

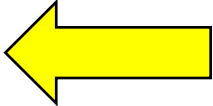
**Folien zur Vorlesung
Grundlagen systemnahes Programmieren
Wintersemester 2016
(Teil 3)**

Prof. Dr. Franz Korf

Franz.Korf@haw-hamburg.de

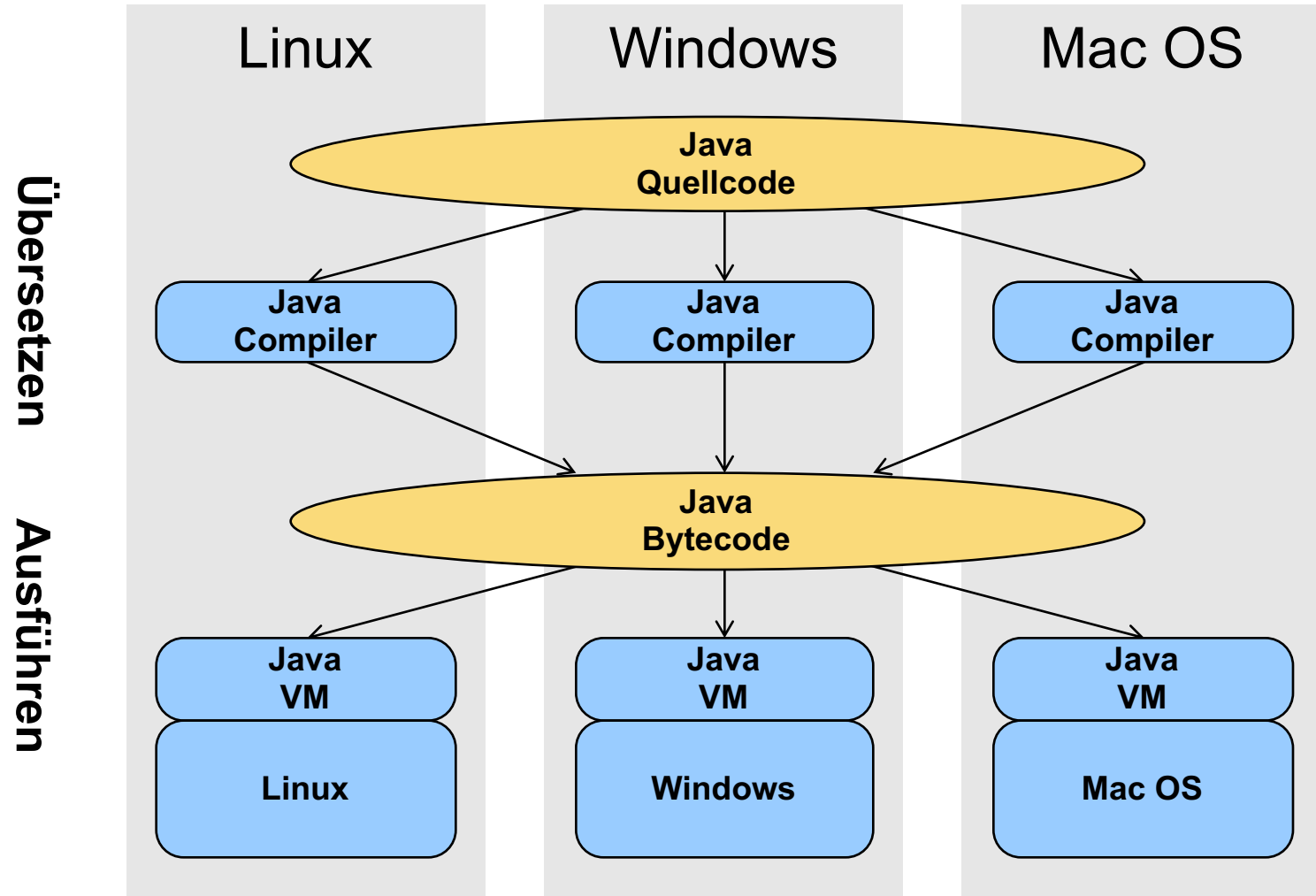
Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung 
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

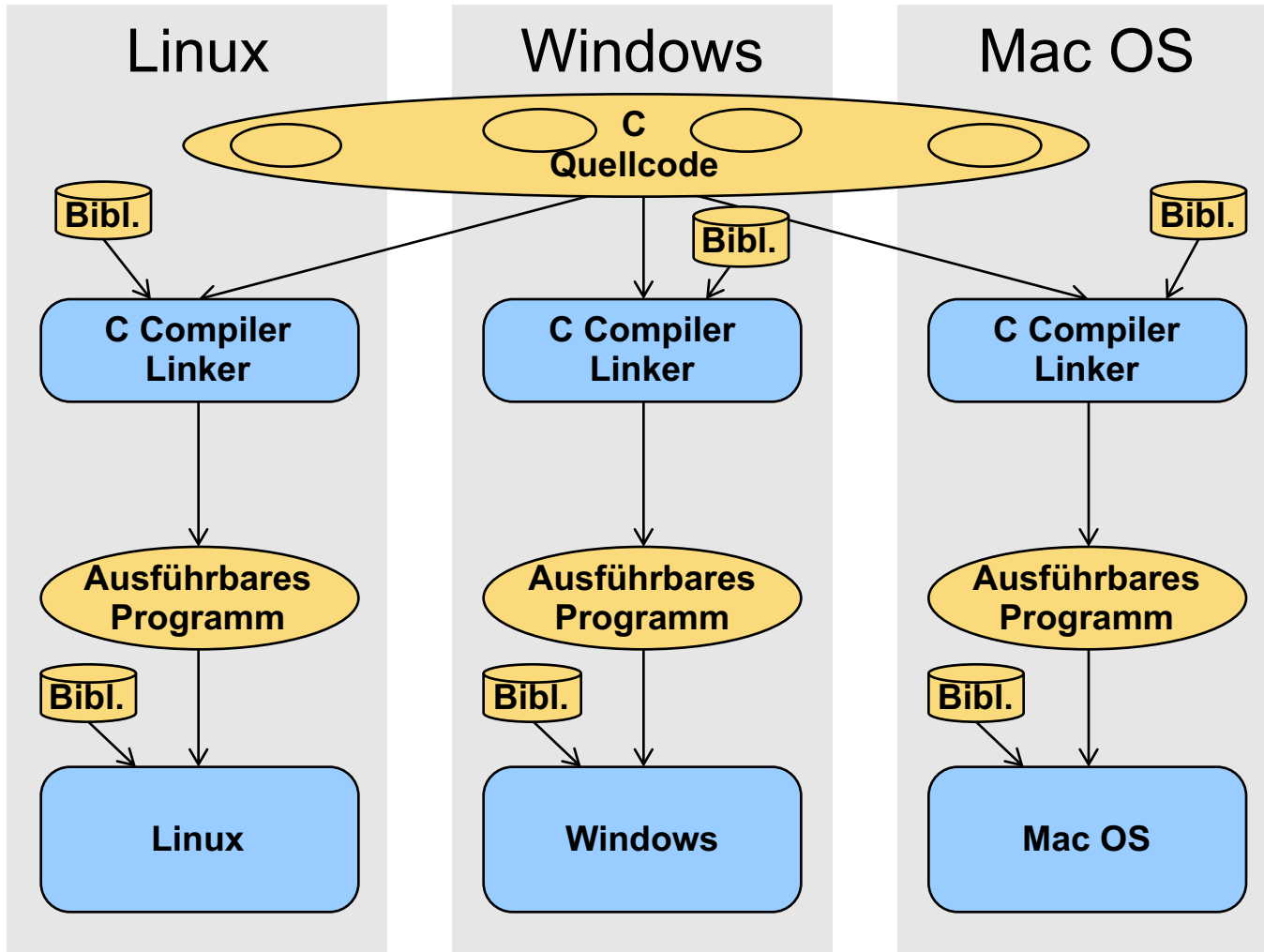
Die Folien zu dieser Vorlesung basieren auf Ausarbeitungen von, Heiner Heitmann, Reinhard Baran und Andreas Meisel

SW Entwicklung mit Java

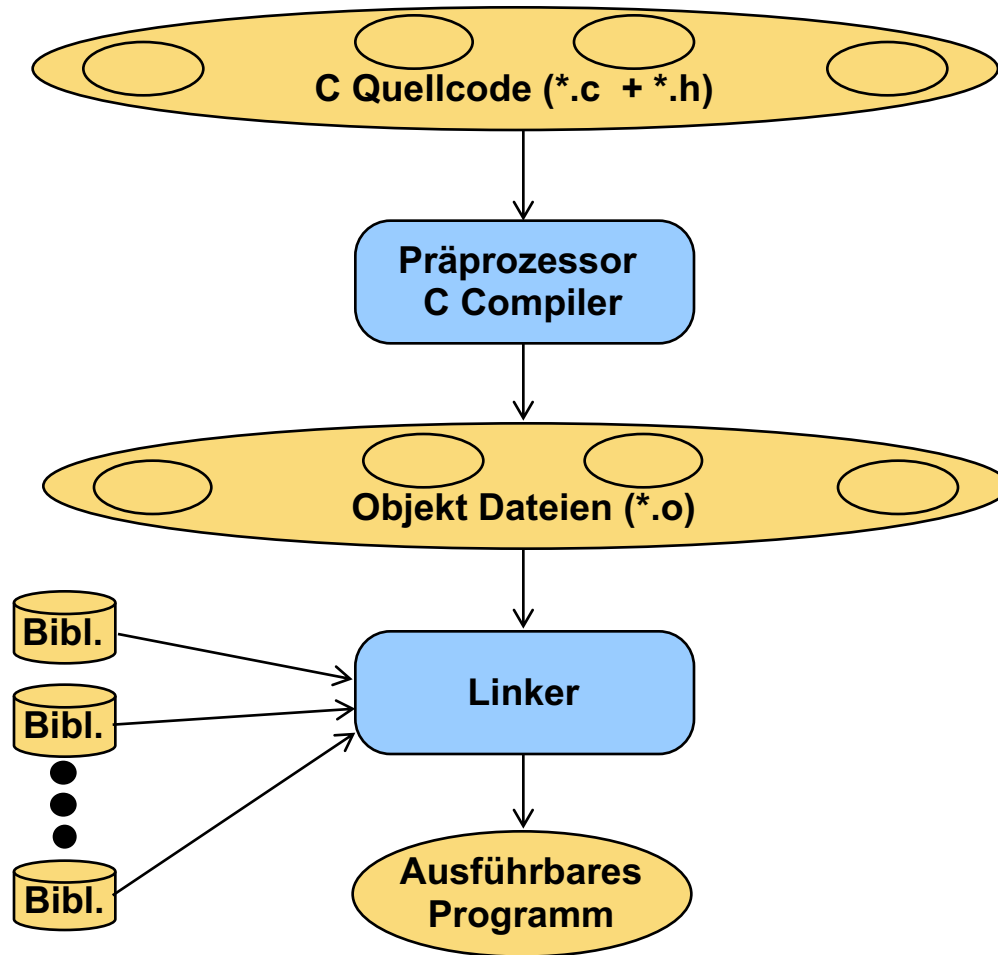


SW Entwicklung mit C (Teil 1)

Übersetzen
Ausführen



SW Entwicklung mit C (Teil 2)



Historie

- C wurde 1972 von Dennis Ritchie bei den AT&T Bell Lab als Systemprogrammiersprache zur Implementierung von UNIX für die PDP-11 entwickelt.
- Ziel von C war die Entwicklung einer Hochsprache für **lesbare** und **portable Systemprogramme**, aber einfach genug, um auf die zugrunde liegende Maschine abgebildet zu werden.
- In 1973/74 wurde C von Brian Kernighan verbessert. Daraufhin wurden viele Unix-Implementierungen von Assembler nach C umgeschrieben.
- Um die Vielzahl der entwickelten Compiler auf einen definierten Sprachumfang festzulegen, wurde C 1983 durch die amerikanische Normbehörde ANSI normiert.
Diese Sprachnorm wird als ANSI-C bezeichnet. C90, C99 C11 (2011 akt. Version)
- ANSI-C sollte von jedem Compiler fehlerfrei übersetzbar sein.

