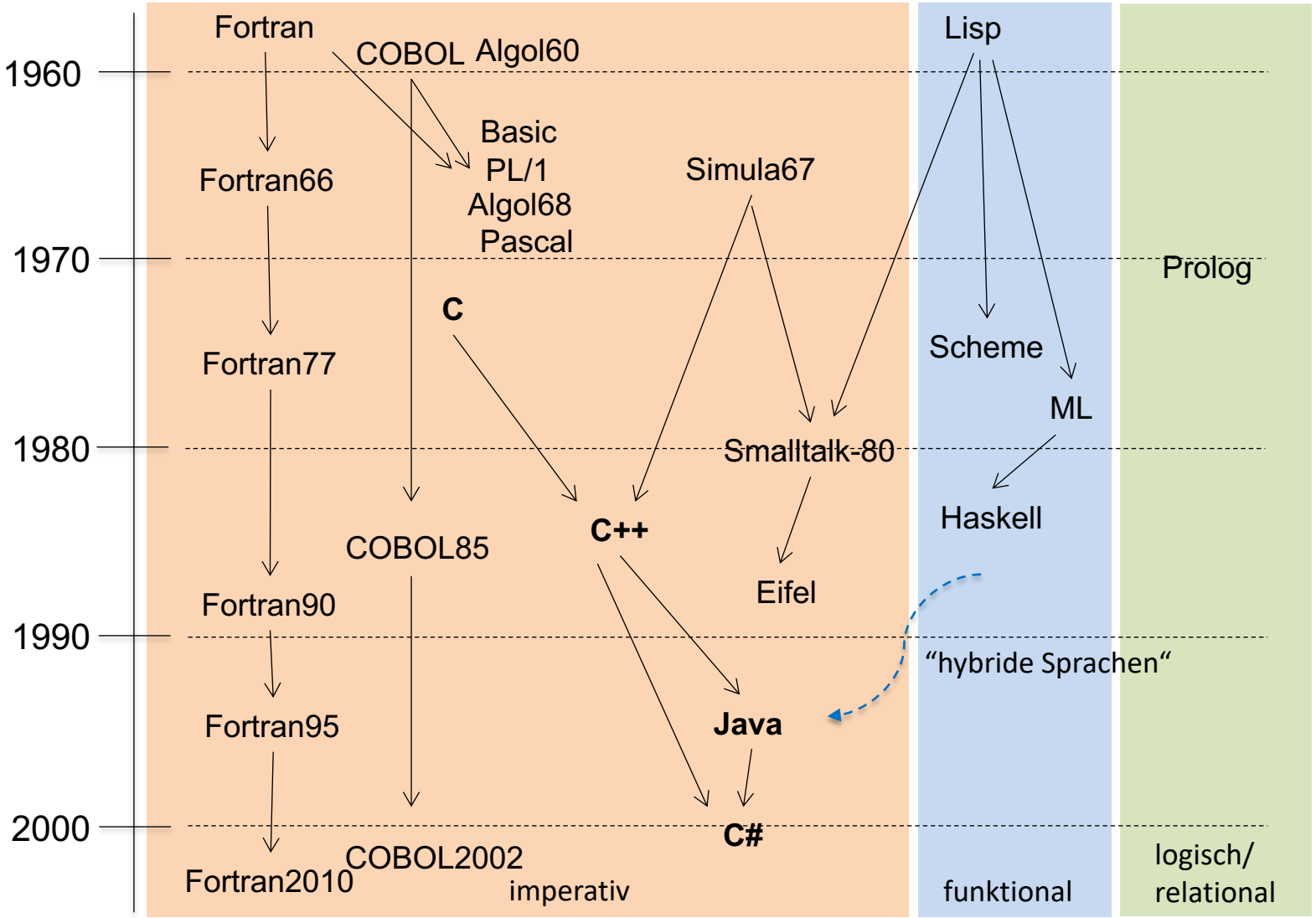
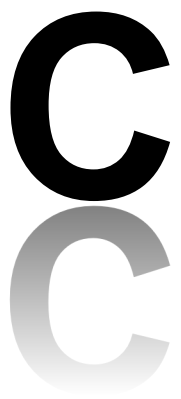
Fachhochschule
Dortmund
University of Applied Sciences

Programmierkurs Anwendungsentwicklung
Einführung C + Grundlagen

nach Unterlagen von: Prof. Dr. Dirk Wiesmann (PK2 / Wintersemester 22/23)


1960
1970
1980
1990
2000
Fortran
COBOL
Algol60
Basic
PL/1
Algol68
Pascal
Simula67
Lisp
Prolog
Fortran66
C
Scheme
Fortran77
ML
Smalltalk-80
Haskell
C++
COBOL85
Eifel
Fortran90
"hybride Sprachen"
Java
Fortran95
C#
COBOL2002
Fortran2010
imperativ
funktional
logisch/
relational

- Visual Studio steht auf den Praktikumsrechnern zur Verfügung
- Für Microsoft Visual Studio ist eine Lizenz erforderlich
- Über eine akademische Lizenz können Sie das Produkt für die Lehre kostenfrei auf dem eigenen Rechner nutzen
  - Hompage des Fachbereichs -> Studierende -> IT-Dienste
    -> Microsoft DreamSpark


C

Historie

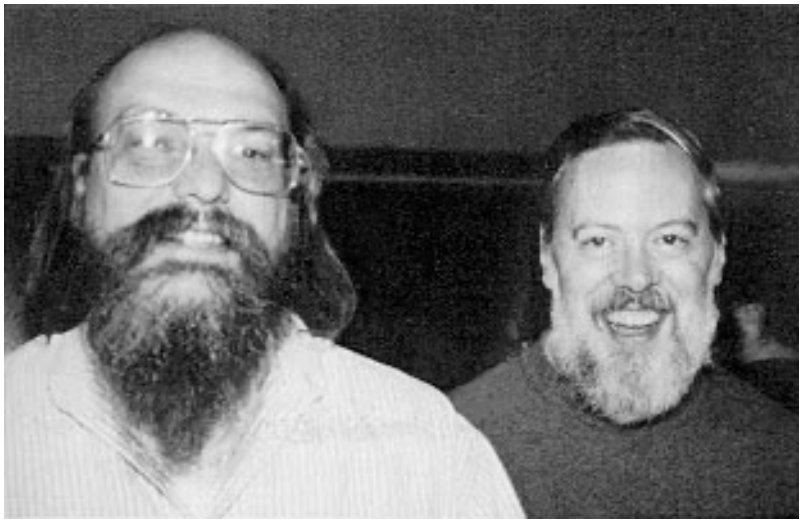
– Ab 1968 startete Ken Thompson mit der Entwicklung eines Betriebssystems für einen DEC PDP-7 Computer

