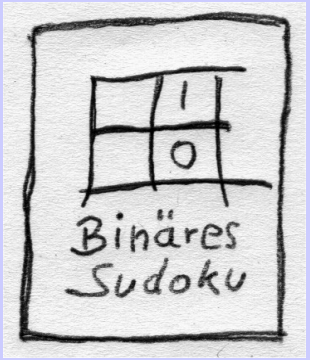
hochschule mannheim

Jörn Fischer
j.fischer@hs-mannheim.de

Willkommen zur Vorlesung

Informatik 1 für Mechatroniker



Vorstellung

Zu meiner Person...

Überblick

Inhalt

1 - Organisatorisches

1 - Organisatorisches

Kapitel 1 - Organisatorisches

- Übungen
- Online-Plichtübungen
- Life-Testate (Pflichtübungen)
- Prüfungsvoraussetzung
- Fragen an mich

1 - Organisatorisches

Übungen

- Übungsstunden finden in 2'er oder 3'er Gruppen statt
- In den Übungen besteht Anwesenheitspflicht!
- Wenn ihr krank seid reicht bitte eine Krankmeldung vom Arzt ein.
- 2 mal unentschuldigt fehlen führt zum Ausschluss von der Klausur
- Wer sich letztes Semester schon für die Klausur qualifiziert hat wende sich bitte per Mail an mich!
- Es gibt alle zwei Wochen neue Übungsblätter

1 - Organisatorisches

Online-Pflichtübungen

- Die Online-Pflichtübungen sollte jeder für sich machen.
- Sie sollen sicher stellen, dass **jeder** von Ihnen die nötigen Kenntnisse für die Klausur erworben hat.
- Verzögertes testieren bzw. bearbeiten der Pflichtübungen ist nur mit ärztlichem Attest möglich!
- Es gibt alle zwei Wochen neue Pflichtübungen

1 - Organisatorisches

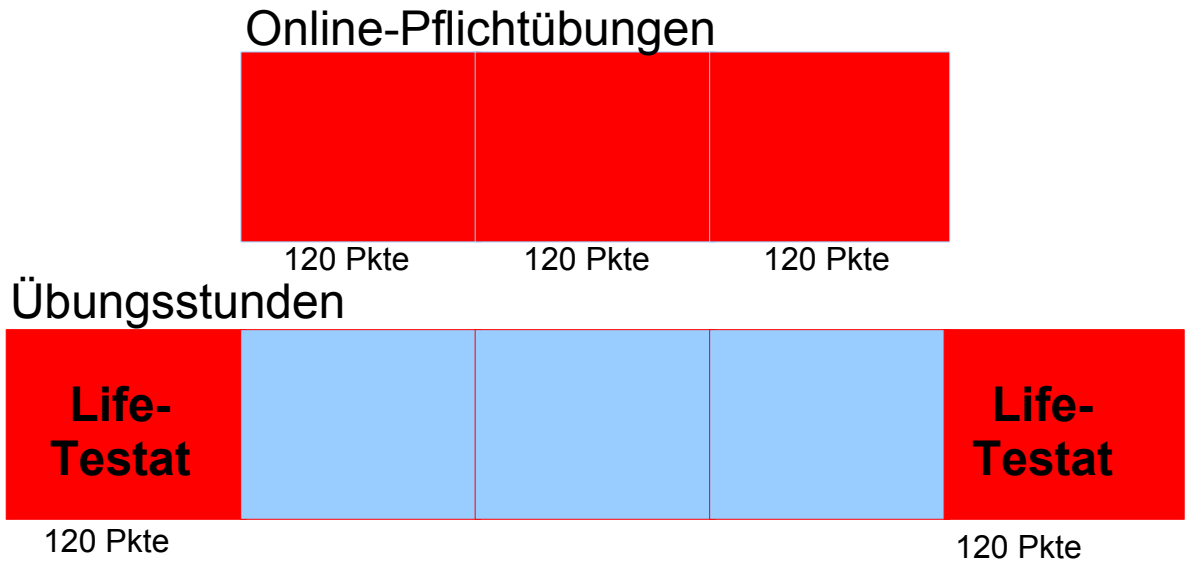
Life-Testate (Pflichtübungen)

- Die 1. Übung wird im Robotiklabor statfinden. Dazu werden 2 Termine bekanntgegeben. Den ersten Termin kommen bitte alle, die in Gruppe A eingeteilt sind, zum zweiten Termin alle die in Gruppe B eingeteilt sind.
- Die 5. Übung wird genauso in 2 Terminen ablaufen. Auch hier bitte zum ersten Termin die Studenten der Gruppe A und zum 2. Termin die in der Gruppe B.
- Die 1. und 5. Übung sind Life-Testate bei denen jeweils 120 Punkte erreicht werden können. Sie finden anstelle der Online-Plichtübungen statt.

1 - Organisatorisches

Prüfungsvoraussetzung

- Anwesenheit in den Übungsstunden
- Mindestens 500 von 600 Punkten in den **Pflichtübungen**



1 - Organisatorisches

Fragen an mich...

- Fragen können gerne auch per Email an mich gerichtet werden oder aber bei Diskussionsbedarf einfach per Mail einen Termin vereinbaren...
- Rückkopplung ist ausdrücklich erwünscht !!!!!
- Zwischen den Vorlesungen bin ich oft kurz angebunden; darum besser in den Übungsstunden fragen.

1 - Organisatorisches

Haben Sie Fragen?

2 - Einführung

Kapitel 2 - Einführung

- Welches Buch kann ich lesen?
- Wie funktioniert ein Computer?
- Welche Zahlensysteme werden benutzt?
- Die Programmiersprache C und C++.

2 - Einführung

Welches Buch kann ich lesen

- Es gibt unglaublich viele Bücher über C/C++ (auch online)
- Sinnvoll für dieses Semester ist ein Buch über reine C-Programmierung (**kein C++**)
- Auf **gar keinen Fall** ein Buch über „Microsoft Visual C++“
- Sinnvoll für dieses Semester ist, wenn Sie sich erst einmal ein Buch aus der Bibliothek über C-Programmierung ausleihen.
- Auf meiner Homepage gibt es auch noch ein WIKI zur C-Programmierung (leider lange nicht aktualisiert)

2 - Einführung

Welches Buch kann ich lesen

- The C Programming Language, second edition, by Brian Kernighan and Dennis Ritchie (**schweres aber umfassendes Standardwerk**) (http://cgip.inf.unideb.hu/eng/rtornai/Kernighan_Ritchie_Language_C.pdf)
- http://de.wikibooks.org/wiki/C-Programmierung
- RRZN: Die Programmiersprache C.
  Erhältlich im Rechenzentrum bei Frau Fontagnier
- M. Dausmann, U. Bröckl, J. Goll: C als erste Programmiersprache
  Elektronisch in unserer Bibliothek vorhanden

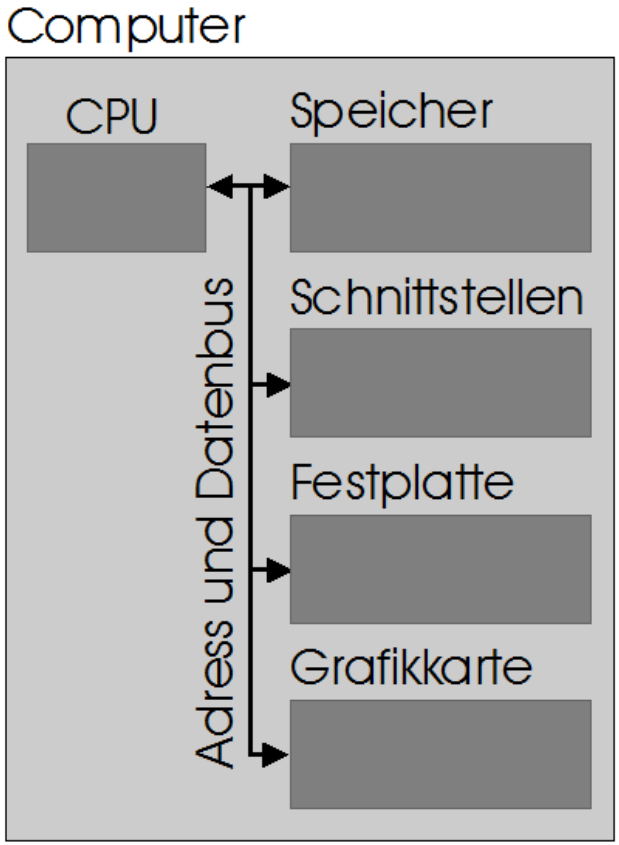
2 - Einführung

Wie funktioniert ein Computer

- Ein Computer folgt dem EVA Prinzip: Eingabe, Verarbeitung, Ausgabe
- Im Computer gibt es i.a.
  - CPU (Central Processing Unit), die Recheneinheit
  - RAM (Random Access Memory)
  - ROM (Read only Memory) bzw. Flash-Memory für das BIOS (Basic Input Output System)
  - Festplattenspeicher oder SSD (Solid State Disk)
  - Grafikkarte
  - Interfaces (Schnittstellen)
  - Bus Systeme (z.B. USB = Universal Serial Bus, PCI-Express) meistens bestehend aus Daten- und Adressbus

2 - Einführung

Wie funktioniert ein Computer

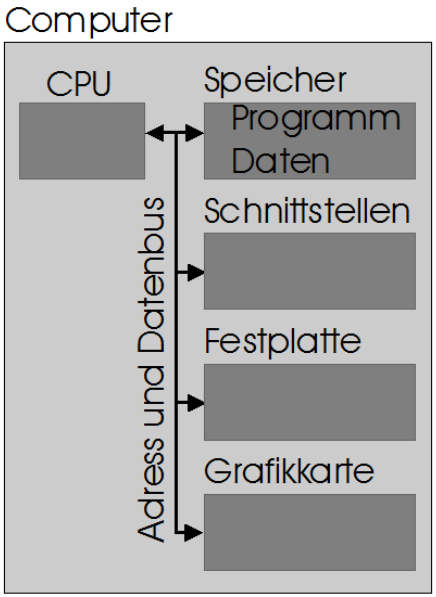


2 - Einführung

Wie funktioniert ein Computer

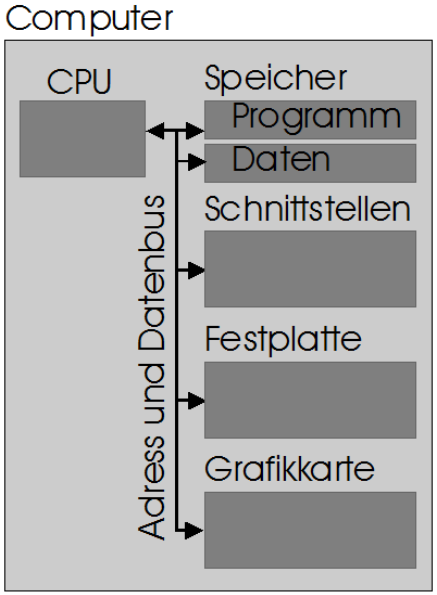
- Es gibt unzählige Computerarchitekturen. Wichtig für die Programmierung ist beispielsweise die Unterteilung in

Von Neumann Architektur



Moderner PC

Harvard Architektur



manche Digitalen Signal Prozessoren

2 - Einführung

Zahlensysteme und Umwandlung

- Zahlensysteme sind Stellenwert-Systeme
- Jede Stelle ist ein Vervielfachungsfaktor in Form einer Potenzzahl zugeordnet
- Beim Dezimalsystem ist jeder Stelle einer Zehnerpotenz zugeordnet (10 mögliche Ziffern)
- Beim Binärsystem ist jeder Stelle einer Zweierpotenz zugeordnet (2 mögliche Ziffern pro Stelle)
- Beim Hexadezimalsystem ist jeder Stelle einer Sechzehnerpotenz zugeordnet (16 mögliche Ziffern)

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Dezimal in Hexadezimal:

```
4025 / 16 = 251 Rest 9  -> 9 (letzte Stelle ganz rechts)
251  / 16 = 15  Rest 11 -> B
15   / 16 = 0   Rest 15 ->   (erste Stelle ganz links)
```

Dez	16^5	16^4	16^3	16^2	16^1	16^0
	1048576	65536	4096	256	16	1
4025	0	0	0	F	B	9

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Dezimal in Binär:

```
185 / 2 = 92 Rest 1 (Least Significant Bit -> LSB)
92  / 2 = 46 Rest 0
46  / 2 = 23 Rest 0
23  / 2 = 11 Rest 1
11  / 2 = 5  Rest 1
5   / 2 = 2  Rest 1
2   / 2 = 1  Rest 0
1   / 2 = 0  Rest 1 (Most Significant Bit -> MSB)
```

	2^7	2^6	s	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
Binär	1	0	1	1	1	0	0	1

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Binär in Dezimal:

```
185 = 128 + 32 + 16 + 8 + 1
```

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
Binär	1	0	1	1	1	0	0	1

- Summiert man die Zahlen, über den Einsen einer Binärzahl, so erhält man die Dezimalzahl (s.o.)

Alternativ kann man auch verschachtelt vorgehen:

185 = (((((((1)*2+0)*2+1)*2+1)*2+1)*2+0)*2+0)*2+1)

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Binär in Hexadezimal:

$1011_{bin} = 11_{dez} = B_{hex}$

$1001_{bin} = 9_{dez} = 9_{hex}$

$1011\ 1001_{bin} = B9_{hex}$

	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
	8	4	2	1	8	4	2	1
Binär	1	0	1	1	1	0	0	1

- Man teilt die Binärzahl von rechts nach links in 4‘er Gruppen und wandelt jede Hex. Ziffer einzeln um

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Hexadezimal in Binär:

$B9_{hex}$ enthält Ziffern „B“ und „9“

$B_{hex} = 11_{dez} = 1011_{bin}$

$9_{hex} = 9_{dez} = 1001_{bin}$

$B9_{hex} = 1011\ 1001_{bin}$

	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
	8	4	2	1	8	4	2	1
Binär	1	0	1	1	1	0	0	1

- Man wandelt jede Hex. Ziffer einzeln in Binär um!

