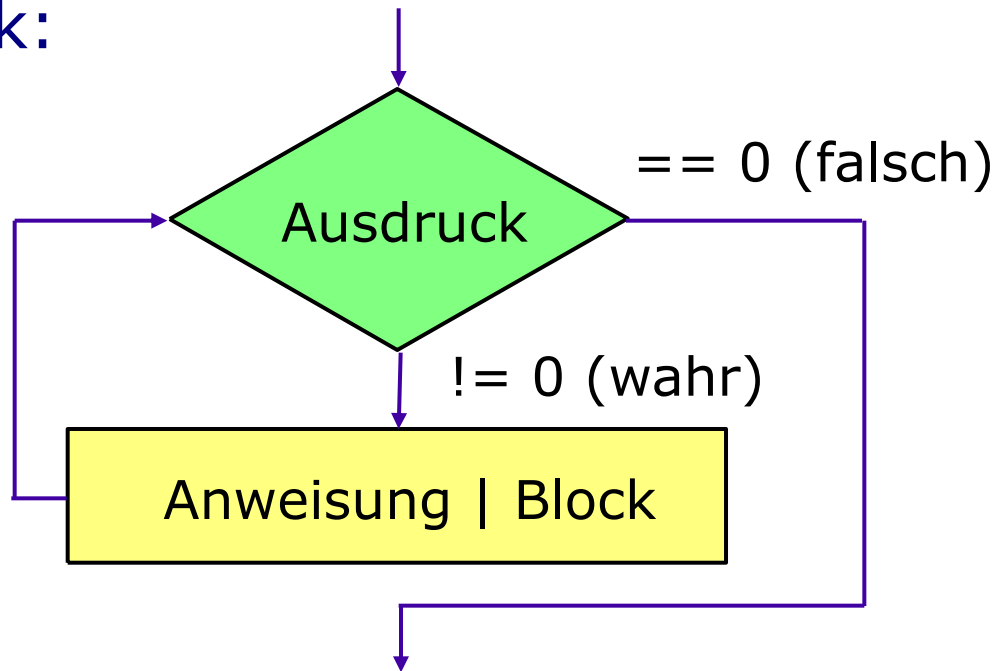


Kontrollstrukturen (5/10)

Syntax der kopfgesteuerten while-Schleife:

```
while (beliebiger Ausdruck)  
    Anweisung | Block
```

Semantik:



Kontrollstrukturen (6/10)

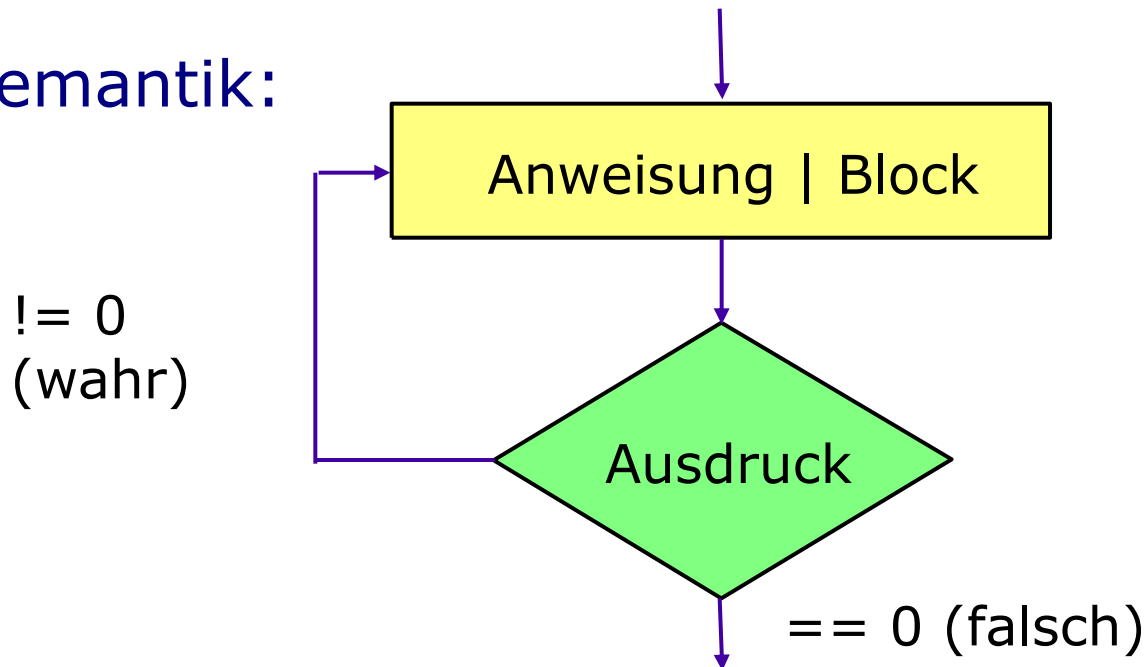
Syntax der fußgesteuerten while-Schleife:

do

Anweisung | Block

while (beliebiger Ausdruck) ;