8 KByte RAM
18 Bit Wortbreite

Quelle: http://www.columbia.edu/cu/computinghistory/pdp7.html

– Dieses Betriebssystem war der Vorgänger von Unix und wurde zunächst in Assembler für den PDP-7 Prozessor geschrieben

- Es wurde aber erkannt, dass eine höhere Programmiersprache für die Systemprogrammierung zwei Vorteile im Vergleich zu Assembler bietet (welche ?)
- Aus der Sprache BCPL (*Basic Combined Programming Language*) wurde von Thompson die Sprache B abgeleitet
- Von Dennis Ritchie wurde in den Jahren 1969 – 1973 aus B die Programmiersprache C abgeleitet (C erweitert B um Typen)



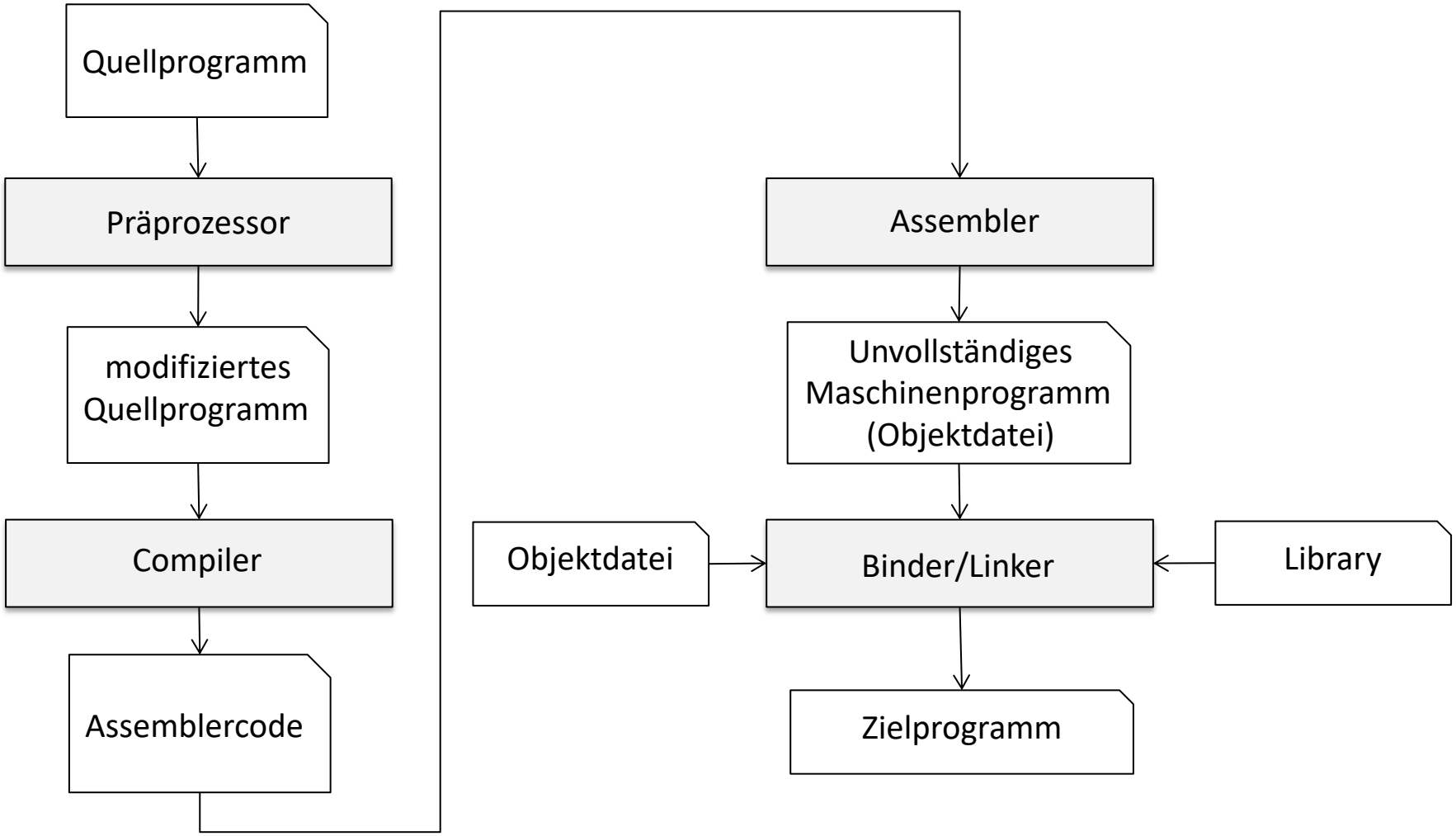
Ken Thompson (links) und Dennis Ritchie (rechts)
Quelle: de.wikipedia.org

- Ab 1973 wurde der Unix-Kernel für die PDP-11 in C implementiert
- Im weiteren Verlauf wurde C auf verschiedene Rechnerplattformen portiert
- Im weiteren Verlauf wurde die Sprache standardisiert (ANSI 89, ISO/IEC 9899-1990)
- C ist einer der am meisten eingesetzten Programmiersprachen (Unix/Linux-Umfeld, Programmierung von Microcontroller, ...)

