ANSI-C

ANSI-C Programmierung

Autor: Dipl.-Math.
E. Engelhardt

Stand: 10. Dezember 2007

Vorbemerkungen

- Diese Präsentation ist keine vollständige ANSI-C-Sprachbeschreibung!
- Es werden wesentliche Teile von ANSI-C in knapper Form vorgestellt
- Die Präsentation dient zur Kursdurchführung
- Begleitend zu dieser Präsentation ist die Präsentation „ANSI-C Übungen", die nicht an die Studenten verteilt wird

Inhalt

1 Einleitung

1.1 Entwicklungsgeschichte von „C"
1.2 „C" und „UNIX"
1.3 Der Geist von „C" nach ANSI-Kommitee
1.4 Eigenschaften von „C"
1.5 Phasen der Programmentwicklung
1.6 Erstes C-Programm
1.7 Kürzestes C-Programm
1.8 Grundsätzlicher Aufbau eines C-Programms

2 ANSI-C Sprachelemente

2.1 Syntay-Beschreibungsmöglichkeiten
2.2 Elementare Datentypen
2.3 Vereinbarungen elementarer Datentypen
2.4 Arithmetische Operatoren

Inhalt

2.5 Zuweisungen
2.6 Vergleichsoperatoren, logische Operatoren
2.7 Ausdruck, Anweisung
2.8 Semantik von Kontrollstrukturen
2.9 ANSI-C-Kontrollstrukturen
2.10 Strukturierte Programmierung

3 Funktionen

3.1 Arbeit mit Funktionen
3.2 Funktionen-Deklaration
3.3 Rückkehrwert (return-Wert)
3.4 Standardfunktionen
3.5 „call by value", „call by reference"
3.6 Rekursive Funktionen
3.7 Parameterübergabe an „main"
3.8 Zeiger auf Funktionen
3.9 Funktionen mit variabler Parameterzahl

Inhalt

4 Felder (Arrays, Vektoren), Zeiger (Pointer)
- 4.1 Arbeit mit Feldern
- 4.2 Zeichenketten
- 4.3 Zeiger, Pointer
- 4.4 Zeiger-Arithmetik

5 Zusammengesetzte Datenstrukturen
- 5.1 Aufbau einer Struktur
- 5.2 Zugriff auf Bestandteile der Struktur
- 5.3 Zeiger auf Strukturen
- 5.4 Typedefinition
- 5.5 „sizeof"-Operator

6 Dateiarbeit
- 6.1 Filepointer
- 6.2 Datei unformatiert schreiben
- 6.3 Datei unformatiert lesen
- 6.4 Übersicht Dateiarbeit

Inhalt

7 Der Präcompiler
- 7.1 include-Direktive
- 7.2 define-Direktive
- 7.3 Makros

8 Dynamische Datenstrukturen
- 8.1 Übersicht zur Mini-Datenbank
- 8.2 Das klassische MVC-Konzept
- 8.3 Doppelt verkettete Liste (DVL)
- 8.4 Typedefinition für DVL
- 8.5 Anlegen eines Datensatzes (DS)

9 Zeitfunktionen
- 9.1 Struktur „tm"
- 9.2 Datumberechnungen
- 9.3 Zeitberechnungen

Inhalt

10 Programmprojekte
- 10.1 Programmverwaltung mit Tools
- 10.1 Makefile, make
- 10.2 IDEs
- 10.3 Projekte

11 Fehlerbehandlung
- 11.1 Fehlernachrichten

12 Programmierhinweise
- 12.1 Kommentare
- 12.2 Hinweise zur Schreibweise (Syntax)
- 12.3 Hinweise zum Programmierstil (Semantik)

1.1 Entwicklungsgeschichte von „C"

- 1958 „FORTRAN" (formula translation)
  - definiert durch J. W. Backus
  - wichtigste Sprache für techn.-wissenschaftliche Anwendungen
- 1960 ALGOL 60 (ALGOrithmic Language)
  - entwickelt von einem internationalen Komitee
  - erste exakt definierte Programmiersprache für mathematisch Aufgabenstellungen (Backus-Naur-Syntax)
- 1963 CPL (Combined Programming Language)
  - Cambridge University und University of London

Entwicklungsgeschichte von „C“

- 1967 BCPL (Basic CPL)
  - Martin Richards (Cambridge) zum Schreiben von Compilern und Betriebssystemen
- 1970 BCPL => B
  - Ken Thompson bei Bell-Laboratorien (Murray Hill, USA)
  - Systemprogrammiersprache ohne Typkonzept für DEC PDP-7 (Digital Equipment Corporation)
  - Definiert durch J. W. Backus
- 1972 aus B und BCPL => „C“
  - Durch Dennis M. Ritchie auf einer PDP-11
  - Für Implementierung von UNIX

Entwicklungsgeschichte von „C“

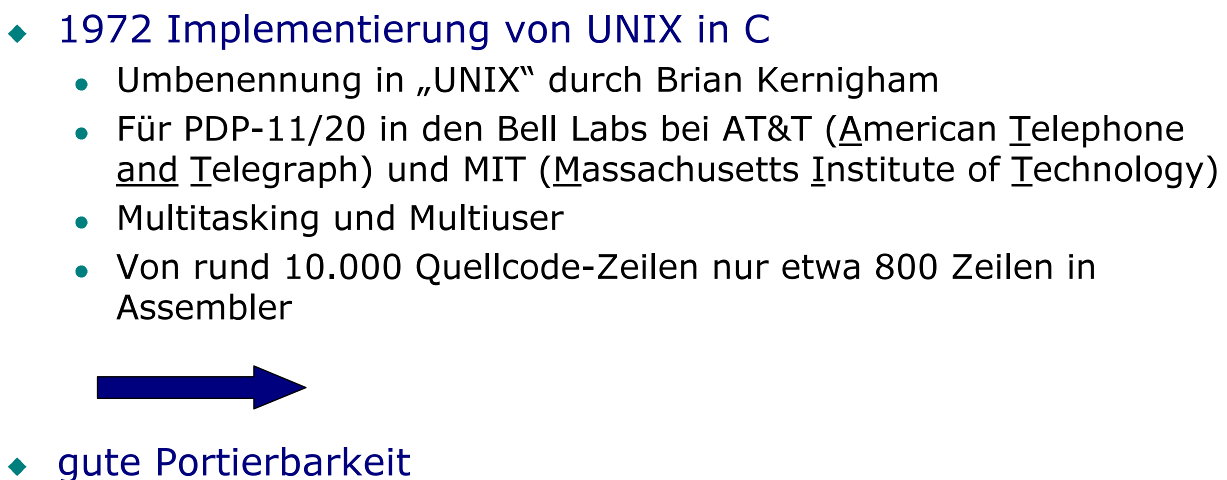
- 1978 „The C Programming Language“
  - Brian W. Kernigham und Dennis M. Ritchie (K&R)
  - Quasi-Standard (C-Bibel)
- 1983 Beginn der Standardisierung
  - American National Standard Institute (ANSI)
  - Komitee X3J11
- 1989 Verabschiedung als ANSI-C („C89“)
- 1990 Ergänzungen und Korrekturen des ANSI-C („C90“), Untermenge von C++ (bis auf wenige Details)
- 1999 Überarbeitung und weitere Veränderungen („C99“), keine Untermenge von C++ mehr!

1.2 „C“ und „UNIX“

- ab 1960 MULTICS
  - MULTiplexed Information and Computing System
  - Bei AT&T, MIT und General Electrics
- 1969 Computerspiel „Space Travel“
  - Ken Thompson für Mainframe
- 1969 - 1971 erste Version von UNIX (UNICS) in Assembler auf PDP-7
  - UNICS (UNiplexed Information and Computing System)
  - Ken Thompson und Dennis Ritchie
  - Zwei Benutzer (Spieler)

„C“ und „UNIX“

- 1972 Implementierung von UNIX in C
  - Umbenennung in „UNIX“ durch Brian Kernigham
  - Für PDP-11/20 in den Bell Labs bei AT&T (American Telephone and Telegraph) und MIT (Massachusetts Institute of Technology)
  - Multitasking und Multiuser
  - Von rund 10.000 Quellcode-Zeilen nur etwa 800 Zeilen in Assembler



- gute Portierbarkeit

1.3 The Spirit of C (ANSI-Komitee)

- Keep it small and simple (KISS).
  - Halte es klein und einfach.
- Trust the programmer, don't do any runtime checks.
  - Vertraue dem Programmierer, mache keine Runtime-Checks.
- Make it fast, even if it is not guaranteed to be portable.
  - Mache es schnell, auch wenn es möglicherweise nicht portabel ist.
- Provide only one way to do an operation (orthogonal).
  - Stelle nur einen Weg zur Verfügung, um eine Aktion / Operation zu machen.
- Don't prevent the programmer from doing what needs to be done.
  - Hindere den Programmierer nicht daran, das zu tun, was getan werden muss.

Der „Geist“ von „C“ (sinngemäß)

- Der Programmierer weiß was er tut (Er hat die Verantwortung für das, was er tut).
- Größtmögliche Freiheiten (Formatfreiheit, wenige Einschrän-kungen)
- Sprache klein und einfach halten (32 Schlüsselwörter und 44 Operatoren bei C89/C90)
- Seit C99 zusätzlich „_Bool“, „_Imaginary“, „restrict“, „Complex“ und „inline“
- Redundanzfreiheit (mit wenigen Zeichen viel ausdrücken)
- Effektive und effiziente Programmierung (Arbeit mit Adressen, Bitoperatoren, ...)

1.4 Eigenschaften von „C“ (1/3)

Als höhere Programmiersprache (Middle-Level-Sprache)

- Datenobjekte: Zeichen, ganze Zahlen und reelle Zahlen (Gleitkommazahlen)
- Zusammengesetzte Datenobjekte: Zeiger, Felder (Vektor, Array), Verbund (structures), Variante (unions), Funktionen
- Arithmetik, Vergleichsoperatoren
- Strukturierte Sprache mit Kontrollstrukturen (anders als z. B. Fortran IV, Basic und Cobol)
- Globale und lokale Variablen
- Betriebssystem- und Rechnerunabhängigkeit

Eigenschaften von „C“ (2/3)

Als Systemprogrammiersprache (ähnlich Assembler-Sprache)

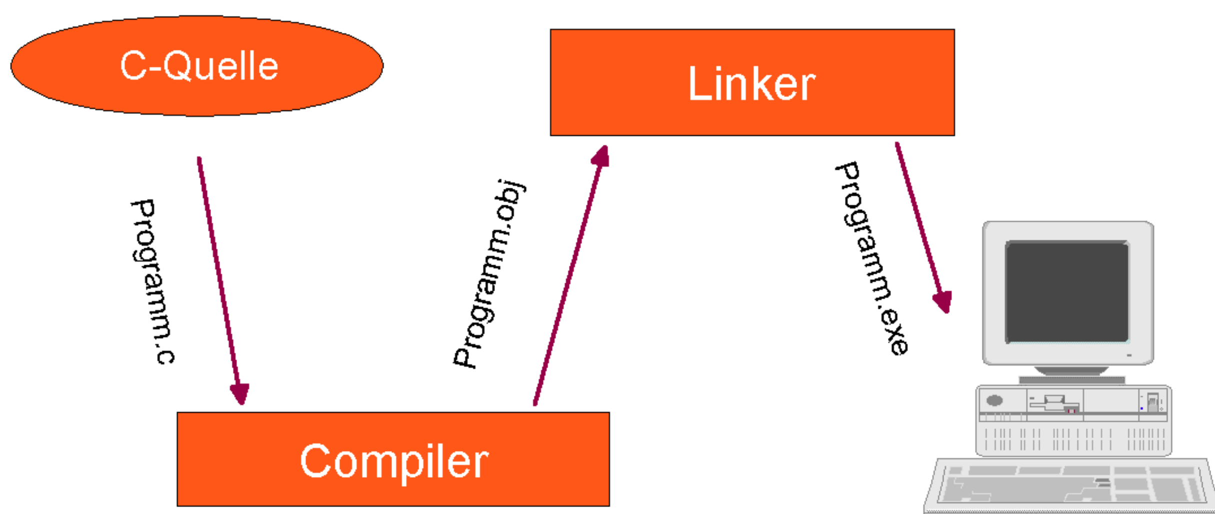
- Wenig Fehlerüberprüfung zur Laufzeit (nicht High-Level)
- Nicht streng typgebunden (automatisches Casting), u. a. deshalb nicht High-Level-Sprache
- Datenobjekte entsprechend der meisten Maschinen (Zeichen, Zahlenwerte, Adressen)
- Keine Elemente für E/A, Dateizugriffe, Speicherverwaltung und Felder
- Adressenarithmetik (Zeigerarithmetik)
- Hardwarenahe Befehle (Bit-Operatoren, Register)