Stärken und Schwächen von C

Stärken:

- standardisiert (ANSI)
- sehr effizient (hardwarenah implementiert)
- universell verwendbar, da C verschiedene Programmierparadigmen beinhaltet (prozedural, modular und C++ objektorientiert)
- sehr weit verbreitet, speziell in der technischen Informatik
- Typüberprüfung (strong typing)

➤ Schwächen:

- C Code kann beliebig unlesbar geschrieben werden, da Fragen des Programmierstils nur in (freiwilligen) Konventionen festgelegt sind.
→ Erfahrung und persönlicher Stil haben entscheidenden Einfluss auf die Softwarequalität !
- teilweise etwas kryptische Notation

Schwäche von C: Gefahr des stillosen Codes

```

/* ASCII to Morsecode */
#include<stdio.h>
#include<string.h>
main()
{
    char*O,I[999]="``acgo\177~|xp .-\0R^8)NJ6%K4~A2~UID57$3G1FBL";
    while(O=fgets(I+45,954,stdin)){
        *I=O[strlen(O)[O-1]=0,strlen(O)-1];
        while(*O)switch((*I&0xf)-(*O))-!*I){
            case-1:{char*I;O+=strlen(O,I+12)+1)-2,O=34;
                while(*I&3&&(O=(O-16<<1)+*I---'-')<80);
                putchar(O&93?*I&8||!(I=memchr(I,O,44))?'?:I-I+47:32);
                break;
            case 1: ;*I=(*O&31)[I-15+(*O>61)*32];
                while(putchar(45+*I%2),(*I=*I+32>>1)>35);
            case 0: putchar((++O,32));}
        putchar(10);}
}

```


Fundamentale Eigenschaften von C

C ist

- **klein**
 - 32 Schlüsselwörter und
 - 40 Operatoren
- **modular**
 - alle Erweiterungen stecken in Funktionsbibliotheken
 - unterstützt das Modulkonzept
- **maschinennah**
 - geht mit den gleichen Objekten um wie die Hardware: Zeichen, Zahlen, Adressen, Speicherblöcke

C lebt von vielen Funktionen, die in Bibliotheken gesammelt sind.

Das berühmte erste Programm “Hello World”

```
#include <stdio.h>

#define MY_STR "Hello World\n"

int main( )
{
    printf(MY_STR);
    return 0;
}
```

- Genau eine main-Funktion wird stets benötigt, damit der Compiler den Beginn des Hauptprogramms erkennt. Das Programm steht zwischen { ... }.
- Der Typ des Rückgabewerts der Funktion `main` und somit des Programms ist `int` (ganze Zahl).
- Der Rückgabewert dieses Programms ist 0 vom Typ `int` (`return 0;`)
- Rückgabewert der Funktion `main` wird an das Betriebssystem weitergereicht. Er signalisiert dem Betriebssystem, ob das Programm fehlerfrei abgelaufen ist:
 - 0: fehlerfrei
 - sonst: Fehlerkode

Präprozessor

- `#include <stdio.h>` (im obigen Beispielprogramm)
- `#` in der ersten Spalte: Präprozessor-Anweisungen
- Präprozessor-Anweisungen sind kein direkter Bestandteil der Sprache C - sondern ein Befehle des **Präprozessors**.
- Der Präprozessor (cpp) führt vor dem eigentlichen Übersetzungsvorgang eine **Textersetzung** durch und erlaubt die Steuerung des Übersetzungsvorganges.
- Aufgrund der `#include` Anweisung setzt der Präprozessor den Text der Datei `stdio.h` an der Stelle ein, wo die `#include` Anweisung im Programm steht.
- Aufgrund der `#define` Anweisung setzt der Präprozessor den Text `"Hello World\n"` überall anstelle von `MY_STR` ein.

Beispiel Präprozessor

```
/* Der Präprozessor expandiert #include <stdio.h>
   und ersetzt MY_STR
*/
```

```
/*
 * stdio.h      Standard I/O functions
 *
 * Copyright by QNX Software Systems Limited 1990-1995. All rights reserved.
 *
 * Copyright (c) 1994-2000 by P.J. Plauger.  ALL RIGHTS RESERVED.
 * Consult your license regarding permissions and restrictions.
 */

...
extern int      perror( FILE *__fp );
extern int      fflush( FILE *__fp );
extern int      fgetc( FILE *__fp );
...
```

```
int main( )
{
    printf("Hello World\n");
    return 0;
}
```

Ausgabe

➤ Mit `printf("Hello World\n")` wird der in “ ” stehende Text ausgegeben.

➤ “\n” ist eine sog. Escape-Sequenz und bedeutet „Zeilenumbruch“.

➤ Einfache Datentypen und formatierte Ergebnisausgabe

```
#include <stdio.h>
int main () /* Beginn des Hauptprogramms */
{
    int      i = 10;
    double   db = 1.23;

    printf("Zahl=%d",i);    /* integer Zahl wird ausgegeben */
    return 0;
}
```

Einfache Datentypen und formatierte Ergebnisausgabe

```
#include <stdio.h>
int main () /* Beginn des Hauptprogramms */
{
    int      i = 10;
    double   db = 12345.23;

    printf("i=>>%3d<<", i);
    printf("\n");
    printf("db=>>%10.21f<<", db);
    return 0;
}
```

Ausgabe:

i=>> 10<<

db=>> 12345.23<<

➤ **Formatierte Ergebnisausgabe**

- Ergebnisse können durch Formatbeschreiber (%d, %10.21f) im Formatbeschreiberstring formatiert ausgegeben werden.
- %3d bedeutet, eine Integerzahl wird in einem Feld von 3 Zeichen ausgegeben.
- %10.21f bedeutet, eine double-Zahl wird in einem Feld von 10 Zeichen, mit 2 Nachkommastellen ausgegeben.

Weitere Elemente des Beispielprogramms

➤ **Zeilenende:** ;

- Alle Kommandos werden mit einem ";" abgeschlossen.

➤ **Kommentare:** /* */

- Kommentare sind in /* Kommentar ... */ eingeschlossen.
- Ab C99: // kennzeichnet den Rest einer Zeile als Kommentar

Variablendefinitionen

```
#include <stdio.h>
int main () /* Beginn des Hauptprogramms */
{
    int      i = 10;
    double   db = 1.23;

    printf("Zahl=%d", i);
    printf("\n");
    printf("%10.2lf", db);
    return 0;
}
```

- Alle verwendeten Variablen müssen definiert werden, d.h. Datentyp und Name der Variablen werden bekannt gemacht.
 - `double` definiert z.B. eine reelle Zahl.
 - `int` definiert einer ganze Zahl.

Achtung: Die Variablen können nicht beliebig große Werte annehmen (wird später genauer betrachtet.)

Tastatureingabe

```
#include <stdio.h>
int main ()
{
    int i;

    printf("Bitte geben Sie eine Zahl ein : ");
    scanf("%d",&i);      /*Wartet auf Eingabe*/
    printf("Die Eingabe ist %d\n",i);
    return 0;
}
```

➤ Tastatureingabe:

- Mit `scanf()` können Texte von der Tastatur eingelesen werden. Der Formatbeschreiber (hier `"%d"`) gibt den Typ des einzulesenden Werts an.
- Der Variablen muss der Adressoperator `&` vorangestellt werden.

Übungsaufgabe

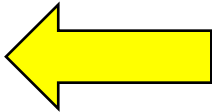
Schreiben Sie ein C Programm, das zwei int Zahlen einliest und wieder ausgibt.

```
#include <stdio.h>

int main () /* Beginn des Hauptprogramms */
{
    int    zahl1, zahl2;
    printf("Geben Sie die erste Zahl jetzt ein : ");  fflush(stdout);
    scanf("%d", &zahl1);
    printf("\n");
    printf("Geben Sie die zweite Zahl jetzt ein : ");  fflush(stdout);
    scanf("%d", &zahl2);
    printf("\n");
    printf("zahl1 = %d\n zahl2 = %d\n ", zahl1, zahl2);
    return 0;
}
```

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C 
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Die sechs Wortklassen von C

1. Bezeichner (identifizier)

2. Reservierte Worte (Schlüsselwörter, z.B. `if`)

3. Konstanten

- Ganzzahlkonstanten (z.B. `512`, `-33`)
- Gleitkommakonstanten (z.B. `3.14`)
- Zeichenkonstanten (z.B.: `'d'`)

4. Strings (z.B.: `"Hello World\n"`)


5. Operatoren (z.B. `+`)

6. WhiteSpaces (Trennzeichen) (Leerzeichen, Zeilenumbruch, Kommentare)

Bezeichner

Bezeichner (identifier) benennen auf eindeutige Weise

- Variablennamen,
- Konstantenbezeichner,
- Typbezeichner und
- Funktionsnamen



**Beachten Sie
den C-
Coding-Style**

Syntax eines identifiers:

- Ein Bezeichner wird aus den Zeichen `_`, `a`, `b`, ..., `z`, `A`, `B`, ... `Z`, `0`, `1`, ... `9`, zusammengesetzt.
- Das erste Zeichen eines Bezeichners darf keine Zahl sein.
- C ist **case-sensitiv**, d.h. C unterscheidet Groß- und Kleinbuchstaben.
- Bezeichner dürfen **beliebig lang sein, wobei mindestens die ersten 31 Zeichen signifikant** sind.
- Schlüsselwörter dürfen nicht als Bezeichner verwendet werden.

Reservierte Wörter

Die Menge der **reservierten Wörter (Schlüsselwörter)** von C:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

Darüber hinaus kann es Compiler-spezifische reservierte Wörter geben.

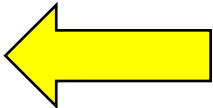
ÜBUNG Bezeichner

Welche der folgenden Bezeichner sind nicht erlaubt?