Eigenschaften von C

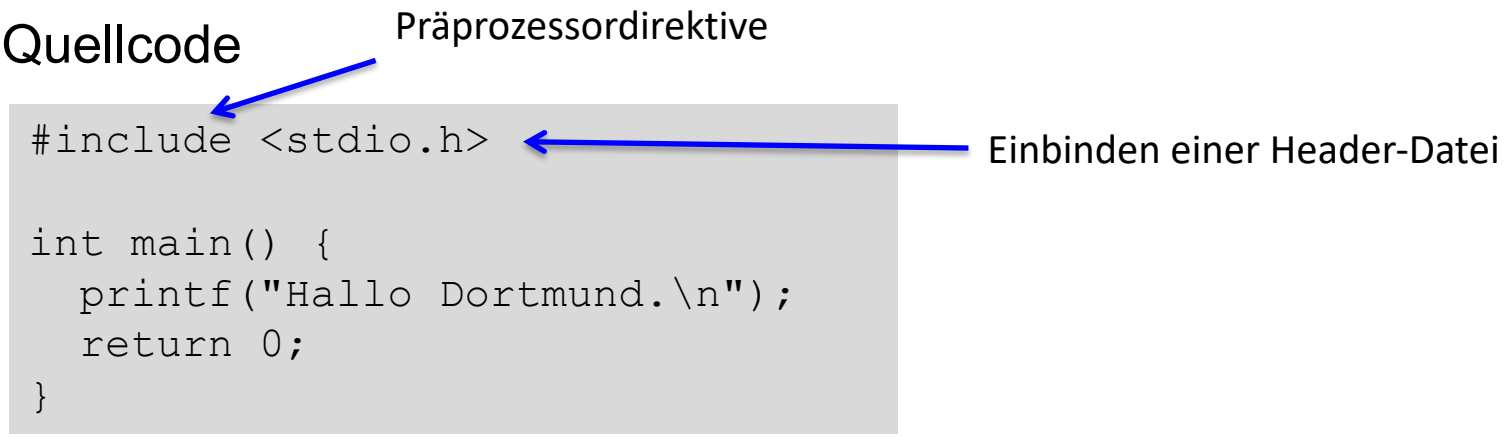
- ist eine Prozedurale Sprache und gehört damit zur Klasse der Imperativen Sprachen
- wird compiliert
- statisch typisiert
- schwach typisiert (Typkonvertierungen ohne weitere Prüfung möglich)
- kein eigener Datentyp für Wahrheitswerte und Zeichenketten
- bietet Zeiger
- manuelle Speicherverwaltung
- geringer Sprachumfang (Nutzung von Bibliotheken/Libraries)

Compilierung



Das erste C-Programm

– Quellcode



– Compilierung mit dem GNU C-Compiler auf einem Linux-System in ein ausführbares Programm

```
> gcc hallo.c -o hallo
```

– Ausführung des Programms

```
> ./hallo
Hallo Dortmund.
```

Das zweite C Programm

- Berechnung des GGT nach dem Algorithmus von Euklid
  1. Gegeben sind zwei Zahlen $a, b \in N$
  2. Falls $a = b$ ist, dann gib a aus und stoppe
  3. Ansonsten ersetze den größeren der Werte durch die Differenz der Werte und mache bei Schritt 2 weiter

- Mögliche Implementierung in C

```
int ggt(int a, int b)
{
   while (a != b){
      if (a > b) a=a-b;
         else b=b-a;
   }
   return a;
}
```

ggt.c

- Einschub: Inspektion des Assembler-Codes
  - Mit Option `-S` wird der Compiler nach der Übersetzung in Assemblercode gestoppt

```
> gcc -S ggt.c
```

  - Assemblercode ist in Datei `ggt.s` gespeichert

```
 .file  "ggt.c"
    .text
    .globl ggt
    .type  ggt, @function
ggt:
    pushl  %ebp
    movl   %esp, %ebp
    jmp    .L2
.L4:
    movl   8(%ebp), %eax
    cmpl   12(%ebp), %eax
    jle    .L3
    movl   12(%ebp), %eax
    subl   %eax, 8(%ebp)
    jmp    .L2
.L3:
    movl   8(%ebp), %eax
    subl   %eax, 12(%ebp)
.L2:
    movl   8(%ebp), %eax
    cmpl   12(%ebp), %eax
    jne    .L4
    movl   8(%ebp), %eax
    popl   %ebp
    ret
    .size  ggt, .-ggt
    .ident "GCC: (SUSE Linux) 4.3.2 [gcc-4_3-branch revision 141291]"
    .section .comment.SUSE.OPTs,"MS",@progbits,1
    .ascii   "ospwg"
    .section    .note.GNU-stack,"",@progbits
```

ggt.s

- Erzeugen einer Objektdatei `ggt.o`
  - Die Option `-c` startet den Compiler (inkl. Assembler) und unterdrückt den Binder/Linker. Ausgabe in Datei `ggt.o`

```
> gcc -c ggt.c
```

- Um die Funktion `ggt` später in anderern Quelldateien aufrufen zu können, benötigen wir eine Header-Datei

```
int ggt(int a, int b);
```

ggt.h

– Das Hauptprogramm

```
#include <stdio.h>
#include "ggt.h"

int main(){
  printf("GGT von 9 und 21 ist %d\n",ggt(9,21));
  printf("GGT von 27 und 9 ist %d\n",ggt(27,9));
  return 0;
}
```

Einbinden der Funktion ggt über die Header-Datei

calcggt.c

– Übersetzen und Binden des Hauptprogramms

```
> gcc -o calcggt calcggt.c ggt.o
```