2 - Einführung

Zahlensysteme und Umwandlung

Wandlung von Hexadezimal in Dezimal:

$FB9_{hex}$ enthält Ziffern „F“, „B“ und „9“

$F_{HEX} = 15_{dez}$

$B_{hex} = 11_{dez}$

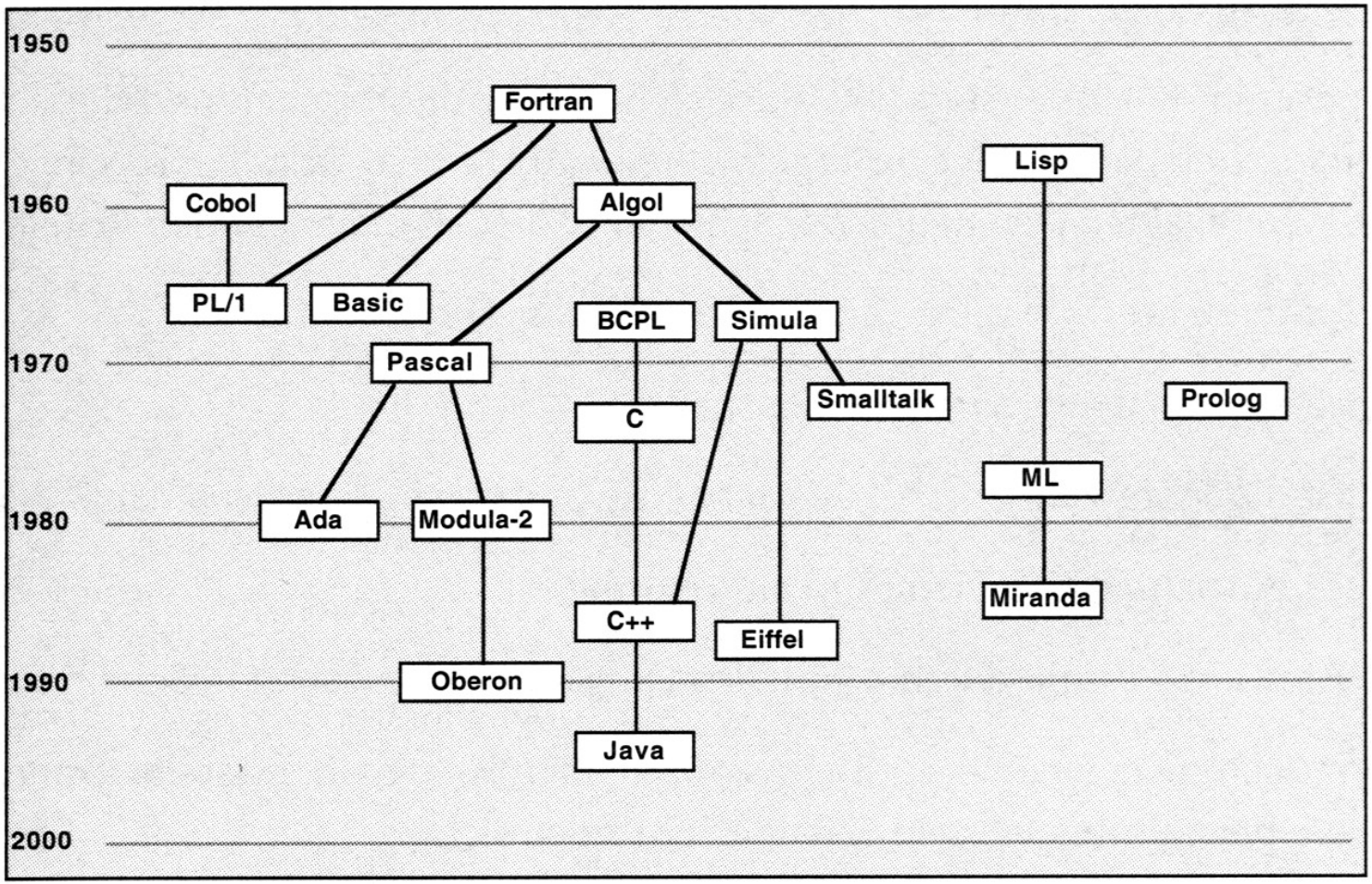
$9_{hex} = 9_{dez}$

Dez	16^5	16^4	16^3	16^2	16^1	16^0
	1048576	65536	4096	256	16	1
4025	0	0	0	F	B	9

$FB9_{hex} = 15 * 256 + 16 * 11 + 9 * 1 = 4025_{dez}$

2 - Einführung

Die Programmiersprache C und C++



2 - Einführung

Die Programmiersprache C und C++

- Entwicklung in Assembler sehr aufwendig (schlecht portier- und wartbar) → Fortran
- Programmiersprache B, angelehnt an BCPL (Basic Combined Programming Language)
- Entwicklung von C, im Gegensatz zu B typisiert
- Viele Betriebssysteme in C implementiert:
- Unix-Kernel, Linux, Windows
- 1979: Dennis Ritchie und Brian Kernighan: The C Programming Language Quasi-Standard
- 1988: Standardisierung durch ANSI-Komitee? ANSI-C bzw. C-89

2 - Einführung

Die Programmiersprache C und C++ (warum?)

- einfach
- portierbar
- sehr performant (vergleichbar mit Assembler)
- hardwarenah (wichtig für Embedded-Anwendungen → "Hochsprachen-Assembler„)
- Standardbibliothek auf allen Compilern verfügbar
  - Ein-/ Ausgabe, Dateioperationen
  - Zeichenkettenverarbeitung
  - Mathematik
  - Speicherverwaltung

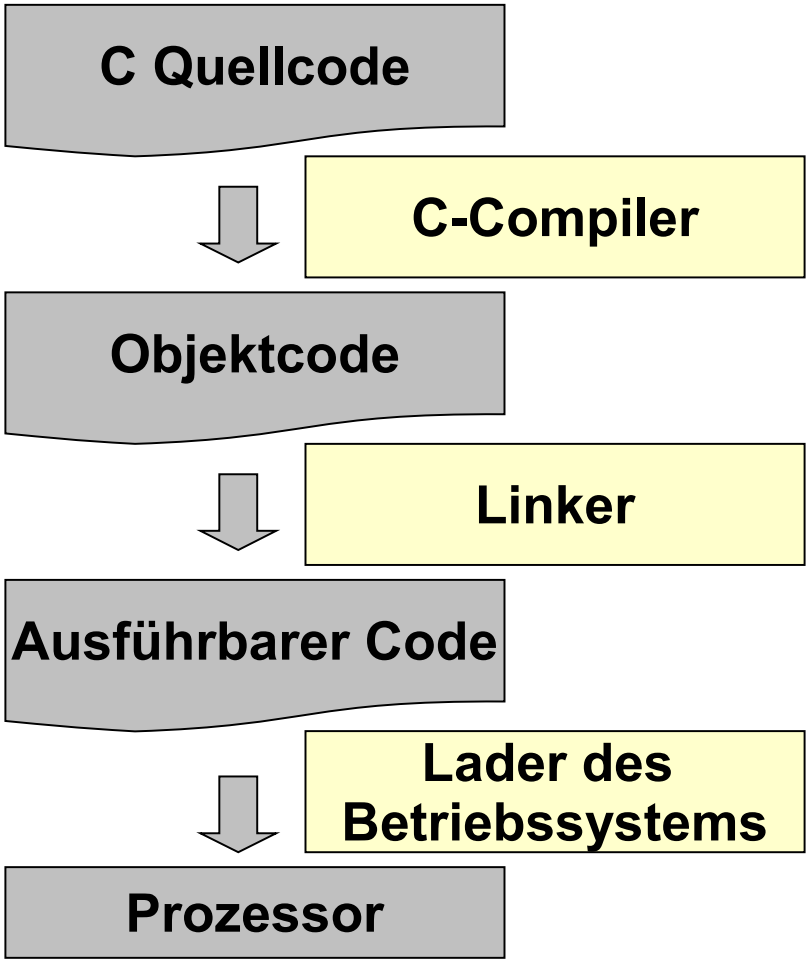
2 - Einführung

Die Programmiersprache C und C++ (aber...)

- fehlende Objektorientierung → Trend zu C++
- fehlende standardisierte Klassenbibliotheken für
  - GUI-Programmierung (z.B. QT)
  - Komplexe Datenstrukturen (Listen, Bäume ...)
- hoher Programmieraufwand
- "Low Level-Programmierung"
  - von Vorteil für Embedded-Anwendungen
  - von Nachteil für große Softwaresysteme

2 - Einführung

Was passiert mit einem selbst geschriebenen Programm?



- C-Programme liegen als Quellcode vor
- Ein Programm kann aus mehreren Quellcodedateien bestehen.
- int main(int argc, char argv[][])wird beim Starten automatisch aufgerufen
- Wir werden zunächst nur mit Programmen arbeiten, die aus einer einzelnen Datei bestehen.

3 - Programmentwurf

3 - Programmentwurf

Kapitel 3 - Programmentwurf

- Algorithmenbegriff
- Nassi-Schneiderman Diagramm
- Flußdiagramm
- Pseudocode

3 - Programmentwurf

Algorithmenbegriff:

- Ein **Algorithmus** ist ein Verfahren mit einer
  - präzisen (d.h. in einer genau festgelegten Sprache formulierten)
  - endlichen Beschreibung unter Verwendung
    - effektiver (d.h. tatsächlich ausführbarer)
    - elementarer Verarbeitungsschritte
- Ein Algorithmus benötigt nur endlich viele Ressourcen:
  - Rechenzeit
  - Speicher

Ein Algorithmus ist unabhängig von der Programmiersprache, in der er programmiert wird!

3 - Programmentwurf

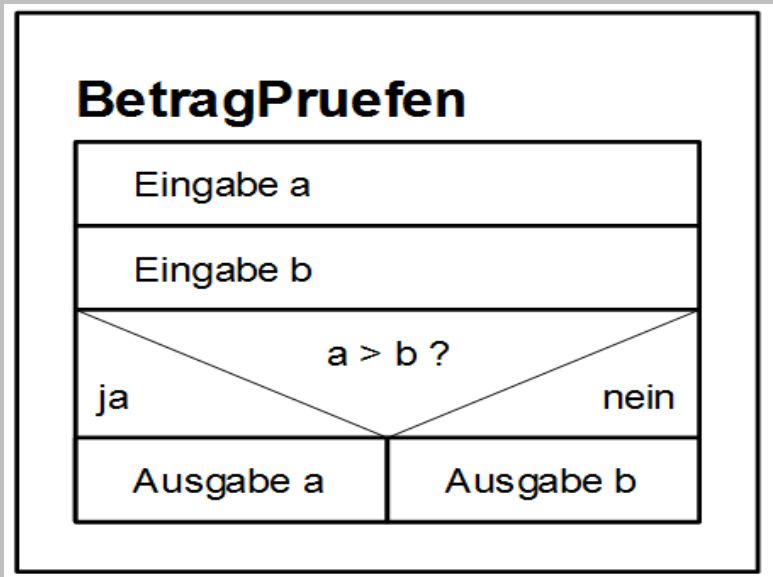
Nassi-Schneiderman Diagramm

3 - Programmentwurf

Nassi-Schneiderman Diagramm

- 1973 eingeführt von I. **Nassi** und B. **Shneiderman** als Darstellungsmittel für Algorithmen
- 11 Symbole + Erweiterungen
- DIN 66261

Beispiel:



3 - Programmentwurf

Nassi-Schneiderman Diagramm

Prozess / Anweisung

(Zuweisungen, Berechnungen, Ein-/Ausgabebefehle, Unterprogrammaufrufe)

Mehrere Arbeitsschritte werden durch mehrere Strukturblöcke dargestellt

Beispiel:

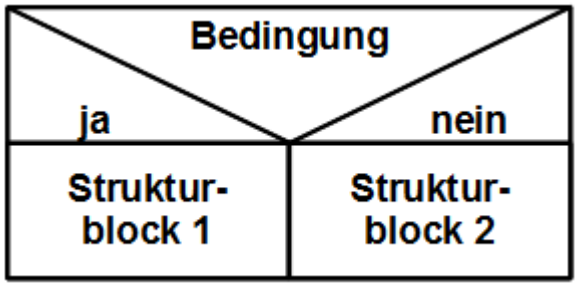
a = b + 1
Eingabe Gewicht
Berechne Preis

C-Code:

```
a = b + 1;
printf("Bitte Gewicht eingeben\n");
scanf("%f",weight);
calculatePrice();
```

3 - Programmentwurf

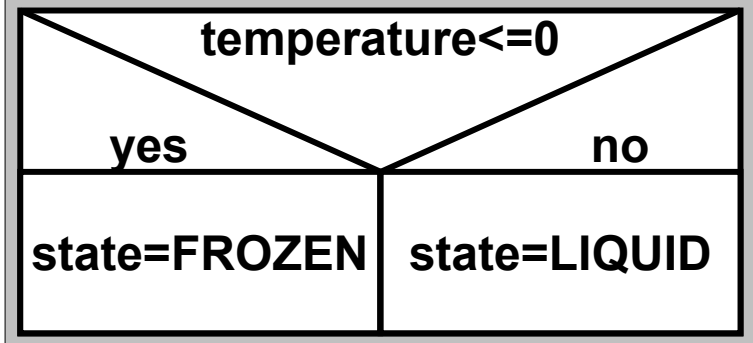
Nassi-Schneiderman Diagramm



Alternative, Verzweigung, Fallunterscheidung

- Bedingung erfüllt → weiter bei Strukturblock 1
- Bedingung nicht erfüllt → weiter bei Strukturblock 2

Beispiel:

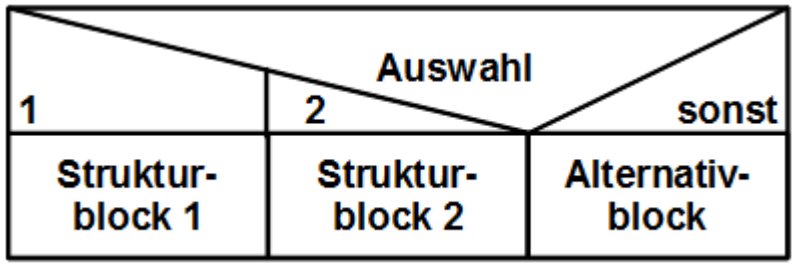


C-Code:

```
If (temperature<=0){
    state = FROZEN;
}
else{
    state = LIQUID;
}
```

3 - Programmentwurf

Nassi-Schneiderman Diagramm

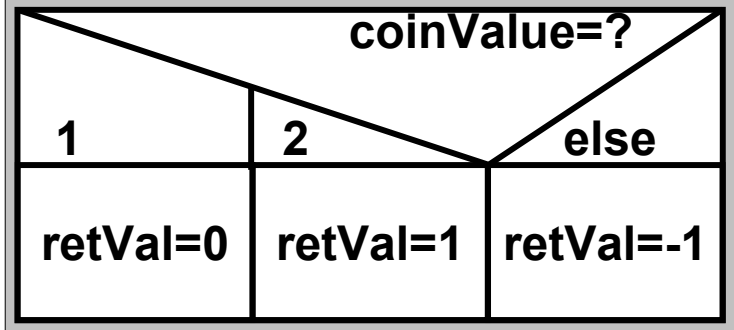


Mehrfachauswahl, Fallunterscheidung

- Bedingung jeweils erfüllt → weiter bei jeweiligem Strukturblock

  Bedingung nicht erfüllt → weiter bei Alternativblock

Beispiel:

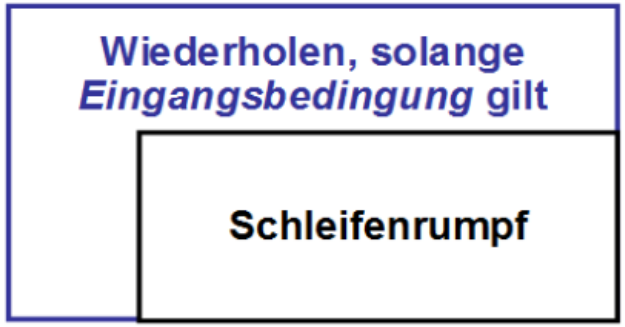


C-Code:

```
switch (coinValue){
    case 1 :{retVal=0; break;}
    case 2 :{retval=1; break;}
    default:{retval=-1};
}
```

3 - Programmentwurf

Nassi-Schneiderman Diagramm

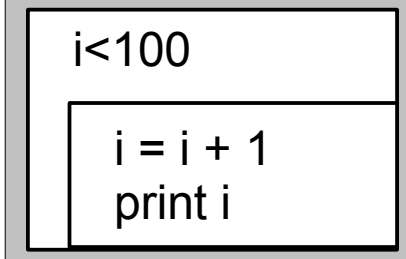


Bedingung steht im Schleifenkopf
- kopfgesteuerte Schleife
- abweisendes Verhalten

Abweisende Schleife:

- Falls *Eingangsbedingung* **nie** erfüllt : Schleifenrumpf 0x ausgeführt
- Falls *Eingangsbedingung* **immer** erfüllt : Endlosschleife

Beispiel:

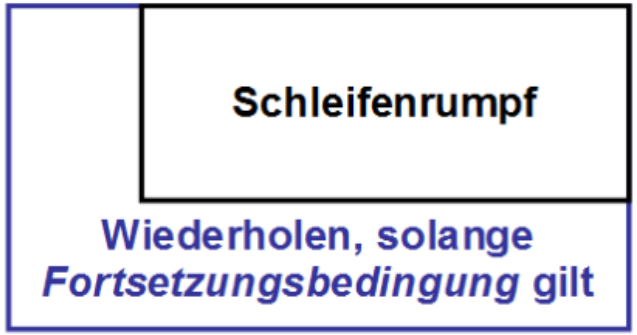


C-Code:

```
while(i<100){
    i = i + 1;
    printf("%d",i);
}
```

3 - Programmentwurf

Nassi-Schneiderman Diagramm

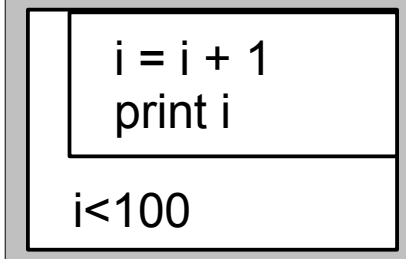


Bedingung steht im Schleifenfuß
- fußgesteuerte Schleife
- akzeptierendes Verhalten

Akzeptierende Schleife:

- Falls *Fortsetzungsbedingung* nie erfüllt : Schleifenrumpf genau 1x ausführen
- Falls *Fortsetzungsbedingung* immer erfüllt : Endlosschleife

Beispiel:



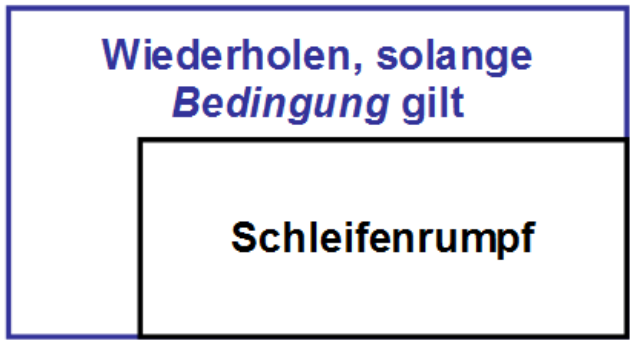
C-Code:

```
do{
    i = i + 1;
    printf("%d",i);
}while(i<100);
```

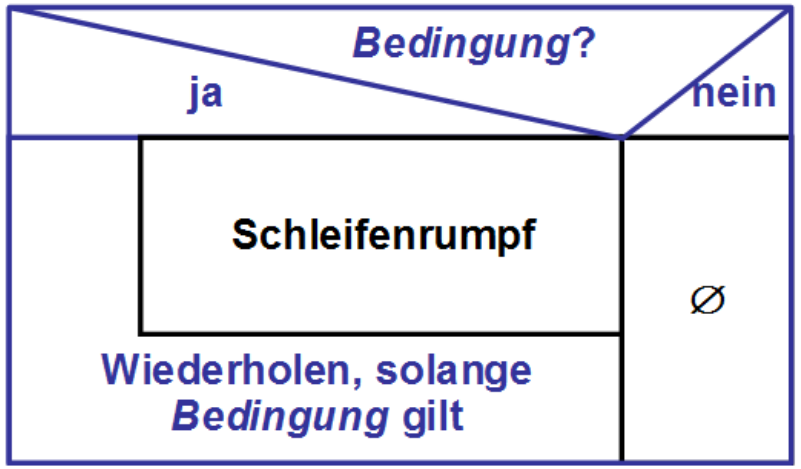
3 - Programmentwurf

Nassi-Schneiderman Diagramm

Abweisende Schleife …



...realisiert mit akzeptierender Schleife

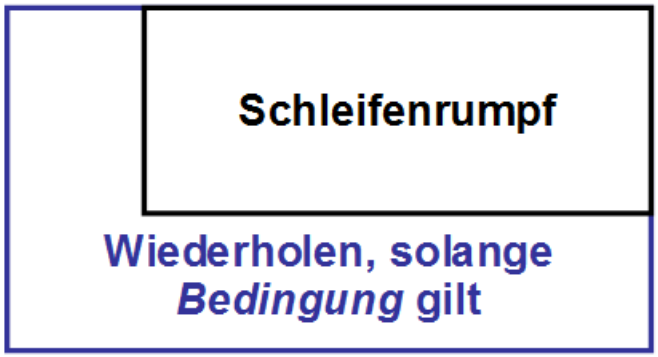


Man kann akzeptierende Schleifen z.B. in abweisende Schleifen überführen…

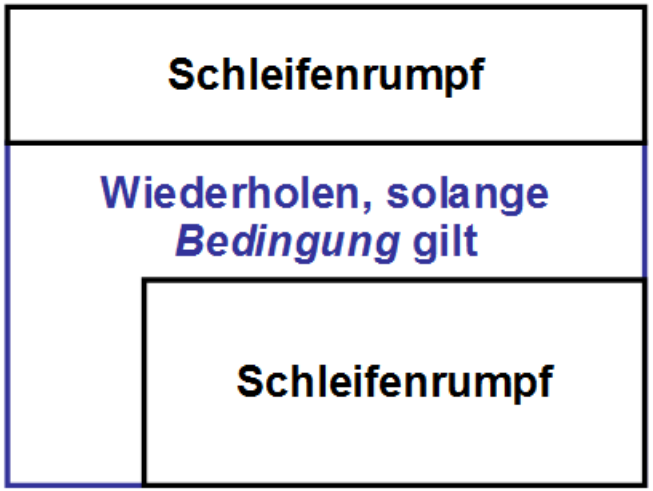
3 - Programmentwurf

Nassi-Schneiderman Diagramm

Akzeptierende Schleife …



...realisiert mit abweisender Schleife

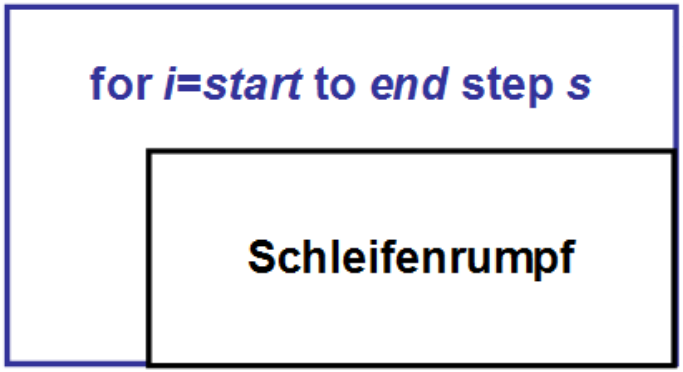


…oder aber abweisende Schleifen in akzeptierende Schleifen überführen…

3 - Programmentwurf

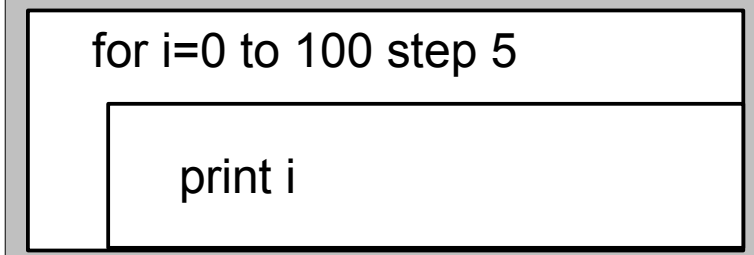
Nassi-Schneiderman Diagramm

Zählschleife:



Wertebereich für Laufvariable *i*
Steht im Schleifenkopf

Beispiel:



C-Code:

```
for (i=0; i<=100; i+=5){
    printf("%d",i);
}
```

3 - Programmentwurf

Nassi-Schneiderman Diagramm

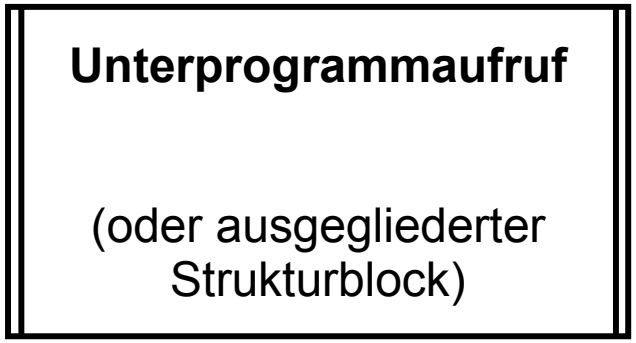
Zählschleife:

- Für jeden Wert *i* von *start* bis *end* mit Schrittweite *s*
  - Schleifenrumpf ausführen
    - Schleifenzähler *i* um Schrittweite *s* erhöhen
    - **step *s*** kann weggelassen werden → Defaultwert *1*
    - *s* kann auch negativ sein → rückwärts zählen → nur sinnvoll, wenn *end<start*
- Falls *s < 0* : rückwärts zählen
- Falls *s==0* : nicht sinnvoll
- Falls *s > 0* : vorwärts zählen

3 - Programmentwurf

Nassi-Schneiderman Diagramm

Unterprogrammaufruf:

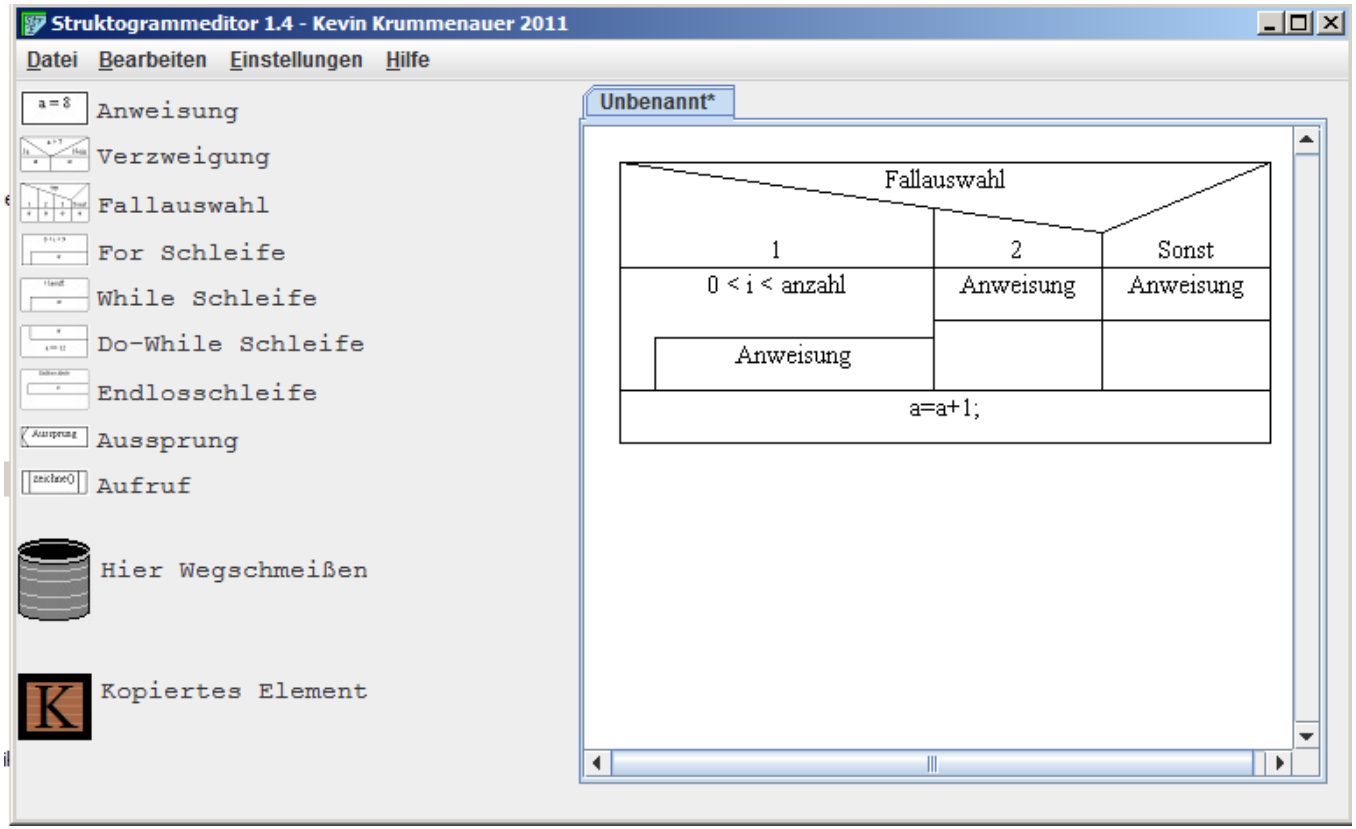


- Aufruf
  - Unterprogramm, Funktion, Prozedur. Methode
- aber auch „wenn der Platz nicht reicht“
  - in dem Fall aber besonders kennzeichnen
  - Struktogramme sollen auf DIN A4 passen

3 - Programmentwurf

Nassi-Schneiderman Diagramm

http://whiledo.de/programm.php?p=struktogrammeditor



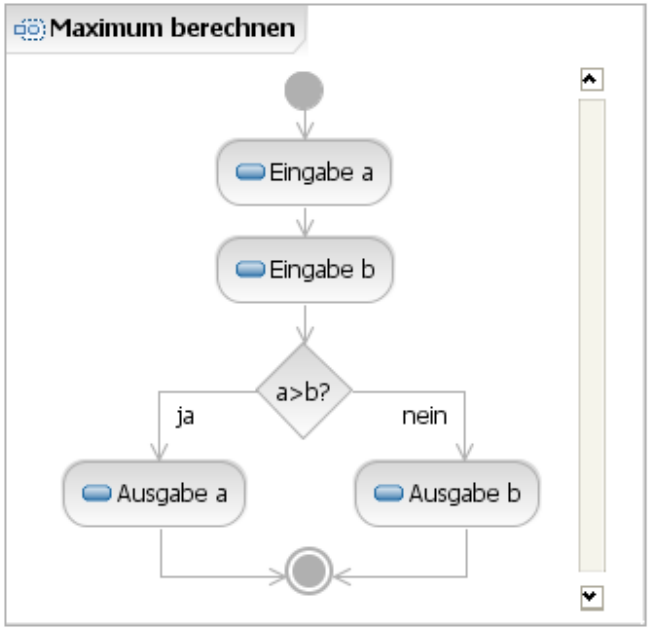
3 - Programmentwurf

Flußdiagramm

3 - Programmentwurf

Flußdiagramm

- Standardisierte Notation für Abläufe
  - Bestandteil der UML (Unified Modeling Language)
  - Basiert auf Programmablaufplan / Flussdiagramm
- Beispiel:



3 - Programmentwurf

Flußdiagramm

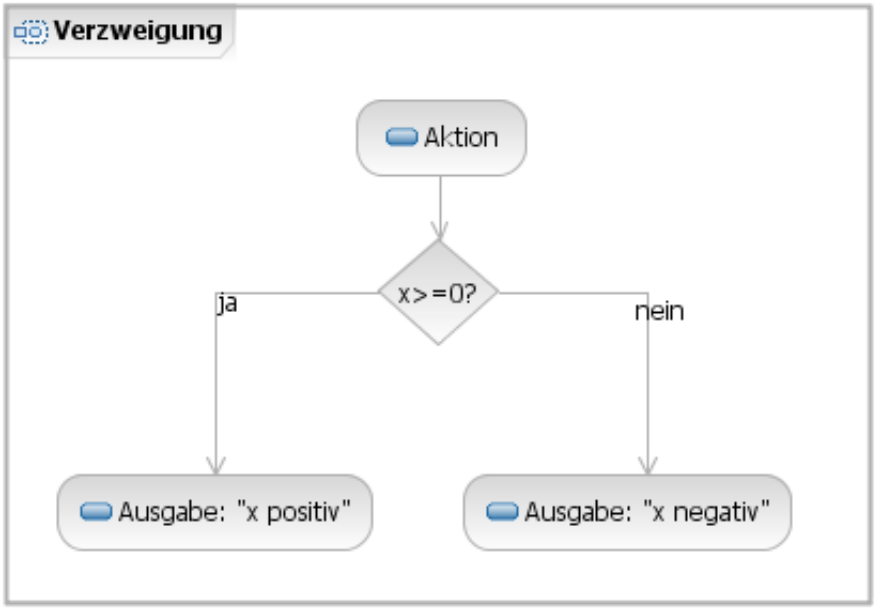
- Grafische Darstellung als geschlossener Block
  - Muss mit Startknoten ● beginnen
  - Muss mit Endknoten ◉ enden
  - Aktionen: abgerundete Rechtecke 
  - Fallunterscheidung: Raute mit Bedingung 

  - Schleife: Fallunterscheidung mit Rücksprung
  - Programmfluss: Kanten ←

3 - Programmentwurf

Flußdiagramm

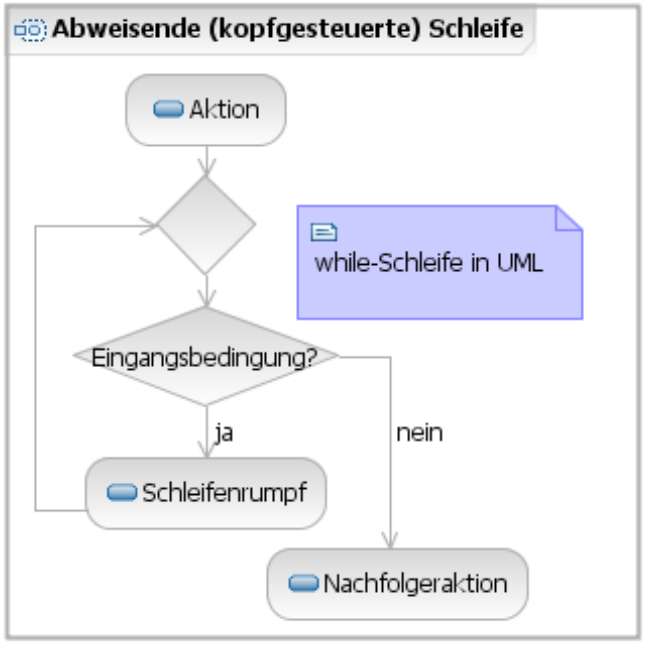
- Verzweigungen werden durch Rauten dargestellt
  - Ja-Ausgang wird genommen, wenn Bedingung zutrifft
  - Nein-Ausgang wird genommen, wenn Bedingung nicht zutrifft
  - Bei Fallunterscheidungen entsprechend weitere Ausgänge

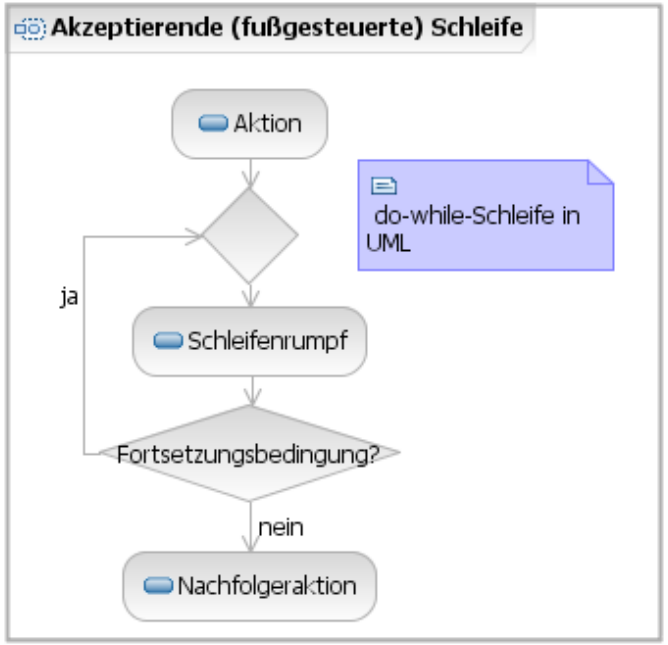


3 - Programmentwurf

Flußdiagramm

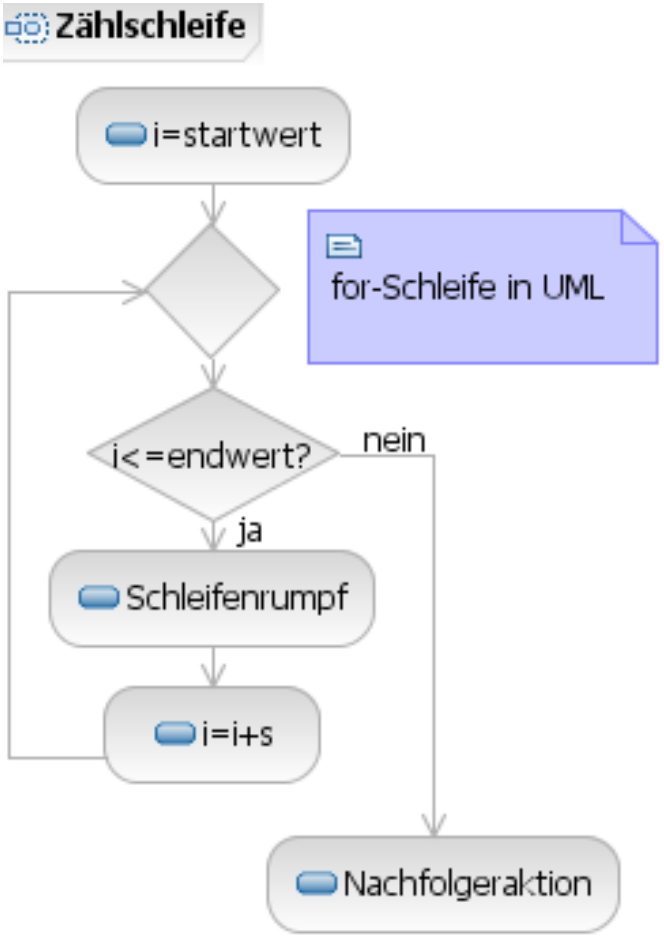
- Schleife mit Verzweigung und Zusammenführung simulieren
  - Bedingungsabfrage durch Verzweigung
  - Rücksprung an Schleifeneintritt durch Zusammenführung
  - Immenser Platzverbrauch im Diagramm