Semantik:



Kontrollstrukturen (7/10)

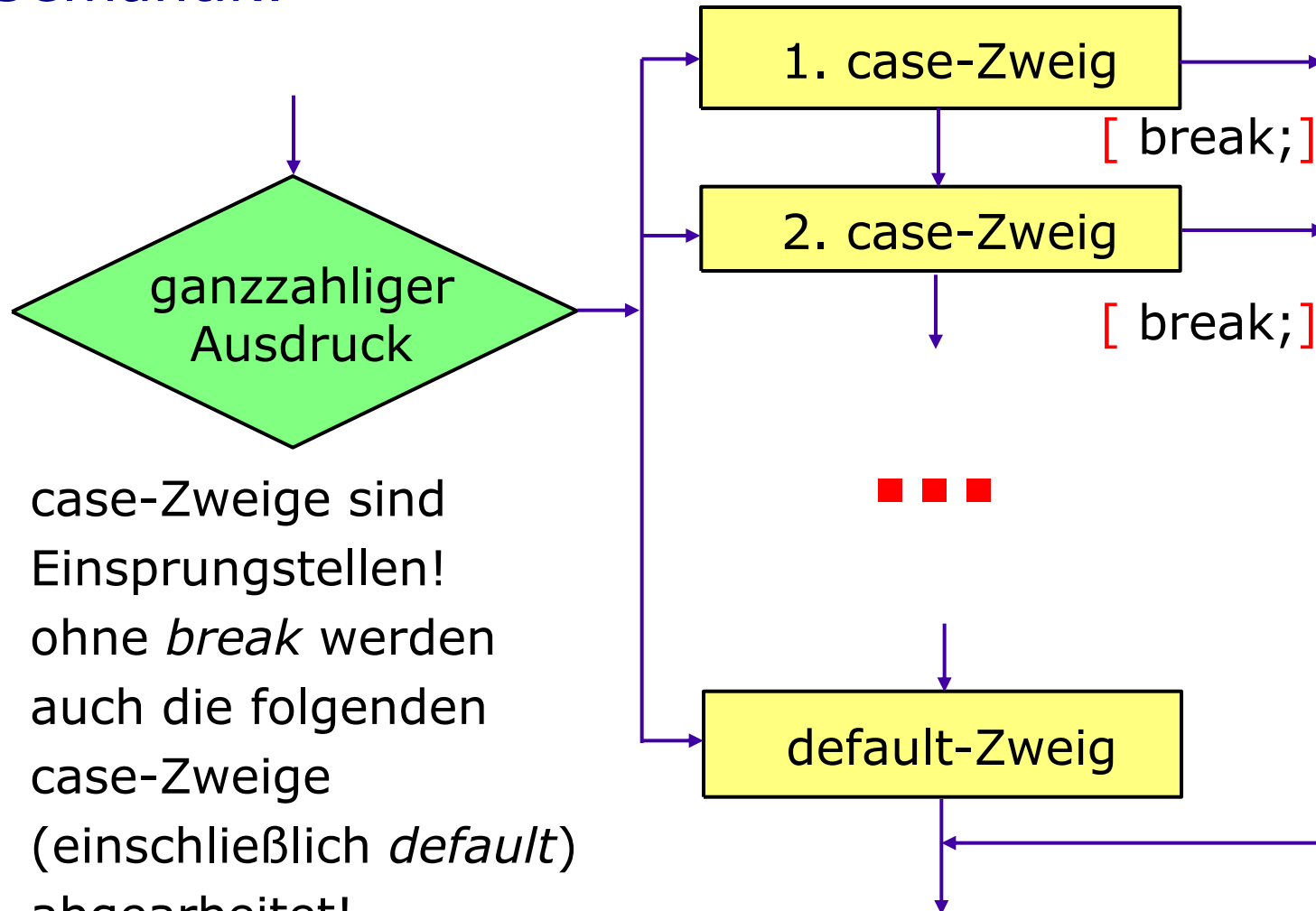
Syntax der switch-Struktur:

```
switch (ganzzahliger Ausdruck)
{
    case genau eine ganzz. Konstante:
        {Anweisung}
        {Block}
        [break;]
}

[ default: {Anweisung}
           {Block} ]
}
```