– Ausführen des Hauptprogramms

```
> ./calcggt
GGT von 9 und 21 ist 3
GGT von 27 und 9 ist 9
```

make

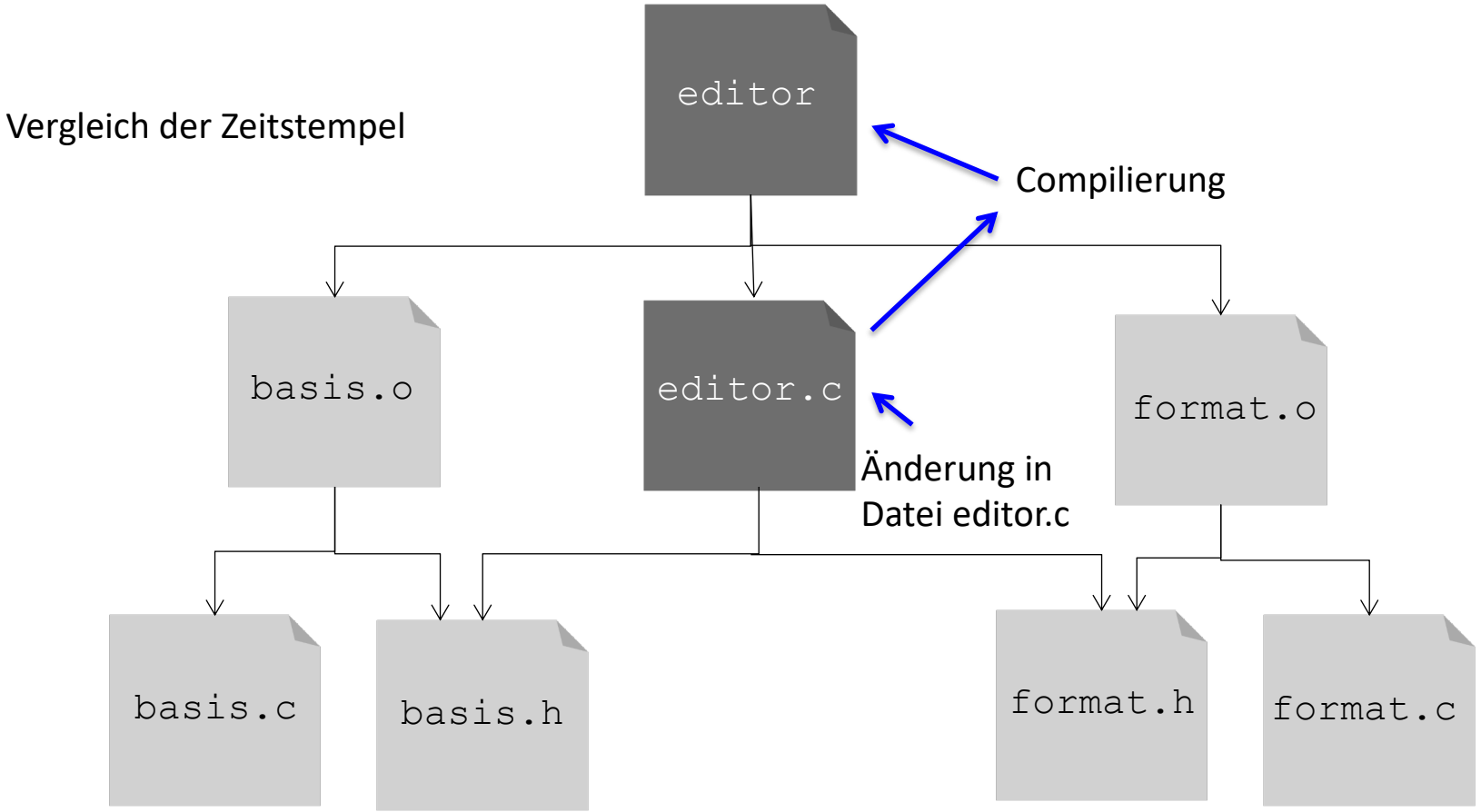
- Ein C-Programm setzt sich in der Regel aus einer Vielzahl von Quellcode- und Header-Dateien zusammen
- Diese müssen in Objektdateien compiliert werden
- Die Objektdatein werden dann (evtl. mit weiteren Bibliotheken) zu einem ausführbaren Programm gebunden

Die wiederkehrende Eingabe der Befehlsfolge zur Compilierung ist zeitaufwendig

- Im Entwicklungszyklus werden häufig einzelne Dateien geändert
- Bei Änderung einer Datei müssen aber nicht zwangsläufig alle Quellcode-Dateien neu übersetzt werden
- Es müssen nur die Dateien neu übersetzt werden, die von einer Änderung direkt oder indirekt beeinflusst werden

Eine selektive Compilierung ist fehleranfällig, wenn sie manuell durchgeführt wird

– Bei einer nachträglichen Änderung in der Datei `editor.c` müssen die Dateien `basis.o` und `format.o` nicht neu übersetzt werden



- Der Prozess der Erzeugung von ausführbaren Programmen sollte daher durch ein Werkzeug unterstützt werden
- Im Umfeld von C und C++ kommt häufig das Programm `make` zum Einsatz

- `make` steuert die folgenden Aktivitäten
  - Kopieren von Dateien
  - Bedarfsgerechte (inkrementelle) Compilierung
  - Linken

- Der Erstellungsprozess wird formal in einem *Makefile* beschrieben
- Im *Makefile* sind alle Abhängigkeiten beschrieben und alle Aktivitäten definiert

– Ein *Makefile* besteht aus Regeln der folgenden Form

```
Ziel : Voraussetzung ... ...
    Befehl
    ...
    ...
```

makefile

Jede Befehlszeile muss mit einem Tabulatorzeichen beginnen

– Mit `make` wird auf der Kommandozeilenebene die Verarbeitung der Datei `makefile` gestartet

– Beispiel: Das Programm `editor` besteht aus den drei Quellcode-Dateien `editor.c`, `basis.c` und `format.c`

```
editor: basis.o format.o editor.c basis.h format.h
    gcc editor.c -o editor basis.o format.o

basis.o: basis.c
    gcc -c basis.c

format.o: format.c
    gcc -c format.c

clean:
    rm editor basis.o format.o
```

makefile

Bezeichner und Schlüsselwörter

- Bezeichner
  - bestehen aus Buchstaben, Ziffern und dem Unterstrich _
  - das erste Zeichen muss ein Buchstabe sein
  - Groß- und Kleinschreibung wird unterschieden
  - Schlüsselwörter dürfen nicht als Bezeichner verwendet werden
  - abhängig vom Compiler kann die Anzahl der zur Unterscheidung signifikanten Zeichen eines Bezeichners beschränkt sein