Eigenschaften von „C“ (3/3)

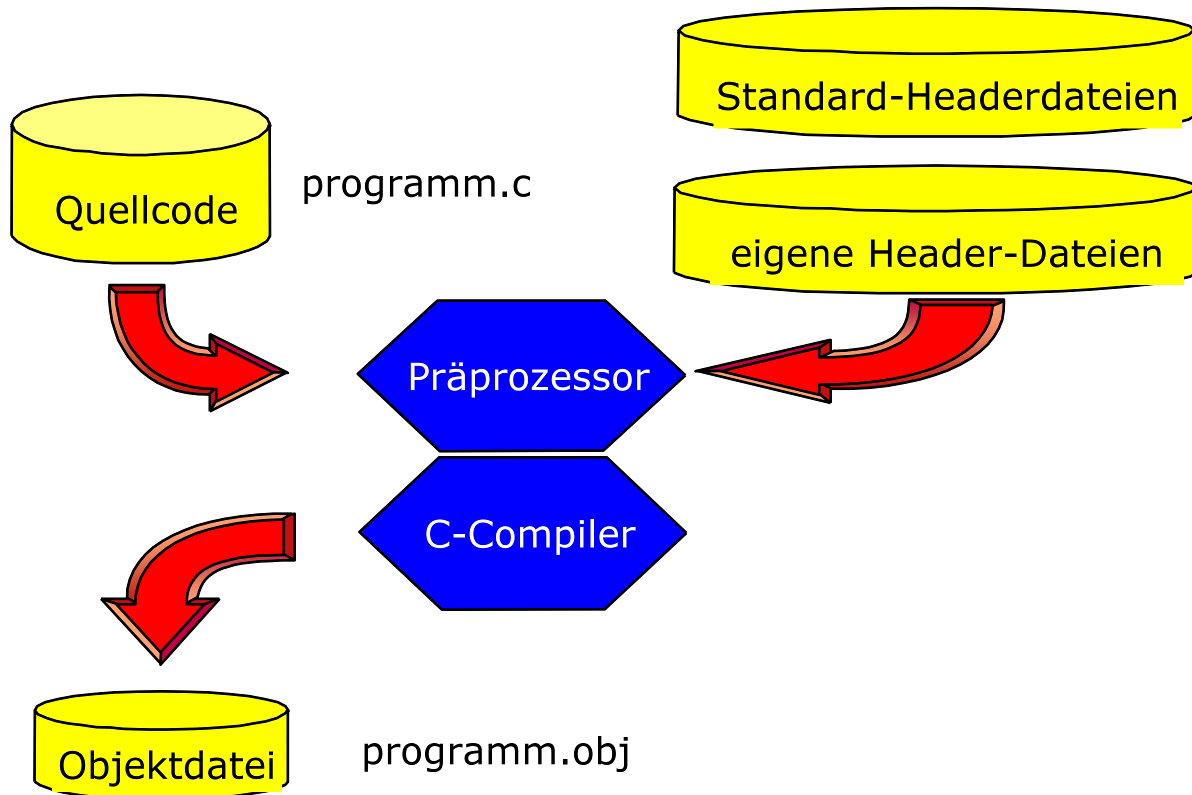
Weitere Eigenschaften

- C ist portable Sprache (leicht übertragbar auf andere Rechner/Betriebssysteme)
- Funktionenorientierte Sprache (auch E/A-Operationen, Speicherverwaltung und Bearbeitung von Feldern als Funktionen)
- C ist „general purpose programming language“ (vielseitig einsetzbare, universelle Programmiersprache)
- Entwickelt von Systemprogrammierern für Systemprogrammierer

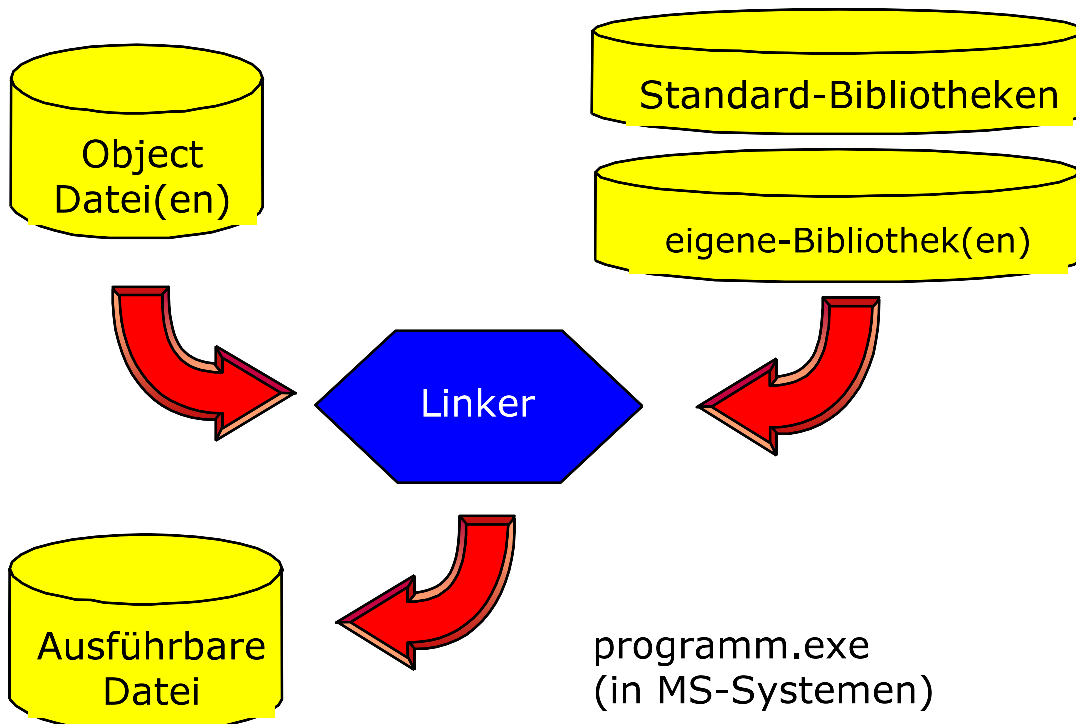
1.5 Phasen der Programmentwicklung (1/3)



Ein C-Programm ausführen (2/3)



Ein C-Programm ausführen (3/3)



1.6 Erstes C-Programm („hallo.c“)

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

```
/* erstes C-Programm */

#include <stdio.h>

int main(void)
{
   printf("Hallo Welt!\n");
   return 0;
}
```

Kommentar

Einbindung von Header-Dateien

Haupt-Funktion

Ausgabefunktion

Rückgabewert

1.7 Kürzestes C-Programm

- Mit Warnungen

```
main(){}
```

- ANSI-gerecht

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

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

1.8 Grundsätzlicher Aufbau

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

Grundsätzlicher Aufbau

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

Grundsätzlicher Aufbau

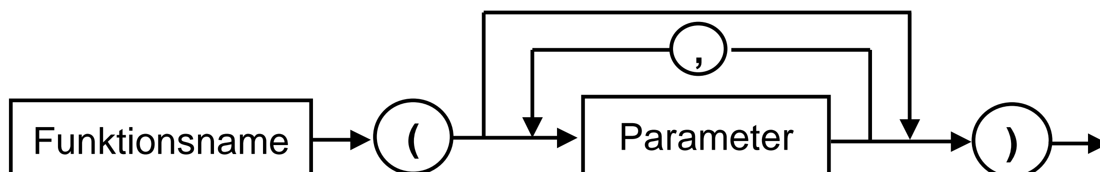
- ein Block besteht aus:
  - Öffnender und schließender geschweifter Klammer
  - Vereinbarungen (Definition von Variablen und Feldern); diese müssen am Anfang des Blockes stehen!
  - Anweisungen bzw. Kontrollstrukturen
  - Weiteren Blöcken (Blöcke können geschachtelt werden)
- Ab C99 können Vereinbarungen und Anweisungen gemischt untereinander stehen
  - Nur für for-Schleife: for (int i= 0; ...) erlaubt

Syntax eines Blockes:

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


2.1 Syntax-Beschreibungsmöglichkeiten

- Zur Beschreibung der Syntax der Programmiersprache ANSI-C können (wie auch zur Beschreibung vieler anderer Programmiersprachen) verschiedene Möglichkeiten genutzt werden
- In dieser Folienpräsentation wird auf eine rein formale Syntax-Beschreibung verzichtet
- Es wird eine Mischung aus verbaler Beschreibung und Beschreibung mit Hilfe einer vereinfachten erweiterten *Backus-Naur-Form* (EBNF) verwendet
- Eine weitere Möglichkeit wäre eine Beschreibung mit Hilfe von *Syntaxdiagrammen* (Bsp. Funktionsaufruf)



Erweiterte Backus-Naur-Form (EBNF)

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

2.2 Elementare Datentypen

- ganzzahlige Datentypen
  - short (short int) 2 Byte -2^{15} .. $+2^{15} - 1$ (-32 768 ... 32 767)
  - unsigned short 2 Byte 0 .. $2^{16} - 1$ (0 ... 65 535)
  - int 2 Byte, 4 Byte oder 8 Byte (implementationsabhängig)
  - long (long int) 4 Byte -2^{31} .. $+2^{31} - 1$ (-2 147 483 648 ... 2 147 483 647)
  - unsigned long 4 Byte 0 .. $2^{32} - 1$ (0 ... 4 294 967 295)
  - char 1 Byte -128 .. +127
  - unsigned char 1 Byte 0 ... 255 (ascii)

Elementare Datentypen

- Gleitkommazahlen
  - float 4 Bytes -3.4E+38 ... 3.47E+38
    mindestens 6 Stellen Genauigkeit (dezimal)
  - double 8 Bytes -1.7E+308 … 1.7E+308
    mindestens 15 Stellen Genauigkeit (dezimal)
  - long double 16 Bytes -1.1E+4932 … 1.1E+4932
    mindestens 19 Stellen Genauigkeit (dezimal)
- Es gibt keine elementaren Datentypen „String" (Zeichenkette, Text) und „byte"!

Konstanten

- Einzelne Zeichen: (sind auch ganze Zahlen)
  - 'A', '1', '\n', '\011', '\x84', ('A' ungleich "A"!)
  - der Gegenschrägstrich macht aus einem normalen Zeichen ein Sonderzeichen oder ein Sonderzeichen zu einem normalen Zeichen
- ganze Zahlen (Standard ist „int")
  - dezimal: 123, -12, 45L (long), 678U (unsigned)
  - oktal: 0101, 048, 0142, 055L (long),
    4 000 000 000UL (unsigned long)
  - hexadezimal: 0x20, 0xFF, 0x12L (long)
- Real-Zahlen (Standard ist „double")
  - 123.45, .667, -3.5e-5, 65.43F (float), 12345678912345.0L (long double)

Neue Datentypen seit K&R (ANSI)

- const: konstanter Datentyp
  - Variable kann nach der Vereinbarung nicht mehr verändert werden (auch nicht versehentlich)
- void: unbestimmt, nicht existent
- volatile: flüchtig
  - Entsprechend vereinbarte Variablen können auch durch Ereignisse außerhalb des Programms verändert werden
  - „const" und „volatile" können kombiniert werden, d.h. die Variable kann nicht durch das Programm, jedoch durch Ereignisse von außerhalb verändert werden
- Ab 1999 („C99")
  - long long 8 Byte -2^{63} .. $+2^{63}$ - 1
  - unsigned long long 8 Byte 0 .. $+2^{64}$ - 1
  - complex, imaginary (inkompatibel zu C++)

2.3 Vereinbarungen

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

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

2.4 Arithmetische Operatoren

- Für numerische Datentypen (auch Zeichen)
- „+“ - Addition, „-“ - Subtraktion, „*“ - Multiplikation
- „/“ - Division (bei ganzen Zahlen wird abgerundet)

- zusätzlich bei ganzzahligen Datentypen
- „%“ - Restdivision (modulo)
- „++“ - Inkrement (Bsp.: i++, ++i)
- „--“ - Dekrement (Bsp.: j- -, - -j)

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

2.5 Zuweisungen, Bedingungsoperator

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

Syntax:

Vergleichsausdruck ? Ausdruck_1 : Ausdruck_2

2.6 Operatoren

- Vergleichsoperatoren
  - < kleiner,   <= kleiner gleich
  - > größer,   >= größer gleich
  - == gleich,   != ungleich
- logische Operatoren
  - && logisches „und“ (and)
  - || logisches „oder“ (or)
  - ! logisches „nicht“ (not)
- logische Werte ab „C99“
  - Datentyp „bool“ (als Makro in <stdbool.h>, _Bool)
  - „true“ und „false“

2.7 Ausdruck, Anweisung

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

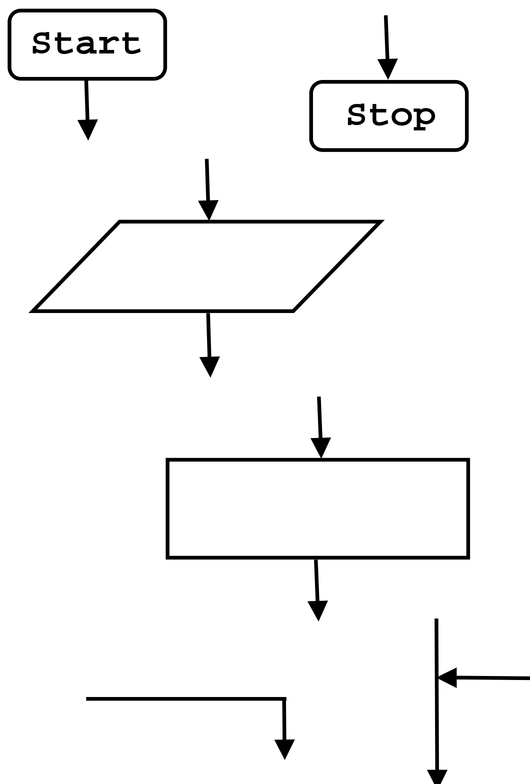
Ausdrücke mit Wert ungleich 0 werden als „wahr“ (true) gewertet. Gleich 0 ergibt „falsch“ (false)!