Kontrollstrukturen (8/10)

Semantik:



Kontrollstrukturen (9/10)

◆ `continue`

- Nur für Schleifen *for*, *while* und *do ... while* gültig
- Der aktuelle Schleifendurchlauf wird abgebrochen, die Re-Initialisierung wird ausgeführt (bei der *for*-Schleife) und es wird zum Schleifentest gesprungen

◆ `break`

- Nur für Schleifen *for*, *while*, *do ... while* und *switch*
- Nur die unmittelbar umgebende Schleife oder *switch* wird verlassen (kein Sprung aus mehreren ineinander geschachtelten Schleifen möglich!)

Kontrollstrukturen (10/10)

◆ goto

- äußerst sparsam verwenden!
- Im Sinne strenger strukturierter Programmierung nicht erlaubt!

◆ *Syntax:* goto Marke; . . . Marke:

- return

- Rückkehr in unmittelbar aufrufende Funktion (keine Funktion!)

- ◆ *Syntax:* return [Ausdruck];

- ◆ `exit` (ganzzahliger Ausdruck); Standardfunktion!

11 Funktionen

- ◆ Wiederverwendbarkeit
 - Beliebig oft ausführbar
 - Programm wird kürzer und einfacher
- ◆ Modularität
 - Einfachere Pflegbarkeit
 - Wiederverwendbarkeit
- ◆ Portabilität
 - nicht-ANSI-Programmteile in extra Funktionen packen
- ◆ Funktionen sind alle gleichberechtigt, sie können nicht geschachtelt definiert werden (wie z.B. in Pascal)
- ◆ Funktionen sind global, das heißt überall bekannt
- ◆ *main* ist die Anfangsadresse des Programms

Arbeit mit Funktionen

- ◆ 1. Funktionen-Deklaration (Prototyp der Funktion)
 - Bekanntgabe des Namen der Funktion und der Typen des Rückkehrwertes und der Parameter der Funktion (Parameternamen sind nicht erforderlich)
 - Belegt keinen Speicherplatz
 - Ermöglicht dem Compiler die Überprüfung von Übereinstimmung mit der Funktions-Definition
- ◆ 2. Funktionen-Definition
 - Enthält die Implementation der Funktion (Programmzeilen)
 - Der Funktionsblock kann auch leer sein (nur evtl. entsprechende return-Anweisung).
- ◆ 3. Funktions-Aufruf
 - Ausführen der Funktion

Funktionen-Deklaration

◆ Funktionen-Deklaration nach ANSI-C

- `int max(int a, int b); // Prototyp`
 Parameternamen können fehlen: `int max(int, int);`
- `int max(int a, int b) // Definition`
 `{ return ((a > b) ? a : b); }`

Rückkehrwert (return-Wert)

- ◆ Durch eine Funktion kann genau ein Wert über den Rückkehrwert zurück gegeben werden
 - Sollen mehrere Werte zurück gegeben werden, so muss mit Parametern gearbeitet werden oder es muss ein Feld angelegt werden, von dem die Adresse zurück gegeben wird.
 - Der Typ des Rückkehrwertes ist der Typ der Funktion.
- ◆ Beispiel:

```
double wurzel(double x)    // Definition
{
    ...
    return ergebnis;
}

. . .

erg = wurzel(op1);         // Aufruf
```

Standardfunktionen

◆ 1. Funktionen-Deklaration

- Die Deklaration (Prototyp) steht in der entsprechenden Header-Datei (Verzeichnis „...\\include“)
- Die Header-Datei wird im Programm mittels Präcompiler-Anweisung angegeben. Beispiel: `#include <stdio.h>`
- durch die spitzen Klammern wird das Standard-include-Verzeichnis durchsucht

◆ 2. Funktionen-Definition

- die Funktion ist in der entsprechenden Bibliothek (Verzeichnis „...\\lib“)
- Durch den Linker werden die entsprechenden Funktionen aus den Bibliotheken hinzugefügt