3 - Programmentwurf

Flußdiagramm



Zählschleife:

for *i* = *startwert* to *endwert* step *s*

Schleifenrumpf ausführen

Falls *s<= -1* : rückwärts zählen

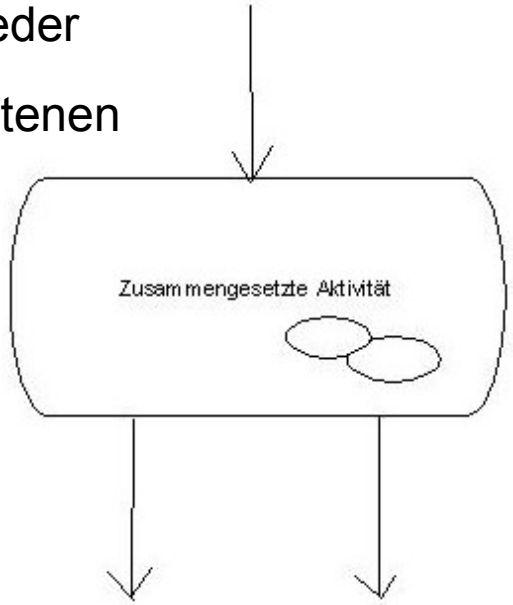
Falls *s== 0* : nicht sinnvoll

Falls *s>= 1* : vorwärts zählen

3 - Programmentwurf

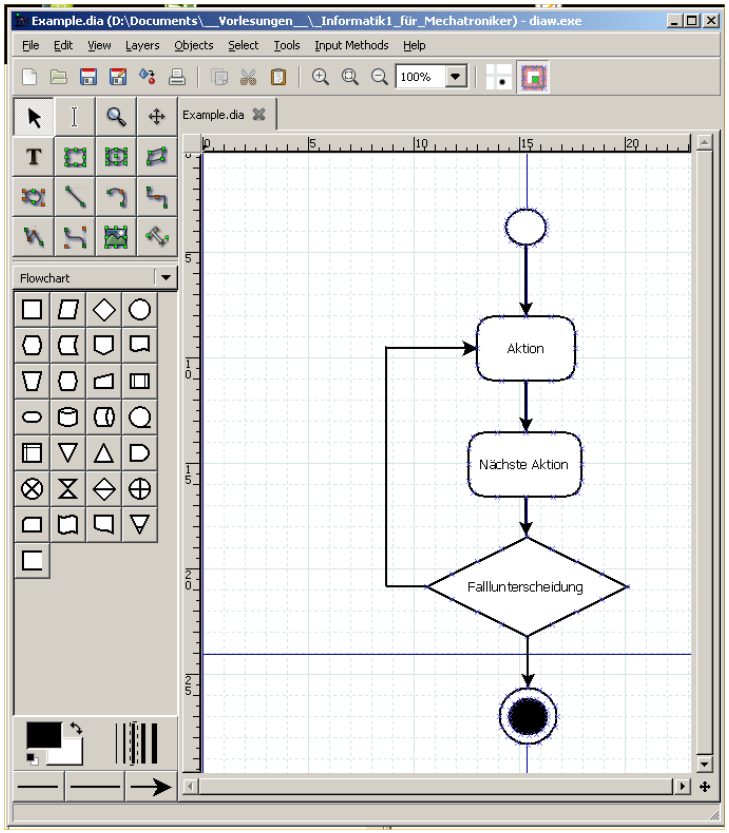
Flußdiagramm

- Aktivitäten lassen sich hierarchisch schachteln
- Eine Aktivität kann wieder aus einer Menge von Detail-Aktivitäten (ggf. dargestellt in weiterem Aktivitätsdiagramm) aufgefasst werden
- **Wichtig:** Die Ein- und die Ausgänge müssen dann aber dementsprechend übereinstimmen!
- Diese Methode spiegelt das Divide and Conquer wieder
- Als Symbol wird ein Aktivitätssymbol mit zwei enthaltenen Aktivitätssymbolen verwendet



3 - Programmentwurf

Flußdiagramm



Dia-Portable

- Auf www.portableapps.com
- Freeware
- Drag‘n drop
- beherrscht auch andere UML Darstellungen

3 - Programmentwurf

Pseudocode

3 - Programmentwurf

Pseudocode:

- Pascal-ähnliche Notation:

```
begin BetragPruefen
Eingabe(a);
Eingabe(b);
if a > b then
    Ausgabe(a);
else
    Ausgabe(b);
end if
end BetragPruefen
```

4 - Erste Schritte

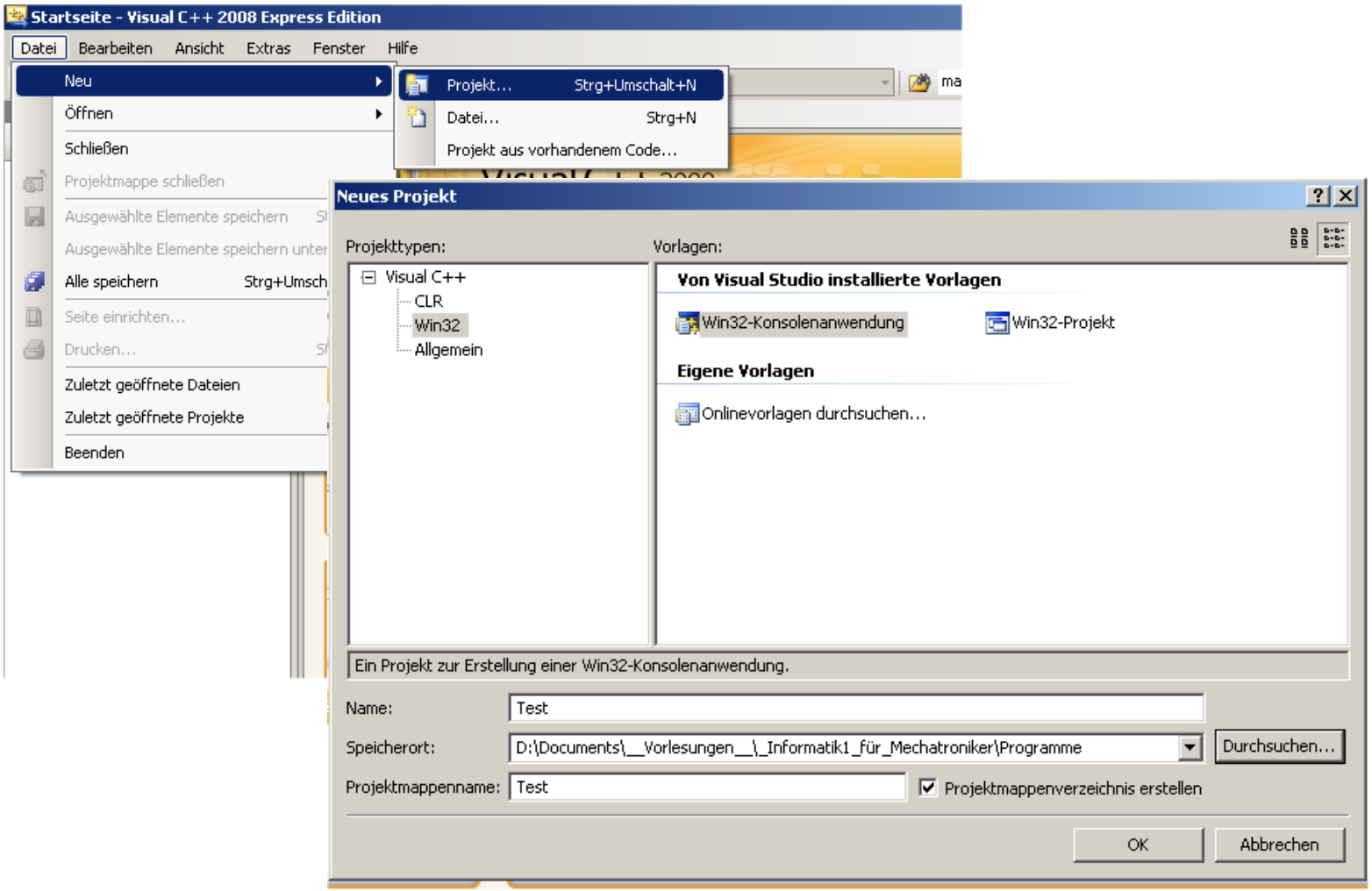
4 – Erste Schritte

Kapitel 4 – Erste Schritte

- Ein erstes Visual C Projekt
- Kommentare
- Namenskonventionen

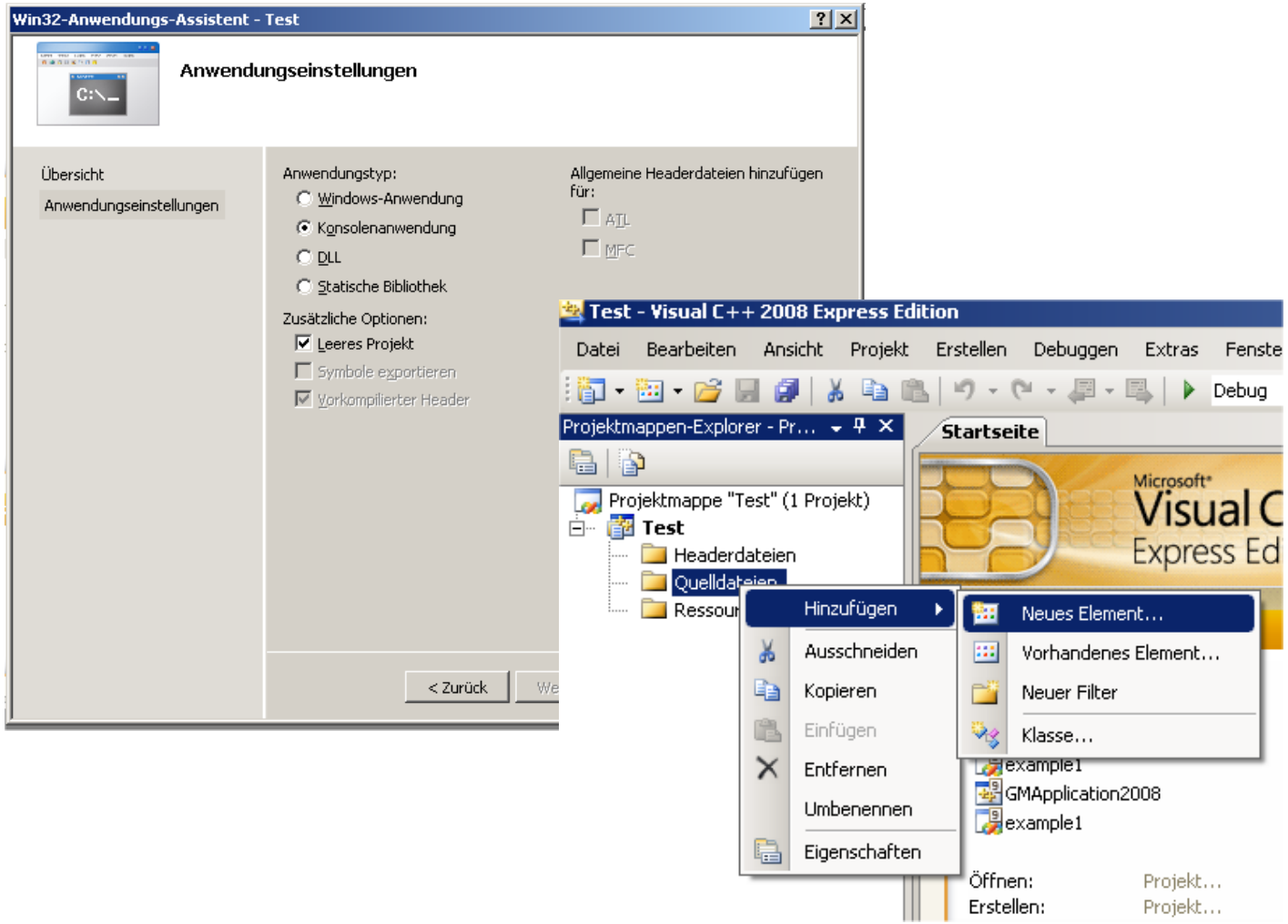
4 - Erste Schritte

Ein erstes Visual C Projekt



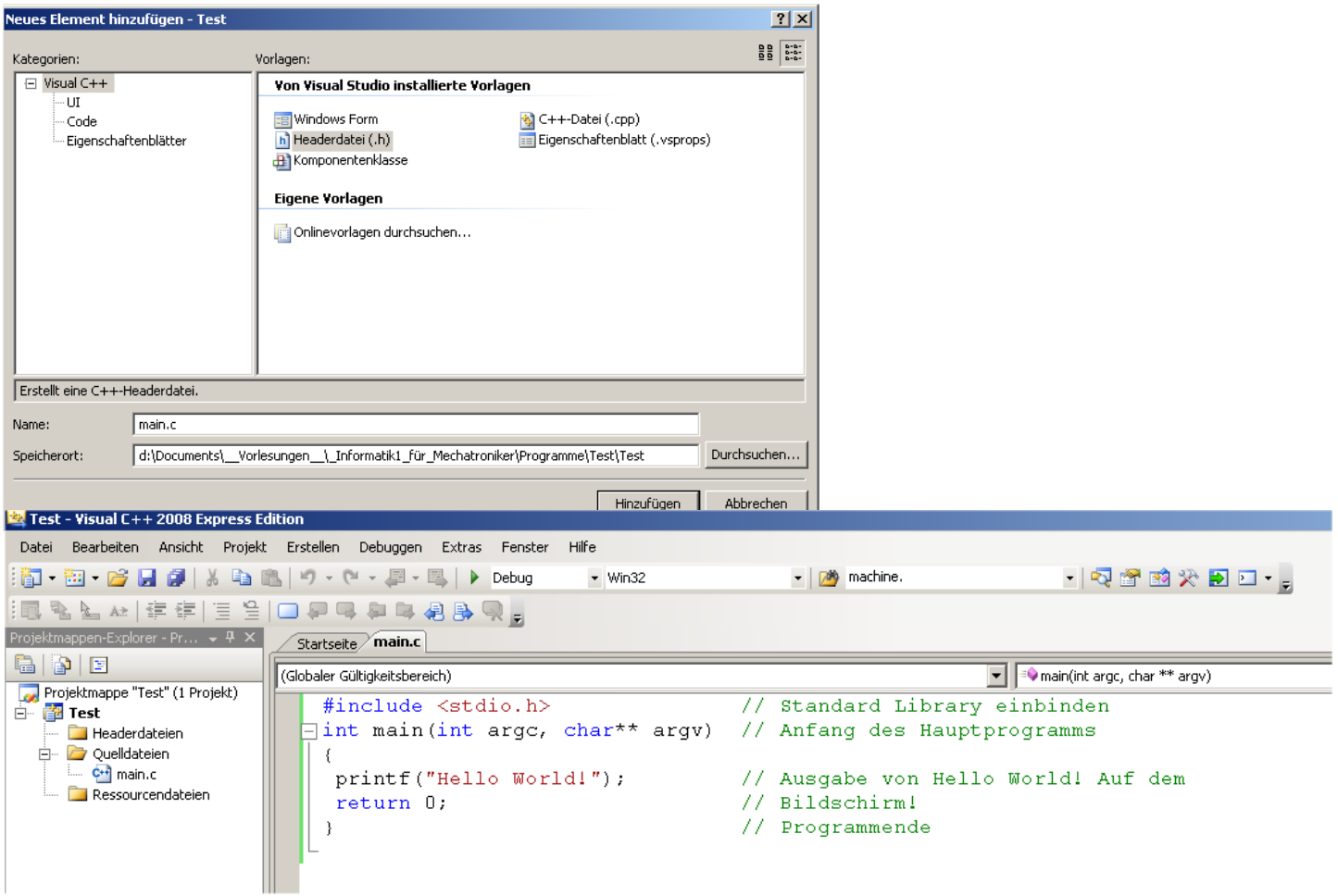
4 - Erste Schritte

Ein erstes Visual C Projekt



4 - Erste Schritte

Ein erstes Visual C Projekt



4 - Erste Schritte

Ein erstes Visual C Projekt

Das erste Programm:

```
#include <stdio.h>                  // include standard library
int main(int argc, char** argv) // start of the main program
{
 printf("Hello World! ");          // output of „Hello World!“ on screen
 return 0;                         // end program without error

}
```

Achtung: C unterscheidet Groß-/Kleinschreibung !!!