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

2.8 Semantik von Kontrollstrukturen

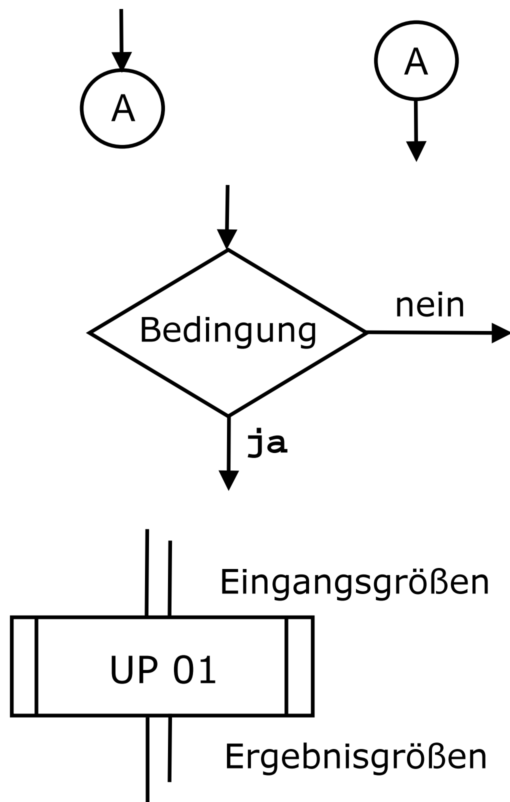
- Die Beschreibung der Sematik von Programmiersprachen wird in vielen Lehrbüchern oft nur verbal vorgenommen
- Eine Beschreibung der Semantik von Kontrollstrukturen kann nicht mit Hilfe von Nassi-Shneiderman-Diagrammen erfolgen, da diese grafischen Elemente mit den Kontrollstrukturen hinsichtlich Abstraktionsebene völlig gleichwertig sind (abgesehen von den Vereinbarungen)
- Zum exakten Verständnis der Funktionsweise der Kontrollstrukturen wird in dieser Folienpräsentation die grafische Darstellung mit Hilfe von PAP-Diagrammen (DIN 66001) verwendet
- Dadurch kann auf umfangreiche verbale Beschreibungen verzichtet werden

Darstellungsformen - PAP (DIN 66 001)



- Anfang / Ende
- Ein- / Ausgabe
- Anweisung (Aktion)
- Ablauf / Zusammenführung

Darstellungsformen - PAP (DIN 66 001)



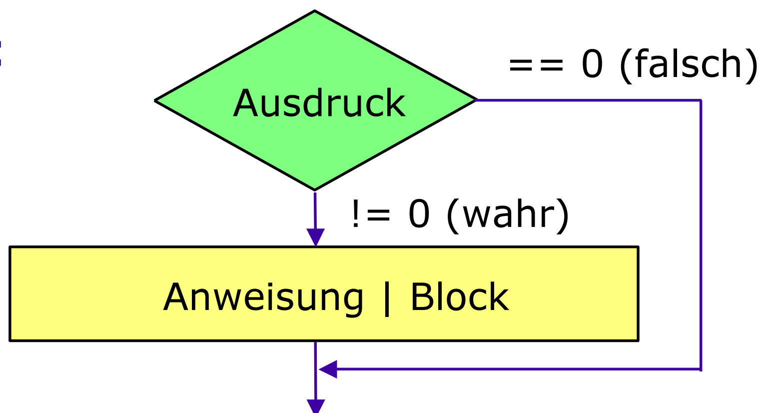
- Anschluss-Stellen
- Verzweigung
- Aufruf von Teilalgorithmen

2.9 Kontrollstrukturen (1/10)

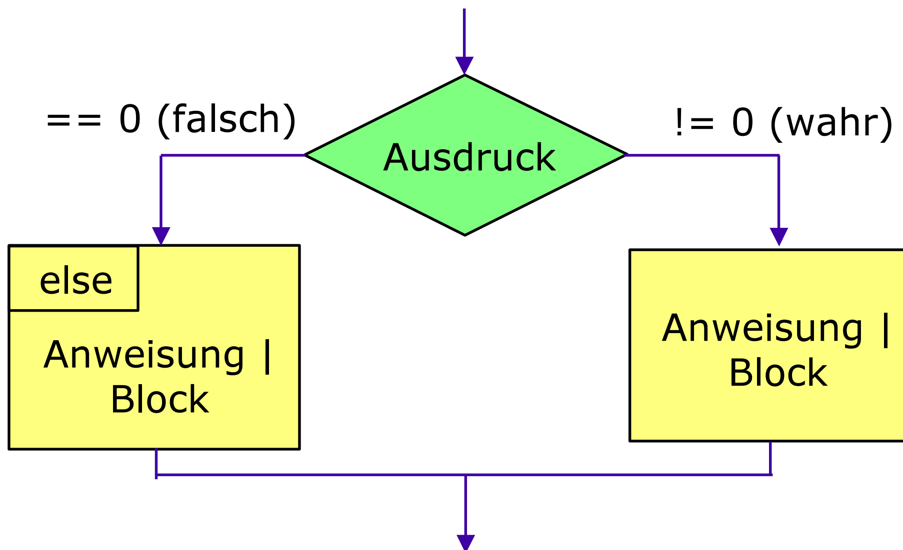
Syntax der if - Verzweigung:

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

Semantik:
(1. Fall)



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



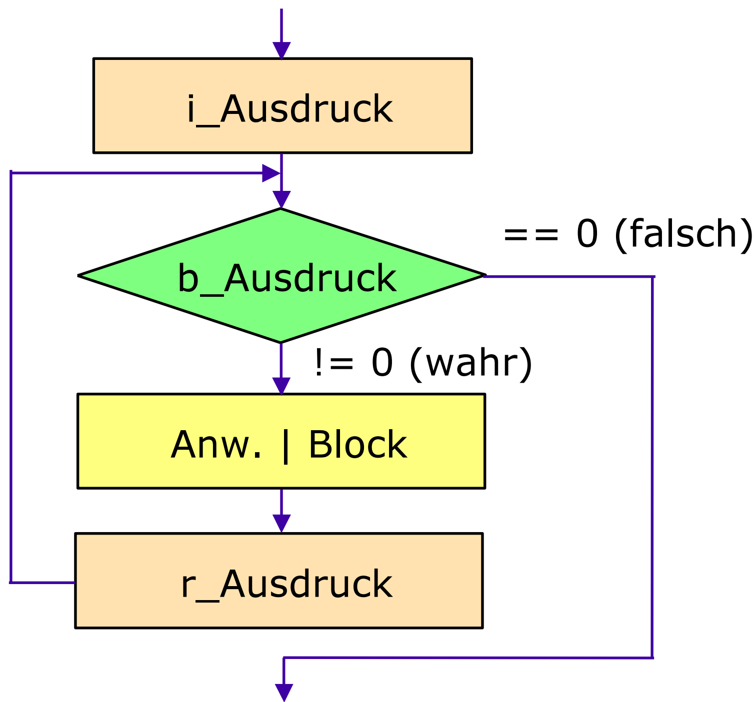
Syntax der for - Schleife:

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

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

Beispiel: **for (;;);** ist zulässig!

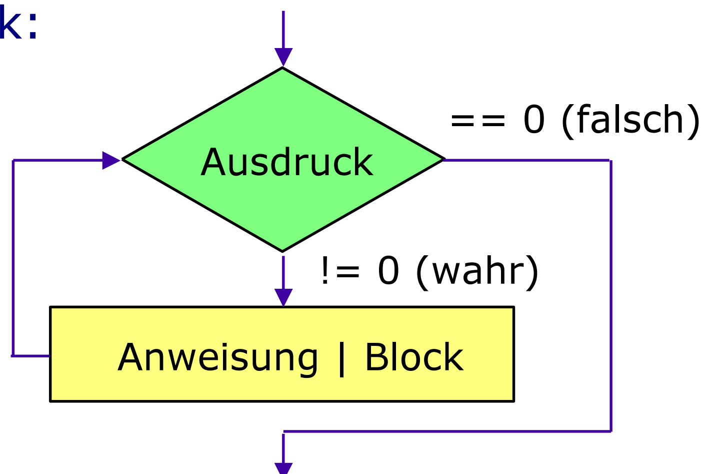
Semantik der for-Schleife:



Syntax der kopfgesteuerten while-Schleife:

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

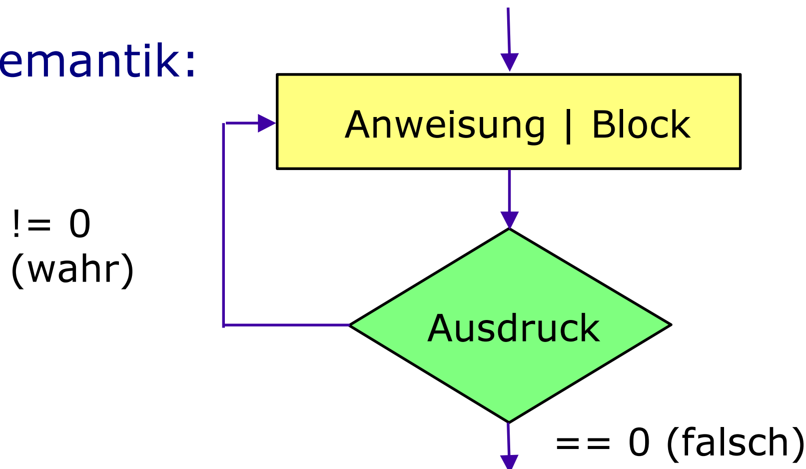
Semantik:



Syntax der fußgesteuerten while-Schleife:

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

Semantik:

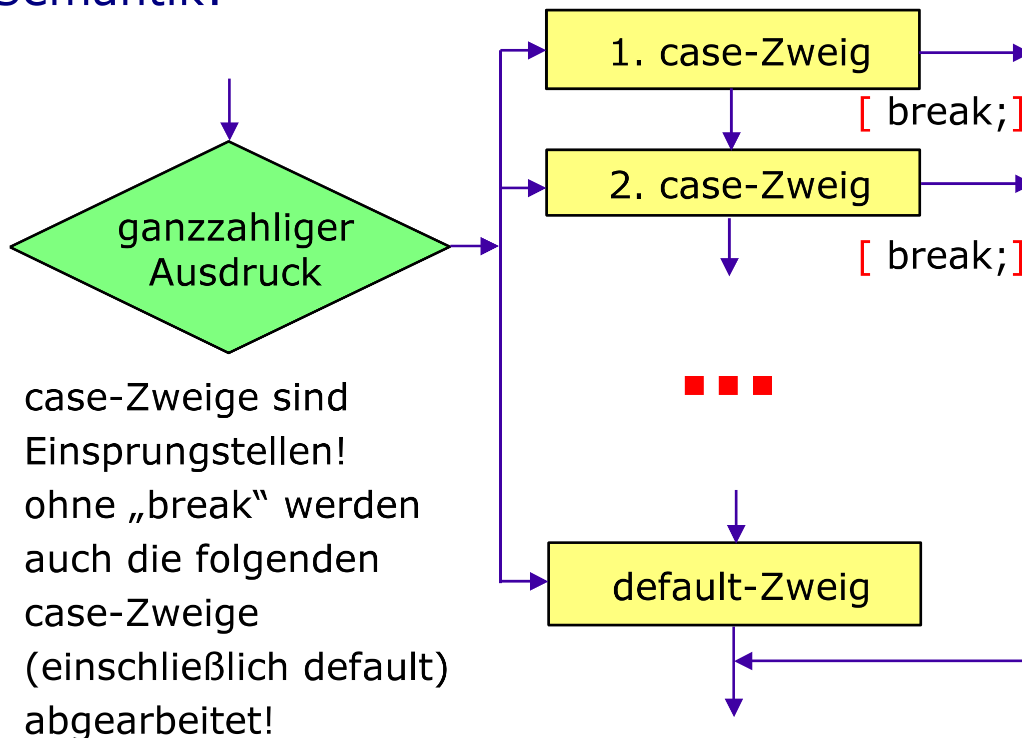


Syntax der switch-Struktur:

```
switch (ganzzahliger Ausdruck)
{
   case genau eine ganzz. Konstante:
{                                    {Anweisung}          }1
                                     {Block}
                                     [break;]

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

Kontrollstrukturen (8/10)

Semantik:



case-Zweige sind Einsprungstellen! ohne „break" werden auch die folgenden case-Zweige (einschließlich default) abgearbeitet!

Kontrollstrukturen (9/10)

- continue
  - Nur für Schleifen „for", „while" und „do ... while" gültig
  - Der aktuelle Schleifendurchlauf wird abgebrochen, die Reinitia-lisierung wird ausgeführt (bei der for-Schleife) und es wird zum Schleifentest gesprungen
- break
  - Nur für Schleifen „for", „while", „do ... while" und „switch"
  - Nur die unmittelbar umgebende Schleife oder „switch" wird verlassen (kein Sprung aus mehreren ineinander geschachtelten Schleifen möglich!)

Kontrollstrukturen (10/10)

- goto
  - äußerst sparsam verwenden!
  - Im Sinne strenger strukturierter Programmierung nicht erlaubt!
- *Syntax:* goto Marke; . . . Marke:
- return
  - Rückkehr in unmittelbar aufrufende Funktion (keine Funktion!)
- *Syntax:* return [Ausdruck];
- exit (ganzzahliger Ausdruck); Standardfunktion!

2.10 Strukturierte Programmierung

- Die ersten höheren Programmiersprachen (z.Bsp. FORTRAN; FORmula TRANslator) waren geprägt durch viele Verzweigungen und Sprünge (goto)
- Dadurch wurden die Programme schwer verständlich und unübersichtlich.
- Es wurde das Konzept der sogenannten „strukturierten Programmierung“ entwickelt, das völlig ohne „goto“ auskommt.
- Dieses Konzept wurde u.a. in den Programmiersprachen ALGOL, ALGOL68, Pascal und C verwirklicht.
- Für die einzelnen Kontrollstrukturen wurden durch Isaac Nassi und Ben Shneiderman entsprechende grafische Darstellungselemente entwickelt.

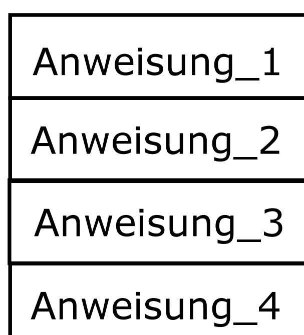
Struktogramme in der Praxis

- Vorüberlegungen mit Zettel und Bleistift können sehr hilfreich sein
- Anfänger versuchen oft die Programme durch reines Probieren zum Laufen zu bringen, was nicht zum Erfolg führt und viel Zeit kostet
- Obwohl theoretisch Kontrollstrukturen und die grafischen Elemente nach Nassi-Shneiderman völlig gleichwertig sind, erfordert das Erstellen von NS-Diagrammen mehr Nachdenken über den zugrunde liegenden Algorithmus
- Die Programmierer werden gezwungen, sich gründlicher mit dem Algorithmus zu beschäftigen
- Programme können mit NS-Diagrammen vereinfacht und übersichtlicher beschrieben werden

Struktogramm

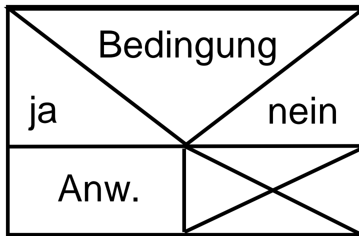
- I. Nassi und B. Shneiderman schlugen 1973 diese Darstellungsart vor, deshalb auch Nassi-Shneider-man-Diagramm (DIN 66 261)
  - Grundform ist ein Strukturblock (Rechteck)
  - Abarbeitungsreihenfolge erfolgt von oben nach unten

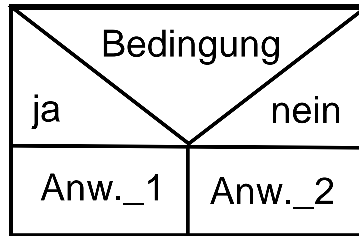
- Anweisung



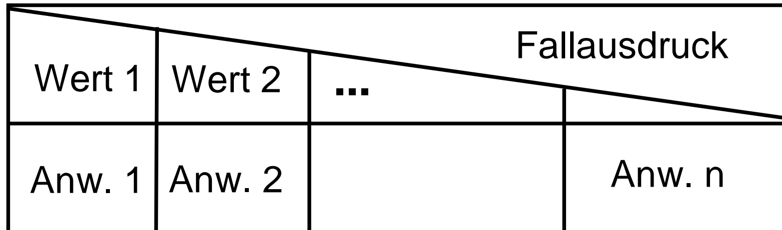
- Sequenz / Folge

Verzweigungen

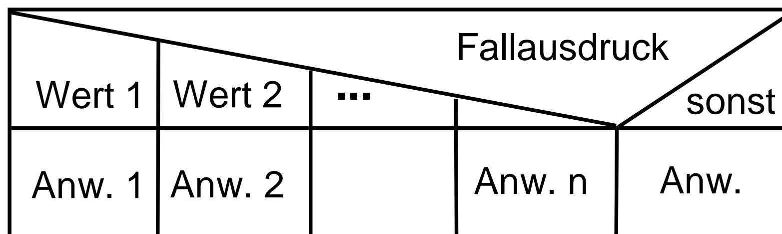




- Selektion / Verzweigung mit und ohne Alternative

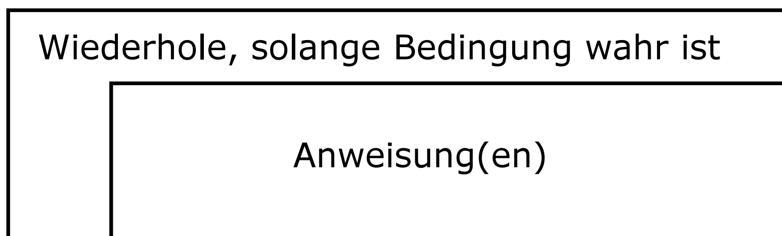


- Mehrfachverzweigung ohne „default"-Zweig

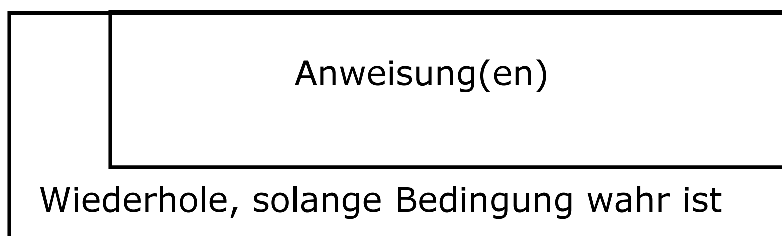


- Mehrfachverzweigung mit „default"-Zweig

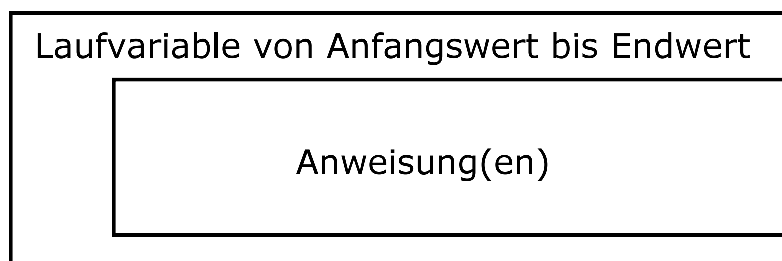
Schleifen



- Kopfgesteuerte Schleife (abweisender Zyklus)



- fußgesteuerte Schleife (mindestens ein Durchlauf)



- Schleife von Anfangswert bis Endwert (Zählschleife)

Verbesserungsvorschlag

- Vorschlag (E. Engelhardt Dez. 2002)
  - Bereitstellung von grafischen Elementen für Programmier-elemente aus C/C++, C#, Java, J++, JavaScript, …
  - In oben genannten Sprachen funktioniert die Mehrfachver-zweigung anders als z.B. in ALGOL, Pascal, ...

- Mehrfachver-zweigung mit „default"-Zweig
- Anweisungen können „break" enthalten oder auch nicht
- (ohne „default"-Zweig analog)

3 Funktionen

- Wiederverwendbarkeit
  - Beliebig oft ausführbar
  - Programm wird kürzer und einfacher
- Modularität
  - Einfachere Pflegbarkeit
  - Wiederverwendbarkeit
- Portabilität
  - nicht-ANSI-Programmteile in extra Funktionen packen

- Funktionen sind alle gleichberechtigt, sie können nicht geschachtelt definiert werden (wie z.Bsp. in Pascal)
- Funktionen sind global, das heißt überall bekannt
- „main" ist die Anfangsadresse des Programms

3.1 Arbeit mit Funktionen

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

3.2 Funktionen-Deklaration

- Funktionen-Deklaration mit K&R-C (nach wie vor erlaubt!)
  - Deklaration und Definition von Funktionen sind die größten Unterschiede zwischen ANSI-C und K&R-C
  - Unter K&R-C ist die Deklaration nicht unbedingt nötig
  - Deklaration nach K&R: `int max();` (keine Parameter!)
  - Definition nach K&R:

```
int max(a, b)
int a, b;
{ return ((a > b) ? a : b); }
```

- Funktionen-Deklaration nach ANSI-C
  - `int max(int a, int b);   // Prototyp`
    Parameternamen können fehlen: `int max(int, int);`
  - `int max(int a, int b)    // Definition`

```
{ return ((a > b) ? a : b); }
```

3.3 Rückkehrwert (return-Wert)

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

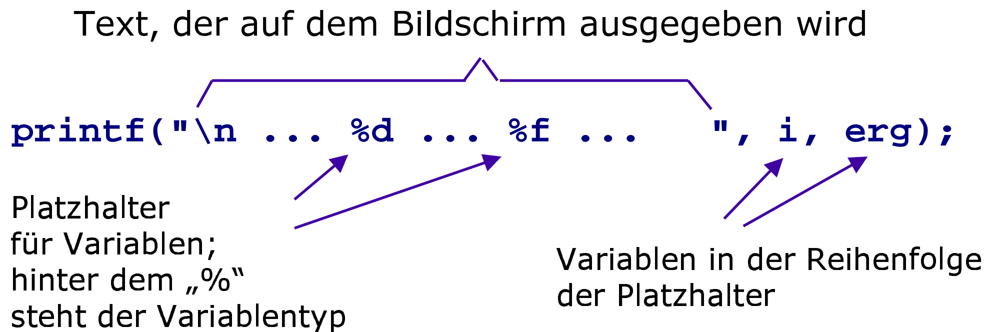
```
double wurzel(double x)   // Definition
{
   ...

   return ergebnis;
}

. . .