◆ 3. Funktions-Aufruf

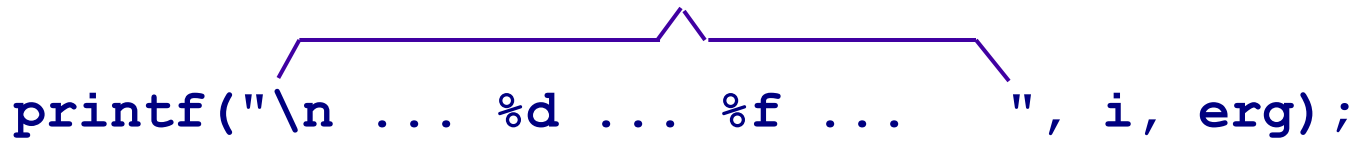
Standardfunktionen

Funktion „printf“

Syntax am Beispiel:

Text, der auf dem Bildschirm ausgegeben wird

`printf("\n ... %d ... %f ... ", i, erg) ;`



Platzhalter
für Variablen;
hinter dem ,%`
steht der Variablentyp

Variablen in der Reihenfolge
der Platzhalter

Bemerkung: `\n` steht für *newline* (neue Zeile)

12 Rekursive Funktionen

- ◆ Alle Berechnungen lassen sich rekursiv beschreiben
- ◆ Jede Rekursion ist umwandelbar in eine Iteration
- ◆ Beispiel der Fakultät wird nur zu Demonstrationszwecken verwendet, da diese Berechnung trivialerweise als Iteration schreibbar ist
- ◆ Lange Zeit war es ein Hauptziel der Informatik rekursive Berechnungen auf Iterationen zurück zu führen. Dies kann sehr aufwändig sein.
- ◆ Manche Programmiersprachen kannten keine Rekursion (Fortran IV).
- ◆ Viele Probleme sind natürlicherweise und elegant rekursiv formulierbar (binäre Bäume).
- ◆ Iterative Lösungen sind meist effizienter.

Beispiel: Fakultät

```
#include <stdio.h>

long long fakul_r(int n)    // rekursive Variante
{
    if (n < 2)
        return 1;

    return (n * fakul_r(n-1));
}

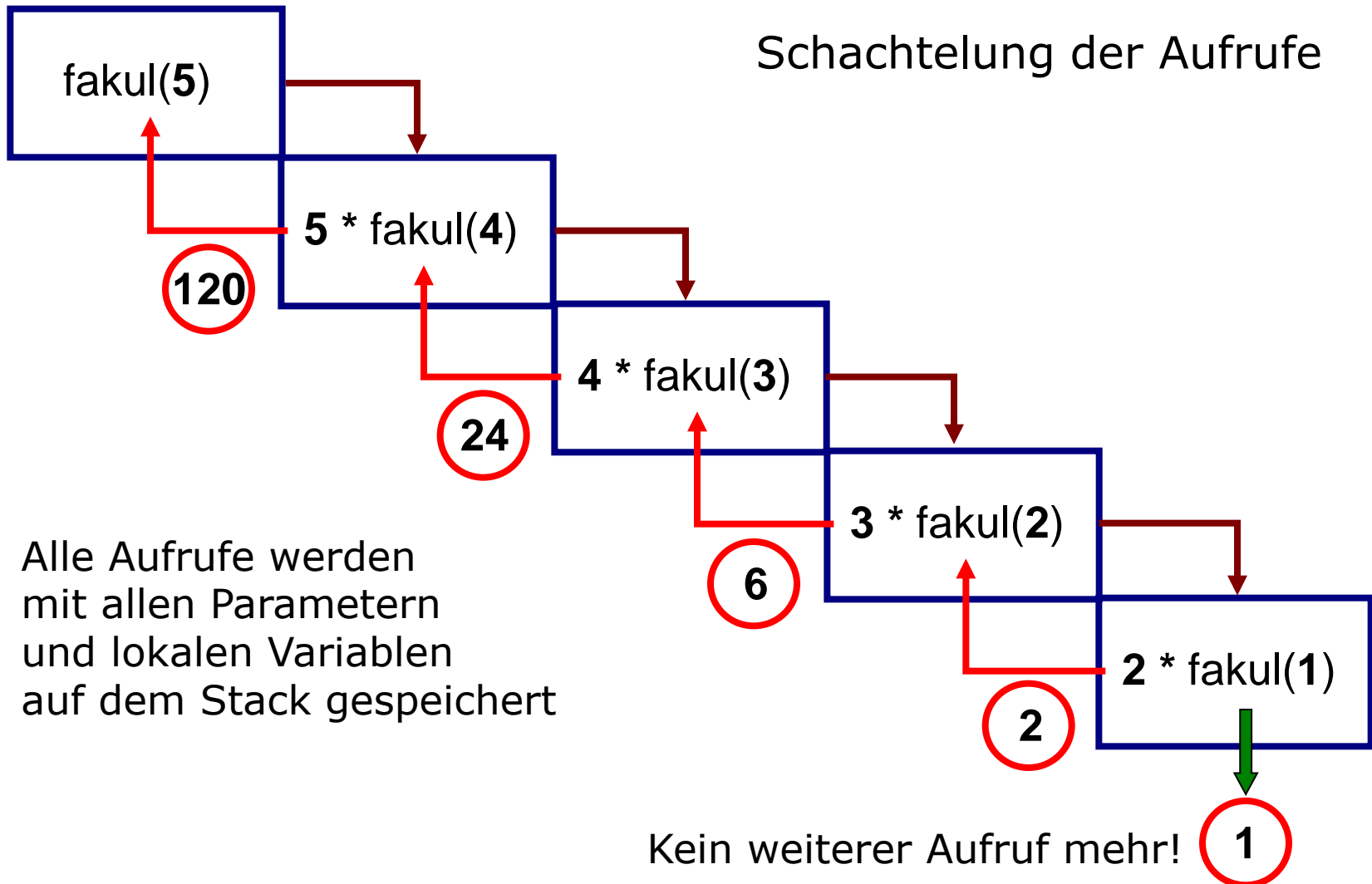
long long fakul_i(int n)    // iterative Variante
{
    long long fak= 1;
    int i;

    for (i= 2; i <= n; i++) fak*= i;

    return fak;
}
```