- Schlüsselwörter

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

ab ANSI-C99

Variablen und Konstanten

Eine Variable ist ein benannter logischer Speicherplatz mit dessen Inhalt

- C statisch typisiert. Jede Variable besitzt einen Datentyp

Ein Datentyp ist eine Zusammenfassung von Wertebereich und Operationen auf diesem Wertebereich

- Definition einer Variablen

```
typ variablenname;
```

Es wird Speicher reserviert. Der Wert der Variablen ist aber undefiniert

- Initialisierung bei der Definition möglich

```
typ variablenname = wert;
```

C stellt nicht die Initialisierung einer Variablen vor der ersten Nutzung sicher

- Auch eine Konstante bezeichnet einen logischen Speicherplatz
- Einer Konstanten darf aber nur einmal ein Wert zugewiesen werden
- Definition einer Konstanten

```
const typ VARIABLENNAME = wert;
```

Initialisierung muss
bei der Definition erfolgen

Immer wenn möglich, sollte man Konstanten verwenden. Warum?

Numerische Datentypen

- Ganze Zahlen: **`int`**
  - Definition

```
int i;
int anzahl, index;
int hausnummer = 42;
```

  - Wertebereich ist abhängig von der Rechnerarchitektur
  - Auf einer *n*-Bit-Architektur ist der Wertebereich häufig $-2^{n-1},\ldots,2^{n-1}-1$
  - Der Wertebereich kann durch vier **Qualifier** beeinflusst werden

1) `short`
2) `long`
3) `signed`
4) `unsigned`

Datentyp	Bits
`short int`	*n*/2
`long int`	*n*

gilt häufig für *n*-Bit-Architekturen

  - `long zaehler;` ist Abkürzung für `long int zaehler;`

- Qualifizierer `signed` und `unsigned` bestimmen, ob der Datentyp neben positiven auch negative Zahlen annehmen kann
- Werden die Qualifizierer `signed` und `unsigned` nicht angegeben, wird automatisch `signed` angenommen

- Operatoren für `int`

Operator	**Rechenart**
+	Addition
-	Subtraktion
*	Multiplikation
/	Division (ganzzahlig)
%	Modulo

- Speicherbedarf eines Datentyps in Byte kann mit dem Operator **`sizeof`** ermittelt werden
  - Es gilt immer `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`
  - Zudem sind die Grenzen des Wertebereichs in der Datei `limits.h` in der Form von Konstanten festgehalten

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

int main(){
  printf("Der Typ int benoetigt %d Byte\n", sizeof(int));
  printf("Wertebereich von int: %d, ... ,%d\n",
           INT_MIN, INT_MAX);

  return 0;
}
```

- Auf einem 32-Bit-System wurden z.B. die folgenden Wertebereiche ermittelt

Datentyp	Wertebereich
`signed short int`	-32768,...,32767
`signed long int`	-2147483648,...,2147483647
`unsigned short int`	0,...,65535
`unsigned long int`	0,...,4294967295

- Fließkommazahlen: **`float, double`**
  - Definition

```
float preis;
double gewicht;
```

  - Länge der Codierung

Datentyp	Länge
`float`	32 Bits
`double`	64 Bits
`long double`	≥ 64 Bits

maschinenabhängig

  - Literale
    - müssen einen Dezimalpunkt enthalten und können mit einem Exponenten geschrieben werden

```
double gewicht = 0.031e2;
float preis = 17.14F;
double d = 10.25l;
```

auch f möglich

optional. Auch L möglich

- Operatoren für `float` und `double`

Operator	Rechenart
+	Addition
-	Subtraktion
*	Multiplikation
/	Division

- Mathematische Funktionen über Standard-Mathematik-Bibliothek. Mit `#include <math.h>` wird die Header-Datei eingebunden

Funktion	Beschreibung
`sin(x)`	Sinus von *x*
`log(x)`	Natürlicher Logarithmus von *x*
`pow(x,y)`	x^y
`sqrt(x)`	Quadratwurzel von *x*
`fabs(x)`	Absolutbetrag von *x*
`floor(x)`	Rundet *x* auf die nächst kleinere Zahl

Ausschnitt

- Wahrheitswerte: „**`int`**“
  - C besitzt ursprünglich keinen eigenen Datentyp für Wahrheitswerte
  - Datentyp `int` übernimmt diese Aufgabe

Wert	Bedeutung
0	falsch (false)
≠ 0	wahr (true)

- Zeichen: **`char`**
  - Definition

```
char c;
char zeichen = 'A';
```

  - Codierung
    - Numerischer Typ mit 8 Bits und Wertebereich -128,..,127
    - speichert ASCII-Code des Zeichens

```
char zeichen = 65;
```

- Typumwandlung
  - implizite Typumwandlung (Gefahr: Verlust von Daten)

```
double d = 12.55;
int i;

i=d;
```

  - explizite Typumwandung

```
long anzahlMonate = 12;
long urlaubstageProJahr = 30;
double urlaubProMonat =
       (double) urlaubstageProJahr / anzahlMonate;
```

Warum ist hier eine explizite Typumwandlung erforderlich?

Anweisungen und Kontrollstrukturen

- Anweisung
  - Verarbeitungsvorschrift (z.B. Kontrollstrukturen)
  - Elementare Anweisung (z.B. Funktionsaufruf)
  - Ausdruck (z.B. Zuweisung)
  - Wird mit Semikolon abgeschlossen

