Einführung in C

Computer Systeme
Wintersemester 2018-2019

Einführung

- Warum C im Rahmen dieser Vorlesung?
  - Relevante Betriebssysteme (Unix, GNU/Linux) sowie Systemtools (z. B. GNU Core Utilities) sind in C programmiert.
  - Mit C können wir “in den Rechner” auf Systemebene schauen und als “Lupe” für die Datenanalyse benutzen
  - Durch Vergleich der C-Konstrukte mit den übersetzten Assemblerbefehlen kann auf die Effizienz des Systems geschlossen werden.
  - Haupteinsatzgebiet: Systemprogrammierung inkl. Erstellung von Betriebssystemen, Programmierung von eingebetteten Systemen
  - Grundlage für C++ als wichtiger Programmiersprache

2

Literatur und Links

- Online-Compiler und Debugger für C zum Testen:
  - http://www.tutorialspoint.com/online_c_compiler.php
  - http://www.onlinegdb.com

Literatur für Fortgeschrittene

- Bücher zu C (ab Standard C99):
  - S. P. Harbison, G. L. Steele: *C: A Reference Manual (5th edition)*. Pearson, 2002.
  - P. Prinz, T. Crawford: *C in a Nutshell*. O'Reilly Media, 2008.
- GNU Referenz zu C und zur C-Library:
  - http://www.gnu.org/software/gnu-c-manual/
  - http://www.gnu.org/software/libc/manual/

3

Literatur und Links

- Detaillierte C-Referenz, nach Themen sortiert:
  - http://en.cppreference.com/w/c
- Übersicht über die C Standard Libraries (empfohlener Standard: C99):
  - http://en.cppreference.com/w/c/header
- Weitere Online-Tutorials für C:
  - http://en.wikibooks.org/wiki/C_Programming
  - http://www.tutorialspoint.com/cprogramming/index.htm
  - http://www.c-howto.de (auf deutsch)
- Literatur zum Umstieg von Java auf C:
  - J. Maassen: C for Java Programmers (Reader)
  - N. Nagarajan: C for Java Programmers (Folien)

4

Hands-on: Aufbau von C-Programmen

- Einfaches *Hello World*-Programm in C:

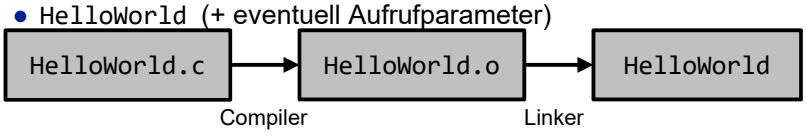
```c
#include <stdio.h>  // use C standard input/output functions

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

- Deklarationen verwendeter Funktionen aus den C Standard Libraries werden durch Präprozessoranweisung `#include` eingebunden
- C-Programme benötigen genau eine globale `main`-Methode
- Das Programm wird durch **return** *Rückgabewert*; beendet.
- Standard für *Rückgabewert*: `-1` für fehlerhafte Ausführung, `0` sonst
- Jede Zeile wird mit **;** abgeschlossen.
- Kommentare sind rechts von *//* möglich

5

Hands-on: Kompilieren von C-Programmen

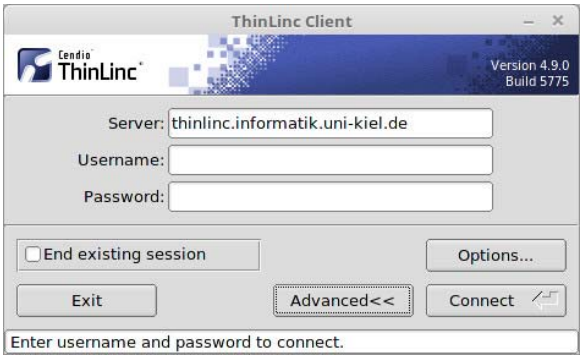
- C-Compiler (unter Linux: `gcc`) erzeugt *Objektdateien* (Assembler-Code) aus C-Quellcode:
  - `gcc -c HelloWorld.c`
- Erstelle ausführbare Datei (*Executable*) aus Objektdatei (muss `main`-Funktion beinhalten), löst Funktionsaufrufe auf (*Linking*):
  - `gcc -o HelloWorld HelloWorld.o`
- Erstellen in einem Schritt (über temporäre Objektdatei):
  - `gcc -o HelloWorld HelloWorld.c`
- Führe Executable über die Kommandozeile aus:
  - `HelloWorld` (+ eventuell Aufrufparameter)



6

ThinLinc-Installation How-To

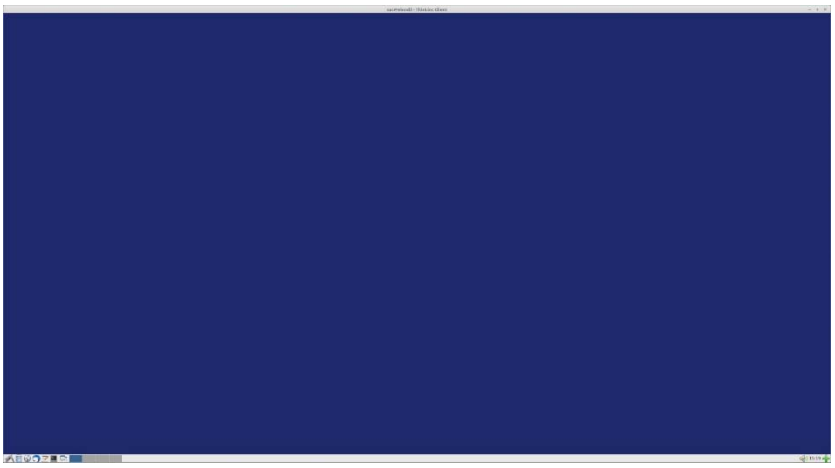
1. Download *ThinLinc* von https://www.cendio.com/thinlinc/download
2. Server: thinlinc.informatik.uni-kiel.de (falls kein Server-Eingabefeld zu sehen ist, zunächst auf ,,Advanced>>" klicken
3. Benutzername / Passwort des Informatik-Rechneraccounts



7

ThinLinc

Nach dem Klick auf ,,Connect" erscheint der Startbildschirm (leer, Icons unten links):



8

Arbeiten mit dem Terminalserver

1. Mit Klick auf das Icon in der Taskleiste den Home-Ordner öffnen.

2. Einen neuen Ordner erstellen: Rechtsklick in den Ordner => *Create New... => Folder*. Dann den Namen *C-files* als Ordnername eingeben. Es kann jeder beliebige Name selber gewählt werden.

3. Text-Editor ,,Kate" öffnen
(Startmenü mit Klick auf das Icon öffnen, dann *Accessories => Kate*).

4. Toolbar einblenden mit *Settings => Show Toolbar*

9

Arbeiten mit dem Terminalserver

5. Inhalt im Kate Editor einfügen:

10

Compilieren und Ausführen in Konsole

1. Die Datei mit dem Namen ***HelloWorld.c*** im Ordner ***C-files*** speichern. Wichtig ist die Endung **.c**, der erste Teil ist beliebig.
2. Konsole starten (Startmenü mit Klick auf das Icon öffnen, dann *System Tools => Konsole*).
3. Die Konsole öffnet sich im Home-Ordner.
4. Mit dem Befehl **`cd C-files`** in den Arbeits-Ordner wechseln. (cd: change directory)
5. Mit dem Befehl **ls** wird der Inhalt des Ordners angezeigt. Hier sollte die Datei *HelloWorld.c* zu sehen sein. (ls = list directory)

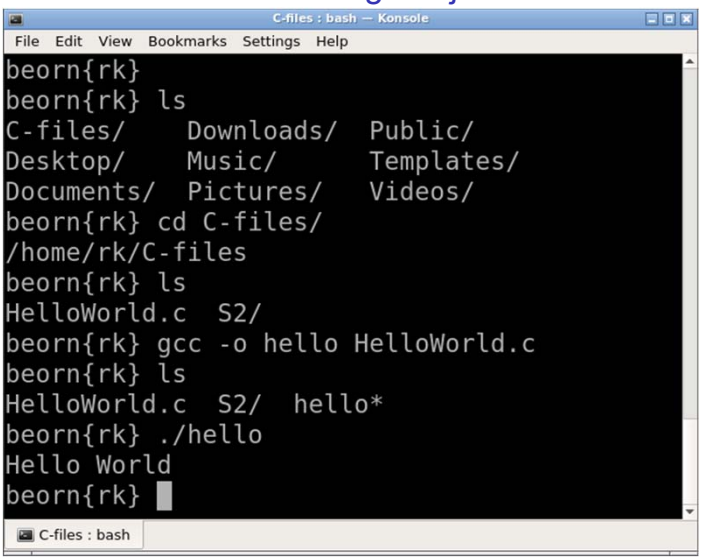
6. Compilieren: Mit dem Befehl **`gcc -o hello HelloWorld.c`** wird das Programm kompiliert und ein *Executable* erstellt. (gcc = Gnu C Compiler) Befehl: gcc –o *name-Executablefile name-Eingabefile*
7. Das Executable wird erzeugt, kenntlich durch ein * am Ende des Namens. **ls** zeigt jetzt: *hello** *HelloWorld.C* auf der Konsole.

8. Mit dem Befehl **`./hello`** wird das Executable ausgeführt. Das Zeichen **./** vor **hello** bedeutet, dass das Programm im aktuellen Ordner liegt.

11

Ergebnis der Ausführung in Konsole

So soll die Konsolenausgabe jetzt aussehen:



12

Verschiedenes

C-Files immer mit Kate-Editor öffnen:

1. Wähle ein C-File (*HelloWorld.c*) mit rechter Maustaste an
2. Unter *Properties*, wähle "*open with*" select *Kate*

Speichere einen snapshot der Konsole:

1. Wähle im Start-Menu  *Accessories -> screenshot*
2. Wähle *grab current window*
3. *Take Snapshot*
4. *Save (z.B. in Pictures)*
5. *Mail picture*

13

Konsolenbefehle – Navigieren und Anzeigen

- `ls`: Zeigt den Inhalt des aktuellen Ordners an
- `ls` <*Ordner*>: Zeigt den Inhalt des angegebenen Ordners an
- `cd` <*Ordner*>: Wechselt in den angegebenen Ordner (`cd` = change directory)
- `pwd`: Gibt den aktuellen Ordner aus (`pwd` = print working directory)
- `cat` <*Datei*>: Gibt den Inhalt der angegeben Textdatei auf der Konsole aus

14

Konsolenbefehle – Erstellen und Löschen

- `mkdir` <*Ordner*>: Erstellt den angegebenen Ordner
- `mv` <*Quelldatei*> <*Ziel*>: Verschiebt die Quelldatei an den Zielort (`mv` = move)
- `cp` <*Quelldatei*> <*Ziel*>: Kopiert die Quelldatei an den Zielort (`cp` = copy)
- `rm` <*Datei*>: Löscht die Datei unwiderruflich (`rm` = remove)

Sollen Ordner und ihr Inhalt verschoben, kopiert oder gelöscht warden, muss dem Programm als erster Parameter die Option `-r` übergeben werden (`r` = recursive). Beispiel:

`mv -r` <*Quellordner*> <*Ziel*>

15

Konsolenbefehle – Spezielle Ordner

Einige häufig benötigte Ordner sind Symbole zu erreichen:

- `.`: Der Punkt steht für das aktuelle Verzeichnis. So sind z.B. die Befehle `ls` und `ls .` analog.
- `..`: Zwei Punkte stehen für übergeordnete Verzeichnis. Beispiel:

  `cd ..`: Wechselt in das übergeordnete Verzeichnis
- `~`: Die Tilde steht für das Home-Verzeichnis. Beispiel:

  `mkdir ~/test`: Erstellt den Ordner `test` im Home-Verzeichnis

Weitere Befehle in der Unixreferenz, z.B. mit unix cheat sheets

https://www.cs.cmu.edu/~213/recitations/fwunixref.pdf

oder ähnliche Quellen

16

Hands-on: printfExample.c

```
#include <stdio.h>  // include standard library header for printf command
int main()    // start main programm, all after // is comments
{
   char name[] = “Willi"; // insert name here
   int studentID = 1234; // insert student ID here