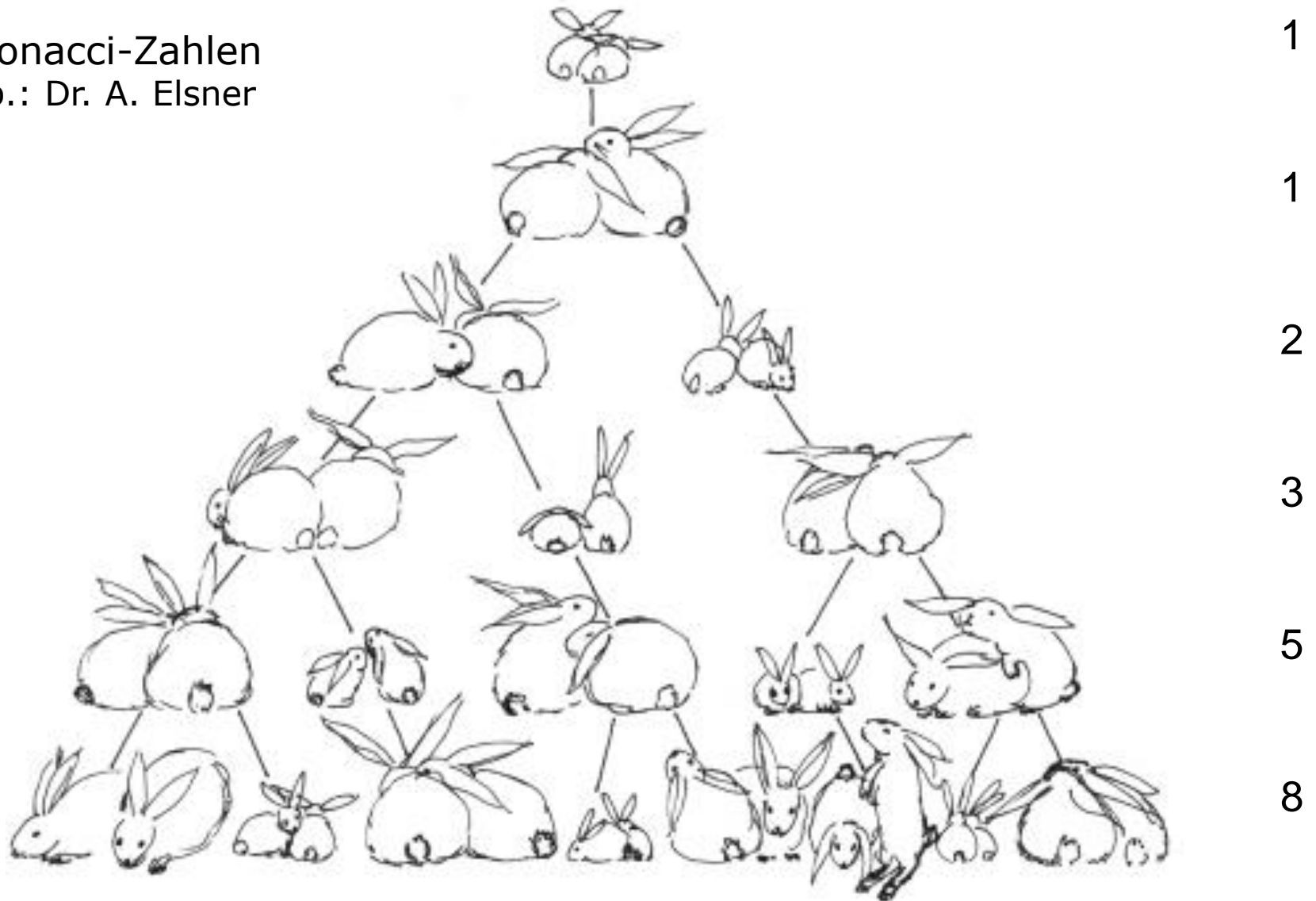

Beispiel: Fakultät

Schachtelung der Aufrufe



Beispiel: Fibonacci-Zahlen

Fibonacci-Zahlen
Abb.: Dr. A. Elsner



Beispiel: Fibonacci-Zahlen

- ◆ Fibonacci: Leonardo von Pisa (1170–1240)
- ◆ Wie viele Kaninchenpaare gibt es nach einem Jahr, wenn es anfangs ein junges Paar gibt, jedes Paar jeden Monat ein neues Paar zur Welt bringt und alle Jungen sich jeweils nach einem Monat fortpflanzen?
- ◆ Zahlenreihe:
1 1 2 3 5 8 13 21 34 55 89 144 ...
- ◆ Mathematische Beschreibung:
 $F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}; n \geq 2$

Beispiel: Fibonacci

```
long long fibo_i(int n)    // iterative Variante
{
    long long fib1= 1, fib2= 1, tmp;
    long double gm;
    int i;

    for (i= 2; i <= n; i++)
    {
        tmp= fib2;
        fib2 += fib1;
        fib1= tmp;
// explizites Type-Casting
        gm= (long double) fib2 / (long double) fib1;
        printf("\n%16lld, %20.12Lf", fib2, gm);
    }
    return fib2;
}
```

13 Felder (Vektoren, Arrays)

- ◆ Ein Feld ist die Aneinanderreihung von Elementen gleichen Typs (elementare Datentypen, Strukturen, selbst definierte Datentypen, ..). Die Elemente stehen im Speicher lückenlos mit aufsteigenden Adressen hintereinander.