- Mitgliedsnummer
- 4U
- Auto
- Read_Me
- double
- Laurel&Hardy
- Ergänzung
- a
- ist_Null_wenn_die_Linie_laenger_als_MAXLEN_ist

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante 
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Konstanten

Beispiele für Ganzzahlkonstanten (Datentyp int)

	Beispiele			
Dezimalkonstante	0	12	+7	-999
Hexadezimalkonstante	0x12	0x00	-0xffff	0xffff
Oktalkonstante	06	-0	+012	

Hex Konstanten:

- Die Konstante gibt das Bitmuster an, dass in die Speicherzelle geschrieben wird.
- Typ und Breite der Variablen geben an, wie das Bitmuster interpretiert wird.
- Der Unäre Operator – wird als Multiplikation mit -1 übersetzt.

	uint32_t	int32_t	uint16_t	int16_t
0xffff	65535	65535	65535	-1
-0xffff	4294901761	-65535	1*	1

- : Compiler erzeugt in der Regel eine Warnung

Achtung: HW nahe Darstellung

Ganzzahlkonstanten (Fortsetzung)

Datentyp	Größe (s.u.)	Wertebereich	Wertebereich ber.
long long int (ist signed) unsigned long long int	meist 8 Byte	$-2^{63} \dots 2^{63}-1$ $0 \dots 2^{64}-1$	
long int (ist signed) unsigned long int	meist 4 Byte	$-2^{31} \dots 2^{31}-1$ $0 \dots 2^{32}-1$	$-2147483648 \dots 2147483647$ $0 \dots 4294967295$
int (ist signed) unsigned int	meist 4 Byte	$-2^{31} \dots 2^{31}-1$ $0 \dots 2^{32}-1$	$-2147483648 \dots 2147483647$ $0 \dots 4294967295$
short int (ist signed) unsigned short int	meist 2 Byte	$-2^{15} \dots 2^{15}-1$ $0 \dots 2^{16}-1$	$-32768 \dots 32767$ $0 \dots 65535$
char (=signed char) unsigned char	1 Byte	$-2^7 \dots 2^7-1$ $0 \dots 2^8-1$	$-128 \dots 127$ $0 \dots 255$

- Größe von int ist maschinenabhängig!
ANSI: Größe int-Variable mindestens 16 Bit.
 - auf 16-bit-Maschinen (z.B. 68000) ist int i.allg. 16 Bit groß,
 - auf 32-bit-Maschinen (z.B. Athlon, PowerPC) ist int i.allg. 32 Bit groß.
 - auf 64-bit-Maschinen (z.B. Intel Core i7) ist int i.allg. 32 Bit groß.
- Hintergrund: C soll als Implementierungssprache auf den verschiedensten Rechnersystemen (Micro-Controller, Großrechner) einsetzbar sein.
- Anm.: Die maschinenspezifischen Größen stehen in "limits.h"

ARM-Keil Compiler verwendet diese Größen

Typen aus <stdint.h>

- Maschinenunabhängige Definition von ganzzahligen Typen

Datentyp	Größe	Wertebereich
int8_t uint8_t	1 Byte	-2 ⁷ 2 ⁷ -1 0 2 ⁸ -1
int16_t uint16_t	2 Byte	-2 ¹⁵ 2 ¹⁵ -1 0 2 ¹⁶ -1
int32_t uint32_t	4 Byte	-2 ³¹ 2 ³¹ -1 0 2 ³² -1
int64_t uint64_t	8 Byte	-2 ⁶³ 2 ⁶³ -1 0 2 ⁶⁴ -1

- Beispiel

```
#include <stdint.h>
#include <stdio.h>
int main () {
    int16_t i;
    i = 0xffff;
    printf("i = %d\n", i);
}
```

Typen von Ganzzahlkonstanten

Suffixe definieren den Typ von Ganzzahlkonstante

- Unsigned-suffix: character `u` or `U`
- Long-suffix: character `l` or `L`
- Long-long-suffix: character sequence `ll` or `LL`

- Beispiele

<code>0xFFFFFFFF</code>	<code>4294967295</code>	<code>4294967295U</code>
<code>0xFFFFFFFFL</code>	<code>4294967295L</code>	<code>4294967295UL</code>
<code>0xFFFFFFFFLL</code>	<code>4294967295LL</code>	<code>4294967295ULL</code>

Bestimmung des Typs einer Ganzzahlkonstante

- Die Syntax einer Konstante definiert deren Typ nicht immer eindeutig.
Beispiel: Welchen Typ hat die Konstante 23 ?
- Typ der Konstante ist der erste Typ der folgenden Listen, in den der Wert passt.

Types allowed for integer literals		
suffix	decimal bases	hexadecimal or octal bases
no suffix	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l or L	long int (until C++11) unsigned long int (until C++11) long int (since C++11) long long int (since C++11)	long int unsigned long int long long int unsigned long long int
both l/L and u/U	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll or LL	long long int	unsigned long int unsigned long long int
both ll/LL and u/U	unsigned long long int	unsigned long long int

Quelle: http://en.cppreference.com/w/cpp/language/integer_literal

Gleitkommakonstanten

Datentyp	Größe	Wertebereich (Betrag)
float	meist 4 Byte	$3.4 \cdot 10^{-38}$ $3.4 \cdot 10^{38}$
double	meist 8 Byte	$1.7 \cdot 10^{-308}$ $1.7 \cdot 10^{308}$
long double	meist 10 Byte	$3.4 \cdot 10^{-4932}$ $3.4 \cdot 10^{4932}$

Beispiele

78.4	-15.125	5.	-0.784e2	-0.10476E+4	3e5
-------------	----------------	-----------	-----------------	--------------------	------------

Gleitkommakonstanten (Fortsetzung)

Anmerkungen

- Die gültigen Größen stehen in `float.h`.
- `math.h` enthält die Konstanten `NAN` (not a Number) und `INFINITY` (für unendlich)
- Liefert eine Berechnung einen Overflow, so ist das Ergebnis `INFINITY`.
- Viele Maschinen besitzen eine 64 Bit Floatingpoint-Unit, so dass der Datentyp **`float`** keine Rechenzeitvorteile bringt (aber Speicherplatzvorteile).
- **`long double`** ist vor allem dann zweckmäßig, wenn eine hohe numerische Genauigkeit erforderlich ist.

Zeichenkonstanten

Zeichenkonstanten (Datentyp ***char***) = einzelne in einfache Anführungsstriche eingeschlossene Zeichen

Beispiele: 'a', 'R', '1', '8'

Der Wert der Konstante ist der numerische Wert des Zeichens im Zeichensatz der jeweiligen Maschine (i. allg. ASCII)

Darüber hinaus sind sog. Escape Sequenzen erlaubt, z.B. (s. ASCII-Tabelle):

\n	Zeilenumbruch (CR)	\r	Wagenrücklauf (LF)
\f	Seitenwechsel (FF)	\b	Backspace (BS)
\0	Nullzeichen (NUL)		

oder mit beliebigen Bitpattern:

\015	Oktalzahl,	ASCII CR
\x0a	Hexadezimalzahl,	ASCII LF

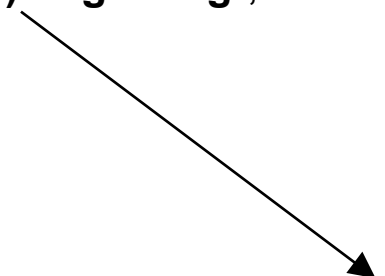
Strings

String (Zeichenkette = Folge von Zeichen umgeben von Anführungszeichen ("...") .

Intern wird **der Zeichenkette ein 0-Zeichen (\0) angehängt**, wodurch das Ende der Zeichenkette markiert wird.

Beispiel: "ABC 123 \n"

Im Speicher steht dann:



0x41	0x42	0x43	0x20	0x31	0x32	0x33	0x20	0x0D	0x00
------	------	------	------	------	------	------	------	------	------

ASCII-Merkregel:

Zahlen beginnen bei 0x30.

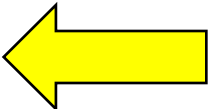
Großbuchstaben beginnen bei 0x41.

Kleinbuchstaben beginnen bei 0x61.

Leerzeichen ist 0x20.

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable 
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Variablen und Vereinbarungen

Variablen = Datenobjekte, deren Wert im Programmverlauf geändert werden kann.

Variablen müssen vor ihrer ersten Benutzung deklariert werden.

Datenobjekte werden nach folgender Syntax vereinbart.

Syntax:

declaration ::= type-spezifizier init-declarator { ", " init-declarator } ", " .

type-spezifizier ::= "int" | "float" | "double" | **Anm.: unvollständig !**


init-declarator ::= variablen-identifizier ["=" initialwert] .

Beispiele:

```
int      zaehler, increment = 5 ;
float    pi = 3.1415926 ;
```

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke 
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Ausgewählte binäre Operatoren

+	Addition	Arithmetik	Zahlen (u. teilw. Zeiger)
-	Subtraktion		"
*	Multiplikation		"
/	Division		"
%	Modulo-Division (Rest)		ganze Zahlen
<	kleiner	Vergleich	alle Typen
<=	kleiner gleich		"
==	gleich		"
!=	nicht gleich		"
>=	größer gleich		"
>	größer		"
-	Vorzeichen (unär)	Arithmetik	Zahlen
+	Vorzeichen (unär)		"
&&	log. AND	Logik	boolesche Werte
	log. OR		"

Overloading: Typ der Argumente und Operator definieren die Funktion, die ausgeführt wird.

Beispiel:

- Der Operator / wird auf zwei `int` Variablen/Konstanten angewendet: ganzzahlige Division
- Der Operator / wird auf zwei `double` Variablen/Konstanten angewendet: Division von Gleitkommazahlen

Vorrangstufen und Auswertungsreihenfolgen von Operatoren

- Bei der Auswertung eines Ausdrucks wird der Vorrang der Operatoren beachtet.
- Aus Java bekannt
- Aus der Mathematik bekannt (Punkt vor Strichrechnung)

Vorrangstufen numerischer Operatoren

<i>Stufe</i>	<i>Operanden</i>	<i>Erläuterung</i>	<i>Auswertung</i>
5	()	Klammern	von links
4	+ -	unären Operatoren	von rechts
3	* % /		von links
2	+ -	binären Operatoren	von links
1	=	Zuweisung	von rechts

Anmerkung zur Auswertungsreihenfolge binärer Operatoren

Stehen mehrere (bezüglich der Vorrangstufe) gleichwertige Operationen hintereinander, werden die Reihenfolgeregeln angewendet.

Beispiel: Die ganzzahlige Berechnung von "3 * 11 / 4" ergibt:

a) von links ausgewertet $(3 * 11) / 4 = 33 / 4 = 8$! (liefert C)

b) von rechts ausgewertet $3 * (11 / 4) = 3 * 2 = 6$!

Das Assoziativgesetz gilt nicht. (Warum ?)

Regel:

Verwendung von Klammern wenn die Auswertungsreihenfolge nicht direkt ersichtlich ist.*

*1 goldene Regel des C-Coding-Styles: Keep it simple.