```
Anweisung;
```

- Kontrollstrukturen
  - Block
  - Auswahl
  - Schleife
  - Aufruf

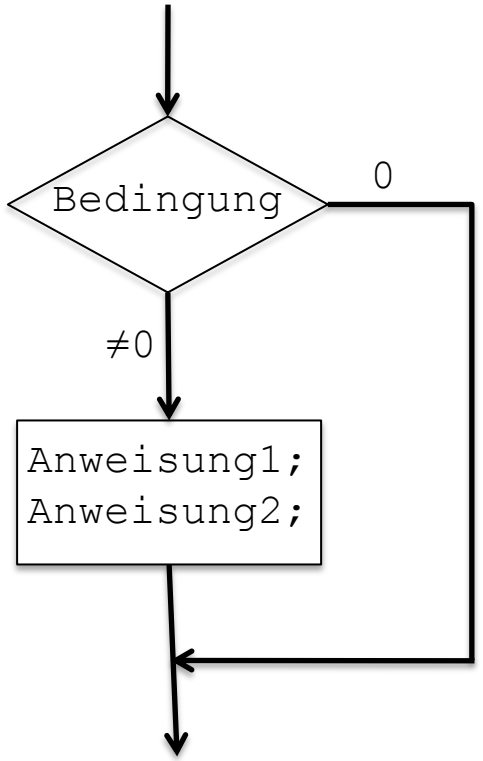
- Block
  - Fasst mehrere Anweisungen (Sequenz) zusammen
  - Kann überall da stehen, wo auch eine einzelne Anweisung erlaubt ist
  - Blöcke können geschachtelt werden

```
{
  Anweisung1;
  Anweisung2;
}
```

- Auswahl
  - Für unterschiedliche Eingaben können unterschiedliche Sequenzen durchlaufen werden
  - Einseitige Auswahl (bedingte Anweisung):

Ausdruck, der zu int ausgewertet wird

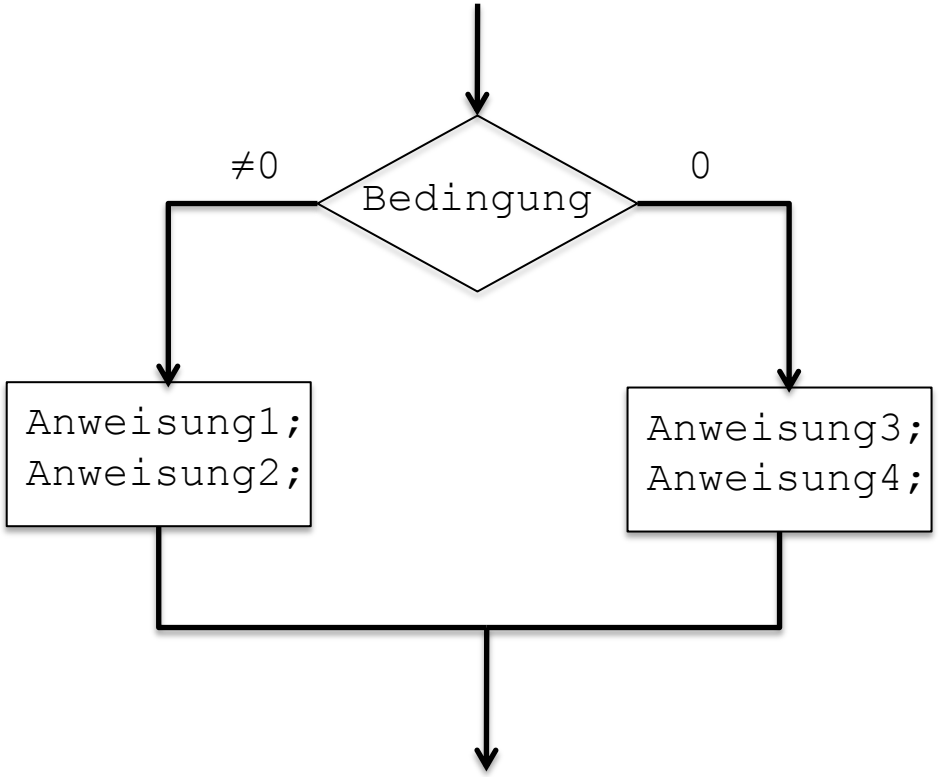
```
if (Bedingung){
  Anweisung1;
  Anweisung2;
}
```



- Zweiseitige Auswahl (Verzweigung):

Ausdruck, der zu int ausgewertet wird

```
if (Bedingung) {
  Anweisung1;
  Anweisung2;
} else {
  Anweisung3;
  Anweisung4;
}
```



- Tertiärer Operator `?` `:` (Abkürzung für `if-else` Anweisung)

```
Bedingung ? if-Ausdruck : else-Ausdruck
```

Beispiel

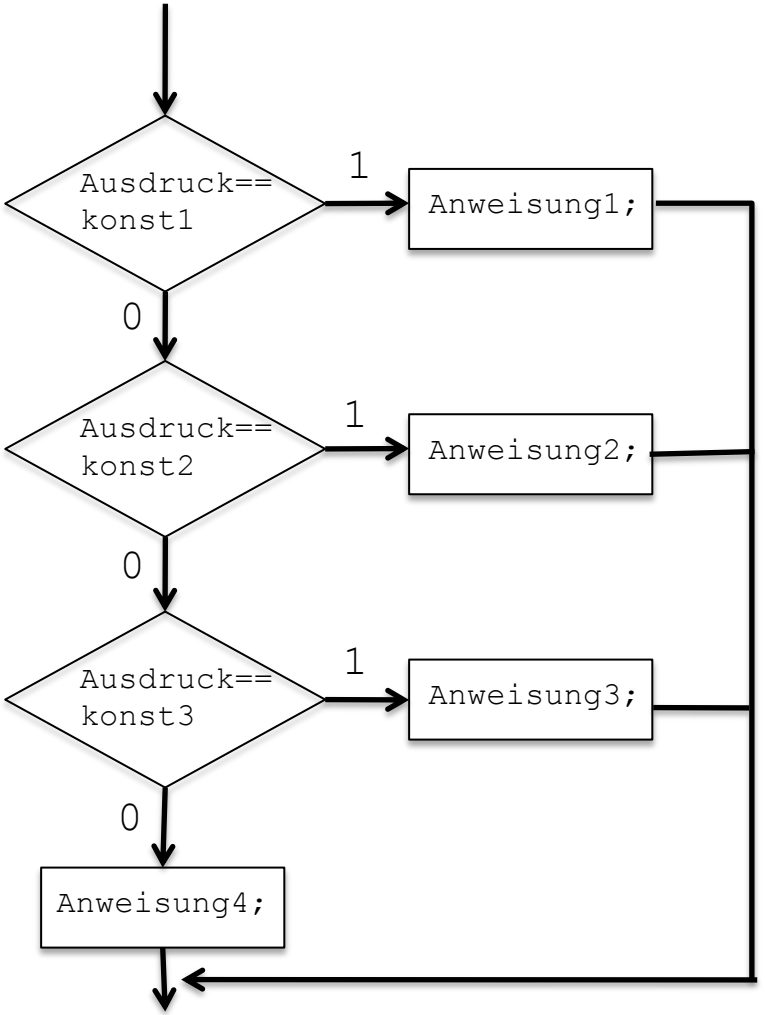
```
...
char vorzeichen;