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


3.4 Standardfunktionen

- 1. Funktionen-Deklaration
  - Die Deklaration (Prototyp) steht in der entsprechenden Header-Datei (Verzeichnis „...\include")
  - Die Header-Datei wird im Programm mittels Präcompiler-Anweisung angegeben. Beispiel: #include <stdio.h>
  - durch die spitzen Klammern wird das Standard-inlude-Verzeichnis durchsucht
- 2. Funktionen-Definition
  - die Funktion ist in der entsprechenden Bibliothek (Verzeichnis „...\lib")
  - Durch den Linker werden die entsprechenden Funktionen aus den Bibliotheken hinzugefügt
- 3. Funktions-Aufruf

Funktion „printf"

Syntax am Beispiel:



Bemerkung: „\n" steht für „newline" (neue Zeile)

- Syntax: %[Flags][Feldbreite][.Genaugikeit]typ
- Variablentypen (typ)
  - **d** (dezimal), ganze vorzeichenbehaftete Zahlen vom Typ „int"
  - **i** (integer), Typ „int"
  - **ld** (long dezimal, long), Typ „long int" bzw. „long"
  - **seit C99: lld** (long long dezimal, long long)
  - **u** (unsigned int)
  - **o** (oktal), **x** (hexadezimal), ganze Zahlen
  - **f** (float/double Gleitkomma)
  - **e, g** (float/double Exponential)
  - **Lf** (long double)
  - **c** (char einzelnes Zeichen)
  - **s** (string/Zeichenkette/Text), Felder von Zeichen mit binärer Null
  - **p** (pointer Zeiger)

Standardfunktionen

- Flags
  - + (Vorzeichen), Leerzeichen (führende Leerzeichen)
  - **-** (linksbündig)
- Feldbreite
  - ganze Zahl für Länge des Ausgabefeldes
- Genauigkeit (Nachkommastellen)
- Beispiele:

```
printf("\n%-20s|%-20s|%8.2f",titel,autor,preis);

ergibt zum Beispiel:

ANSI-C Buch         |B. Müller           |   29.95
```


Standardfunktionen

Funktion „scanf“

Syntax am Beispiel:

```
printf("Eingabe: ");
fflush(stdin);  // Leeren des Eingabepuffers
scanf("%d %f", &anzahl, &operand);
```

Platzhalter
für Variablen;
hinter dem "%"
steht der Variablentyp
keine Textausgabe!

Variablen in der Reihenfolge
der Platzhalter;
Es muss die Adresse übergeben
werden!

3.5 „call by value", „call by reference"

- „call by value":
  - Von den übergebenen Parametern werden lokale Kopien in den Funktionen angelegt.
  - Es können keine in der Funktion geänderten einzelnen Werte über Parameter zurück gegeben werden.
- „call by reference"
  - Es werden die Adressen übergeben, von denen auch Kopien angelegt werden. Auf den Adressen geänderte Werte sind auch im aufrufenden Programm geändert.
- In „C/C++" gilt „call by value" (Wertübergabe)
  - Sollen Werte übergeben werden, die in der Funktion geändert werden, so müssen die Adressen der Werte übergeben werden.
  - Die Adresse wird durch den Adressoperator „**&**" geliefert (Beispiel: „scanf")

Beispiel: „call by value"

```
#include <stdio.h>

void tausch(int a, int b)  // funktioniert so nicht
{
   int hilf;

   hilf= a; a= b; b= hilf;
   printf("tausch: a= %d, b= %d\n", a, b);
}

int main(void)
{
   int x= 5, y= 7;

   tausch(x, y);
   printf("main: x= %d, y= %d\n", x, y);

   return 0;
}
```

Beispiel: „call by reference"

```
#include <stdio.h>

void tausch(int *a, int *b)  // funktioniert
{
   int hilf;

   hilf= *a; *a= *b; *b= hilf;
   printf("tausch: a= %d, b= %d\n", *a, *b);
}

int main(void)
{
   int x= 5, y= 7;

   tausch(&x, &y);
   printf("main: x= %d, y= %d\n", x, y);

   return 0;
}
```

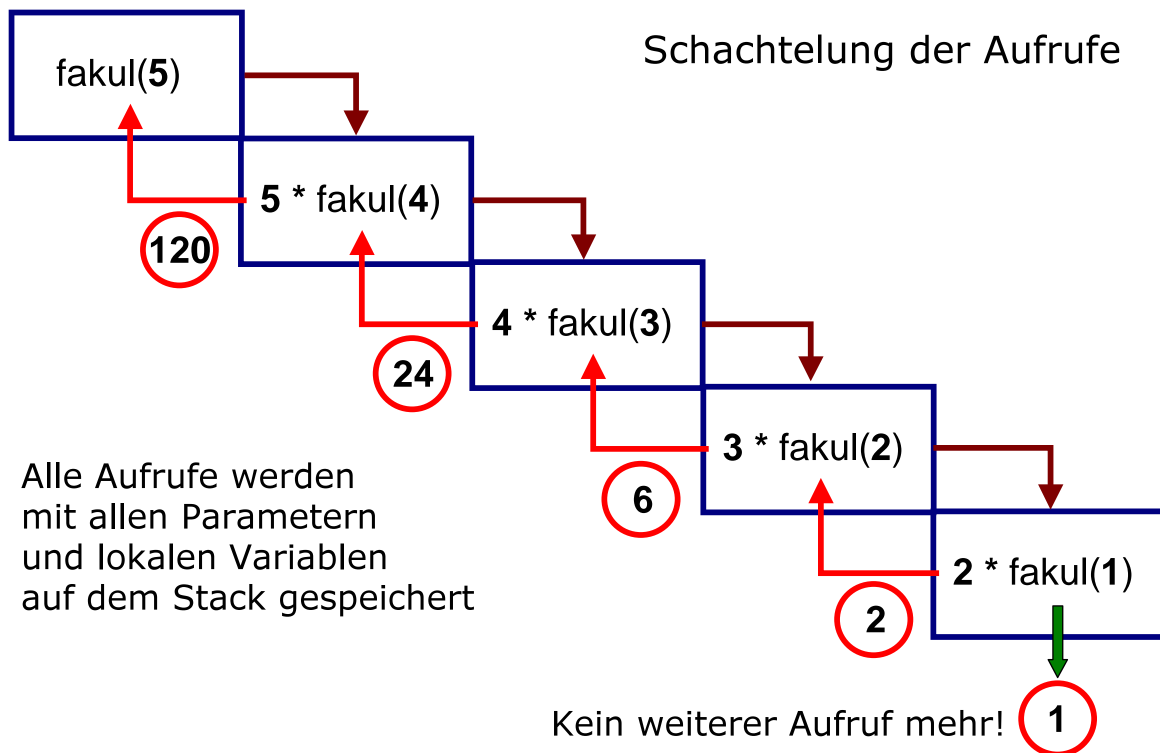

3.6 Rekursive Funktionen

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

Beispiel: Fakultät

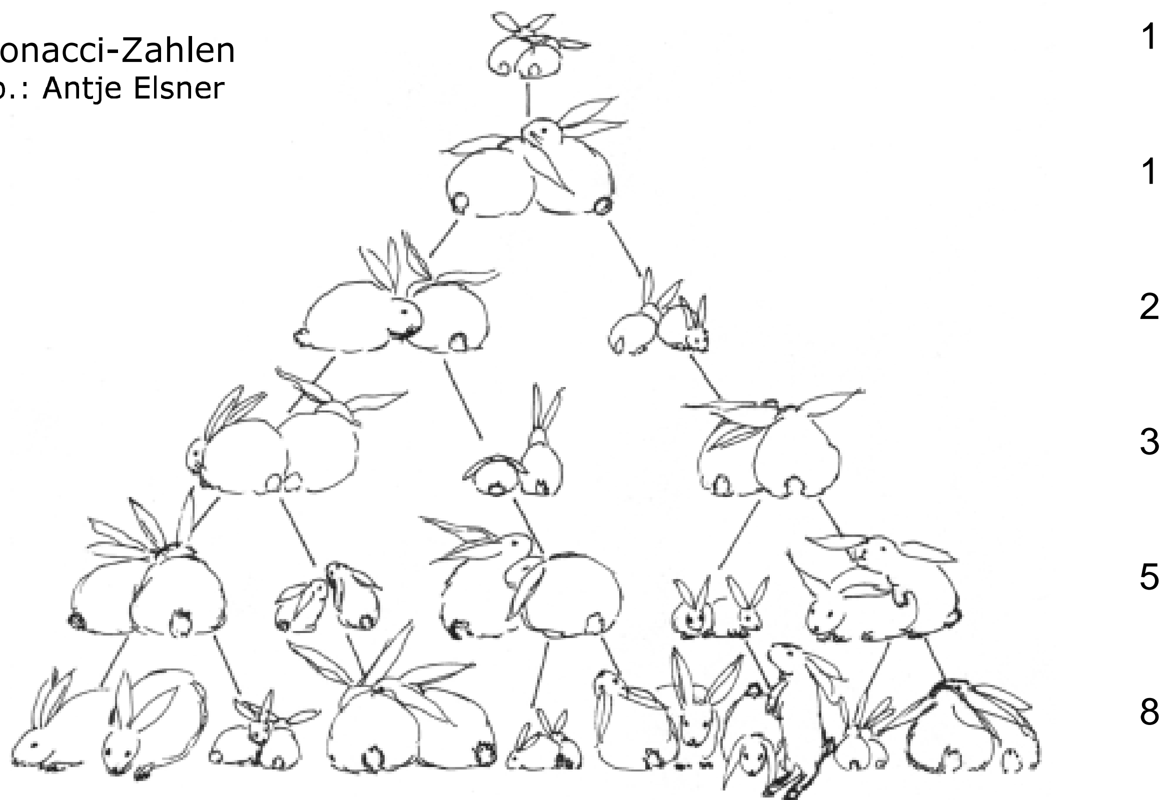
```
#include <stdio.h>

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

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

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

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

   return fak;
}
```


Beispiel: Fakultät



Alle Aufrufe werden
mit allen Parametern
und lokalen Variablen
auf dem Stack gespeichert

Fibonacci-Zahlen
Abb.: Antje Elsner



1

1

2

3

5

8

- Fibonacci: Leonardo von Pisa (1170 - 1240)
- Wieviele Kaninchenpaare gibt es nach einem Jahr, wenn es anfangs ein junges Paar gibt, jedes Paar jeden Monat ein neues Paar zur Welt bringt und alle Jungen sich jeweils nach einem Monat fortpflanzen?

- Zahlenreihe:
  1 1 2 3 5 8 13 21 34 55 89 144 …

- Mathematische Beschreibung:
  $F_0 = 1$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$; $n >= 2$

Beispiel: Fibonacci

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

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


3.7 Parameterübergabe an „main“

- Aufruf des Programms: `> editor datei.dat`

```
#include <stdio.h>

int main(int argc, char **argv, char **env)
{
   FILE *ein_ptr;

   if (argc == 2)
   {
      if ((ein_ptr= fopen(argv[1], "r"))== NULL)
         return 1;
      // Datei einlesen

        • • •
```

Argumentenliste „argv“

- In „argc“ (Argument-Counter) ist die Anzahl der Argumente gespeichert. Dabei wird der Name des Programms selbst mitgezählt.
- „argv“ ist ein Vektor (Feld) von Zeichenketten und enthält die Argumente. Das erste Argument ist der Name des Programms selbst
- Der Vektor „env“ enthält die Environment-Variablen des Systems (ohne Zählvariable)

Index	0	1	2	3	4	5	6	7	8	9
argv[0]	'e'	'd'	'i'	't'	'o'	'r'	'\0'	...		
argv[1]	'd'	'a'	't'	'e'	'i'	'.'	'd'	'a'	't'	'\0'

Funktionen mit variabler Argumentenanzahl

va_arg, va_end und va_start (<stdarg.h>)

- „va_arg“ liefert das nächste optionale Argument der Funktion; als zweiter Parameter muss der Typ dieses Argumentes angegeben werden
- „va_end“ wird aufgerufen, wenn alle optionalen Parameter gelesen wurden; der Zeiger auf die Argumente vom Typ „va_list“ wird auf 0 gesetzt
- „va_start“ initialisiert den Zeiger auf die optionalen Argumente; der zweite Parameter von va_start ist der letzte feste Parameter der Funktion

Funktionen mit variabler Argumentenanzahl

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

int maximum(int anz, ...)
{
   int i, h, erg= 0;
   va_list arg_zeiger;

   va_start(arg_zeiger, anz);  // Initialisierung des Zeigers
   erg= va_arg(arg_zeiger, int);

   for (i= 1; i < anz; i++)
     if (h= va_arg(arg_zeiger, int))
        erg= h;

   va_end(arg_zeiger);       // arg_zeiger wird auf 0 gesetzt

   return erg;
}

x = maximum(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  // Aufruf
```


Zeiger auf Funktionen (Beispiel)

```
// Prototype in <stdlib.h>
void qsort(void * _base,       // Zeiger auf Feld
      size_t _nmemb,           // n Elemente
      size_t _size,            // Elementgroesse
      int (*_compar) (const void *, const void *));
// _compar ist ein Zeiger auf eine Vergleichsfunktion
// mit zwei Zeigern auf zu vergleichende Elemente

// Aufruf:
   qsort(feld, anz, sizeof(int), vergleich);

// Vergleichsfunktion
int vergleich(int *z1, int * z2)
{
   return (*z1 < *z2 ? -1 : 1);
}
```

4 Felder (Vektoren, Arrays)

- Ein Feld ist die Aneinanderreihung von Elementen gleichen Typs (elementare Datentypen, Strukturen, selbst definierte Datentypen, ..). Die Elemente stehen im Speicher lückenlos mit aufsteigenden Adressen hintereinander.

- Es gibt in „C‟ keine Sprachelemente zur Verarbeitung von kompletten Feldern (die eckigen Klammern sind nur Syntaxelemente für die Arbeit mit Indizes)! Die Arbeit mit Feldern erfolgt mit Standardfunktionen oder elementweise.

- Der Zugriff zu den einzelnen Elementen eines Feldes erfolgt über eine Nummerierung (Index), die in „C‟ bei Null beginnt!

4.1 Abeit mit Feldern

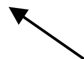
- Der Name des Feldes ist die Anfangsadresse des Feldes, also die Adresse des ersten Elementes (mit Index 0).
  feld = &feld[0]
- Wird die Anfangsadresse des Feldes benötigt (z.B. bei Parametern im Funktionenaufruf), kann einfach der Name des Feldes angegeben werden.

- Beispiel: Feld ganzer Zahlen

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

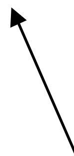
Vereinbarung von Feldern und Zugriff

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



Anzahl der Feldelemente muss bei der Compilierung bekannt sein (bis „C99").
Es kann ein ganzzahliger aber konstanter Ausdruck angegeben werden.
Ab C99 kann die Anzahl der Feldelemente während der Laufzeit ermittelt werden, kann aber danach nicht mehr verändert werden (semidynamisch).

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



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

Feld-Vereinbarung

- Beispiele für Feldvereinbarungen

```
#define ANZ 10

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

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

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

Indizes

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

4.2 Zeichenketten (String, Texte)

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

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

Vereinbarung von Texten

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

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

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

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

- Der Datentyp „char" wird in „C" wie ein ganzzahliger Datentyp behandelt.