ÜBUNG: Arithmetische Operatoren

Gegeben seien:

```
int    j=9, k=-15, m, n;
```

Berechnen Sie

```
m = 9/2;
```

```
n = k%4;
```

```
j = j+1;
```


Implizite Typkonvertierungen

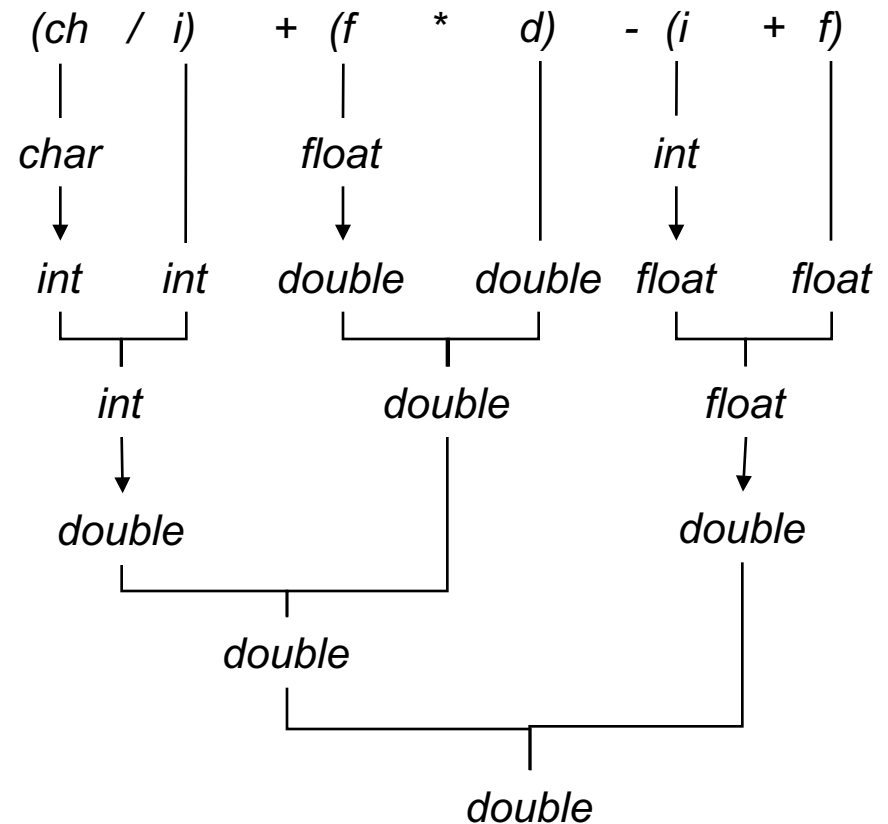
Werden in math. Ausdrücken verschiedene Typen kombiniert, wird bei jeder Operation der "niedrigere" Typ in den höheren konvertiert.

Beispiel:

```
char    ch;  
int     i;  
float   f;  
double  d;
```

Achtung:

- Keine implizite Typumwandlung bei
 - Zuweisungen
 - den logischen Operatoren && und ||



Übung zur impliziten Typkonvertierung

Was gibt das nachfolgende Programm aus?

```
#include <stdio.h>
int main() {
```

```
    char s1 = -1;
```

```
    char s2 = 200; // s2 = -56
```

```
    unsigned char u1 = -1; // u1 = 255
```

```
    unsigned char u2 = 200;
```

```
    int i;
```

```
    unsigned int j;
```

```
    i = s1;          printf( "%d\n", i);    // -1
```

```
    i = s2;          printf( "%d\n", i);    // -56
```

```
    i = s1 + s2;     printf( "%d\n", i);    // -57
```

```
    i = s1 + 255;    printf( "%d\n", i);    // 254
```

```
    i = u1;          printf( "%d\n", i);    // 255
```

```
    i = s1 + u1;     printf( "%d\n", i);    // 254
```

```
    i = u1 + u2;     printf( "%d\n", i);    // 455
```

```
    j = s1 + s2;     printf( "%u\n", j);    // 4294967239
```

```
    j = u1 + u2;     printf( "%u\n", j);    // 455
```

```
}
```

Code widerspricht der goldenen Regel „Don't be clever“.

i.d.R. keine Warnung
des Compilers

Diesen Code sollen sie verstehen aber i.d.R. nicht so schreiben.

Explizite Typkonvertierungen (= casting)

Gelegentlich werden auch explizite Typvereinbarungen benötigt. Hierzu wird vor den zu konvertierenden Ausdruck der Zieltyp in Klammern gesetzt.

Beispiel:

```
int    a=7;
int    b=2;
double Erg;
...
Erg = (double)a/(double)b;    /* Erg = 3.5 */
Erg = a/b;                    /* Erg = 3.0 */
```

Übung zur expliziten Typkonvertierung

Gegeben seien:

```
char    c1='5', c2=25, c3, c4=-1;
int     i=5, j=9, k=-15, m, n, p, q;
double  d1=12.5, d2=2.0E-3, d3=-100, d4, d5, d6;
```

Stil: – Verstoß gegen die goldene Regel: „Be explicit – write your program for people.“.

Berechnen Sie

m = d1*i;	// 62	Stil: -
d4 = 9/2;	// 4.000000	Stil: -
n = k%4;	// -3	Stil: o.k.
j = j+1;	// 10	Stil: o.k.
p = d2*750 + 0.1;	// 1	Stil: -
d5 = (double)j/i + 0.2;	// 2.200000	Stil: --
q = (unsigned char)c4;	// 255	Stil: bei Bedarf o.k.
d6 = 22%5*3%2;	// 0.000000	Stil: --
c3 = (c1-0x30)+c2;	// 30	Stil: -
c3 = 64*8;	// 0	Stil: --
c3 = 64*4 - 1;	// -1	Stil: --

Stil: – Diesen Code sollen Sie verstehen aber i.d.R. nicht so schreiben.

Welche Ausdrücke zeugen von schlechtem Stil ?

Binäre Operatoren: Vergleichsoperatoren

Vergleichsoperatoren liefern ebenfalls ein **numerisches Ergebnis**:

- Vergleich **falsch (FALSE)** : 0
- Vergleich **richtig (TRUE)** : 1

Bei der Auswertung ist auf den Vorrang zu achten.

Vorrangstufen der Vergleichsoperatoren

<i>Stufe</i>	<i>Operanden</i>	<i>Erläuterung</i>	<i>Auswertung</i>
7	()		von links
6	+ -	unären Operatoren	von rechts
5	* % /		von links
4	+ -	binären Operatoren	von links
3	< <= >= >		von links
2	== !=		von links
1	=	Zuweisung	von rechts

Beispiele

3 < 5 - 4

liefert den Wert 0 **(besser: Klammern verwenden)**

(3 < 5) - 4

liefert den Wert - 3 **(Unsinn, Stil: --)**

3 < 5 < 0 < 2

liefert den Wert 1 **(Unsinn, Stil: --)**

Binäre Operatoren: Logische Operatoren

Logische Operatoren (&&, ||, !) liefern ebenfalls ein numerisches Ergebnis:

- Aussage **falsch (FALSE): 0**
- Aussage **richtig (TRUE) : 1**

Logische Operatoren und Vergleichsoperatoren werden oft zusammen eingesetzt.

Bei der Auswertung ist auf den Vorrang zu achten.

Vorrangstufen der log. Operatoren und Vergleichsoperatoren

<u>Stufe</u>	<u>Operanden</u>	<u>Erläuterung</u>	<u>Auswertung</u>
7	!	Negation	
6	< <= >= >		von links
5	==, !=		von links
4	&&	log. Operatoren (AND)	von links
3		" " (OR)	von links
1	=	Zuweisung	von rechts

In C wird jeder Ausdruck ungleich 0 logisch interpretiert als wahr (TRUE) betrachtet.

ÜBUNG Vergleichsoperatoren & logische Operatoren

Was ergeben folgende Ausdrücke:

```
char  b;
```

```
int    i = 7, j = 25;
```

```
b = !(i > 10) && !(i < 5);
```

```
b = (i <= 10) && (i >= 5);
```

```
b = i <= 10 && i >= 5;
```

```
b = (i - 10) && (i - 7);
```

```
b = 5 <= j <= 10;
```

```
b = 5 <= j && j <= 10;
```

```
b = !(i < 10) || i==7 || !(i-7);
```

```
b = !(i=7);
```

Stil: – Verstoß gegen die goldene Regel: „Be explicit – write your program for people.“.

```
// b = 1 Stil: –
```

```
// b = 1 Stil: o.k.
```

```
// b = 1 Stil: noch o.k.
```

```
// b = 0 Stil: --, Unsinn
```

```
// b = 1 Stil: --, Unsinn
```

```
// b = 0 Stil: noch o.k.
```

```
// b = 1 Stil: --
```

```
// b = 0 Stil: --,
```

```
// Achtung :
```

```
= ist Zuweisungsoperator
```

Welche Ausdrücke zeugen von schlechtem Stil? Welche sind unsinnig?

Unäre Operatoren: Übersicht

&	Adresse von ...	Referenzierung	alle Typen
*	Inhalt von ...	Dereferenzierung	Zeiger
+	pos. Vorzeichen	Arithmetik	Zahlen
-	neg. Vorzeichen	"	"
~	bitweise invertieren	Bitoperation	ganze Zahlen
!	log. invertieren	Logik	boolesche Werte
(Zieltyp)	Typumwandlung		
sizeof	Speicherbedarf		Ausdrücke u. Typen
++	Inkrementierung	Prä- und Post-Inkrement/-Dekrement	ganze Zahlen und Zeiger
--	Dekrementierung		

Inkrement/Dekrement Operator

- "+" und "--" sind unäre Operatoren zur Inkrementierung (hoch zählen) und Dekrementierung (runter zählen) von ganzzahligen Variablen.
- Sind auf Java bekannt
- Die Position von "+" und "--" in der Vorrangtabelle ist noch vor den arithmetischen Operatoren "*" und "/"

Beispiel:

```
int i=0, k=0, m=0, n=0;
printf("%2d %2d %2d %2d \n" ,++i, k++, --m, n--);
/*  druckt:  1  0 -1  0  */
printf("%2d %2d %2d %2d \n" ,i, k, m, n);
/*  druckt:  1  1 -1 -1  */
```

Bedingte Ausdrücke

- `if_expr` aus Java bekannt
- Die Priorität des `if_expr` Operators in der Vorrangtabelle ist sehr niedrig - kurz vor der Zuweisung.

Beispiel:

```
// Berechnung des Maximums
```

```
max = (x > y) ? x : y;
```

Spezielle Zuweisungen

- Die speziellen Zuweisungsoperatoren `+=` , `-=` , `*=` , `/=` und `%=` sind aus Java bekannt.
- In der Vorrangtabelle haben diese Operatoren die selbe Priorität wie der Zuweisungsoperator (sehr niedrig).