vorzeichen = (a >= 0) ? '+' : '-';
```

– Mehrfachauswahl:

vom Typ int oder char

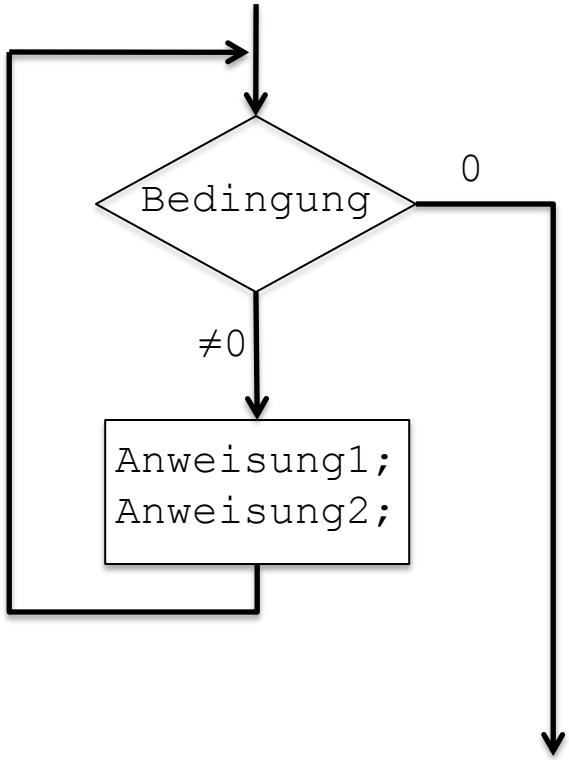
```
switch (Ausdruck){
   case konst1 :
      Anweisung1;
      break;
   case konst2 :
      Anweisung2;
      break;
   case konst3 :
      Anweisung3;
      break;
   default:
      Anweisung4;
      break;
}
```



- Schleife
  - Wiederholen von Sequenzen (abhängig von einer Bedingung)
  - **Kopfgesteuerte/vorprüfende** Schleife:

Ausdruck vom Typ int

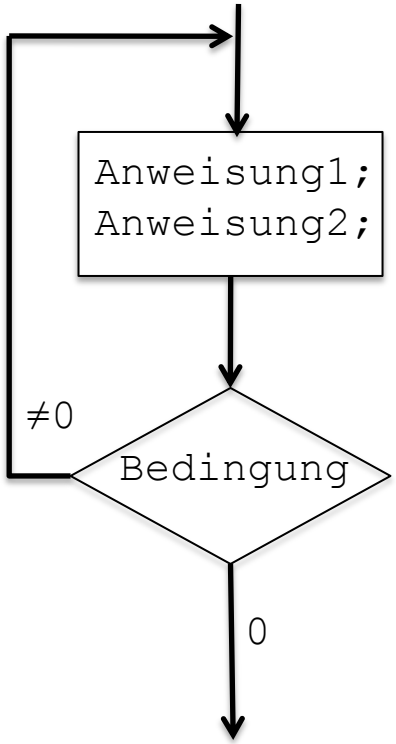
```
while (Bedingung) {
  Anweisung1;
  Anweisung2;
}
```



- Fußgesteuerte/nachprüfende Schleife:

```
do {
  Anweisung1;
  Anweisung2;
} while (Bedingung);
```

Ausdruck vom Typ int



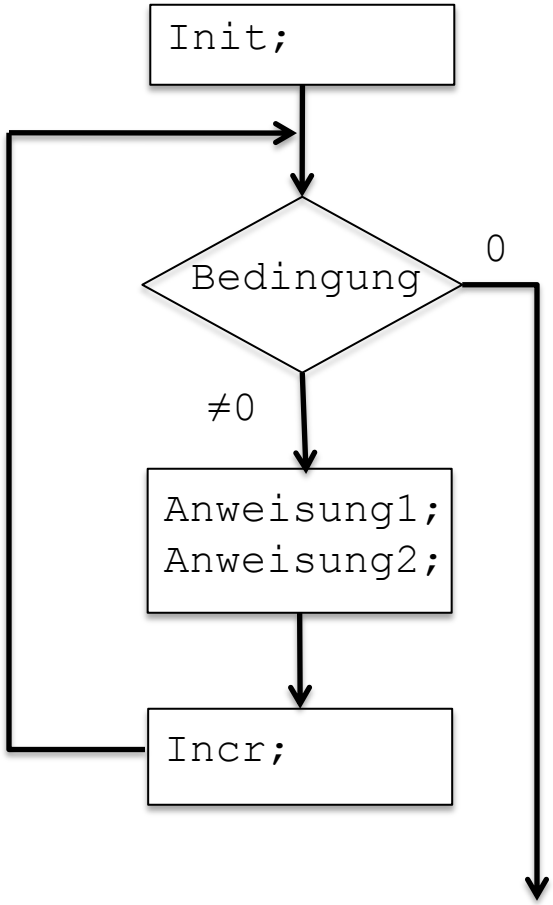
- Zählschleife:

Wird vor dem ersten Durchlauf ausgewertet

Wird vor jedem Durchlauf ausgewertet