- ◆ Es gibt in **C** keine Sprachelemente zur Verarbeitung von kompletten Feldern (die eckigen Klammern sind nur Syntaxelemente für die Arbeit mit Indizes). Die Arbeit mit Feldern erfolgt mit Standardfunktionen oder elementweise.
- ◆ Der Zugriff zu den einzelnen Elementen eines Feldes erfolgt über eine Nummerierung (Index), die in **C** bei Null beginnt!

Arbeit mit Feldern

- ◆ Der Name des Feldes ist die Anfangsadresse des Feldes, also die Adresse des ersten Elementes (mit Index 0).
`feld = &feld[0]`
- ◆ Wird die Anfangsadresse des Feldes benötigt (z.B. bei Parametern im Funktionsaufruf), kann einfach der Name des Feldes angegeben werden.
- ◆ Beispiel: Feld ganzer Zahlen

Index	0	1	2	3	4	5	6	7
Inhalt	12	7	-3	8	-13	0	5	1

Vereinbarung von Feldern und Zugriff

```
int feld1[10];    /* Vereinbarung */
```



Anzahl der Feldelemente muss bei der Compilierung bekannt sein (bis C99).

Es kann ein ganzzahliger aber konstanter Ausdruck angegeben werden.

Ab C99 kann die Anzahl der Feldelemente während der Laufzeit ermittelt werden, kann aber danach nicht mehr verändert werden (semidynamisch).

. . .

```
hilf= feld1[i];    /* Inhalt des Feldelementes */  
                  /* wird kopiert                */
```

. . .