   // Some output in formatted print command, \n is newline, %d,%X
Formats
   printf("Hello %s!\n", name); // print variable ’name’ in format  %s
   printf("Your student ID converted to different bases: \nDecimal: %d\n
Hexadecimal: %X\n", studentID, studentID);  // write all in one line

return 0;  // finish main program
}
```

17

Hands-on: printfExample.c

Erklärungen:
`char name[]=“Willi“;` legt ein Feld mit 5 Bytes an, legt den Text *W i l l i* in die ersten 5 Bytes, und ein Steuerzeichen \0 im 6. Byte zeigt an, wo der Text endet.

int studentID=1234; Hier wird eine ganze Zahl (`int`) mit dem Namen `studentID` und dem Wert `1234` dezimal angelegt.

Mit der Funktion `printf()` wird eine Ausgabe auf der Konsole getätigt. An Stelle des Platzhalters `%s` wird der Inhalt des Byte-Feldes `name` bis zum Zeichen **\0** als ASCII-Zeichenkette (string) ausgegeben.

Um Zahlen mit `printf()` auszugeben, gibt es unter anderem die Platzhalter %d für dezimale Darstellung und %x für hexadezimale Darstellung.

18

Hands-on: printfExample.c

Kompilieren mit `gcc -o printfExample printfExample.c`

Ausführen mit `./printfExample`

Ausgabe:

```
Hello Willi!
Your student ID in converted to different bases:
Decimal: 1234
Hexadecimal: 4d2
```

19

Programmierung in C: Überblick

1. Aufbau von Programmen/Dateien
2. Steueranweisungen
3. Funktionen
4. Mehr zu Datentypen:
   Union, Strukturen Felder, Zeiger

20

1. Aufbau von Programmen/Dateien

Typen von Dateien

Header-Dateien (`.h`): enthalten Deklarationen (auch Signatur, Prototyp, Rumpf) von Funktionen, Variablen, Konstanten eines Programms oder einer Bibliothek, welche in anderen Programm(datei)en genutzt werden können, in der Regel jedoch nicht deren Definition (Implementierung)

```
/* Deklaration einer Funktion */
int sum(int x, int y);
```

Code-Dateien (auch Source- oder Quelldateien, `.cpp`, `.C`, `.c`, etc.): enthalten die Definition von Funktionen (oder Methoden in C++) und ggf. die spezielle Funktion `main` als Einstiegspunkt für ein ausführbares Programm

```
/* Definition einer Funktion */
int sum(int x, int y) { return x+y; }
```

21

Aufbau von Programmen/Dateien /2

Programme können aus zahlreichen Code- und Header-Dateien bestehen

Header-Dateien als Schnittstelle zwischen verschiedenen Code Dateien: *in einer inkludierten Datei deklarierte Funktionen können benutzt werden, sind aber in einer anderen Code-Datei definiert (implementiert)*

Code-Dateien werden zuerst separat übersetzt → Ergebnis:

Objektdateien `.obj` oder `.o`

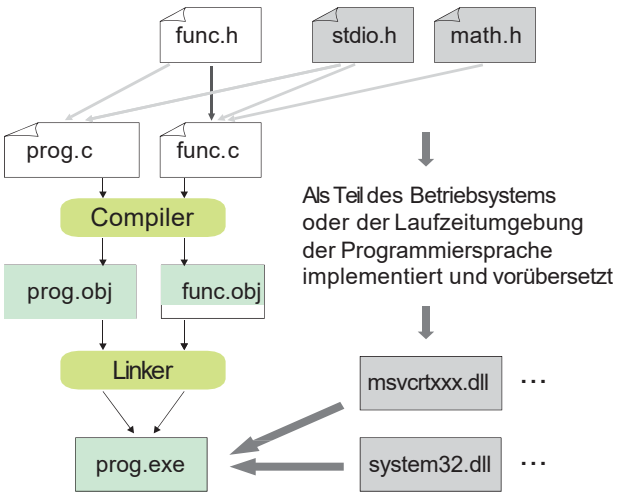
Linker: statische Verbindung verschiedener Objektdateien zu einer Programmbibliothek oder zu einem ausführbaren Programm

Standardbibliotheken der Programmiersprache oder des Betriebssystems sowie Teile komplexer Programme werden meist dynamisch (zur Laufzeit) gelinkt/geladen

Windows: *Dynamic Load Libraries* (DLL) Unix: zum Beispiel `Shared Objects` (`.so`)
Vermeidet Aufblähung ausführbarer Programme und ständige Neuübersetzung immer wieder verwendeter Programmteile

22

C Übersetzungsprozeß



23

1.1 Struktur der Hauptdatei

Im folgenden Fokus auf einfache Programme bestehend aus einer Hauptdatei:

```
#include <...> /* Deklarationen für einzubettende Programmbibliotheke
#define ... /* Konstanten-Definitionen und Makros */

/* Globale Variablen */
int i;
char c[] = ...;

/* Funktionen */
int f1(int a, int b) { ... ;}
void f2() {...;}

/* Hauptprogramm */
int main()
{
   ...
}
```

24

Der Präprozessor

Mit # beginnende Zeilen sind Anweisungen für den Präprozessor (Teilprogramm des Compilers) zur Durchführung einfacher Transformationen des Quelltextes vor der eigentlichen Übersetzung, z.B.

#include-Statements: Einbettung anderer Quelltextdateien (vor allem Header-Dateien von Programmteilen oder Standardbibliotheken)

```
#include <...> /* Datei im INCLUDE-Pfad */
#include "..." /* relativer Pfad */
```

#define-Statements: Festlegung von Werten für Ausdrücke, die z.B. im Quelltext in der Folge ersetzt werden können oder für die Steuerung des Präprozessors verwendet werden können

```
#define PI 3.14159265
```

25

1.2 Programmblöcke

Inhaltlich zusammengehörige Befehlsfolgen werden durch geschweifte Klammern { . . . } zu Programmblöcken zusammengefaßt

Verwendung zum Beispiel für

Definition von Funktionen

```
int f(int x, int y) { ...;}
```

Steueranweisungen wie Schleifen

```
while (<bedingung>) { ...;}
```

Innerhalb eines Programmblocks definierte Variablen, können nur dort verwendet werden

26

Sequenzen

Sequenzen von Befehlen (entsprechend imperativem Programmierparadigma) werden durch Verkettung mit Semikolon `;` gebildet

Dabei können einzelne Befehle

- Definition/Deklaration von Variablen
- Steueranweisungen (Schleifen, Bedingungen, Fehlerbehandlung, etc.)
- Ausdrücke (haben immer ein Ergebnis und einen Ergebnistyp)
  - Literale Werte (Direkte Festsetzung von Werten, z.B. “10”) Variablen (z.B. “x”)
  - Funktionsaufrufe
  - Komplexe Ausdrücke: Verknüpfung von Teilausdrücken durch Operatoren

sein

27

1.3 Variablendeklarationen

Grundlegende Syntax:

```
<datentyp> <variablennamen> [, <variablenname> ...];
```

Datentyp ist:

Basisdatentyp wie `int`, `float`, etc. (siehe Abschnitt Kodierung)
Feld von Variablen eines Datentyps (→)
Zusammengesetzter, nutzerdefinierter Datentyp (→)
Zeiger auf Adresse eines Objektes oder Wertes von Datentyp (→)

Erlaubt Deklaration mehrerer Variablen eines Typs in einem Statement, z.B.