Var += 5 statt Var = Var + 5

Var -= 5 statt Var = Var - 5

Var *= 5 statt Var = Var * 5

Var /= 5 statt Var = Var / 5

Var %= 5 statt Var = Var % 5

ÜBUNG: Spezielle Operatoren

Was wird ausgedruckt? (C-Puzzle, kein ernsthafter Code)

```
int x=1, y=1, z=1;
```

```
x+=++y;
```

```
printf("x:%d    y:%d\n", x, y);
```

Ausgabe: **x:3 y:2**

```
printf("z:%d\n", z+=x<y?y++:x++);
```

```
printf("x:%d    y:%d    z:%d \n", x, y, z)
```

Ausgabe: **z:4**
 x:4 y:2 z:4

```
x=z=1;
```

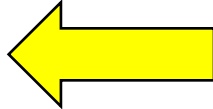
```
while(++z<4){
```

```
    printf("%d %d\n", x++, x++);
```

Ausgabe: **2 1**
 4 3

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen 
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Anweisungen

➤ Sequenzen

- Verbund-Anweisung
- Ausdrücke als Anweisungen
- Funktionsaufrufe

➤ Verzweigungen

- if-else - Anweisung
- switch – Anweisung

Beherrschen Sie diese Anweisungen aus anderen Programmiersprachen?

➤ Schleifen

- while - Schleife
- do-while - Schleife
- for - Schleife

Verbund-Anweisung

- Die Verbundanweisung (compound-Statement) – auch Block genannt – ist eine mit '{' und '}' geklammerte Folge von Vereinbarungen und Statements.
- Der Verbund definiert einen **Namensraum/Gültigkeitsbereich** für die in ihm vereinbarten Objekte

Beispiel:

```
int i = 2, j;  
{  
    int i;  
    i = 4;  
    j = 3 * i;  
}  
j = j + i;
```

- Es wird eine “neue” Variable i angelegt.
- Auf die Variable i der Umgebung kann innerhalb der Verbundanweisung nicht zugegriffen werden.
- Die neue Variable i ist außerhalb der Verbundanweisung nicht sichtbar.

Welchen Wert hat j an dieser Stelle ?

j = 14

C-Coding-Style:

- Verstoß gegen die goldene Regel: „Be explicit – write your program for people.“.
- Lokale Variablen sind gut, aber nicht gleichnamig zu Variablen aus umgebenden Blöcken.

if-else Anweisung

Syntax:

`if_statement ::= "if" "(" expression ")" statement ["else" statement] .`

Beispiele:

```
if ( a < b ) {  
    b = a;  
    c = d;  
}
```

```
if ( a < b ) {  
    b = a;  
}
```

```
if ( a < b ) {  
    b = a;  
} else {  
    a = b;  
}
```

C-Coding-Style:

- Verwenden Sie für den if- und den else-Block stets Verbundanweisungen – auch wenn der Block nur aus einem Statement besteht.

Geschachtelte if-else-Anweisungen und optionaler else Zweig

Beispiel

```
if (A) if (B) statement1; else statement2;
```

Frage: Zu welcher if Anweisung gehört der else Zweig?

Regel in C: Ein else gehört immer zum (rückwärtsgehend) letzten if, welches noch kein else hat.

C-Coding-Style: Bei Verwendung von Verbundanweisungen tritt diese Frage gar nicht auf.

```
if (A) {  
    if (B)  
        statement1;  
    else  
        statement2;  
}
```

else-if Ketten

Beispiel:

```
if (a<0) {  
    V=-1;  
    printf("negative! \n");  
}  
else if (a==0) {  
    V=0;  
    printf("zero! \n");  
}  
else {  
    V=1;  
    printf("positive! \n");  
}
```

C-Coding-Style:

- Nur in diesem Fall ist es üblich, dass die geschachtelten if Anweisungen nicht in Verbundanweisungen eingebettet werden.

Übungen: if-else-Anweisung

Welcher Wert wird ausgegeben?

Achtung: Schlechter Code
- widerspricht dem
C-Coding-Style

```
int i=0, a=0, b=2;  
if (i==0)  
    a=7;  
else  
    b=15;  
    a=b+1;  
printf("%d", a);
```

Ausgabe: 3

Die Anweisung `a=b+1;` ist eingerückt, aber gehört nicht zum ELSE Teil.

C-Coding-Style:

➤ Mit Verbundanweisungen ist der Code an diese Stelle besser lesbar.

Code gemäß C-Coding-Style:

```
int i=0, a=0, b=2;  
if (i==0) {  
    a=7;  
} else {  
    b=15;  
}  
a=b+1;  
printf("%d", a);
```

Übungen: if-else-Anweisung (Fortsetzung)

Welcher Wert wird ausgegeben?

```
int i=0,b=10,a;  
  
if (i=0) {  
    a=10;  
} else if (i=2) {  
    b=15;  
    a=b+1;  
} else {  
    a=0;  
}  
  
printf("%d",a);
```

Ausgabe: 16

Gern gemachter C Fehler:

Der Vergleichoperator ist == und nicht =

Hier wird i der Wert 0 zugewiesen. Das Ergebnis dieser Operation ist der zugewiesene Wert, also 0 in diesem Fall. 0 wird als FALSE interpretiert und somit wird der ELSE Fall durchlaufen.

Hier wird i der Wert 2 zugewiesen. Das Ergebnis dieser Operation ist der zugewiesene Wert, also 2 in diesem Fall. 2 wird als TRUE interpretiert und somit wird der THEN Fall durchlaufen.

C-Coding-Style:

- In C wird diese manchmal als Kurzschreibweise bei der Fehlerbehandlung verwendet.
- Wenn überhaupt, verwenden Sie diese Kurzschreibweise nur für diese Art der Fehlerbehandlung.

Übungen: if-else-Anweisung (Fortsetzung)

Welcher Wert wird ausgegeben? (**Achtung:** C-Puzzles, kein ernsthafter Code)

```
int i=1, a=5, b=10;  
if (i==0);  
{  
    a=10;  
    b=a+1;  
}  
printf("%d", a);
```

; ist auch ein Anweisung – nämlich die „leere“ Anweisung. Somit wird im Fall `i==0` die leere Anweisung ausgeführt. Anschließend wird mit der darauf folgenden Anweisung fortgefahren, in diesem Fall die Verbundanweisung.

Gern gemachter Schreibfehler in C.

Ausgabe: 10

switch-Anweisung

Diese Kontrollstruktur ist als Mehrfachverzweigung oder Verteiler gedacht.

Beispiel:

```
switch ( c ){
    case 'a' :
    case 'A' :
        alpha = 'a';
        break ;
    ....
    case 'z' :
    case 'Z' :
        alpha = 'z';
        break ;
    default : alpha = '0';
}
```

Achtung: Das `break` ist notwendig, damit nicht die darauf folgenden Fälle abgearbeitet werden (s.u.)!

Übungen: switch-Anweisung

Was wird ausgegeben?

```
int in=2;
switch(in) {
    case 1:
    case 2:
    case 4:
    case 6:
        printf("A");
    case 9:
        printf("u");
        printf("t");
    case 10:
        printf("o");
        break;
    case 11:
        printf("ma");
    default:
        printf("t\n");
}
```

Ausgabe: Auto

Übungen: switch-Anweisung

Geben Sie zu folgendem Programmstück eine äquivalente switch-Anweisung an:

```
if ((n<=8) && (n>=5))
    printf("schlecht");
else if ((n==3) || (n==4))
    printf("mittel");
else if ((n<=2) && (n>0))
    printf("gut");
else
    printf("unmoeglich");
```

```
switch (n) {
    case 1:
    case 2:
        printf("gut");
        break;
    case 3:
    case 4:
        printf("mittel");
        break;
    case 5:
    case 6:
    case 7:
    case 8:
        printf("schlecht");
        break;
    default:
        printf("unmoeglich");
}
```


while-Schleife (kopfgesteuerte Schleife)

Aus Java bekannt.

Beispiel:

```
while( a < b ){  
    a = a + 2;  
}
```

C-Coding-Style:

- Verwenden Sie für den Rumpf einer while-Schleife stets eine Verbundanweisung – auch wenn er nur aus einem Statement besteht.

do-while-Schleife (fußgesteuerte Schleife)

Beispiel:

```
do{  
    summe = summe + 1;  
    i = i + 1;  
} while (i <= 100);
```

Anmerkung:

- Der Schleifenrumpf wird mindestens einmal ausgeführt.
- Die Schleifenrumpf wird solange ausgeführt wie die Bedingung (der Prüfausdruck) hinter `while` wahr ist (`!= 0`).
- Dies ist ein Unterschied zum Nassi-Shneiderman Diagrammen und `repeat until` Schleifen.

for-Schleife

Aus Java bekannt.

Beispiel:

```
sum = 0;
for (i=1; i<=100; i++){
    sum = sum + i;
}
```

Initialisierung

Bedingung

Inkrementierung/
Dekrementierung

Äquivalente while Schleife:

```
sum = 0;
i = 1;
while (i <= 100){
    sum = sum + i;
    i++;
}
```

Übung: for-Schleife (Fortsetzung)

Schreiben Sie ein Programm, welches alle Zahlen des "2-aus-5-Code" tabellarisch ausgibt, also:

```
00011
00101
00110
01001
01010
01100
10001
10010
10100
11000
```

Beobachtungen:

- Die Zahlen sind aufsteigend sortiert (von 0 bis 31)
- Eine Zahl wird ausgegeben, wenn ihre Binärdarstellung genau zwei 1 enthält.

Übung: for-Schleife (Fortsetzung)

```
#include <stdlib.h>
#include <stdio.h>
```

```
#define BIT_0_IS_1    0x01
#define BIT_5_IS_1    0x10
```

Schwer lesbarer Code für so ein kleines Problem.
Über Funktionen lässt sich der Code besser strukturieren (s.u.).

```
int main(int argc, char *argv[]) { // Programm 2 aus 5 Code
    unsigned int i ;
    // Durchlaufe die Zahlen 0 bis 31 in einer for Schleife
    for (i = 0; i < 32; i++) {
        // Test, ob die Binaerdarstellung von i genau zwei 1 enthaelt
        int count = 0;
        unsigned int x = i;
        while (0 != x) {
            if (BIT_0_IS_1 == (x & BIT_0_IS_1)) { count++; }
            x = x >> 1;
        }
        // count speichert die Anzahl von 1 in der Binaerdarstellung von i
        if (2 == count) {
            // Ausgabe von i
            int j = 0;
            x = i;
            for (j = 0; j < 5; j++) {
                if (BIT_5_IS_1 == (x & BIT_5_IS_1)) { printf("1"); } else { printf("0"); }
                x = x << 1;
            }
            printf("\n"); fflush(stdout);
        }
    } // end of for loop
    return 0;
}
```

break und continue Anweisung

break Anweisung

- Die **break** Anweisung beendet eine Schleife (while, do, for) vorzeitig. Geschachtelte Schleifen: Nur die innerste Schleife wird abgebrochen.
- Typischer Anwendungsfall ist der Abbruch einer Schleife, wenn eine besondere Bedingung vorliegt.
- I.allg. sollte eine strukturierte Lösung dem break vorgezogen werden. Gelegentlich kann der Code durch break auch übersichtlicher werden.
- Weiterhin beendet break eine **switch-Anweisung (s.o.)**. Hier ist break sinnvoll und üblich.

continue Anweisung

- dient in Schleifen (while, do, for) dazu, die nächste Wiederholung der umgebenden Schleife sofort zu beginnen.
for Schleife: expr3 wird noch ausgeführt.
- I.allg. sollte eine strukturierte Lösung dem continue vorgezogen werden. Gelegentlich kann der Code durch continue übersichtlicher werden.

break und continue Anweisung (Fortsetzung)

Beispiel:

```
/* drucke alle Zahlen von 0 ... 99, mit Ausnahme  
   der durch 5 teilbaren Zahlen.  
*/  
  
int i;  
for(i=0; i<100; i++){  
    if(i%5 == 0) continue;  
    printf("%2d \n",i);  
}
```

Frage:

Wie könnte man das Programm besser (ohne `continue`) schreiben?