Nummer des Feldelementes kann durch einen beliebigen ganzzahligen Ausdruck angegeben werden.

Feld-Vereinbarung

- ◆ Beispiele für Feldvereinbarungen

```
#define ANZ 10

int feld1[ANZ];    // zufällige Anfangswerte
int feld2[]={ 1, 3, -5, 7, -2, 0, 9, 4, -1, 10 };
float feld3[15];

int k= 8;
double feld4[k];   // nicht erlaubt bis C90
                  // ab C99 erlaubt
```

- ◆ Die Länge des Feldes ist mit der Vereinbarung festgelegt, sie kann auch nicht nachträglich verändert werden!
- ◆ Dynamische Speicherverwaltung kann mit den Funktionen *malloc*, *calloc* und *free* realisiert werden.

Indizes

- ◆ Der Zugriff auf die Feldelemente erfolgt über den Index
- ◆ Dabei kann der Index ein beliebiger ganzzahliger Ausdruck sein
- ◆ Es erfolgt keinerlei Überwachung der Überschreitung von Feldgrenzen, d.h. weder bei der Compilierung noch bei der Ausführung wird geprüft, ob auf das Feld über die vereinbarten Grenzen hinaus zugegriffen wird
- ◆ Der Index kann größer sein als Anzahl der Elemente -1 und der Index kann kleiner als Null sein (widerspricht nicht den Syntaxregeln)
- ◆ Die Verantwortung für die Arbeit mit Feldindizes liegt beim Programmierer!

14 Zeichenketten (String, Texte)

- ◆ Zeichenketten (Texte, Strings) in **C** sind Felder einzelner Zeichen mit dem Ende-Kennzeichen „binäre Null“ als letztes Zeichen
- ◆ Der Speicherplatz für das Ende-Kennzeichen muss durch den Programmierer berücksichtigt werden.
- ◆ Da es in **C** keinerlei Sprachelemente für die Arbeit mit Feldern gibt, muss die Verarbeitung von Texten in **C** mittels der dafür vorgesehenen Standardfunktionen oder in eigenen Programmstücken mit Schleifen elementweise erfolgen.

Index	0	1	2	3	4	5	6	7
Inhalt	'H'	'a'	'l'	'l'	'o'	'\0'	...	

Vereinbarung von Texten

```
char text1[80+1];    /* Vereinbarung */
```



Anzahl der maximal benötigten einzelnen Zeichen plus Speicherplatz für die binäre Null.

- ◆ Die binäre Null ist nicht das Zeichen (Ziffer) `,0'`. Dieses wird intern als `ascii-Code 48, 0x30` oder `060` gespeichert.
- ◆ Bei der binären Null sind alle Bits mit 0 belegt.

```
char zeichen= 0;    oder char zeichen= '\0';
```

- ◆ Der Datentyp *char* wird in **C** wie ein ganzzahliger Datentyp behandelt.

15 Der Präcompiler

- ◆ Der Präcompiler (Präprozessor) wird vor der eigentlichen Compilierung ausgeführt
- ◆ Die Arbeit des Präcompilers besteht aus Texteingfügungen, Textersetzungen und Anweisungen für bedingte Compilierung
- ◆ Die Originaldatei wird nicht verändert! Es wird nur eine temporäre Datei erzeugt, die dann compiliert wird
- ◆ Der Präcompiler erfüllt folgende Aufgaben:
 - Hinzufügen von Dateien zur Quellcode-Datei (`# include`-Direktive)
 - Definition von Konstanten und Bezeichnern (`# define`-Direktive)
 - Definition von Makros (`# define`-Direktive)
 - Anweisungen für bedingte Compilierung (`# if`-Direktive, ...)
 - Neuvergabe der Zeilennummern (`# line`-Direktive)

include-Direktive

- ◆ Alle Direktiven (Anweisungen) für den Präcompiler beginnen mit dem Doppelkreuz (Raute) ,#`
- ◆ Mit der include-Direktive können im Prinzip beliebige Textdateien zur Quelle hinzugefügt werden. Es werden üblicherweise nur sogenannte Header-Dateien mit Extension ,.h` eingefügt.
- ◆ Standard-Headerfiles werden in spitze Klammern eingeschlossen.
- ◆ Eigene Headerfiles stehen in Anführungszeichen

```
#include <stdio.h>
#include <string.h>
#include "typedefinition.h"
#include "dateiarbeit.h"
```

define-Direktive

- ◆ Alle Bezeichnungen in den define-Direktiven werden üblicherweise mit Großbuchstaben geschrieben. Damit werden die Namen von ‚normalen‘ Variablen und Funktionen unterschieden.
- ◆ Es können Konstanten und Bezeichner definiert werden
- ◆ So kann mit Datentypen gearbeitet werden, die in **C** nicht vorhanden sind (ab C99 gibt es den Datentyp *bool* als Makro in „<stdbool.h>“ aus „_Bool“).

```
#define PI 3.141592653
#define MWST 0.16
#define BOOL int
#define TRUE 1
#define FALSE 0
```