```
int x, y, z;
```

Erlaubt Zuweisung initialer Werte, z.B.

```
int solution=42;
```

28

Elementare Datentypen

- Zeichen (*Character*): `char` (1 Byte)

(signed) char: 8 Bit 2-Komplement; unsigned char: 8 Bit natürliche Zahl

unsigned char a ={0,…,255} char b= {-128,…,127}

- Ganzzahl (*Integer*): `int` (4 Byte), `short` (2 Byte), `long` (4/8 Byte)

  (signed) int: 32 Bit 2-Komplement; unsigned int: 32 Bit natürliche Zahl
- Gleitkommazahl: `float` (4 Byte), `double` (8 Byte)
- Ohne Vorzeichen: `unsigned <typ>`
- Typumwandlung (cast) zwischen Datentypen implizit (bei Zuweisung) und explizit (per Befehl) möglich: (int -> float) ist OK, (float -> int) kann Datenverlust bedeuten (Nachkommaanteil weg).

`sizeof(`*Typ*`)` liefert Größe eines Datentyps in Byte zurück!

29

Elementare Datentypen

`sizeof(`*Typ*`)` liefert Größe eines Datentyps in Byte zurück!

- Elementare Datentypen:
  - Zeichen (*Character*): `char` (1 Byte)
  - Ganzzahl (*Integer*): `int` (4 Byte), `short` (2 Byte), `long` (4/8 Byte)
  - Gleitkommazahl: `float` (4 Byte), `double` (8 Byte)
  - Ohne Vorzeichen: `unsigned char`, `unsigned int`, …
- Felder (*Arrays*) mit fester Länge, z. B. Integer-Array `int myValues[10]`
- Zeiger (*Pointer*), z. B. Integer-Pointer `int* myIntPointer`
- `void`-Datentyp (kein Wert, undef.), z. B.: `void*` für Pointer ohne Typ
- Verbunddatentypen (*struct, union*)
- Funktionsdatentypen (z. B. `int(int,int)` für Funktion `binomial`)
- `const` für Konstanten (z. B. `const int num = 10`)

30

Konvertierung (cast) von Datentypen

- Implizite Typumwandlung (*Cast*) elementarer Datentypen:
  - Ohne Datenverlust: `int i = 10; double d = i;` → `10.0`
  - `short s = 10; int i = s;` → `10`
  - Genauigkeitsverlust: `double d = 1e-50; float f = d;` → `0.0f`
  - Abrunden: `double d = 10.64; int i = d;` → `10`
  - Überlauf: `int i = 70000; short s = i;` → `4464`
  - `unsigned char u = 255; char c = u;` → `-1`
- Explizite Typumwandlung von Datentypen mit (*Typ*), z. B.:
- `int x = 6, y = 8; double q = x / (double)y;`

Ergibt `6 / 8.0 = 0.75`

31

Zeichen und Zeichenketten in C

Alle **Zeichen** werden in C durch Datentyp **`char`** abgebildet

Mögliche Zustände 0-255 werden für erweiterten ASCII benutzt
Wird entsprechend Programmkontext als Zeichen oder Wert interpretiert

Zeichenketten
fester Länge durch Feld (Array) mit bestimmter Größe, z.B.

```
char login[9];
```

Initialisierung der Zeichenkette (string) der Länge 13 und 1 Byte Ende-Markierung

```
char message[14] = ”Hello, World!”;
```

32

Char-Zeichencodes im ASCII-Format

ASCII (American Standard Code for Information Interchange)

Bit 0 – 3:	Bit 4-7:	000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
0	0000	NUL	DLE	SP	0	@	P	‘	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	“	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	´	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	SKIP	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	HOME	.	>	N	^	n	~
F	1111	SI	NL	/	?	O	_	o	DEL

Ein-/Ausgabe C-style: printf()

Grundlegende Ein-/Ausgabe für C definiert in **`stdio.h`** als Funktionen

Ausgabe: Funktion **`printf()`**

```
printf(<Formatstring>[,<Ausgabeliste>])
```

- Formatstring besteht aus literalem Ausgabetext und typspezischen Platzhaltern beginnend mit % für formatierte Ausgabe von Werten
- Ausgabeliste besteht aus freier Anzahl von kommaseparierten Parametern welche auszugebende Variablenwerte darstellen und mit Anzahl und Typ der Platzhalter übereinstimmen muss
- Achtung: Den Formatstring in einer Zeile eingeben. Falls ein Zeilenumbruch bei der Eingabe gewünscht ist, dann diesen mit „\“ abschließen.

```
printf(“Text \
        Weiterschreiben \n”, Argumente…)
```

34

Ein-/Ausgabe C-style: printf()

- Verwende `printf` zur formatierten Konsolenausgabe
  - Aufruf: `printf(`*Format-String*`,` *Argumente*`…)`
  - Rückgabewert: Anzahl der ausgegebenen Zeichen, `-1` für Fehler
  - Analog zur Methode `format` in Java
- Übliche Platzhalter für auszugebende Argumente im Format-String:
  - `%d` Ganzzahl, für Argumente vom Typ `int`, `short`, usw.
  - `%s` String, für Argumente vom Typ `char*`
  - `%c` Zeichen, für Argument vom Typ `char`
  - `%x` Hexadezimale Darstellung für Ganzzahlen
  - `%f`, `%g` Kommazahl, für Argumente vom Typ `float`
  - `%p` Adresse, für Pointer
  - `%ld`, `%lf`, `%lg` Platzhalter für `long` und `double`

Referenz für `printf` siehe z. B.: http://en.cppreference.com/w/c/io/fprintf

35

Ein-/Ausgabe C-style: printf()

- Sonderzeichen in Strings können durch Escape-Sequenzen beschrieben werden (wie in Java):
  - `\n` Zeilenumbruch (“new line”)
  - `\r` Zum Zeilenanfang zurücksetzen (“carriage return”)
  - `\t` Tabulator
  - `\\` Backslash
  - `\%` Prozentzeichen
  - `\'` Einfaches Anführungszeichen
  - `\"` Doppeltes Anführungszeichen
  - `\` Für Formatstring über mehrere Zeilen
  - `\x`*HH* ASCII-Zeichen zum Hexadezimalcode *HH*
- Hinweis: Diese Escape-Sequenzen stellen in Strings jeweils *ein einzelnes* Zeichen (`char`) dar!

Referenz für `printf` siehe z. B.: http://en.cppreference.com/w/c/io/fprintf

36

Ein-/Ausgabe C-style: scanf()

Eingabe: Funktion `scanf()`