Übung: Wahl der richtigen Kontrollstruktur

Vereinfachen Sie folgendes Programm

```
int done=i=0;

while(i<100 && !done) {
    if((x=x/2) > 1) {
        i++;
        continue;
    }
    done=1;
}
```

Einige Beobachtungen:

- Die Schleife wird beendet, wenn $i == 100$ ist oder $done != 0$ ist.
- In jedem Schleifendurchlauf wird x halbiert, solange $x > 1$ ist.
- Die Schleife terminiert, wenn $x \leq 1$ ist.
- x wird mindestens einmal halbiert.
- i wird nur erhöht, wenn $x/2 > 1$ ist.

Ein vereinfachtes Programm:

```
int i = 0;

while ( (i < 100) && ((x = x/2) > 1) ) {
    i ++;
}
```


Übung: Wahl der richtigen Kontrollstruktur (Fortsetzung)

Vereinfachen Sie folgendes Programm

```
int i, count = 0;
/* random(100) : Zufallszahl
   zwischen 0 und 100 */
while((i=random(100)) != 0) {
    if(i==50) break;
    if(i%3==0) continue;
    if(i%5==0) count++;
}
```

Einige Beobachtungen:

- Im Schleifenkopf wird der Variablen i eine Zufallszahl zwischen 0 und 100 zugewiesen. Ist diese 0, terminiert die Schleife.
- Für i == 50 terminiert die Schleife auch.
- Ist i != 50 und durch 5 teilbar und nicht durch 3 teilbar, wird count erhöht.

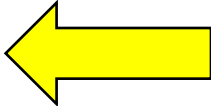
Ein vereinfachtes Programm:

```
int i, count = 0;

i = random(100);
while ((i != 0) && (i != 50)) {
    if ((i%3!=0) && (i%5==0))
        count++;
    i = random(100);
}
```

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1) 
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung

Funktionsdefinitionen

Funktionen sind die Träger von Algorithmen bzw. Berechnungen.
Funktionen werden nach folgender Syntax vereinbart.

Der Rumpf eine Funktion ist
eine Verbundanweisung

Syntax:

`func_def ::= func_head compound_statement .`

`func_head ::= type_spezifier func_identifizier "(" (formal_args | void) ")" .`

`formal_args ::= type_spezifier var_identifizier ("," type_spezifier var_identifizier)*`

`type_spezifier ::= void | int | double |` Anmerkung: unvollständig

Beispiele:

`void wenig (void) { ... }`

`int inkrementiere (int zahl, int incr) { ... }`

`void update (float druck, int temp, int wert) { ... }`

Funktionsdefinitionen (Fortsetzung)

Unterschied: Deklaration – Definition

Deklaration = Bekanntgabe von **Name** und **Typ** eines Objektes an den Compiler.

Funktionsdeklaration: Dem Compiler werden nur Name, Parameterliste und Ergebnistyp der Funktion „mitgeteilt“ – der Funktionskopf enthält genau diese Informationen.

Definition = Detailbeschreibung eines Objektes.

Funktionsdefinition: Funktionskopf und **Rumpf** der Funktion.

Beispiel

```
/* Variablendefinitionen */
int i, j, k;

/* Funktionsdeklaration */
int QuadSum (int x, int y);

int main(){
    ...
    k=QuadSum(m,n); /*Aufruf*/
    ...
}
```

```
/* Funktionsdefinition */
int QuadSum (int x, int y) {
    int res;
    res = x * x + y * y;
    return res;
}
```

return Anweisung

Die **return** Anweisung beendet die Ausführung einer Funktion und gibt, sofern für die Funktion ein Ergebnistyp vereinbart wurde, das berechnete Ergebnis an seine Aufrufumgebung zurück.

Syntax:

`return_statement ::= "return" [expression] .`

Der Wert des optionalen Ausdrucks liefert den Return-Wert der Funktion und muss zum Ergebnistyp der Funktion kompatibel sein.

Beispiel:

```
return ( a + b ) / 2 ;
```

Übung: Funktionen zur Strukturierung

Schreiben Sie ein Programm, welches alle Zahlen des "2-aus-5-Code" tabellarisch ausgibt, also:

```
00011
00101
00110
01001
01010
01100
10001
10010
10100
11000
```

Folie 120 (Übungsaufgabe zu for Schleifen) enthält eine Lösung für diese Aufgabe, jedoch ohne Funktionen.

Führen Sie in dieser Lösung Funktionen ein, so dass der Code übersichtlicher wird.

C-Coding-Style: Eine Funktion berechnet genau eine Aufgabe.

Übung: Funktionen zur Strukturierung (Fortsetzung)

```
#include <stdlib.h>
#include <stdio.h>

#define BIT_0_IS_1    0x01
#define BIT_5_IS_1    0x10

/**
*****
*  @brief      Diese Funktion testet, ob die Binaerdarstellung des
*              Arguments x genau zwei "1" enthaelt.
*
*  @param      x Das Argument des Tests
*
*  @return      true (1) genau dann wenn x zwei "1" enthaelt.
*****/
int twoBitsOne(unsigned int x) {
    int count = 0;
    while (0 != x) {
        if (BIT_0_IS_1 == (x & BIT_0_IS_1)) { count++; }
        x = x >> 1;
    }
    return 2 == count;
}
```

Übung: Funktionen zur Strukturierung (Fortsetzung)

```
/**
*****
*  @brief      Diese Funktion gibt die letzten 5 Bit der
*              Binaerdarstellung des Arguments x in einer Zeile aus.
*
*  @param      x Zahl, die gedruckt werden soll.
*
*  @return      void
*****/
void print5Bits(unsigned int x) {
    int i = 0;
    for (i = 0; i < 5; i++) {
        if (BIT_5_IS_1 == (x & BIT_5_IS_1)) {
            printf("1");
        } else {
            printf("0");
        }
        x = x << 1;
    }
    printf("\n");
    fflush(stdout);
}
```


Übung: Funktionen zur Strukturierung (Fortsetzung)

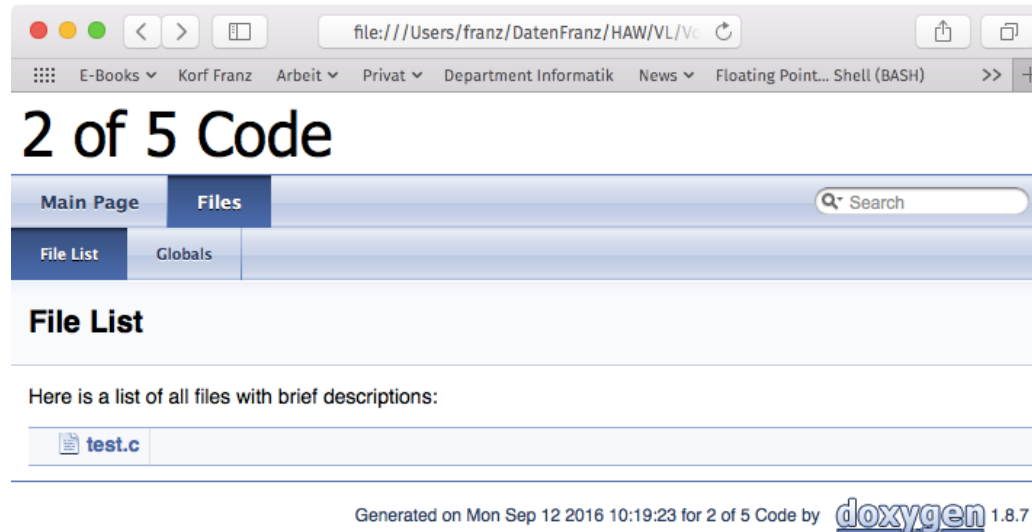
```
/**
*****
*  @brief      Ausgabe des 2 aus 5 Code auf stdout.
*
*****/
int main(int argc, char *argv[]) {
    unsigned int i ;
    // Durchlaufe die Zahlen 0 bis 31 in einer for Schleife
    for (i = 0; i < 32; i++) {
        if (twoBitsOne(i)) {
            // Die Binaerdarstellung von i enthaelt genau zwei "1"
            print5Bits(i);
        }
    }
}

// EOF
```