4 - Erste Schritte

Kommentare

- mit // wird der Rest der Zeile auskommentiert
- mit /* Kommentar */ wird alles zwischen /* und */ auskommentiert und vom Compiler nicht beachtet
- Für die Übung bitte über jedes Programm eine kurze Beschreibung was es macht und welche Funktionen enthalten sind und…
- …über jede Funktion eine kurze Beschreibung was sie tut, was jeder Übergabeparameter bedeutet und was der Rückgabewert bedeutet

4 - Erste Schritte

Kommentare (warum?)

- Software wird meist in Teams erstellt
  - Andere müssen Ihren Code verstehen und nachvollziehen
    - um Fehler zu beheben
    - um Ihre Programme weiter zu entwickeln
  - Sie selbst müssen auch nach längerer Zeit Ihren Code noch
    - verstehen
    - erweitern
    - verbessern
    - Debuggen
- 60 – 70% aller Entwicklungsarbeiten und –kosten sind Wartung und Weiterentwicklung!

4 - Erste Schritte

Kommentare (wie?)

- Jede Funktion sollte in Zukunft auf **Englisch** dokumentiert werden (Beispiel: doxygen kompatible Dokumentation):

```
/*****************************************************************
 \brief <b>Description: </b> generateThread
      <br>  This function generates a thread.
 \param     threadNum   : is of type integer and defines the
                          Threadnumber...
 \return                : SUCCESS  = thread was generated
                          FAILED   = thread couldn't be generated
*****************************************************************/
char generateThread(int threadNum)
{
…
}
```

4 - Erste Schritte

Namenskonventionen

Sprache

- Kommentare und Variablen bitte auf englisch

Variablen sinnvoll benennen

- selbsterklärend
- nicht *i1* oder *k19*
- nicht übermäßig lang (Tipparbeit!)
- Beispiel: *maximum* statt *m*

Abweichung: Schleifenzähler + Hilfsvariablen kurz und prägnant, z.B.:

- *c* für einen *char*-Wert
- *d, e, f* für einen *double*-Wert
- *i*, *j*, *k* für *int*-Werte

4 - Erste Schritte

Namenskonventionen

Namenskonventionen

- werden von der Programmiersprache vorgegeben
- haben sich als vorteilhaft erwiesen
- sollen Verständnis fremden Codes erleichtern
- werden z.B. in mitgelieferten Funktionsbibliotheken eingehalten
- sind einzuhalten in Übungen und Klausur!

Lokale Variablennamen

- beginnen mit Kleinbuchstabe
- weiter mit Kleinbuchstaben
- bei neuem Wortstamm einen Großbuchstaben mittendrin,
  z.B. *numberProfessors*
- Keine Unterstriche (_) mehr → veraltet,
  z.B.: number*Professors statt number_professors*

4 - Erste Schritte

- Die „printf“ Anweisung

```
#include <stdio.h>
int main(int argc, char** argv)
{
 printf("Ganze Zahl = %d \n",5);
 printf("Komma Zahl = %f \n",5.5);
 printf("Buchstabe  = %c \n",'A');
}
```

Fromatierungszeichen	Beschreibung
%d	ganzzahlig
%f	Fließkomma
%c	Character (Zeichen)
%s	String
\n	Zeilenvorschub

- Die „scanf“ Anweisung

```
 #include <stdio.h>
 int main(int argc, char** argv)
 {
  int zahl; float kommaZahl; char buchstabe; // declare variable
  scanf("%d",&zahl);                         // read integer
  scanf("%f",&kommaZahl);                    // read float
  scanf("%c",&buchstabe);                    // read char
 }
```

4 - Erste Schritte

Und noch ein Programm…

```
#include <stdio.h>                      // include standard library
int main(int argc, char** argv)         // start of the main program
{
 int result;                            // variable declaration
 result = (3+4)*7;                      // calculate result
 printf("(3+4)*7=%d",result);           // output of (3+4)*7=49 on
                                        // screen!
 return 0;                              // end program without error

}                                       // end of program
```

5 - Die Programmiersprache C

5 - Die Programmiersprache C

Kapitel 5 – Die Programmiersprache C

- Schlüsselwörter
- Ausdrücke
- Anweisungen
- Token
- Literale

5 - Die Programmiersprache C

Schlüsselwörter:

asm	**auto**	**break**	**case**	*catch*
char	*class*	**const**	**continue**	**default**
delete	**do**	**double**	**else**	**enum**
except	**extern**	*finally*	**float**	**for**
friend	**goto**	**if**	*inline*	**int**
long	*new*	*operator*	*private*	*protected*
public	**register**	**return**	**short**	**signed**
sizeof	**static**	**struct**	**switch**	*template*
this	*throw*	*try*	**typedef**	**union**
unsigned	*virtual*	**void**	**volatile**	**while**

Zusätzlich in C++

5 - Die Programmiersprache C

Ausdrücke:

- liefern einen Wert als Ergebnis
- haben bestimmten Typ (z.B. *int*, *char*)

z.B.

- Literale (Wertkonstanten), z.B. *5.0*
- Variablen, z.B. *x*
- Arithmetische Ausdrücke, z.B.: *5*x+3*y*
  - liefern ganze Zahl oder Fließkommazahl
- Boolesche Ausdrücke, z.B.: *(x && y)* || *z*
  - liefern *true* (in C: *1*) oder *false* (in C: *0*)
- Vergleichsausdrücke, z.B.: *x>3*
  - liefern *true* (in C: *1*) oder *false* (in C: *0*)

5 - Die Programmiersprache C

Anweisungen

- definieren Programmablauf
- enthalten / nutzen Ausdrücke

z.B.

- Zuweisungen
- Folgen von Anweisungen
- Prozedur-, Funktions-, Methoden-Aufrufe
- Bedingte Abfragen
- Schleifen
- etc.

5 - Die Programmiersprache C

Token

- zusammenhängendes Wort über einem Alphabet
  - Alphabet: z.B. Buchstaben, aber auch: Zahlen, _

- getrennt vom nächsten Token durch Trennzeichen
  - Whitespaces: Leerzeichen, Zeilenumbruch, Tabulator
  - Weitere mögliche Trennzeichen:
    - Klammern
    - Komma
    - Semikolon

5 - Die Programmiersprache C

Literale

- konkrete Angabe eines Zahl-, Zeichen- oder Zeichenkettenwertes im Quellcode z.B. “Hello World 1234“
- Wertkonstante z.B. 6.4564

6 - Variablen und Datentypen

6 - Variablen und Datentypen

Kapitel 6 – Variablen und Datentypen

- Datentypen
- Variablen
- Arrays
- Strings in C

6 - Variablen und Datentypen

Datentypen

Datentyp	Wertebereich	Beschreibung
char	-128..127	ganzzahlig (1 ASC II Zeichen)
short	Bitbreite halb so groß oder gleich int meist: -32768..32767	ganzzahlig
int	Bitbreite des Targets mindestens 16 Bit	ganzzahlig
long int	Bitbreite doppelt so groß wie int	ganzzahlig
float	compilerabhängig	Fließkommazahl
double	compilerabhängig	Fließkommazahl

6 - Variablen und Datentypen

Datentypen

In C99 sind noch folgende Typen definiert in der <stdint.h>:

Datentyp	Wertebereich	Beschreibung
int8_t	-128..127	ganzzahlig
int16_t	-32768..32767	ganzzahlig
int32_t	-2147483648..2147483647	Ganzzahlig

- Zusätzlich gibt es die include Files: <limits.h> und <float.h> in denen die Min und Maxwerte der Datentypen stehen.
- Steht ein unsigned vor dem Datentyp, so beschreibt der neue Datentyp eine positive Zahl, dessen Wertebereich sich in den positiven Bereich verschiebt.

6 - Variablen und Datentypen

Variablen

- Platzhalter eines definierten Datentyps.
- Muss vor dem Gebrauch deklariert werden
- Sollte vor Gebrauch initialisiert werden

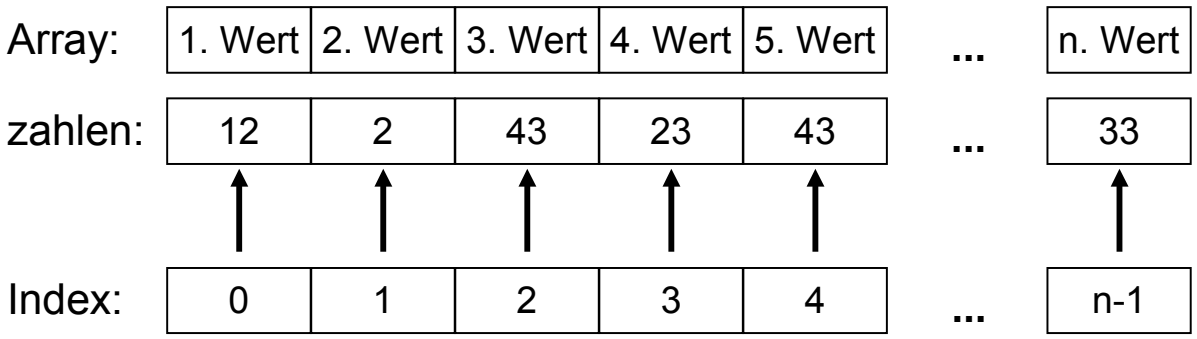
Beispiele:

```
float zahl1;
int   zahl2 = 5, zahl3 = 7;
```

6 - Variablen und Datentypen

Arrays (eindimensional)

- Array hat feste Länge, die bei seiner Definition festgelegt wird
- Auf die verschiedenen Elemente eines Arrays kann mittels Indizierung in beliebiger Reihenfolge zugegriffen werden
- In der Programmiersprache C beginnen die Indizes immer mit **0**



Beispiel für den Zugriff: zahlen[2] == 43
zahlen[n-1] == 33
zahlen[n] == **Fehler!**

6 - Variablen und Datentypen

Arrays (eindimensional)

- Arrays sind Variablen, werden also wie diese mittels einer Definition angelegt:
- Syntax: <datentyp> <variablenname>[groesse]
- Beispiele:
  - *int quadrate[30]; //* legt ein Array aus 30 Integers an
  - *float noten[40]; //* legt ein Array aus 40 Floats an
- Arraylänge nachträglich nicht mehr änderbar!

6 - Variablen und Datentypen

Arrays (mehrdimensional)

- Arrays können auch mehr als eine Dimension haben:

```
short feldVonElementen[5][3]; // 5 x 3 elements of type short

feldVonElementen[0][0]=-1;

feldVonElementen[4][1]=-1; // is neighbour to [4][0] and [4][2]

feldVonElementen[4][2]=-1; // last element!
```

- Die Anzahl der Klammern [ ] geben die Dimension vor.
- Bei dem am weitesten rechts stehende Index werden benachbarte Indizes auf benachbarte Speicherstellen abgebildet.

6 - Variablen und Datentypen

Strings in C

- String-Literal
  - beginnt und endet mit "Anführungszeichen"
  - dazwischen dürfen beliebig viele Zeichen (auch Escape-Codes) stehen
- Wenn String-Literal nicht in eine Zeile passt:
  - Zeile mit **** beenden
  - in nächster Zeile weiterschreiben
- Beispiele für String-Literale:
  - *"Hallo Mannheim"*
  - *"Einen schönen guten Morgen!\nHast Du ausgeschlafen?\n"*
- Anwendung z.B.:
  - *printf("Einen schönen guten Morgen!\nHast Du ausgeschlafen?\n");*

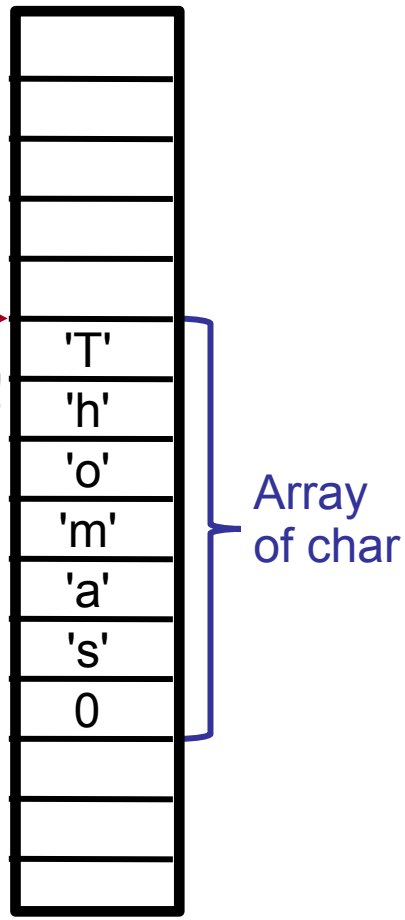
6 - Variablen und Datentypen

Strings in C

- sind ein Array of *char*
  - für jedes Zeichen ein Arrayelement
  - Stringende durch Nullbyte gekennzeichnet

char *myString = "Thomas";

  - String mit **n** Zeichen benötigt Array mit **n+1** Zeichen!
- Array = Zeiger auf ersten Buchstaben im Speicher



6 - Variablen und Datentypen

Strings in C

- als Array
  - ohne Initialisierung
    char vorname[20]; // Platz für 19 Zeichen plus Nullbyte
  - mit Initialisierung
    char vorname[]=„Jörn"; // 4 Zeichen plus Nullbyte
- als Zeigervariable (auf das erste Zeichen)
  - ohne Initialisierung und ohne Speicherbelegung
    *char *vorname;* // Kein Speicher für String belegt!
  - ohne Initialisierung, aber mit Speicherbelegung
    *char *vorname = malloc(20);* // Platz für 19 Zeichen plus Nullbyte
  - mit Initialisierung
    *char *vorname =„Jörn";* // 4 Zeichen plus Nullbyte

7 - Arithmetische Ausdrücke

7 - Arithmetische Ausdrücke

Kapitel 7 – Arithmetische Ausdrücke

- Auswertungsreihenfolge
- Rechnen mit unterschiedlichen Datentypen
- Operatoren
- Bitweise Operatoren

7 - Arithmetische Ausdrücke

Auswertungsreihenfolge

- Ein berechnender Ausdruck wird von links nach rechts ausgewertet
- Klammern werden von innen nach außen ausgewertet
- Zuweisungen werden von rechts nach links ausgewertet.