```
scanf(<Formatstring>[,<Eingabeliste>])
```

Erlaubt Erfassen komplex strukturierter Eingaben mit Textmustern und Platzhaltern entsprechend `printf()`, aber im folgenden einfache Eingabe meist einzelner Werte.
Bei strings und Feldern entweder 1. Element referenzieren oder direkt Stringname angeben.

```
scanf("%i", &meinInteger);
scanf("%c", &meinChar);
scanf("%s", &meinString[0]); // Zeichen-
scanf("%s", meinString);     // ketten
```

Vorsicht: Eingabeliste besteht aus Adressen der Variablen (vorgestelltes &), da Werte durch Funktion modifiziert werden müssen (*pass by reference* → Funktionen)

37

Zeichenketten als **`char`**-Array

Umsetzung von Zeichenketten (engl. *Strings*) als Folge von **`char`**-Werten (1 Byte), welche ASCII-Zeichen kodieren

Größe bestimmt **maximale** Länge der Zeichenkette → tatsächliches Ende der Zeichenkette ist durch speziellen Wert 0 gekennzeichnet (null-terminierte Zeichenketten)

Deshalb: tatsächliche maximale Länge ist Größe des Arrays minus ein Zeichen für Endemarkierung

Initialisierung als Liste wie
normale Arrays oder als literaler Stringwert in Hochkommas

```
char s[6]= {'H','A','L','L','O','\0'};
```

ist äquivalent zu

```
char s[] = "HALLO";
```

Beispiel Test-ascii.c

38

Kommentare

Natürlichsprachliche Beschreibung/Erklärung des Quelltextes oft hilfreich und erforderlich

Werden vom Compiler bei der Übersetzung vollständig ignoriert

```
/* Kommentar, gerne auch
   mal über  mehrere
   Textzeilen */

x=x+1; // Kommentar bis zum Zeilenende
```

39

1.4 Ausdrücke (Expressions)

- Zusammengesetzt aus Variablen, literalen Werten und Funktionsaufrufen
- Verbunden durch Operatoren
- Beispiele:

```
x=1; // Zuweisung eines literalen Wertes
x++; // Inkrementierung (postfix)
x=x+1; /* Zuweisung und arithmetische Verknüpfung */
x=(x==4); /* Zuweisung und geklammerter Vergleich */
x=quadrat(x+1); /* Zuweisung und Funktionsaufruf */
x & 1; /* Logische Operation, Ergebnis ignoriert */
```

40

Ausdrücke /2

Können durch Kombination und Klammerung beliebig komplex werden → VORSICHT: Lesbarkeit!

VORSICHT: Typ-Kompatibilität der Operanden

- Implizite Konvertierung manchmal unproblematisch, z.B. `int` → `float`
- Teilweise Verlust von Information, z.B. `float` → `int` durch Abschneiden der Nachkommastellen
- Nur zum Teil Fehlermeldungen oder Warnungen bei inkompatiblen Typen

41

Ausdrücke: Zuweisungsoperatoren

Operator	Beispiel	Bedeutung
=	**x=7**	**Standardzuweisungsoperator**
+=	x+=7	Selbstzuweisung mit Addition entspricht x=x+7 (prefix Variante)
=+	x=+7	Selbstzuweisung mit Addition (postfix Variante)
-=	x-=7	Selbstzuweisung mit Subtraktion
*=	x*=7	Selbstzuweisung mit Multiplikation
/=	x/=7	Selbstzuweisung mit Division
%=	x%=7	Selbstzuweisung des Divisionsrestes
...	...	...

Ergebnis einer normalen oder prefix-Zuweisung ist der neue Wert des linken Operanden

42

Ausdrücke: Arithmetische Operatoren

Operator	Beispiel	Bedeutung
+	x+y	Addition
-	x-y	Subtraktion
*	x*y	Multiplikation
/	x/y	Division (bei ganzen Zahlen (x div y))
%	x%y	Rest bei ganzzahliger Division (x mod y)
>>	x >> y	Bit-Shift: x nach rechts um y Stellen (Division durch 2^y), Wert vom MSB wird kopiert (für signed Arithmetik)
<<	x << y	Bit-Shift: x nach links um y Stellen (Multiplikation mit 2^y). Von rechts werden die Bits mit 0 aufgefüllt.

43

Ausdrücke: Arithmetische Operatoren (2)

Operator	Beispiel	Bedeutung
++	y=x++	Inkrementierung (postfix) y=x: y hat alten Wert von x x=x+1: x um eins erhöht
	y=++x	Inkrementierung (prefix) x=x+1: x um eins erhöht y=x: y hat neuen Wert von x
	x++, ++x	Selbst-Inkrement x=x+1
--	y=x--, y=--x, x--, --x	Dekrementierung entsprechend als pre- und postfix anwendbar

44

Ausdrücke: Vergleichsoperatoren

Da in C ein `boolean`-Datentyp fehlt, ist Ergebnis ganzzahliger Wert

1, wenn Vergleich erfolgreich (*TRUE*)
0, andernfalls (*FALSE*)

Operator	Beispiel	Bedeutung
==	x==y	Test auf Wertegleichheit
!=	x!=y	Test auf Ungleichheit
>	x>y	... größer als ...
<	x<y	... kleiner als ...
>=	x>=y	... größer als oder gleich ...
<=	x<=y	... kleiner als oder gleich ...
...	...	...

45

Ausdrücke: Logische Operatoren /1

Zwei Sichtweisen für logische Operationen

- Bitweises Verknüpfen für Bytes
- Logische Verknüpfungen von 1 (*TRUE*) und 0 (*FALSE*) kann auf 1. zurückgeführt werden

Bei Operatoren für logische Verknüpfung: Abbruch (*short cut*) der Auswertung, wenn möglich

- Bei *AND*: Abbruch, wenn erste Bedingung *FALSE*, da Ergebnis nur *FALSE* sein kann
- Bei *OR*: Abbruch, wenn erste Bedingung *TRUE*, da Ergebnis nur *TRUE* sein kann

46

Ausdrücke: Logische Operatoren /2

Operator	Beispiel	Bedeutung
!	!x	Logische Negation (*NOT*)
&&	x&&y	Logische Konjunktion (*AND*)
||	x||y	Logische Disjunktion (*OR*)
&	x&y	Bitweises *AND*
|	x|y	Bitweises *OR*
∧	x∧y	Bitweises *XOR*
~	~x	Bitweise Negation
. ..	...	...

47

Ausdrücke: Sonstige Operatoren

Zur Arbeit mit speziellen Datentypen wie zum Beispiel Feldern, Zeigern, Strukturen (→)

Operator	Beispiel	Bedeutung
[]	a[7]	Zugriff auf Position in Feld
.	window.size	Zugriff auf Komponente einer Struktur oder eines Objektes
*	*i	Dereferenzierung eines Zeigers
&	&i	Bildung der Referenz (Zeiger)
->	window->size	Zugriff auf Komponente über Zeiger
()	(int)myFloat	Explizite Typumwandlung (Casting)
...	...	...

48

Zusammenfassung: Aufbau von Programmen

- Programme können aus zahlreichen Dateien bestehen
  Header-Dateien als Definition der Schnittstelle von Bibliotheken aus Funktionen, Variablen und Konstanten
  Code-Dateien enthalten Implementierung von Funktionen aus Bibliotheken und/oder `main`-Funktion
- Programmblöcke, z.B. Implementierung einer Funktionen, bestehen aus Sequenz von Befehlen
- Befehle sind Deklarationen, Steueranweisungen oder Ausdrücke
- Einfache Ausdrücke sind Funktionsaufrufe, Variablen oder literale Werte
- Komplexe Ausdrücke können durch Verknüpfung von Ausdrücken über Operatoren gebildet werden

49

Steueranweisungen

Bisher Programme mit Funktionen als einfache Folge von Befehlen

Ablauf von Programmen darüber hinaus steuerbar über

Bedingte Ausführung: Ausführung von Programmteilen (Befehlen oder Programmblöcken) nur wenn eine bestimmte Bedingung erfüllt ist → Selektion, Verzweigung

Programmschleifen: mehrfache Wiederholung von Programmteilen so lange oder bis eine bestimmte Bedingung erfüllt bzw. nicht mehr erfüllt ist → Iteration, Schleife

Fehlerbehandlung: Ausführung spezieller Programmteile im Fall eines Fehlers

50

Darstellung von Programmabläufen

Zum Entwurf und zur Illustration von Algorithmen verschiedene formale und informale Beschreibungsmittel:

Pseudocode: informale, textuelle Darstellung des Algorithmus zum Teil mit Mitteln der natürlichen Sprache

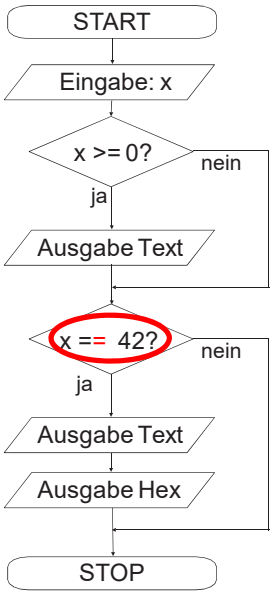
Struktogramme: formale Darstellung von Programmabläufen als Blockdiagramm

Programmablaufpläne (PAP): Beschreibung des Programmablaufs durch spezielle Notation für generische Anweisungsklassen

Im folgenden wird letztgenannter Diagrammtyp gemischt mit Pseudocode zur Illustration verwendet

51

Einfache Bedingte Ausführung: PAP



52

Einfache Bedingte Ausführung

if-Anweisung gefolgt von geklammerter Bedingung als Einführung für einen einzelnen Befehl oder einen Programmblock mit Befehlen. Befehle immer mit ; abschliessen!

```
if (<Bedingung>) <Befehl>
if (<Bedingung>) { <Befehlssequenz> }
```

Bedingung kann über logische Operatoren verknüpft werden und evaluiert entsprechend zu 0 (*FALSE*) oder andernfalls *TRUE* (→ logische Operatoren)

Ausführung entsprechend Ergebnis:

TRUE: Ausführung des von **if** eingeleiteten Befehls oder Programmblocks

FALSE: Programmteil der zum **if**-Block gehört wird nicht ausgeführt und Ausführung danach fortgesetzt

53

Einfache Bedingte Ausführung: Beispiel (C)

```
#include <stdio.h>

int main()
{
   int x;
   printf("x = ");
   scanf("%i",&x);
   if (x >= 0)
      printf("Zahl ist positiv oder gleich 0. \n");
   if (x == 42) {  // Achtung: nicht (x=42) !!!
      printf("Die Zahl ist 42. \n");
      printf(" in hexadezimal %X\n",42);
   }
   return 0;
}
```

Beispiel Test-if.c

54

Bedingte Ausführung mit Alternative

Erweiterter Syntax der **`if`**-Anweisung um **`else`**-Klausel: Ausführung eines alternativen Programmteils, falls Bedingung nicht erfüllt

```
if (<Bedingung>) <Befehl oder Programmblock>
else <Befehl oder Programmblock>
```

In Abhängigkeit von der Bedingung wird nur ein „Zweig“ der Anweisung ausgeführt (Verzweigung)

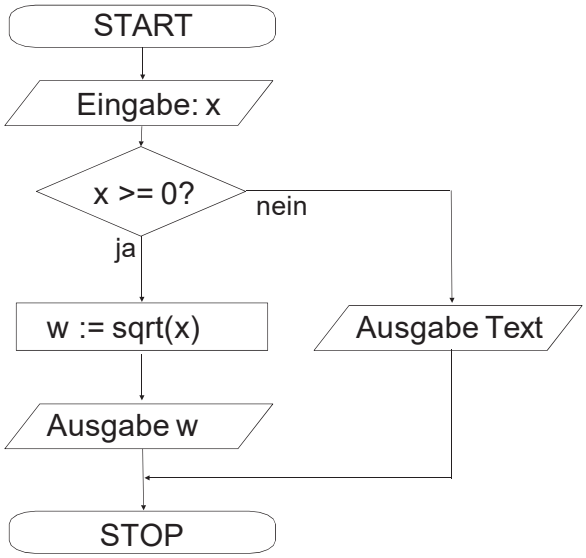
55

Bedingte Ausführung mit Alternative: Beispiel

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

int main()
{
   float x;
   printf(”x =
   ”);
   scanf(”%f”,&x);
   if (x >= 0)
   {
      float w = sqrt(x);
      printf(”Wurzel ist %f \n”,
      w);
   }
   else
      printf(”Wurzelberechnung nicht möglich! \n”);
   return 0;
}
```