4.3 Pointer, Zeiger

- Die Arbeit mit Pointern (Zeigern) macht C-Programme effizienter (Arbeit mit Adressen)
- Mit Pointern kann nur gearbeitet werden, wenn die entsprechenden Daten, worauf die Pointer zeigen vorhanden sind, d.h. vereinbart wurden.
- Pointer selbst belegen auch Speicherplatz, und zwar so viel, wie zur Speicherung einer Adresse nötig ist.
- Die Pointer erhalten den Typ, wie das, worauf sie zeigen (Pointer enthalten Adressen)
- Mit Pointern kann gerechnet werden (Zeiger-Arithmetik)
- Alles was mit Pointern realisiert werden kann, kann auch mit anderen Mitteln (Felder, Indizes, …) realisiert werden.

Pointer, Zeiger

```
int *ptrInt;      // Vereinbarung eines Zeigers,
                  // der auf int-Daten zeigt
int feld[10];
```

Der Stern bei der Vereinbarung bedeutet, dass es sich um die Vereinbarung eines Zeigers (Pointers) handelt. Der Inhalt des Zeigers ist zufällig, kann aber schon bei der Vereinbarung einen Anfangswert erhalten.

```
. . .

ptrInt= feld;     /* Pointer erhält Adresse des */
                  /* Feldes (des 1. Elementes)  */
. . .             /* gleichbedeutend mit:       */
                  /* ptrInt= &feld[0];          */
```

Irgendeine Zuweisung dieser Art muss immer vor der Arbeit mit Zeigern vorhanden sein.

Pointer, Zeiger

- Beispiele für Pointer-Vereinbarungen

```
int *ptrInt;          /* zufälliger Anfangswert */

int *ptrFeld[5];      /* Feld von fuenf Zeigern */

char text[80+1];

char *ptrChar= text; /* Anfangsinitialisierung */
```

- Der Name eines Feldes ist die Adresse auf das erste Element, und damit wie ein Zeiger auf dieses Element
- Der Name eines Feldes darf aber nicht auf der linken Seite einer Zuweisung stehen, da es ein konstanter Zeiger ist

4.4 Zeiger-Arithmetik

- Die Zeigervariablen enthalten Adressen, mit denen gerechnet werden kann.
- Die Verantwortung, dass nach allen Zeigeroperationen die Zeiger auf gültige Daten zeigen liegt beim Programmierer.

```
zeiger++; bzw. ++zeiger;   /* Inkrement */
zeiger--; bzw. --zeiger;   /* Dekrement */
```

- Die Zeigervariable enthält die Adresse des nächsten (bei Inkrement) bzw. vorherigen (Dekrement) Elementes entsprechend des Typs der Zeigervariablen.
- Es werden so viele Byte weiter gezählt, wie es dem Typ der Zeigervariablen entspricht.

Zeiger-Arithmetik

```
Zeiger + (ganzzahliger Ausdruck) bzw.
Zeiger - (ganzzahliger Ausdruck)
```

- Die ganze Zahl (Auswertung des ganzzahligen Ausdrucks) wird mit der Anzahl Byte entsprechend des Typs der Zeigervariablen multipliziert.

```
Anzahl = Zeiger1 - Zeiger2;
```

- Anzahl der Elemente zwischen den Zeigern, wenn die Adresse *Zeiger1 vor Adresse *Zeiger2 liegt (Zeiger1 enthält größere Zahl).

Sinn von Zeigern (Pointern)

- Die effizientere Arbeit und dynamische Datenstrukturen sind der Hauptgrund für die Verwendung von Pointern in „C"
- Berechnung der Zieladresse:

```
i++;                /* Indexvariable erhoehen   */
feld[i] = ...;      /* Zugriff auf das Element */
```

feld + i * (Anzahl Byte entsprechend Datentyp)

⇩

(Anfangsadresse des Feldes)

```
ptrFeld++;          /* Zeigervariable erhoehen */
*ptrFeld= ...;      /* Zugriff auf Inhalt      */
```

Die Zeigervariable enthält bereits nach der Erhöhung die richtige Adresse

Zusammengesetzte Datentypen

- In „C" gibt es zwei Arten von zusammengesetzten Datentypen, Strukturen (Verbund) und Unions (Variante).
- Bei Strukturen stehen die einzelnen Bestandteile im Speicher hintereinander, während bei Unions die einzelnen Bestandteile an der selben Speicheradresse beginnen.
- Die Länge einer Struktur wird durch die Addition der einzelnen Längen bestimmt.
- Bei Unions ist die Länge einer Union gleich der Länge des längsten Bestandteils
- Die Länge sollte auf jeden Fall mit „sizeof" ermittelt werden, wenn sie benötigt wird, da Strukturen oft an Maschinenwortgrenzen angelegt werden. Die tatsächliche Länge ist dann ggf. anders als eine Summierung der Längen der einzelnen Bestandteile ergeben würde.

5.1 Aufbau einer Struktur

```
struct M_Person
{
   char name[80+1];
   char vorname[80+1];
   int  geburtsjahr;
} person;
```

Aufbau der Struktur, die einzelnen Bestandteile (Strukturvariablen) werden hintereinander geschrieben

Variable dieser Struktur, Name und Speicherplatzreservierung

- Es können auch mehrere Variablen dieser Struktur angelegt werden
- Die Bezeichnung „M_Person“ (Musterstruktur) ist keine Variable, sondern bezeichnet das Aussehen der Struktur. Dabei wird noch kein Speicherplatz reserviert, das erfolgt erst beim Anlegen durch „person“.

5.2 Zugriff auf Bestandteile der Struktur

```
struct M_Person person1, person2;

struct M_Person it1[44]; // Feld von Strukturen
```

- Mit Hilfe der Musterstruktur (Strukturbeschreibung können weitere Strukturen angelegt werden.
- Es können auch Felder von Strukturen angelegt werden

```
strcpy(person.name, "Meier");

person.geburtsjahr= 1980;
```

- Der Separator (Trennzeichen) für den Zugriff auf die Bestand-teile (Strukturvariablen) ist der Punkt.

Zugriff auf Bestandteile der Struktur

```
scanf("%d", &person.geburtsjahr);

it1[i].geburtsjahr= 1980;
```

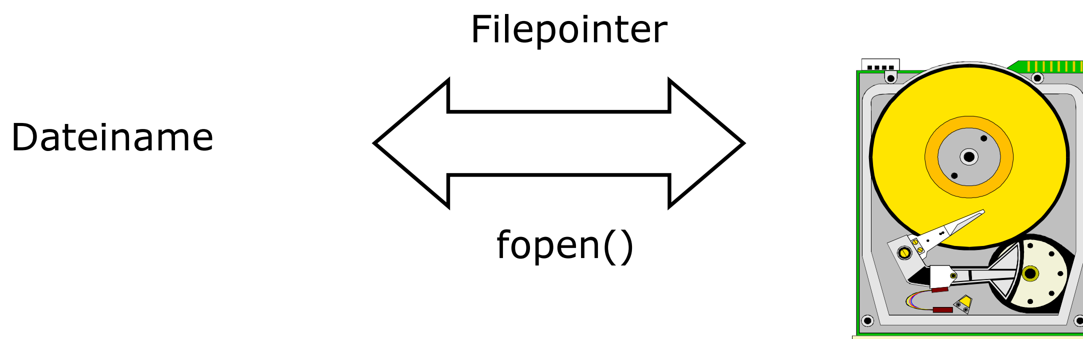
- Auch bei Strukturen muss der Adressoperator (wenn Struktur-variable kein Feld ist) bei „scanf" angegeben werden! Der Adressoperator bezieht sich auf den Bestandteil der Struktur (geburtsjahr).
- Der Zugriff auf einzelne Strukturen eines Feldes von Strukturen erfolgt über den Index. Dieser wird vor den Separator (Punkt) für die Bestandteile geschrieben.

5.3 Zeiger auf Strukturen

```
struct M_Person
{
   char name[80+1];
   char vorname[80+1];
   int  geburtsjahr;

} it1[44], *ptrStudent;

ptrStudent = it1;
 …

gets((*ptrStudent).name); // so waere die Schreib-
                          // weise bisher, wird
                          // aber so nicht gemacht!
sondern:

gets(ptrStudent->name);   // mit "Pfeil" wird sym-
                          // bolisiert, dass davor
                          // eine Zeigervariable
                          // steht
```

6 Dateiarbeit

- In „C" ist jede Ein- Ausgabe (Files, Sockets, Schnittstellen, usw.) Dateiarbeit
- In „C" gibt es keinerlei Datensatz-Strukturen (Records) Alles muss durch den Programmierer selbst angelegt werden
- Ein Filepointer stellt die Verbindung zwischen dem Dateinamen und dem Speicherplatz auf der Platte her



6.1 Filepointer

- Typdefinition in „stdio.h" (so oder so ähnlich)

```
typedef struct
{
   int level;                   /* File besetzt    */
   unsigned flags;              /* File-Status     */
   char fd;                     /* File Descriptor */
   unsigned char hold;          /* Zeichenpuffer   */
   int bsize;                   /* Puffergroesse   */
   unsigned char *buffer;       /* Puffer          */
   unsigned char *curp;         /* aktiver Zeiger  */
   unsigned istemp;             /* Temporary File  */
   short token;                 /* Gueltigkeit     */
} FILE;                         /* Typname         */
```

6.2 Datei unformatiert schreiben (DVL)

```
int datei_schreiben(char *datei)
{
   FILE *aus_fp;                    // Filepointer
   if ((aus_fp= fopen (datei,"wb")) == NULL) return -5;
   dvl_aktuell= dvl_anfang;

   while (dvl_aktuell))             // solange nicht dvl_ende
   {
      dvl2struct();                 // DS aus DVL ohne Zeiger
      if (!fwrite((char *) &buch,sizeof(buch),1,aus_fp))
         return -6;
      dvl_aktuell= dvl_aktuell->danach;
   }
   dvl_aktuell= dvl_anfang;
   fclose(aus_fp);                  // Datei schliessen
   return 0;
}
```


6.3 Datei unformatiert lesen (DVL)

```
int datei_lesen(char *datei)
{
   FILE *ein_fp;                       // Filepointer
   if ((ein_fp= fopen (datei,"rb")) == NULL) return -3;
                                       // Vorlesen
   fread ((char *) &buch,sizeof(buch),1,ein_fp);
   while (!feof(ein_fp))               // Test auf Dateiende
   {
      ds_anlegen();                    // DS fuer DVL anlegen
      struct2dvl();                    // DS mit Daten fuellen
                                       // Nachlesen
      fread((char *) &buch,sizeof(buch),1,ein_fp);
   }
   dvl_aktuell= dvl_anfang;
   fclose(ein_fp);                     // Datei schliessen
   return 0;
}
```