7 - Arithmetische Ausdrücke

Rechnen mit unterschiedlicher Datentypen

Datentypen werden automatisch einander angeglichen

- bei der Anwendung von Rechenoperationen

Beispiel:

```
3.0f + 4
3.0f + 4.0f
7.0f
```

- bei der Ausführung von Zuweisungen

Beispiel:

```
double d;
d = 3.0f + 4;
d = 3.0f + 4.0f;
d = 7.0f;
d = 7.0d;
```

7 - Arithmetische Ausdrücke

Rechnen mit unterschiedlicher Datentypen

Wenn Zieldatentyp „größer“ bzw. genauer als Datentyp des Wertes

- Automatische Datentypumwandlung
  short → int → long → float → double
  char

Wenn Zieldatentyp „kleiner“ bzw. ungenauer als Datentyp des Wertes

- Datentypumwandlung muss durch Type Cast erzwungen werden
- sonst: Verlust einzelner Bits oder des Vorzeichens!

- Compiler merkt das **nicht**, schluckt das kommentarlos!
- Schreibweise:
  (<*Zieldatentyp*>) <*Wert*>

Beispiel:

```
short sh = 18;
long l;
l = (long) sh;
```

7 - Arithmetische Ausdrücke

Rechnen mit unterschiedlicher Datentypen

Rundungsfehler:

- double → float: nächste darstellbare Zahl
- double → long, int, short oder char: Nachkommastellen abschneiden
- float → long, int, short oder char: " "

Rechen- und Vorzeichenfehler

- Ursache: Abschneiden überzähliger vorderer Bits
- long → int, short oder char: vordere 32, 48 oder 56 Bits abschneiden
- int → short oder char: " 16 oder 24 Bits abschneiden
- short → char: " 8 Bits abschneiden

Beispiele:

(char)256 → 0
(char)120 → 120
(char)128 → -128

7 - Arithmetische Ausdrücke

Operatoren (Auswertreihenfolge)

Bezeichnung	Operatorsymbol	Priorität
Klammern	() []	13
Negation Inkrement Dekrement	- ! ~ ++ --	12
Arithmetische Operatoren	* / %	11
	+ -	10
Shift Operatoren	<< >>	9
Vergleichsoperatoren	> >= < <=	8
	== !=	7
Bitweise Operatoren	&	6
	^	5
	|	4
Logische Operatoren	&&	3
	||	2
Zuweisungsoperator	= += -= *= /= %= >>= <<= &= ^= |=	1

7 - Arithmetische Ausdrücke

Operatoren

Für häufig gebrauchte Operationen:

- abkürzende Schreibweisen für Zuweisung
- **Beispiel:**
  a = a + 1;
  kürzer geschrieben:
  a += 1;

Das erlaubt die (bereits bekannte) Syntaxregel
AssignmentOperator: one of
= *= /= %= += -= <<= >>= &= ^= |=

Wert einer verkürzten Zuweisung = zugewiesener Wert

7 - Arithmetische Ausdrücke

Operatoren

Weitere abkürzende Schreibweisen der Zuweisung

- **Inkrement**-und **Dekrement**-Operator **++** und **--**
- **Beispiel:**
  a = a + 1; oder auch *a += 1;*
  kürzer geschrieben:
  *a**++**;*
  oder
  ***++**a;*

Präfix-Operatoren: Stehen vor Variable, z.B: **++**a

Postfix-Operatoren: Stehen nach Variable, z.B: a**--**

7 - Arithmetische Ausdrücke

Operatoren

Unterschied zwischen Präfix- und Postfix-Operatoren:

Präfix-Operator

- **erst** inkrementieren / dekrementieren
- **dann** mit neuem Wert im Ausdruck weiterrechnen

Postfix-Operator

- mit altem Wert im Ausdruck weiterrechnen
- **dann** erst dekrementieren / inkrementieren

Beispiel:

```
int a, b = 1;
a = b++;    // b == 2,  a == 1
a = ++b;    // b == 3,  a == 3
```

7 - Arithmetische Ausdrücke

Operatoren

Sinnvolle Verwendung von Inkrement und Dekrement

- Zählschleifen
- Arrayindizierung

Inkrement und Dekrement in komplexen Ausdrücken

- erschwert Lesen von Programmen
- erschwert Fehlersuche
- vorsichtig und sparsam einsetzen!

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Operator	Beschreibung
|	bitweise oder (OR)
&	bitweise und (AND)
~	bitweise nicht (NOT)
^	Ausschliessendes oder (XOR)
<<	links Verschiebung (shift left)
>>	rechts Verschiebung (shift right)

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Bitweise ODER (OR):

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Beispiel:

$$\begin{array}{ll} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} = 553_{dez} \\ | & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 1188_{dez} \\ = & 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1_{bin} = 1709_{dez} \end{array}$$

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Bitweise UND (AND):

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel:

$$\begin{array}{ll} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ _{bin} = 553_{dez} \\ \& & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ _{bin} = 1188_{dez} \\ = & 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ _{bin} = 32_{dez} \end{array}$$

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Bitweise NICHT (NOT):

A	~A
0	1
1	0

Beispiel:

~ 0 1 0 0 0 1 0 1 0 0 1 $_{bin}$ = 553_{dez}

= 1 0 1 1 1 0 1 0 1 1 0 $_{bin}$ = 1494_{dez}

Bitbreite ist entscheidend für das Ergebnis

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Bitweise ausschließendes ODER (XOR):

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Beispiel:

$$\begin{array}{ll} & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} = 553_{dez} \\ \mathbf{\hat{}} & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 1188_{dez} \\ \mathbf{=} & 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1_{bin} = 1677_{dez} \end{array}$$

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Links verschieben (SHIFT LEFT):

Nach links verschieben um x Stellen entspricht der Multiplikaton mit 2^x

Beispiel:

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} << 1 = 553_{dez} * 2$

$0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0_{bin} = 1106_{dez}$

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin} << 2 = 553_{dez} * 4$

$1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0_{bin} = 2212_{dez}$

7 - Arithmetische Ausdrücke

Bitweise Operatoren

Rechts verschieben (SHIFT RIGHT):

Nach rechts verschieben um x Stellen entspricht der Integer-Division durch 2^x

Beispiel:

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin}$ >> 1 = 553_{dez} / 2

$0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0_{bin}$ = 276_{dez}

$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1_{bin}$ >> 2 = 553_{dez} / 4

$0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0_{bin}$ = 138_{dez}

8 - Anweisungen

8 - Anweisungen

Kapitel 8 – Anweisungen

- Ein-Ausgabe Anweisungen
- Schleifen
- Fallunterscheidungen
- Sprungbefehle

8 - Anweisungen

Ein-Ausgabe Anweisung (printf)

```
#include <stdio.h>
int main(void)
{
 int zahl;    float kommaZahl;
 char character;   char text[100];

 zahl = 5;
 komaZahl = 5.5;
 character = 'A';
 sprintf(text, "Dies ist ein Text!");
 printf("integer zahl    = %d \n",zahl);
 printf("float kommaZahl = %f \n",kommaZahl);
 printf("char character  = %c \n",character);
 printf("string text     = %s \n",text);
}
```

Fromatierungszeichen	Beschreibung
%d	ganzzahlig (bzw. 1 ASCII Zeichen)
%f	Fließkomma
%c	Character (Zeicher)
%s	String
\n	Zeilenvorschub

8 - Anweisungen

Ein-Ausgabe Anweisung (scanf)

```
#include <stdio.h>


int main(void
{
 int zahl; float kommaZahl;  char character;  char text[100];
 scanf("%d",&zahl);
 scanf("%f",&kommaZahl);
 scanf("%c",&character);
 scanf("%s",&text[0]);
}
```

8 - Anweisungen

Schleifen (For)

```
for (zaehler = 0; zaehler < 100; zaehler++){
   printf("zaehler hat den Wert:%d \n",zaehler);
}
```

Schleifen (While)

```
zaehler = 0;
while (zaehler < 100){
    printf("zaehler hat den Wert:%d \n",zaehler++);
}
```

Schleifen (Do-While)

```
zaehler = 0;
do{
   printf("zaehler hat den Wert:%d \n",zaehler++);
}while(zaehler<100);
```

8 - Anweisungen

Fallunterscheidungen (If-Else)

- Mit der if-else Anweisung ist es möglich eine wenn dann (sonst) Beziehung auszudrücken.

```
int variable;

if (variable==1){

   printf(„variable ist gleich 1\n");

} // der nachfolgende „else“ Teil ist nicht notwendig

else{

   printf(„variable ist ungleich 1\n");

}
```

8 - Anweisungen

Fallunterscheidungen (Switch-case)

Die Switch-case Anweisung bietet eine schnelle Möglichkeit viele Vergleiche in einer Anweisung auszuwerten:

```
switch(variable){

   case 1: { /* Die Variable ist gleich 1 */ break;}

   case 2: { /* Die Variable ist gleich 2 */ break;}

   default: { /* die Variable ist ungleich 1 und ungleich 2*/ }

}
```

Fallunterscheidungen (? Operator)

Der Fragezeichenoperator ? ermöglicht eine sehr verkürzte Schreibweise einer Fallunterscheidung : `Bedingung ? Anweisung1 : Anweisung2`

Ist die Bedingung erfüllt, so wird Anweisung1 ausgeführt, sonst Anweisung2.

```
A > B ? printf("A ist größer"); : printf("B ist größer");
```

8 - Anweisungen

Sprungbefehle (goto)

- Die goto-Anweisung ermöglicht einen direkten Sprung zu einem vorher definierten Label. Ein Label ist ein name, der mit einem Doppelpunkt : abgeschlossen wird.
- Die goto-Anweisung sollte nicht verwendet werden.

9 - Pointer und Speicher

9 - Pointer und Speicher

Kapitel 9 – Pointer und Speicher

- Pointer
- Adressoperator
- Pointer auf pointer
- Speicher reservieren

9 - Pointer und Speicher

Pointer

- Zeiger (englisch: pointer) sind Variablen, deren Inhalt eine Speicheradresse ist
- Sie sind auch von einem definierten Typ, wobei der Typ nichts mit dem Inhalt der Variablen zu tun hat, sondern nur mit deren Inkrementierung und Dekrementierung

z.B. wenn float auf einem Computer 4 Byte einnimmt

```
float *pointer1 = &variable;

char   *pointer2 = &variable;

pointer1++; // pointer1 wird um 4 inkrementiert

pointer2++; // pointer2 wird um 1 inkrementiert
```

9 - Pointer und Speicher

Pointer

- Da ein Pointer nach der Deklaration irgendwo in den Speicher zeigt birgt das die Gefahr, dass man über den Pointer irgendwo in den Speicher schreibt.
- Darum ist es wichtig den Zeiger zu initialisieren.
- Falls die Variable, auf die gezeigt werden soll noch nicht bekannt ist sollte der Pointer mit NULL initialisiert werden:

```
int *pZeiger;

pZeiger = NULL;
```

9 - Pointer und Speicher

Adressoperator

Mit dem Adressoperator & bekommt man die Adresse einer Variablen:

```
int a,b,*pZeiger;
a=5;
pZeiger=&a; // pZeiger zeigt auf die Adresse von a
*pZeiger=6; // schreibt in die Adresse die Zahl 6
            //(äquivalent zu a=6;)
pZeiger=&b; // pZeiger zeigt auf die Adresse von b
*pZeiger=2; // schreibt 2 in b (äquivalent zu b=2;)
```

9 - Pointer und Speicher

Pointer auf pointer

Es können auch Pointer auf Pointer definiert werden:

```
int a,*pZeiger,**ppZeiger;

pZeiger = &a;                 // zeigt auf Adresse von a

ppZeiger = &pZeiger;          // zeigt auf Adresse von p Zeiger a=5;

*pZeiger = 6;                 // entspricht a=6;

**ppzeiger = 7;               // entspricht a=7;
```

9 - Pointer und Speicher

Pointer (Operatorreihenfolge)

Kennt jemand den Unterschied zwischen:

```
wert = *pointer++
```

und

```
wert = ++*pointer
```

…und was machen folgende Zeilen?

```
char variable[4] = {0,1,2,3};

char *pointer = &variable[0];

*pointer = *pointer++ + ++*pointer;
```

9 - Pointer und Speicher

Speicher reservieren

Statt Speicher mit z.B. *int varArray[100]* zu reservieren kann man auch die Funktion *malloc()* benutzen.

```
<datentyp> * pVarArray;

int groesse = sizeof(<datentyp>)*numberOfElements;

pVarArray = (<datentyp>*)malloc(groesse);

if (pVarArray){

   // Speicher wurde reserviert!

}
```

Da der Speicher nicht automatisch freigegeben wird, muss er mit free() wieder freigegeben werden. Sonst können Memory Leaks (Speicherlecks) entstehen.

```
free(pVarArray);
```

9 - Pointer und Speicher

Speicher reservieren

- Der gleiche Code z.B. für den Datentyp char könnte auch folgendermassen heissen:

```
char *pVarArray;
char *pointer;
if (pVarArray = (char*)(malloc(sizeof(char)*10))){
   pointer = pVarArray;
   *pointer++='A';
   *pointer++='B';
   *pointer=0;
   printf("%s\n",pVarArray);
   free(pVarArray);
}
```

10 - Funktionen

10 - Funktionen

Kapitel 10 – Funktionen

- Syntax
- Beispiele

10 - Funktionen

Syntax

- In C versucht man ein Programm in Funktionen aufzuteilen.
- Funktionen können Übergabeparameter und einen Rückgabewert haben.
- Funktionsnamen dürfen wie Variablen nur mit Buchstaben oder mit Unterstrichen beginnen.
- Die Syntax einer Funktion lautet folgendermassen:

```
TypRueckgabe funktionsname(typ1 nameParam1, typ2 nameParam2...)
{
     ...Implementierung...
     return Rueckgabewert;
}
```

10 - Funktionen

Syntax

Achtung: Dem Compiler muss die Funktion vor ihrem Aufruf bekannt sein!