56

Bedingte Ausführung mit Alternative: PAP



57

Schachtelung von `if`-Anweisungen

Verzweigungen können beliebig geschachtelt werden

```
if (alter >= 0)
  if (alter < 120)
    printf(”Alter korrekt \n”);
```

Können durch Zusammenfassen der Bedingungen oft vereinfacht werden: Sequenz von **`if`**-Bedingungen entspricht logischer Konjunktion

```
if (alter >= 0 && alter < 120)
  printf(”Alter korrekt \n”);
```

58

Schachtelung von **`if`**-Anweisungen /2

Schachtelung erlaubt separate Behandlung der logischen Alternativen im **`else`**-Zweig

```
if (alter >= 0)
   if (alter < 120)
      printf(”Alter korrekt \n”);
   else
      printf(”Alter zu hoch! \n”);
else
      printf(”Alter zu gering! \n”);
```

59

Sequenz von **`if`**/**`else if`**-Anweisungen

Sequenz von **`if`**/**`else if`**-Anweisungen erlaubt Mehrfachverzweigung

```
if (alter < 1)  printf(”Baby \n”);
else if (alter < 4)
   printf(”Kleinkind \n”);
else if (alter < 6)  printf(”Vorschulkind \n”);
else if (alter < 18)
   printf(”Schulkind \n”);
else
   printf(”Führerschein! \n”);
```

60

Mehrfachverzweigung mit **switch**

switch-Steueranweisung und **case**-Sprungmarken setzen einfache Sprungtabelle um

```
switch(<Variable>) {
   case  <Wert1>  :  <Befehlssequenz>
   case  <Wert2>  :  <Befehlssequenz>
   case <Wert3> : <Befehlssequenz>
   ...
   default: <Befehlssequenz>
}
```

switch-Variable muss ordinalen Wert haben, d.h. Integer (inklusive **char**)

case-Marken sind Sprungmarken, d.h. Ausführung ab da bis zum Ende des **switch**-Statements → sollen nur einzelne Fälle ausgeführt werden, muss der **switch**-Block mit **break** verlassen werden

Falls kein Wert in der Liste, wird zur optionalen **default**-Marke gesprungen

61

Mehrfachverzweigung mit **switch**: Beispiel /1

```
#include <stdio.h>
#define PI 3.141592653589793

int main()
{
   int auswahl;
   float
   radius,durchmesser,umfang,flaeche;
   printf(" Radius des Kreises = ");
   scanf("%f",&radius);
   printf(" 1) Berechnung des Durchmessers\n");
   printf(" 2) Berechnung des
   Umfangs\n");  printf(" 3)
   Berechnung der Fläche\n");
   printf(" Auswahl: ");
   scanf("%d",&auswahl);
   ...
```

62

Mehrfachverzweigung mit **switch**: Beispiel /2

```
   ...
   switch(auswahl) {
      case 1:
         durchmesser = 2*radius;
         printf("Durchmesser ist: %f\n", durchmesser);
         break;
         case 2:
         umfang = 2*radius*PI;
         printf("Umfang ist: %f\n", umfang);
         break;
         case 3:
         flaeche = radius*radius*PI;
         printf("Fläche ist: %f\n", flaeche);
         break;
      default:
         printf("Ungültige Auswahl!\n");
   }
   return 0;
}
```

63

Mehrfachverzweigung mit **switch**: Beispiel /3

Vorsicht bei fehlenden **break**-Steueranweisungen:

```
   switch(auswahl) {
      case 1:
         printf("Durchmesser ist: %f\n", durchmesser);
      case 2:
         printf("Umfang ist: %f\n", umfang);
      case 3:
         printf("Fläche ist: %f\n", flaeche);
   }
```

```
Radius des Kreises = 23
1)Berechnung des Durchmessers
2) Berechnung des Umfangs
3)Berechnung der
Fläche  Auswahl: 1
Durchmesser ist: 46.000000
Umfang ist: 144.513260
Fläche ist: 1661.902466
```

64

Programmschleifen

Auch Iteration oder Zyklen: Wiederholung von Programmteilen (Blöcken oder einzelnen Befehlen) entsprechend bestimmten Bedingungen

- **`while`**-Schleife: kopfgesteuerte Schleife mit beliebiger Bedingung für Wiederholung, die vor dem Programmteil ausgwertet wird
- **`for`**-Schleife: Zählschleife zum Durchlaufen von Wertebereichen für eine Variable
- **`do-while`**-Schleife: fußgesteuerte Schleife mit beliebiger Bedingung für Wiederholung, die am Ende des Programmteils ausgewertet wird

65

`while`-Schleife

Kopfgesteuerte Schleife

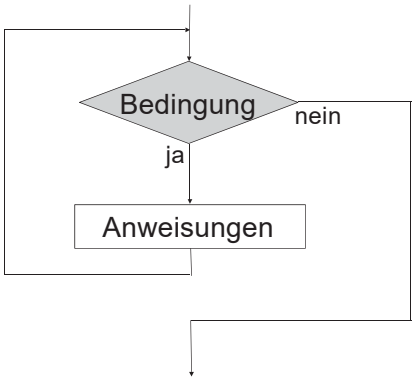
```
while (<Bedingung>) <Befehl oder Programmblock>
```

Bedingung wird jeweils vor Ausführung des folgenden Befehls/Programmblocks ausgewertet

Abweisende Schleife: wird die Bedingung vor der ersten Ausführung nicht erfüllt, wird der Schleifentext gar nicht ausgeführt

66

while-Schleife: PAP



67

while-Schleife: Beispiel (C)

```
#include <stdio.h>

int main()
{
   int min,max;
   int sum = 0;
   printf(”Kleinere Zahl = ”);
   scanf(”%i”, &min);
   printf(”Größere Zahl = ”);
   scanf(”%i”, &max);
   printf(”Summe aller Zahlen von %i bis %i ist ”, min,max);
   while(min <= max) {
      sum = sum + min;
      min++;
   }
   printf(”%i . \n”, sum);
   return 0;
}
```