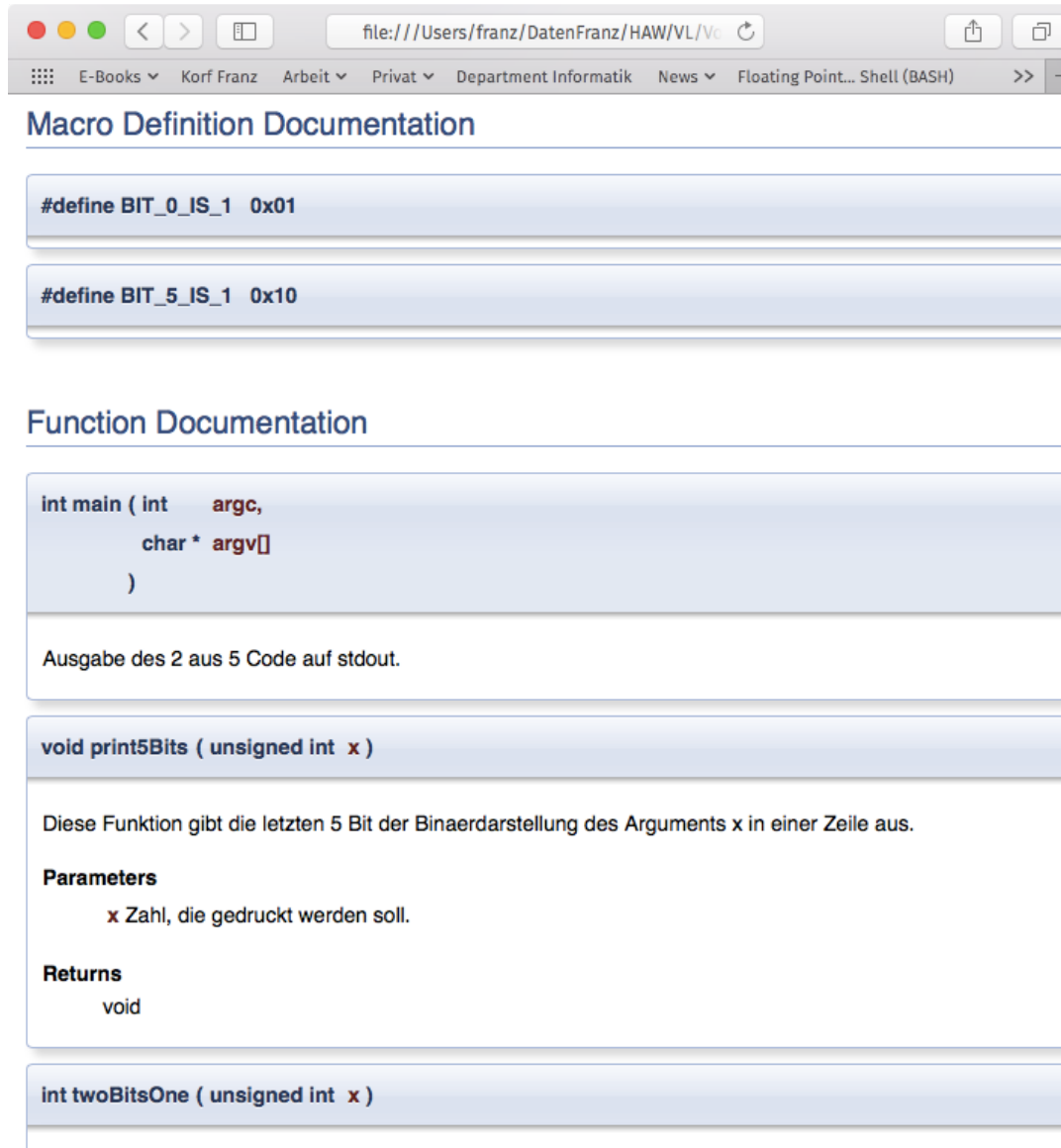
Dokumentation mit doxygen erzeugen

- Analog zu den Java Tools zieht doxygen (openSRC) eine Dokumentation aus dem src Code.
- Unterstützt mehrere Sprachen, viele Funktionen
- Wenn Sie die Funktionen wie im Beispiel dokumentieren, kann schnell eine einfache Dokumentation erzeugt werden.
- Vorgehensweise
 - `doxygen -g myDoxygenConfig` erzeugt Standardkonfiguration
 - Ändern Sie folgende Parameter der Konfiguration
 - `PROJECT_NAME` = "Name des Projekts"
 - `OPTIMIZE_OUTPUT_FOR_C` = YES
 - `EXTRACT_ALL` = YES
 - `EXTRACT_PRIVATE` = YES
 - `EXTRACT_STATIC` = YES
 - `INPUT` = Liste der C Dateien
 - `doxygen myDoxygenConfig` erzeugt eine einfache html und latex Dokumentation

Auszüge aus der html Dokumentation (index.html)



Auszüge aus der html Dokumentation (index.html)



The screenshot shows a web browser window with a file path in the address bar: `file:///Users/franz/DatenFranz/HAW/VL/Vc`. The browser's menu bar includes 'E-Books', 'Korf Franz', 'Arbeit', 'Privat', 'Department Informatik', 'News', 'Floating Point...', and 'Shell (BASH)'. The page content is divided into two main sections: 'Macro Definition Documentation' and 'Function Documentation'.

Macro Definition Documentation

```
#define BIT_0_IS_1 0x01
```

```
#define BIT_5_IS_1 0x10
```

Function Documentation

```
int main ( int    argc,  
           char * argv[]  
)
```

Ausgabe des 2 aus 5 Code auf stdout.

```
void print5Bits ( unsigned int x )
```

Diese Funktion gibt die letzten 5 Bit der Binaerdarstellung des Arguments x in einer Zeile aus.

Parameters

`x` Zahl, die gedruckt werden soll.

Returns

void

```
int twoBitsOne ( unsigned int x )
```

Übungsaufgabe zur Nachbereitung

Schreiben Sie eine mit C Programm zur Berechnung von

$$S = x - \frac{1}{3} \cdot \frac{x^3}{1} + \frac{1}{5} \cdot \frac{x^5}{1 \cdot 2} - \frac{1}{7} \cdot \frac{x^7}{1 \cdot 2 \cdot 3} + \dots$$

Es soll solange summiert werden, bis der relative Zuwachs, d.h.

$$\left| \frac{S_{neu} - S_{alt}}{S_{neu}} \right|$$

kleiner als 10^{-10} ist.

- Strukturieren Sie das Programm durch Funktionen.
- Verwenden Sie doxygen zur Dokumentation.
- Diskutieren Sie die Wahl der Datentypen.

Übungsaufgabe zur Nachbereitung (Lösung)

Umwandlung der Darstellung der Reihe

$$S = x - \frac{1}{3} \cdot \frac{x^3}{1} + \frac{1}{5} \cdot \frac{x^5}{1 \cdot 2} - \frac{1}{7} \cdot \frac{x^7}{1 \cdot 2 \cdot 3} + \dots$$

$$\begin{aligned} S = x &+ (-1)^1 * \frac{1}{(2*1+1)} * \frac{x^{2*1+1}}{1!} \\ &+ (-1)^2 * \frac{1}{(2*2+1)} * \frac{x^{2*2+1}}{2!} \\ &+ (-1)^3 * \frac{1}{(2*3+1)} * \frac{x^{2*3+1}}{3!} \dots \end{aligned}$$

$$S_n(x) = x + \sum_{i=1}^n (-1)^i * \frac{1}{(2*i+1)} \frac{x^{2*i+1}}{i!}$$

$$S_0(x) = x \quad \text{Für } n \in \mathbb{N}, n > 0: S_n(x) = S_{n-1}(x) + (-1)^n * \frac{1}{(2*n+1)} * \frac{x^{2*n+1}}{n!}$$

Übung: Berechnung einer Reihe

```
#include <stdio.h>
#include <math.h>
#include <stdint.h>

/**
*****
*  @brief      Diese Funktion berechnet die Fakultät des
*              Arguments i.
*
*  @param      i Die Fakultät von i wird berechnet.
*
*  @return     Die Fakultät von i oder INFINITY bei einem Overflow.
*****/

double fakultaet(unsigned int i) {
    // Wann tritt eine Overflow auf, wenn der
    // Ergebnis den Typ long long ist hat?
    double erg = 1;
    while (0 != i) {
        erg = erg * i;
        i--;
    }
    return erg;
}
```

Übung: Berechnung einer Reihe

```
#define STOP      10E-10 // Abbruchkriterium

/**
*****
*  @brief      Diese Funktion berechnet die Reihe aus der
*              Aufgabenstellung und gibt sie auf stdout aus.
*
*  @param      x Argument x der Reihe
*
*  @return      void
*****/

void reihe (const double x) {
    double sOld = 0; // s_i
    double sNew = x; // s_(i+1)
    double a;       // s_(i+1) = s(i) + a;
    double test;     // Speichert die Abbruchbedingung
    int i = 0;       // Laufindex der Reihe
```


Übung: Berechnung einer Reihe

```
do {
    sOld = sNew;
    i = i + 1;
    // Berechne Element i der Reihe
    a = 1;
    for (int j = 1; j <= (2*i + 1); j++) {a = a * x;}
    a = a / fakultaet(i);
    a = a / (2 * i + 1);
    a = a * ((i % 2) == 1) ? -1 : 1);
    sNew = sOld + a;
    if (INFINITY == fabs(sNew)) {
        printf("Overflow in der Reihenberechnung.\n");
        return;
    }
    printf("s(i = %d, x = %lf) = %lf\n", i, x, sNew);
    test = fabs((sNew - sOld) / sNew);
    if (INFINITY == fabs(test)) {
        printf("Overflow in der Abbruchbedingung.\n");
        return;
    }
} while (test >= STOP);
}
```

Übung: Berechnung einer Reihe

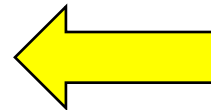
```
/**
*****
*  @brief      Berechnung der Reihe ...
*
*****/
int main(int argc, char *argv[]) {
    double x = 0; // Parameter x der Reihe
    // Einlesen der Parameters x
    printf("Bitte geben Sie den Parameter x der Reihe ein : \n");
    scanf("%lf",&x);
    printf("\nBerechnung der Reihe fuer x = %lf\n", x);
    reihe(x);
}

// EOF
```

Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Funktionen (Teil 1)
- Gültigkeitsbereich & Speicherklassen
- Zusammenfassung



Geltungsbereiche und Speicherklassen

Einführung: Ein C Programm besteht im allgemeinen aus mehreren Dateien.

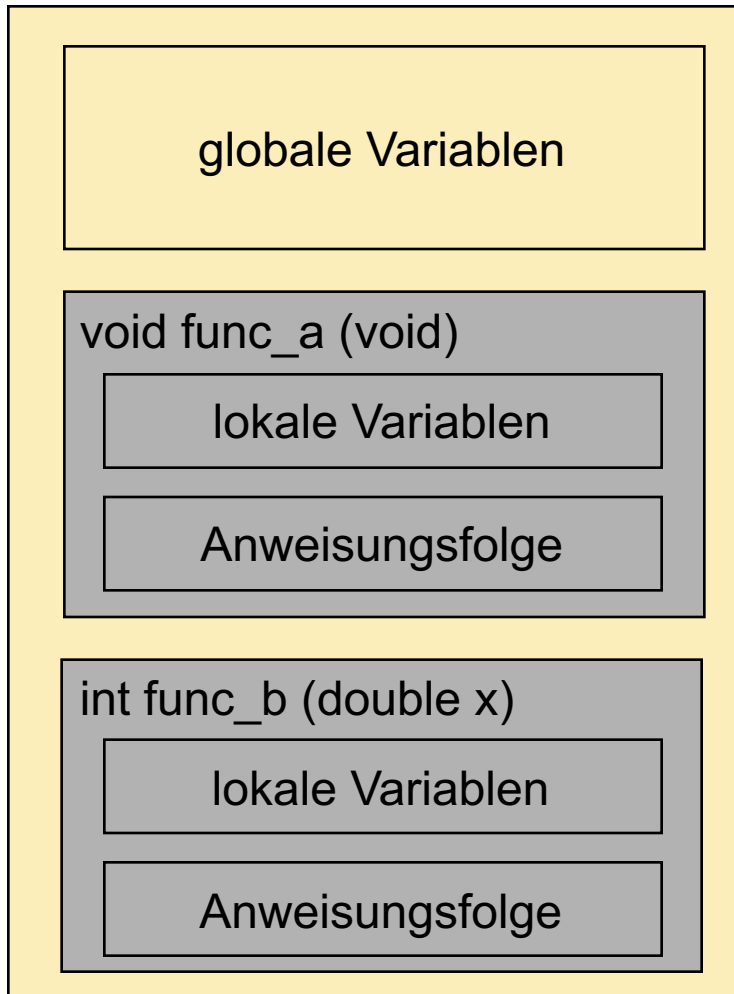
Weiterhin werden Funktionen aus Bibliotheken eingebunden – das sind in 0-ter Näherung Mengen von „früher“ übersetzten Funktionen.