6.4 Übersicht Dateiarbeit

	Bildschirm E/A	Datei E/A	System-E/A	String E/A
Eingabe	scanf	fscanf	read	sscanf
formatiert	gets	fgets		
	getchar	fgetchar		
	getc	fgetc		
unformatiert		fread		
		fopen	open / create	
		freopen		
Ausgabe	printf	fprintf	write	sprintf
formatiert	puts	fputs		
	putchar	fputchar		
	putc	fputc		
unformatiert		fwrite		
	perror	clearerr		
		fclose	close	
		ferror	remove	
		feof	rewind	
Positionierung		ftell / fseek	rename	
		fgetpos		
		fsetpos		

7 Der Präcompiler

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

7.1 include-Direktive

- Alle Direktiven (Anweisungen) für den Präcompiler beginnen mit dem Doppelkreuz (Raute) „#“
- Mit der include-Direktive können im Prinzip beliebige Text-dateien zur Quelle hinzugefügt werden. Es werden üblicher-weise nur sogenannte Header-Dateien mit Extension „.h“ eingefügt.
- Standard-Headerfiles werden in spitze Klammern einge-schlossen.
- Eigene Headerfiles stehen in „Anführungszeichen“

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


7.2 define-Direktive

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

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

7.3 Makros

- Da Makros vor der Compilierung in den Quelltext eingesetzt werden, werden sie schnell ausgeführt. Es entfällt der Overhead beim Funktionsaufruf.
- Makros sind typunabhängig, das heißt, sie funktionieren mit int-Variablen, float-Variablen, usw.
- Wenn Makros längere Programmstücke enthalten und oft im Quelltext enthalten sind, wird das Programm länger.
- Makros werden üblicherweise für kleinere, mehrfach vorkommende Programmstücke eingesetzt
- Makros können bei sinnvoller Anwendung die Lesbarkeit von C-Programmen erhöhen
- Makros sind anfällig gegenüber Programmierfehlern (z.B.Klammerung)

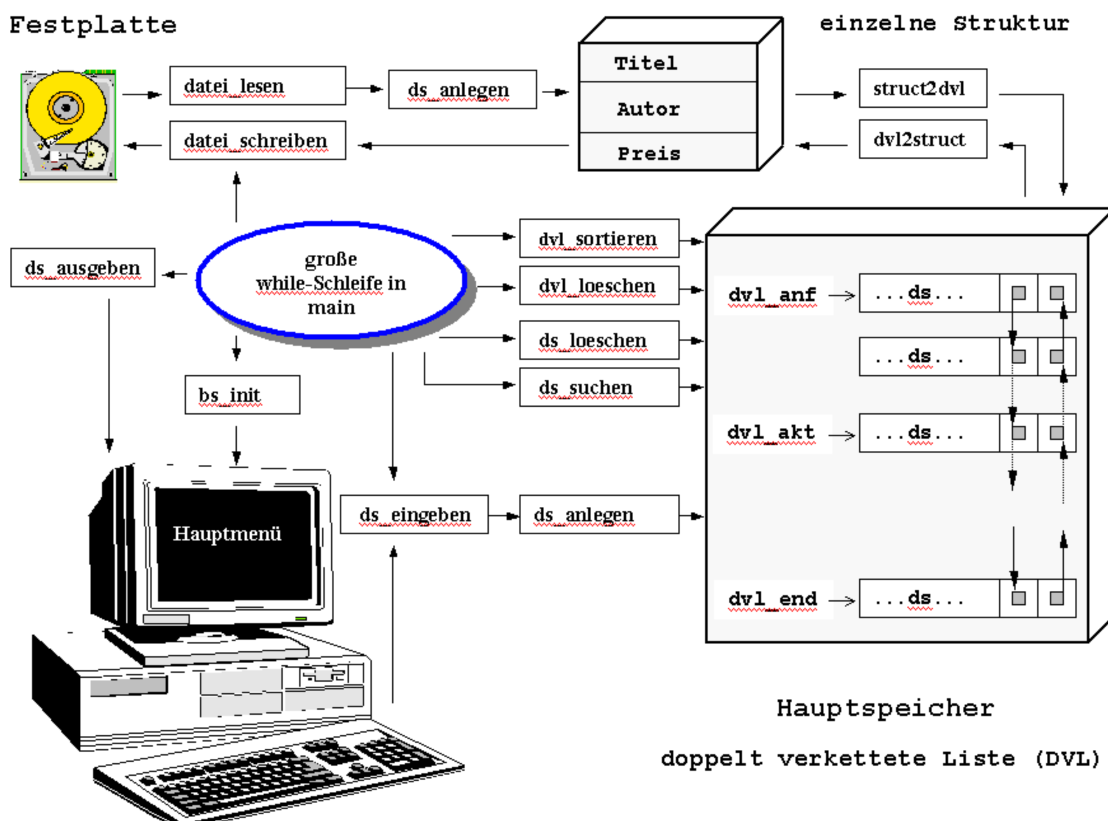
Makros

```
#define QUADRAT(x) (x)*(x)       // Klammerung!
#define ABS(x)     ((x) > 0) ? (x) : -(x)
#define MAX(x,y)   (((x) > (y)) ? (x) : (y))
...

   c= (a > b) ? a : b;

   c= MAX(a,b);
```

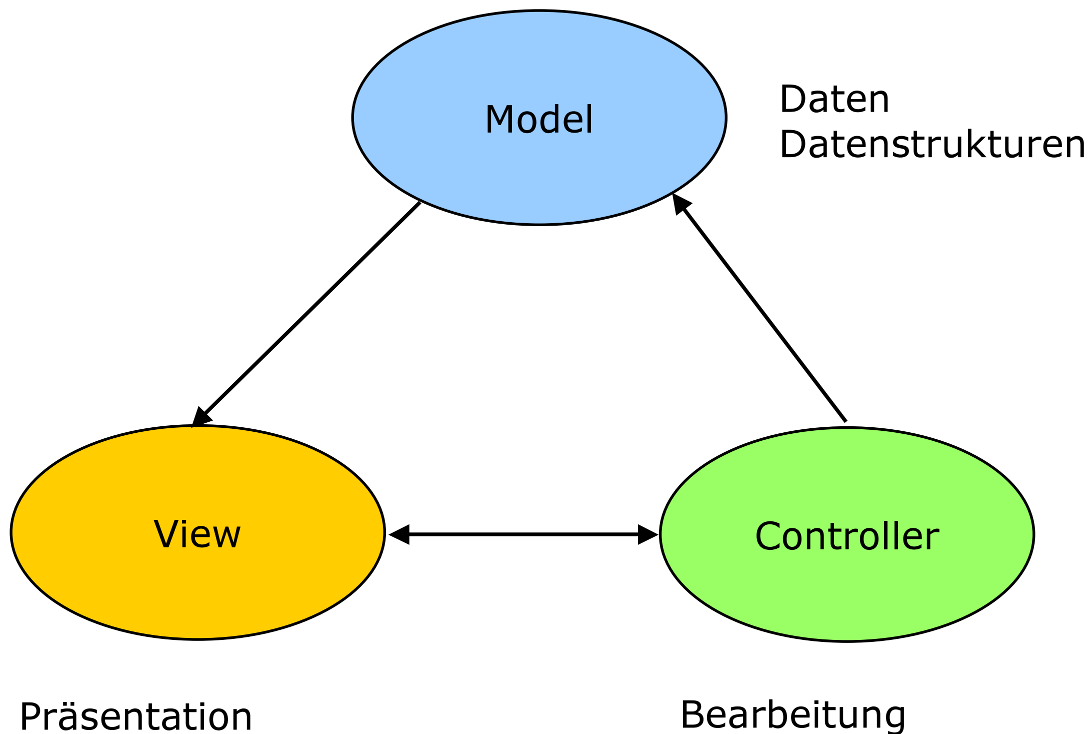
8 Dynamische Datenstrukturen

- Dynamische Datenstrukturen werden benötigt, um bei Bedarf im Programm Speicher für die Speicherung von Daten bereit zu stellen
- Felder (Arrays) sind in „C" statisch, bzw. ab „C99" semi-dynamisch
- Es gibt in „C" keine Standardfunktionen für dynamische Datenstrukturen (Stack, Baumstruktur, verkettete Liste, Hash-Tabelle, usw.)
- Natürlich gibt es Programmpakete und Bibliotheken, die solche Datenstrukturen bereit stellen
- Beispielhaft soll hier anhand einer doppelt verketteten Liste (DVL) das Anlegen einer dynamischen Datenstruktur vor-geführt werden

8.1 Übersicht zur Mini-Datenbank



8.2 Das klassische MVC-Konzept

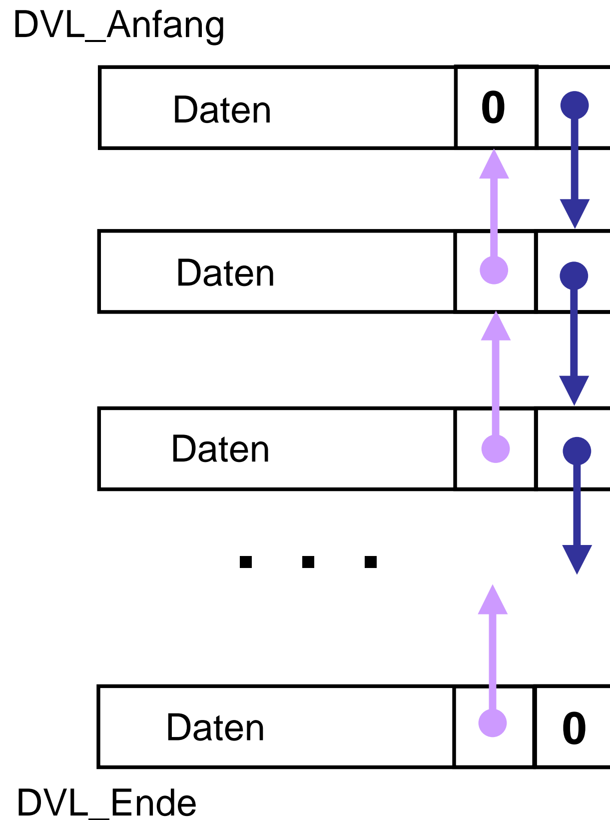


Anwendung des MVC-Konzeptes

- Die strenge Anwendung des MVC-Konzeptes bewirkt:
  - Modularer Aufbau von Programmen
  - Bessere Aufteilung der Aufgaben (mehrere Programmierer)
  - Bessere Pflegbarkeit der Programme
  - Austauschbarkeit von Funktionen
  - ... (siehe auch Programmierprojekte)

8.3 Doppelt verkettete Liste (DVL)

- Bei einer DVL ist ein Zeiger auf den nächsten Datensatz und ein Zeiger auf den vorherigen Datensatz bereits im Datensatz enthalten.
- Dazu muss die Struktur für die Datensätze entsprechend aufgebaut sein.



8.4 Typedefinition für DVL

```
struct M_Buch   // Muster/Modell
{
   char   titel[80+1];
   char   autor[80+1];
   float preis;
   struct M_Buch *davor;
   struct M_Buch *danach;

};

typedef struct M_Buch T_Buch;    // Name des Typs

// globale Daten (in allen Funktionen bekannt)

T_Buch *dvl_anfang= 0;    // Zeiger auf DVL-Anfang
T_Buch *dvl_ende=   0;    // Zeiger auf DVL-Ende
T_Buch *dvl_aktuell=0;    // Zeiger auf aktuellen DS
```

8.5 Anlegen eines Datensatzes (DS) (1/2)

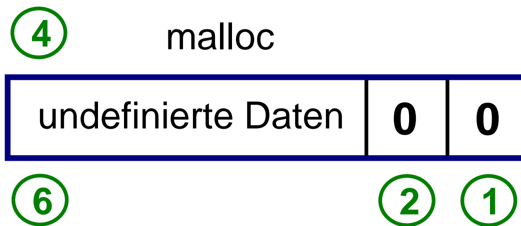
```
int ds_anlegen(void)     // DS wird an das Ende angehaengt
{
   if (!(dvl_aktuell= (T_Buch *) malloc(sizeof(T_Buch))))
   {
      fehlerAusgabe("ds_anlegen", -2);
      return -2;
   }
   dvl_aktuell->danach= 0;                 // 1
   dvl_aktuell->davor=  dvl_ende;          // 2

   if (dvl_anfang == 0)                    // 3
      dvl_anfang= dvl_aktuell;             // 4
   else
      dvl_ende->danach= dvl_aktuell;       // 5

   dvl_ende= dvl_aktuell;                  // 6

   return 0;
}
```


Anlegen eines Datensatzes (DS) (2/2)