Beispiel Test-Schleife.c

68

for-Schleife

Zählschleife

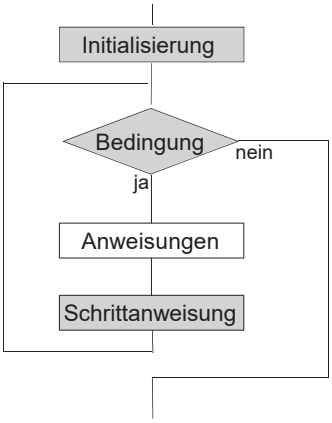
```
for ([<Initialisierung>];[<Bedingung>];[<Schrittanweisung>])
   <Befehl oder Programmblock>
```

Initialisierung: Deklaration (oder kommaseparierte Folge von Deklarationen) und Defintion der Laufvariable(n), die einmalig vor Beginn der Iteration ausgeführt wird Bedingung: logischer Ausdruck, der am Anfang jeden Iterationsschritts erfüllt sein muss
Schrittanweisung: Modifikation der Laufvariablen am Ende jeden Iterationsschrittes, bei mehreren auch kommaseparierte Liste von Modifikationen

Alle drei Teile sind optional

69

for-Schleife: PAP



70

for-Schleife: Beispiel (C)

```
#include <stdio.h>

int main()
{
   int max;
   printf("Maximalwert = ");
   scanf("%i", &max);
   int sum = 0;
   for (int i=1; i <= max; i++)
      {  sum = sum + i;
      printf("Summe der natürlichen Zahlen bis %i : %i \n",
            i, sum);
   }
   return 0;
}
```

Beispiel Test-for.c

71

for-Schleife: Beispiele möglicher Schleifenköpfe

Weitere Beispiele

```
for (int i=1; i <= 10; i++) ... // 1 2 3 ... 10
for (int i=10; i > 0; i--) ... // 10 9 8 ... 1
for (int i=1; i <= 256; i*=2) ... // 1 2 4 8 ... 256
for (int i=1, int j=10; i < j; i++,j--) ...
                  // (1,10) (2,9) ... (5,6)
for (;;) ... // Endlosschleife
for (;!gefunden;) ... // entspricht while(!gefunden)
...
```

72

`do-while`-Schleife

Fußgesteuerte Schleife

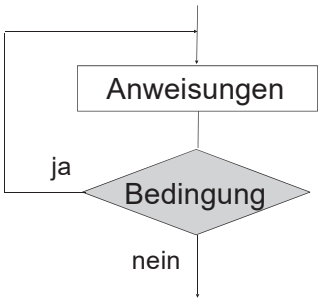
```
do <Befehl oder Programmblock> while (<Bedingung>);
```

Bedingung wird jeweils nach Ausführung des Schleifenteils überprüft

Nicht-abweisende Schleife: der Schleifentext wird mindestens immer einmal ausgeführt, bevor die Bedingung am Ende überprüft wird

73

`do-while`-Schleife: PAP



74

do-while-Schleife: Beispiel (C)

```
#include <stdio.h>

int main()
{
   int summand;  int
        sum = 0;

         do
      {
      printf("Aktuelle Zwischensumme : %i \n", sum);
      printf("Nächster Summand (0 für Abbruch) = ");
      scanf("%i", &summand);
      sum = sum + summand;
   }
   while (summand != 0);

   return 0;
}
```

75

Schachtelung von Schleifen

Schleifen gleicher oder unterschiedlicher Art können beliebig geschachtelt werden

76

Steueranweisung in Schleifen

Spezielle Steueranweisung zur Kontrolle des Schleifenablaufs

`break`: Abbruch der Schleifenausführung, Programm wird nach dem Schleifentext fortgesetzt (entsprechend schon bei Mehrfachverzweigung mit switch verwendet)

`continue`: Abbruch des aktuellen Schleifendurchlaufs, d.h. Code bis zum Ende des Schleifenblocks wird ignoriert, und mit dem nächsten Iterationsschritt (Anfang der Schleife) fortgesetzt

77

Schleifen mit **`break`**: Beispiel (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MIN 1
#define MAX 100
#define MAX_VERSUCHE 20

int main()
{
   srand(time(NULL)); // Initialisierung Zufallszahlen
   int gesucht = MIN+rand()%(MAX-MIN+1);
   printf("Zahlenraten zwischen %i und %i \n",MIN,MAX);
   printf("Gesuchte Zahl = %i \n",gesucht);
   int versuch = 0;
   ...
```

Schleifen mit **`break`**: Beispiel (C) /2

```
   ...
   while(versuch <
      MAX_VERSUCHE) {
      versuch++;
      int test = MIN+rand()%(MAX-MIN+1);
      printf("Versuch %i : %i \n", versuch,test);
      if (test == gesucht)
         {
         printf("Gefunden!
         \n");  break;
      }
   }
   return 0;
}
```

79

Schleifen mit **`continue`**: Beispiel (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MIN 1
#define MAX 10

int main()
{
   srand(time(NULL));
   int gesucht = MIN+rand()%(MAX-MIN+1);
   printf("Zahlenraten zwischen %i und %i \n",MIN,MAX);
   int test;
   ...
```

80

Schleifen mit **`continue`**: Beispiel (C) /2

```
   ...
   int gefunden=false;
   while(!gefunden) {
      printf("Geratene Zahl = ");
      scanf("%i",&test);
      if (test < MIN || test > MAX)  {
         printf("Zahl ist außerhalb des Bereichs! \n");
            continue;
      }
      if (test == gesucht) break;
      printf("Leider falsch geraten.
      \n");
   }
   printf("Gefunden! \n");
   return 0;
}
```

81

Zusammenfassung: Steueranweisung

- Bedingte Ausführung
  - **`if`**: einfache bedingte Ausführung
  - **`if-else`**: mit Alternative
  - Schachtelung und Sequenz von Verzweigungen
  - **`switch-case`**: Sprungtabelle für Mehrfachverzweigung
- Schleifen
  - **`for`**: Zählschleife
  - **`while`**: kopfgesteuerte Schleife
  - **`do-while`**: fußgesteuerte Schleife
  - Steuerung mit **`break`** und **`continue`**

82