Möchte man die Funktion z.B. hinter der main() Funktion implementieren, so muss man den Funktionskopf bzw. Funktionsprototypen vorher deklarieren:

```
double quadrat(double x); // functionprototype
int main(int argc, char** argv)
{
 printf("4.4^2=",quadrat(4.4));
}
double quadrat(double x)
{
  return x*x;
}
```

10 - Funktionen

Syntax

Die Übergabeparameter können dabei unterschiedlicher Natur sein:

- Feste Werte (call by value)
- Zeiger auf Variablen (call by address)

Der TypRückgabe kann ein beliebiger Datentyp sein:

char, int, long int, double, float….

Oder **void**, wenn nichts zurückgegeben wird!!!

10 - Funktionen

Beispiele

Beispiel für call by value:

```
double quadrat(double x)
{
  double y; // new variables only valid for the function
  y = x*x;  // commands may be applied
  return y; // variables may be returned
}
int main(int argc, char **argv)
{
 printf("quadrat(3.3)=%f", quadrat(3.3) ); // function-call inside
                                           // the printf command
 return 0;
}
```


Jörn Fischer - j.fischer@hs-mannheim.de - Raum 113a

10 - Funktionen

Beispiele

Beispiel für call by value:

```
void init(int var) // Variable ist eine Kopie von a auf dem Stack
{
 var = 5; // variable is only changed on the stack
          // leaving the function, the variable is lost
}
int main(int argc, char **argv)
{
 int a=3;
 init(a); // contents of a is given to the function init()
 printf("der Wert von a ist %d",a); // output: 3
 return 0;
}
```

10 - Funktionen

Beispiele

Beispiel für call by address:

```
void init(int *var)
{
 *var = 5; // contents of the variable may be changed
}
int main(int argc, char **argv)
{
 int a;
 init(&a); // adress of a is given to the function init()
 printf("der Wert von a ist %d",a); // 5 wird ausgegeben
 return 0;
}
```

11 - Präprozessor

11 – Präprozessor

Kapitel 11 – Präprozessor

- Mehrere Quelldateien
- #include
- #define
- #if #else #endif
- #pragma

11 – Präprozessor

Mehrere Quelldateien

- Wenn man ein Programm in Funktionen unterteilt erscheint es schnell sinnvoll, z.B. mathematische Funktionen von Text manipulierenden Funktionen zu trennen
- So kann man in der Programmiersprache C in einem File die mathematischen und im anderen die Text manipulierenden Funktionen programmieren
- Dann muss jedoch zu jedem dieser Files ein Header File existieren, in dem die Deklaration der Funktionen stattfindet. Diese werden vom Linker gebraucht:

```
double cos(double phi); // prototype declaration
```

11 – Präprozessor

- Benutzt nun ein Quellcode die Funktion, so muss das Headerfile mit der Prototypendeklaration der Funktion mit Hilfe des #include Befehls eingebunden werden:

```
#include "eigeneRoutinen.h"
```

11 – Präprozessor

#include

Was macht der #include Befehl?

Was ist der Unterschied zwischen

```
#include "string.h"  // sucht zunächst im lokalen Ordner
#include <string.h>  // (c) sucht im standard Include Pfad
#include <string>    // (c++) sucht im standard Include Pfad
```

11 – Präprozessor

#define

Was macht der #define Befehl?

Was tun folgende Befehle?

```
#define TEXT "Hallo Welt"
#define TRUE 1
#define FALSE 0
#define DEBUG TRUE
#define MAX(a,b) a>b?a:b
#define MAXX(a,b) if (a>b)                          \
                      printf("a ist groesser"); \
                  else                              \
                      printf("b ist groesser")
```

11 – Präprozessor

#if #else #endif

Was machen die Befehle #if, #else,#elif, #endif, #ifdef, #ifndef?

```
#ifndef BERECHNUNG_H
#define BERECHNUNG_H
// obige Zeilen sogen dafür, dass das File
// nur einmal eingebunden wird, weil ab dann
// H_BERECHNUNG definiert ist.
...
#if DEBUG == TRUE
   printf("debugvalue=%d\n",debugvalue);
#else
   printf("\n");
#endif
...
#endif // BERECHNUNG_H
```

11 – Präprozessor

#pragma

Was macht der Befehl #pragma?

- Der #pragma Befehl veranlasst den Präprozessor eine implementierungsspezifische Aktion (z.B. den Coprozessor mit zu benutzen)

```
// turn on code generation for the coprozessor
#pragma NEO
```

12 - Die C-Standard Bibliothek

12 - Die C-Standard Bibliothek

Kapitel 12 – Die C-Standard Bibliothek

- Strings
- Dateizugriffe

12 - Die C-Standard Bibliothek

Strings

Länge eines Strings abfragen mit der Funktion
strlen(<String>)

- liefert Anzahl der Zeichen im übergebenen String als int-Wert.

Beispiele:

```
char *name = "Jörn Fischer";
printf ("%d\n",strlen(name));
```

→ **12**

```
char *a = "a";
printf ("%d\n",strlen(a));
```

→ **1**

12 - Die C-Standard Bibliothek

Strings

String kopieren mit der Funktion
strcpy(<kopie>, <quelle>)

```
// Speicher für neuen String belegen
char *quelle = "Beispieltext";
char *kopie = malloc(strlen(quelle)+1);   // +1 Nullbyte
// String kopieren
strcpy(kopie,quelle);
```

12 - Die C-Standard Bibliothek

Strings

n Zeichen des Strings kopieren mit der Funktion
strncpy(<kopie>, <quelle>, <n>)

```
// reserve memory for the string

char *kopie = malloc(4);

// copy string

strncpy(kopie,quelle, 4);
```

12 - Die C-Standard Bibliothek

Strings

Zwei Strings aneinanderhängen (konkatenieren) mit der Funktion
strcat(<ziel>, <quelle>)

- Zielstring bekommt Quellstring hinten drangehängt

```
char *vorname = „Jörn";

char *nachname =„Fischer";

// Speicher für Zielstring belegen

char *name = malloc(20);       // Platz für 19 Zeichen plus
Nullbyte

strcpy(name,vorname);          // Vornamen kopieren

strcat(name, " ");             // Leerzeichen dazwischen

strcat(name, nachname);        // Nachnamen hinten dran hängen

printf("%s\n",name);           // Gibt „Jörn Fischer" aus
```

12 - Die C-Standard Bibliothek

Strings

n Zeichen des Quellstrings an Zielstring anhängen mit der Funktion
strncat(<ziel>, <quelle>, <n>)

- Zielstring bekommt n Zeichen des Quellstrings hinten drangehängt

```
char *vorname = „Jörn";

char *nachname =„Fischer";

// Speicher für Zielstring belegen

char *name = malloc(20);       // Platz für 19 Zeichen plus
Nullbyte

strcpy(name,vorname);          // Vornamen kopieren

strcat(name, " ");             // Leerzeichen dazwischen

strncat(name, nachname, 3);    // 3 Buchst. des Nachnamens anh.

printf("%s\n",name);           // Gibt „Jörn Fis" aus
```

12 - Die C-Standard Bibliothek

Strings

Inhalt zweier Strings vergleichen mit der Funktion
strcmp(<String1>, <String2>)

- liefert **0**, wenn beide Strings zeichenweise gleich
- liefert +1, wenn String1 größer String2 (lexikalisch nach String2)
- liefert -1 wenn String1 kleiner String2 (lexikalisch vor String2)

!strcmp(<String1>, <String2>)

- liefert **1**, wenn beide Strings zeichenweise gleich, sonst 0.

```
char *t1=„Jörn";
char *t2=„Jörn";
char *m = "Markus";
printf("!strcmp(%s, %s)=%d\n", t1, t2, !strcmp(t1,t2)); // returns 1
printf("!strcmp(%s, %s)=%d\n", t1, m, !strcmp(t1,m));   // returns 0
```

12 - Die C-Standard Bibliothek

Strings

Suchstring im Quellstring herausfinden

char *strstr(<quellString>, <suchString>)

- liefert den Quellstring ab der Stelle, an der der Suchstr. gefunden ist
- liefert NULL, wenn Suchstring im Quellstring nicht enthalten

Beispiel:

```
char *name = "Thomas Specht";
char *specht = strstr(name,"Specht");
printf ("%s\n",specht);       // Specht
char *specht2 = strstr(name,"specht");
printf ("%s\n",specht2);      // (null)
char *specht3 = strstr(name,"Thomas");
printf ("%s\n",specht3);      // Thomas Specht
```

12 - Die C-Standard Bibliothek

Dateizugriffe (Fopen(), fprintf(), fclose())

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

int main(int argc, char *argv[])
{
   int t;
   FILE *out;
   out = fopen("Filename.txt","wb");
   if (out != NULL){
      for (t=0;t<100;t++){
          fprintf(out, "Test\n");
      }

      fclose(out);//closes the File
   }
   return 0;
}
```

```
open("file",xxx);
xxx = w für schreiben
xxx = r für lesen(File existiert)
xxx = r+ für lesen + schreiben
xxx = w+ für lesen + schreiben
xxx = a+ für lesen und erweitern
xxx = wb für binäres schreiben
xxx = rb für binäres lesen

fprintf() schreibt das, was
bei printf() auf dem Bildschirm

landet ins File!
```

12 - Die C-Standard Bibliothek

Dateizugriffe (Fscanf())

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   FILE *in;
   char letter;
   in = fopen("File.txt","rb");
   if(in != NULL){
        while(fscanf(in,"%c",&letter)!=EOF){
            printf("%c",letter);
        }
        fclose(in);
   }
   return 0;
}
```

12 - Die C-Standard Bibliothek

Dateizugriffe (Fgetc())

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

int main(int argc, char *argv[])
{
   char letter;
   FILE *in;
   in = fopen("Filename.txt","rb");
   if (in != NULL){
      while(!feof(in)){
         letter=fgetc(in);
         printf("%c",letter);
      }
      fclose(in);
   }
   return 0;
}
```

fgetc liest ein char aus dem File

12 - Die C-Standard Bibliothek

Dateizugriffe (Fputc())

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

int main(int argc, char *argv[])
{
   int t;
   FILE *out;
   out = fopen("Filename.txt","wb");
   if(out != NULL){
        for (t=0;t<100;t++){
            fputc('A',out);
        }
        fputc('\n',out);
        fclose(out);
   }
   return 0;
}
```

fputc schreibt ein char ins File

12 - Die C-Standard Bibliothek

Dateizugriffe (Fseek())

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   FILE *out;
   out = fopen("Filename.txt",“w");
   if(out != NULL){
      fputs("Dies ist ein Irrtum.",out);
      fseek(out, 13, SEEK_SET);
      fputs("Beispiel!" ,out);
   }
   fclose(in);
   return 0;
}
```

```
fseek(FILE*stream, long int
offset, int origin)
```

offset = Anzahl der bytes vom Ausgangspunkt

origin = SEEK_SET (Fileanfang)
= SEEK_CUR (aktuelle Pos)
= SEEK_END (Fileende)

13 – Typedefs, Strukturen und Unions

13 – Typedefs, Strukturen und Unions

Typedefs, Strukturen und Unions

- Typedefs
- Strukturen
- Unions

13 – Typedefs, Strukturen und Unions

Typedefs

Mit der typedef Anweisung kann man einem Datentyp einen anderen Namen geben:

```
typedef int xxxInteger, xyzInteger;

xxxInteger var1;

xyzInteger var2;
```

13 – Typedefs, Strukturen und Unions

Strukturen

Strukturen werden in C dazu benutzt, Daten verschiedenen Typs, die zusammen gehören, in einer Datenstruktur zusammenzufassen.

```
struct schwein
{
 char name[20];
 float groesse;
 float gewicht;
}rudi,nele; // rudi und nele sind Instanzen von Schwein
```

13 – Typedefs, Strukturen und Unions

Strukturen

Eine Instanz eines structs benutzt man folgendermassen:

```
struct schwein berta; // berta is an instance of schwein
strcpy(berta.name,"Berta");
berta.groesse=0.95;
berta.gewicht=200;


struct schwein *pZeigerAufBerta;
pZeigerAufBerta=&berta;
pZeigerAufBerta->groesse += 0.1; // berta.groesse is enlarged
pZeigerAufBerta->gewicht  = 210; // berta.gewicht = 210 kg
```

13 – Typedefs, Strukturen und Unions

Unions

Im Gegensatz zu einer Struktur kann eine Union in einer Variablen unterschiedliche Typen speichern:

```
union eineVariable
{
 // Instance of eineVariable may be of type int or array
 // of float, but not both

 int intWert;
 float fliesskomaWert[10];
};
```