1. Fall: noch keine DVL vorhanden

dvl_anfang = 0
dvl_ende = 0
dvl_aktuell = 0

dvl_anfang
dvl_aktuell
dvl_ende



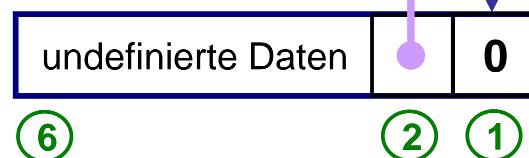
2. Fall: Ein DS der DVL vorhanden
dvl_anfang = dvl_ende
dvl_aktuell = dvl_anfang

3. Fall: mehrere DS der DVL vorhanden

wie 2. Fall!

dvl_anfang
dvl_ende

dvl_aktuell
dvl_ende



9 Zeitfunktionen

- Grundlage der Datumsformate ist die Struktur „tm" aus <time.h>

```
struct tm {
  int tm_sec;        // Sekunden
  int tm_min;        // Minuten
  int tm_hour;       // Stunde (0-23)
  int tm_mday;       // Monatstag (1-31)
  int tm_mon;        // Monat (0-11)
  int tm_year;       // Jahr (Kalenderjahr minus 1900)
  int tm_wday;       // Wochentag (0-6; Sonntag = 0)
  int tm_yday;       // Jahrestag (0-365)
  int tm_isdst;      // 0, wenn Sommerzeit
                     //    deaktiviert ist
};
```


Funktion für deutsches Datumformat

```
char * zeit(char *datum)
{
   time_t sekunden;
   struct tm *zeit;            // tm aus time.h

   const char tag[7][2+1]= {"So","Mo","Di","Mi","Do","Fr","Sa"};

   const char monat[12][3+1]=
                      {"Jan","Feb","Mae","Apr","Mai","Jun",
                       "Jul","Aug","Sep","Okt","Nov", "Dez"};
   time(&sekunden);            // Sekunden seit dem 1. Januar 1970

   zeit= localtime(&sekunden);      // Fuellen der Struktur tm

   sprintf(datum, "%s,%2d. %s %4d %02d:%02d",
            tag[zeit->tm_wday],
            zeit->tm_mday, monat[zeit->tm_mon],
            zeit->tm_year + 1900, zeit->tm_hour,
            zeit->tm_min);
   return datum;
}
```

10 Programmprojekte

- Größere Programme sollten in mehrere Module (Dateien) aufgeteilt werden aus folgenden Gründen:
  - Übersichtlichkeit der Programme
  - Kleinere Dateien sind besser editierbar
  - Möglichkeit der getrennten Compilierung
  - Zusammenfassung von Funktionen für gleichartige Aufgaben
  - Zusammenstellung von Bibliotheken
  - Wiederverwendbarkeit für andere Programme
  - Zusammenfassung der nicht-ANSI-Funktionen (Konsolen-E/A)
  - Bessere Pflegbarkeit und Wartbarkeit der Programme
  - Bessere Erweiterbarkeit
  - Schneller portierbar auf andere Plattformen (verschiedene Betriebssysteme, verschiedene Compiler)
  - Mehrere Programmierer können gleichzeitig am selben Programm (an verschiedenen Modulen) arbeiten

10.1 Programmverwaltung mit Tools

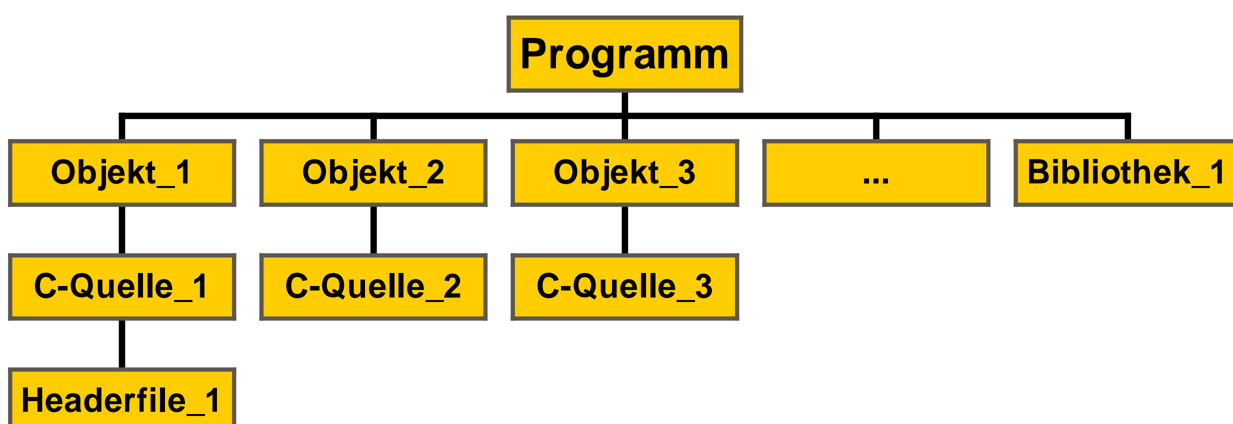
- Gründe für den Einsatz von Werkzeugen zur Programmverwaltung:
  - Bei sehr komplexen Programmen, die aus sehr vielen Teilen bestehen, muss das Bilden der Programme automatisiert werden (Übersichtlichkeit)
  - Es sollen nur die Teile neu gebildet werden, die sich zur vorhergehenden Version geändert haben
  - Es können verschiedene Varianten des Programms gebildet werden (Debug-Variante, abgespeckte Variante, …)
  - Verwaltung von verschiedenen Versionen (Bearbeitungsständen)

- Für die Programmverwaltung stehen verschiedene Werkzeuge (Tools) zur Verfügung. Eines der Entwicklungswerkzeuge ist das aus der UNIX-Welt stammende Tool „make".

10.2 Makefile, make

- „make" ist ein seit langem und nach wir vor häufig (besonders auf UNIX-Plattformen) eingesetztes Tool.
- Die Syntax eines „Makefile" ist standardisiert, d.h. auf verschiedenen Plattformen kann dasselbe Makefile verwendet werden.
- „make" ist das Tool, das die Informationen aus einer Datei „Makefile" (kann auch anders heißen) auswertet und das Programm bildet.
- Die Datei „Makefile" (Beschreibungsdatei) enthält alle Informationen (Abhängigkeiten), Teile und Kommandos, die zur Bildung des Programms benötigt werden.

Abhängigkeiten (dependencies)



11 Fehlerbehandlung

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

Fehlerbehandlung

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

Fehlerarten / Einteilung

- Während der Laufzeit eines Programmes können die verschiedensten Fehler auftreten. Diese Fehler lassen sich in die folgenden Kategorien einteilen:

1. Fehler aufgrund falscher bzw. unkorrekter Benutzer-eingaben
2. Fehler bei Verwendung von Systemrecourcen (malloc, free, fopen, fread, fwrite, fclose, ...)
3. Fehler bei internen Programmabläufen (Division, sqrt, Endlosschleifen, Fehlerhafte Speicherzugriffe, ...)
4. Fehler durch fehlerhafte Compilierung, Fehler im Betriebssystem und Hardwarefehler

Fehlernachrichten

- Die Fehlernachrichten werden oft zentral in einer dafür vorgesehenen Funktion aufgelistet und bei Bedarf ausgegeben (siehe Beispiel)
- Die Reaktion auf Fehler hängt stark vom jeweiligen Fehler ab. Auch hier gibt es verschiedene Möglichkeiten:
  - Direkte Behandlung in der Funktion, in der der Fehler auftritt (evtl. nur Nachricht ohne weitere Reaktion)
  - Weiterreichen des Fehlers mittels return-Wert (meist als negative ganze Zahl) an die aufrufende Funktion und Reaktion dort
  - Zentrale Behandlung von gleichartigen Fehlern in einer dafür vorgesehenen Funktion

11.1 Ausgabe von Fehlernachrichten

```
#define FANZ 7

void fehlerAusgabe(char *funktion, int fehlerNummer)
{
   int fehlerIndex= (-fehlerNummer) - 1;

   const char fehlerNachricht[FANZ][60+1]=
      { " -1 Doppelt verkettete Liste ist leer",
        " -2 Speicherplatz konnte nicht bereit gestellt werden",
        " -3 Datei konnte nicht fuer Lesen geoeffnet werden",
        " -4 Datensatz konnte nicht gelesen werden",
        " -5 Datei konnte nicht fuer Schreiben geoeffnet werden",
        " -6 Datensatz konnte nicht geschrieben werden",
        " -7 Datei konnte nicht geschlossen werden" };

   if (fehlerIndex >= 0 && fehlerIndex < FANZ)
   {
      printf("\nFunktion: %s\n", fehlerNachricht[fehlerIndex]);
      perror(funktion);   // Standardfunktion fuer Fehlerausgabe
   }
}
```


11.2 Beispiel für Nutzereingabe

```
   ...
   int fehler;

   do
   {
      fehler= 0;

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

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

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

   } while(fehler);
```

12 Programmierhinweise

- Softwareentwicklung ist mehr als nur die Implementation von Algorithmen (reine Umsetzung in Programmiersprache).
- Der Prozess der Softwareentwicklung umfasst Problem-analyse, Planung (Pflichtenheft), Implementation, Dokumen-tation, Tests, Einführung und Pflege/Wartung.
- Sehr oft ist die Softwareentwicklung Teamarbeit, damit gehört auch Projektmanagement zum Softwareentwicklungsprozess.
- Zum Lehrgebiet „Systementwicklung" gehört damit auch die Vermittlung von Regeln, Schreibweisen, Richtlinien usw. für die Programmierung.
- Richtlinien ermöglichen u.a. Teamarbeit und Pflegbarkeit/Er-weiterbarkeit der Software.
- Durch Einhaltung von Richtlinien wird oft die Verständlichkeit und Übersichtlichkeit der Programme erhöht.
- Bereits beim Programmieren erleichtern Richtlinien die Fehler-suche und erleichtern die Programmierung.

12.1 Kommentare

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

12.2 Hinweise zur Schreibweise (Syntax)

- In Softwarefirmen gibt es oft sehr genaue und restriktive Regeln für die Schreibweise von Programmen.
- Bereits bei ersten Programmierübungen sollten Mindestanforderungen erfüllt werden:
  - Bei eigenen Namen (Funktionsnamen, Variable, Makros, ...) sollte einheitlich entweder deutsch oder englisch gewählt werden.
  - Für die Schreibweise von Variablen gibt es allgemein bekannte Richtlinien, z.B. ungarische Notation
  - Namen aus Präcompiler-Direktiven werden meist in Großbuchstaben geschrieben (ANZ, MAX, MIN, ...)
  - Möglichst selbsterklärende/selbstsprechende Namen verwenden
  - Einrückungen einheitlich, d. h. entweder Leerzeichen oder Tabulatoren (kein mathematisches „ODER")
  - Feste Anzahl von Leerzeichen bzw. Tabulatoren in allen Ebenen
  - Einheitliche Klammerung („{" und „}")

Hinweise zur Schreibweise (Syntax)

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

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

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

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

12.3 Hinweise zum Programmierstil

- Es kommt in normalen Anwendungen nicht darauf an, unter Ausnutzung aller Möglichkeiten und Tricks (z. B. Seiten-effekte) der Programmiersprache das Programm möglichst effizient zu gestalten.
  - Programmiertricks machen den Programmcode für andere Programmierer schwer lesbar (erschwert Fehlersuche, Teamarbeit, Wartung/Pflege und Erweiterungen)
  - Übertragbarkeit in andere Programmiersprachen kann dadurch extrem erschwert werden (z. B. nach Java)
- Beispiel:

```
while (*zieltext++ = *quelltext++) ;
```

- Ist für interne Systembibliothek u. U. gerade noch tragbar wegen der Effizienz, aber für „normale" Anwendungen keineswegs zu empfehlen

Hinweise zum Programmierstil (Semantik)

- Ressourcenverschwendende Programmierweise wird nicht geduldet.
- Beispiel:

```
...
char quelltext[80+1], zieltext[80+1];
...
for (i= 0; i <= 80; i++)
    zieltext[i]= quelltext[i];
```

- Funktioniert zwar, ist aber in vielen Fällen fast unsinnig ineffizient
- Programmteile, die fast trivialerweise besser (effizienter und/oder eleganter) schreibbar sind, sollten auch besser geschrieben werden.
- Es lohnt sich oft, über eventuell bessere Realisierung und Implementation nachzudenken

Literaturangaben

[1] Kernighan/Ritchie: *„Programmieren in C“*,
Carl Hanser Verlag München Wien,
ISBN 3-446-15497-3

[2] D. Frischalowski, J. Palmer:
*„**ANSI C 2.0** Grundlagen der Programmierung“*,
Herdt-Verlag für Bildungsmedien GmbH,
CANSI2 30-0-07-03-01