Zwei Fragestellungen:

- Wie werden Variablen während der Übersetzungszeit korrekt vereinbart, so dass sie in allen Dateien des C Programms sichtbar sind (oder nicht)?
- Wie werden Funktionen aus Bibliotheken bzw. anderen Dateien des C Programms korrekt referenziert?

Der **Geltungsbereich** (scope) eines Namens ist der Teil des Programms, wo das dem Namen zugeordnete Objekt sichtbar ist – über den Namen referenzierbar ist.

Geltungsbereiche und Speicherklassen (Fortsetzung)



Dargestellt ist ein Modul – eine *.c Datei.

Die **globalen Variablen** sind in allen Funktionen aus der Datei bekannt.

Die globalen Variablen sind für die Funktionen **extern**.

Die **lokalen Variablen** haben die gleiche Lebensdauer wie ihre Funktion.

Wenn eine Funktion aufgerufen wird, werden die lokalen Variablen (auf dem Stack) angelegt. Sie werden entfernt, wenn die Funktion beendet wird.

In diesem Sinn unterscheidet C zwischen **statischen** und **automatischen** Variablen.

Globale Variablen sind stets statisch.

Geltungsbereiche und Speicherklassen (Fortsetzung)

Es gibt vier Speicherklassen:

auto Lokale Variablen einer Funktion werden erzeugt wenn die Funktion aufgerufen wird und gelöscht, wenn die Funktion beendet wird. Solche Variablen heißen automatische Variablen, sie haben die Speicherklasse **auto**. Die default Speicherklasse von lokalen Variablen aus Funktionen ist **auto**.

Dies gilt ebenso für die Parameter einer Funktion.

static **Eine lokale Variable einer Funktion, die in der Speicherklasse static ist**, wird schon beim Start des Programms angelegt. Sie wird **nicht** beim Start der Funktion angelegt und sie wird **nicht** bei der Termination der Funktion gelöscht. Somit behält sie Ihren Wert zwischen den Funktionsaufrufen – sie ist eine statische Variable (s. Java).

Das Schlüsselwort **static** vor einer Variablen Definition legt fest, dass die Variable statisch ist.

Achtung Für globale Variablen hat das Schlüsselwort **static** eine andere Bedeutung (s nächste Folie).

Geltungsbereiche und Speicherklassen (Fortsetzung)

- static** Bei **globalen Variablen** legt das Schlüsselwort `static` den **Geltungsbereich** fest. **Eine globale Variable, die in der Speicherklasse static ist,** ist nur in der C Datei / in dem Modul sichtbar, in der sie definiert ist. Der Geltungsbereich dieser Variablen umfasst andere Module (*.c Dateien) des Programms **nicht**.
- extern** Globale Variablen stehen allen Funktionen aller Module des Programms zur Verfügung.
Dies ist die default Speicherklasse globaler Variablen.
- register** Variablen, die möglichst in den Prozessorregistern gehalten werden sollen.

Geltungsbereiche und Speicherklassen (Fortsetzung)

Frage: In der Datei c1.c wird die globale Variable `global_var` definiert. Eine Funktion aus der Datei c2.c möchte diese Variable benutzen. Wie werden Namen und Typ von `global_var` in der Datei c2.c bekannt gemacht? Der Compiler, der die Datei c2.c separat übersetzt, benötigt diese Informationen.

Antwort: In der Datei c1.c wird `global_var` **definiert**, d.h. Typ und Name werden festgelegt **und zur Laufzeit wird der Speicherplatz angelegt**. In der Datei c2.c wird `global_var` **deklariert**, d.h. Typ und Name werden festgelegt – aber zur Laufzeit wird **kein** Speicherplatz angelegt. Stattdessen wird auf den in c1.c definierten Speicherplatz zugegriffen. Eine globale Variable wird genau einmal definiert und dann beliebig oft in anderen Dateien/Modulen des Programms deklariert.

Das vorgestellte Schlüsselwort **`extern`** deklariert eine globale Variable – es wird kein Speicherplatz zur Laufzeit angelegt.

```
/* Datei c1.c */  
  
// Definition von global_var  
int global_var;
```

```
/* Datei c2.c */  
  
// Deklaration von global_var  
extern int global_var;  
...
```


Geltungsbereiche und Speicherklassen (Fortsetzung)

Im allgemeinen werden externe Objekte in mehreren Modulen verwendet.

Daher werden diese extern Deklarationen in einer h Datei zusammengefasst und in den anderen Modulen inkludiert.

Das bedeutet für das Beispiel der letzten Folie:

```
/* Datei c1.c */  
#include "c1.h"  
// Definition von global_var  
int global_var;
```

```
/* Datei c2.c */  
  
#include "c1.h"  
...
```

```
/* Datei c1.h */  
#ifndef _C1_H  
#define _C1_H  
// Deklaration von global_var  
extern int global_var;  
#endif
```

Standardtechnik zur Vermeidung doppelter geschachtelter includes.

c1.h als Exportschnittstelle

c1.h als Importschnittstelle

Funktionsdeklarationen werden entsprechend in h Dateien eingetragen.

Geltungsbereiche und Speicherklassen: Beispiel

```
/* Modul_1.c */
#include "Modul_2.h"

int calc_i(); /* Fkt.-Deklaration */

int i=0;
```

```
int main(){
    int i=1, m;
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext());
    printf("k=%d \n",k);
    return 0;
}
```

```
int calc_i(){
    return i;
}
```

```
/* Modul_2.c */
#include "Modul_2.h"

int k=15;
static int i=2;
```

```
int calc_i_ext(){
    return i;
}
```

```
/* Modul_2.h */
#ifndef _MODUL_2_H
#define _MODUL_2_H
extern int k;
int calc_i_ext();
#endif
```

Ausgabe: i=1

i=0

i=2

k=15

Geltungsbereiche und Speicherklassen: Beispiel

```
/* Modul_1.c */
// #include "Modul_2.h"

/* Modul_2.h */
extern int k;
int calc_i_ext();

int calc_i(); /* Fkt.-Deklaration */

int i=0;
```

```
int main(){
    int i=1, m;
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext() );
    printf("k=%d \n",k);
    return 0;
}
```

```
int calc_i(){
    return i;
}
```

```
/* Modul_2.c */
// #include "Modul_2.h"

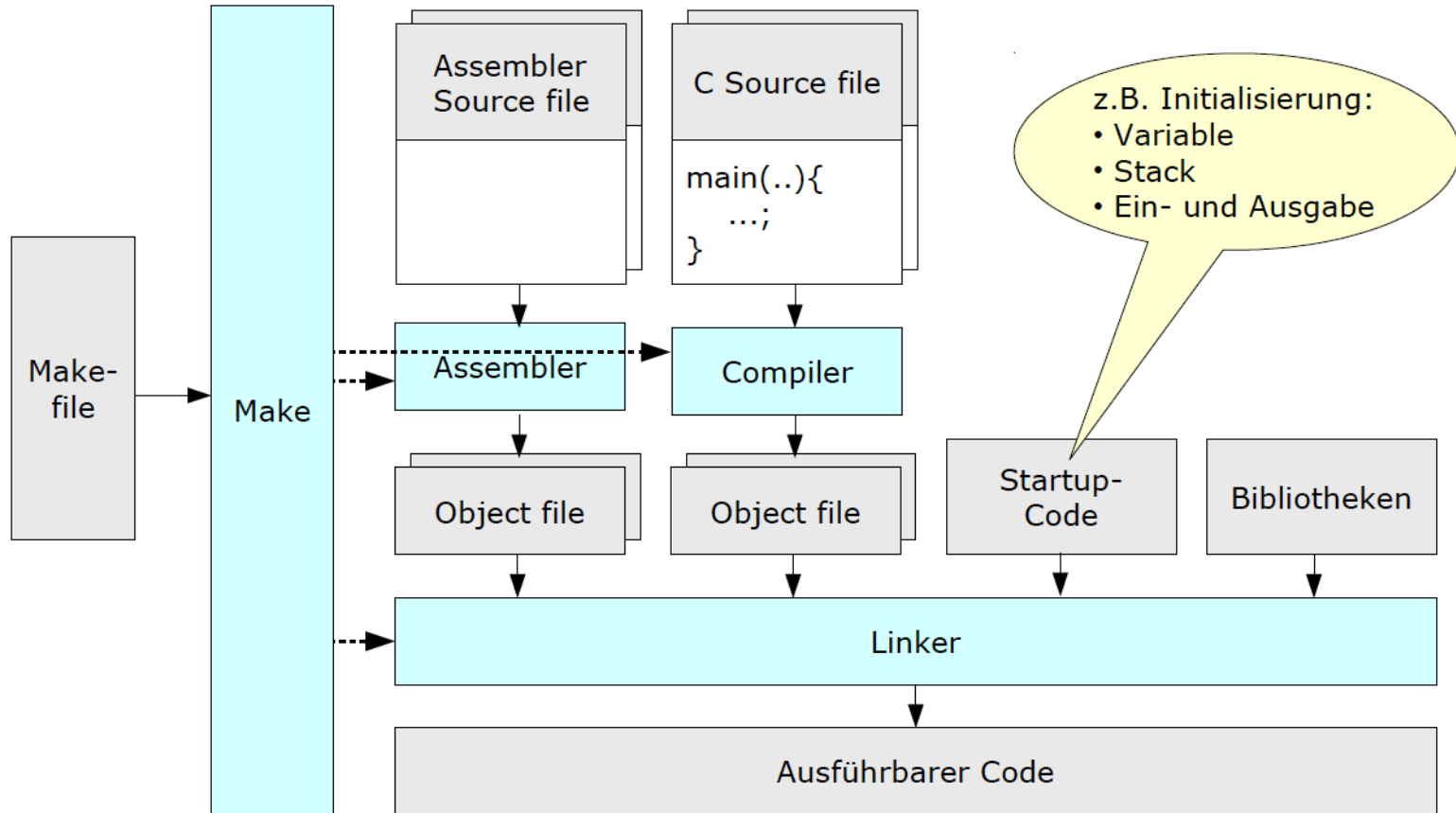
/* Modul_2.h */
extern int k;
int calc_i_ext();
int k=15;

static int i=2;
```

```
int calc_i_ext() {
    return i;
}
```

•static scope

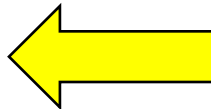
Build der SW Komponenten des TI Board



Kapitel 3: Elementare Aspekte der Programmiersprache C

Gliederung

- Einleitung
- Sprachelemente von C
- Konstante
- Variable
- Operatoren, Ausdrücke
- Anweisungen
- Zusammenfassung



Zusammenfassung