Fehlerbehandlung

- ◆ Eine sorgfältige und umfassende Fehlerbehandlung ist ein sehr wichtiger Teil der Programmierung
- ◆ Dieser Teil der Programmierung wird oft sehr stiefmütterlich behandelt und oft als lästig empfunden
- ◆ Eine sorgfältige Fehlerbehandlung ist eine gute Voraussetzung für qualitativ gute Programme
- ◆ Die Fehlerbehandlung kann in mehreren Stufen erfolgen:
 - 1. Fehlervermeidung durch Abfangen von fehlerhaften und unsinnigen Eingaben, Test auf ungültige Zahlenbereiche, Vermeidung ungültiger Operationen, usw.
 - 2. Aussagekräftige und ausreichende Fehlernachrichten
 - 3. Sinnvolle Reaktionen auf Fehler (Wiederholung von Aktionen, Abbruch von Aktionen, Abbruch des Programms)

Fehlerbehandlung

- ◆ Die verschiedenen Stufen der Fehlerbehandlung erfordern verschiedene Herangehensweisen bei der Programmierung
- ◆ Die Vermeidung von fehlerhaften und unsinnigen Nutzereingaben erfolgt meist in den Funktionen, wo die Eingabe erfolgt
- ◆ Die Ausgabe von Fehlermeldungen sollte **nicht** dort erfolgen, wo der Fehler auftritt, aus folgenden Gründen:
 - Redundanz (viele gleichartige Fehler werden in verschiedenen Funktionen mehrfach dokumentiert)
 - Funktionen ohne Ein- Ausgabe erhalten Fehlerausgaben
 - Programme werden unübersichtlich, schwer wartbar und unnötig lang
 - Die Fehlermeldungen lassen sich schwer systematisieren, gruppieren und auflisten

Beispiel für Nutzereingabe

```
...
int fehler;

do
{
    fehler= 0;

    printf("\n Preis in der Form xxx.xx: ");
    fflush(stdin);

    if (!scanf("%f", &preis))
        fehler= 1;

    if (preis < 0.0 || preis > 1000.0)
        fehler= 1;

} while(fehler);
```

Kommentare

- ◆ Kommentare am Anfang von Funktionen dienen zur Identifizierung (Datum, Version, Name des Autors, usw.)
- ◆ Der Zwang, kommentieren zu müssen, führt dazu, dass oft schon bei der Überlegung nach der richtigen Formulierung Denkfehler auffallen.
- ◆ Der Zeitaufwand für die Kommentare ist oft verschwindend gegenüber dem Zeitaufwand für die Fehlersuche, also keine Entschuldigung, für das „Einsparen“ der Kommentare.
- ◆ Selbst wenn Programmteile eventuell später wieder gelöscht werden, ist das sofortige Kommentieren keine Zeitverschwendung.
- ◆ Kommentare helfen dabei, dass später noch (auch für den Programmierer selbst) die Gedanken nachvollziehbar sind.
- ◆ Auch Kommentieren will gelernt sein.

Hinweise zur Schreibweise (Syntax)

1. Variante: `if (...) // sehr oft in C/C++`
`{`
 `...`
`}`

2. Variante: `if (...) { // in Java stark verbreitet`
 `...`
`}`

3. Variante:
 `if (...)`
 `{`
 `...`
 `}`

Weitere Varianten sind denkbar, wichtig ist die Entscheidung durchgängig für eine Variante!

Literaturangaben

- [1] Kernighan/Ritchie: „*Programmieren in C*“, Carl Hanser Verlag München Wien, ISBN 3-446-15497-3
- [2] D. Frischalowski, J. Palmer: „**ANSI C 2.0** Grundlagen der Programmierung“, Herdt-Verlag für Bildungsmedien GmbH, CANSI2 30-0-07-03-01