```
for (Init;Bedingung;Incr){
  Anweisung1;
  Anweisung2;
}
```

Wird nach jedem Durchlauf ausgewertet



- `break`-Anweisung
  - mit `break;` kann eine Schleife vorzeitig verlassen werden

- `continue`-Anweisung
  - mit `continue;` kann vorzeitig der nächste Schleifendurchlauf ausgelöst werden

- `goto`-Anweisung und Marken
  - der Quellcode kann mit Sprungmarken der Form *`ziel`*`:` versehen werden
  - eine Marke hat die gleiche syntaktische Form wie ein Variablenname
  - eine Marke muss vor einer Anweisung in der gleichen Funktion wie `goto` stehen
  - mit `goto` *`ziel`*`;` wird direkt zur ersten Anweisung hinter der Marke *`ziel`* gesprungen

Auf die goto-Anweisung kann in jedem Fall verzichtet werden

Funktionen

- Über Funktionen können eigene Anweisungen definiert werden
- Definition einer Funktion

```
Typ Funktionsname (Parameterliste)
{
   Funktionsrumpf
}
```

darf leer sein

keine Überladung

- Der Funktionsname muss eindeutig sein
- Für jeden Parameter muss Typ und Variablenname angegeben werden
- Nach der Definition einer Funktion kann der Funktionsname als Anweisung oder Teilausdruck (wenn Typ nicht `void` ist) verwendet werden
- Beim Aufruf einer Funktion wird die Sequenz abgearbeitet, die im Funktionsrumpf steht

```
#include <stdio.h>

long power(long x, long y)
{   long ergebnis = x;
    long i;

    if (y==0) return 1;

    for (i=1; i < y; i++){
      ergebnis = ergebnis * x;
    }
    return ergebnis;
}

int main()
{   int i;
    for (i=0; i < 8; i++){
      printf("%ld\n",power(2,i));
    }
    return 0;
}
```

nur für y>=0

Rückgabe eines Funktionswertes vom Typ long

Aufruf der Funktion

– Eine Funktion darf sich selber aufrufen. Man erhält dadurch rekursive Funktionen

```
int ggt(int a, int b)
{
  if (a==b) return a;
  if (a>b) ggt(a-b,b);
     else ggt (a,b-a);
}
```

Aufgabe:
Diese rekursive Implementierung der Funktion ggt ist fehlerhaft.
Korrigieren Sie den Fehler!

Hinweis: Funktionsdefinitionen können in C nicht geschachtelt werden

Deklaration und Definition

- Zwei unterschiedliche Konzepte in C
  - Alle Namen müssen dem Compiler bekannt gemacht werden
  - Falls eine Name vor der Definition verwendet wird, muss er deklariert werden

- Deklaration
  - Bekanntgabe einer globalen Variablen, über Typ und Name
  - Bekanntgabe eines Funktionsnamens über die Signatur

```
Rückgabetyp Funktionsname (Parameterliste);
```

  - Keine Reservierung von Speicher

- Definition
  - Implementierung einer Funktion
  - Reservierung von Speicher für Variablen

– Beispiel

```
#include <stdio.h>


double quadrat(double);

void ausgabe(){
  printf("5.0 hoch 2 ist %.1f\n", quadrat(5.0));
}

double quadrat(double x){
  return x*x;
}

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

Deklaration
Wird häufig in Header-Datei ausgelagert

Definition

Ausgabe

- Keine Befehle für die Aus- und Eingabe im direkten Sprachumfang von C
- Entsprechende Funktionen werden über die Standardbibliothek geliefert
- Um diese Funktionen nutzen zu können, muss die Header-Datei `stdio.h` eingebunden werden
- Die Ausgabefunktion `printf` hat den folgenden Aufbau

```
printf(Zeichenkettenliteral, Parameter1, Parameter2);
```

- Aufruf kann auch nur mit einem Literal erfolgen

```
printf("Willkommen im PK2.");
```

- Das Zeichenkettenliteral kann Platzhalter enthalten (werden mit % eingeleitet)
  - für jeden Platzhalter muss nach dem Zeichenkettenliteral ein weiterer Parameter stehen
  - der Parameter muss einen Typ aufweisen, der zu dem Ausgabeformat passt
  - die Reihenfolge der Parameter muss genau zu der Reihenfolge der Platzhalter passen
  - bei der Ausgabe wird der Platzhalter durch den Wert des Parameters ersetzt

```
int anzahl = 100;
printf("Die Variable anzahl hat den Wert %d.", anzahl);
```

Datentyp	Platzhalter	Ausgabeformat
`int`	`%d` (oder `%i`)	dezimal
`long`	`%ld`	dezimal
	`%lo`	oktal
	`%lx`	hexdezimal
`float`	`%f`	Punktnotation
	`%e`	wissenschaftliche Notation
	`%g`	variable Notation
`double`	wie `float`	wie `float`
`char`	`%c`	Zeichen
`char *`	`%s`	Zeichenkette

– Zeichenkettenliteral kann Escape-Sequenzen enthalten (werden mit \ eingeleitet)

Escape-Sequenz	Ausgabe
\n	Neue Zeile
\"	"
\\	\
\t	Tabulator
\r	Cursor auf Anfang der Zeile
\b	Signalton

```
#include <stdio.h>

int main()
{  long x=10, y=5;

   printf("%ld * %ld = %ld\n", x, y, x*y);
   return 0;
}
```

- Formatierungsmöglichkeiten
  - Anzahl der Nachkommastellen bestimmen

```
double zahl = 12.3456;
printf("Auf zwei Nachkommastellen gerundet: %.2e\n", zahl);
```

  - Ausrichtung (hier: rechtsbündig mit einer Breite von vier Zeichen)

```
int i = 1;
printf("%4d",i);
```

- Abkürzung: `putchar(c);` statt `printf("%c", c);`

Eingabe

- Die Eingabefunktion `scanf` ist ebenfalls in der Standardbibliothek enthalten

```
scanf(Zeichenkettenliteral, &Variable);
```

- Die Eingabe wird von der Tastatur gelesen und mit der Return-Taste abgeschlossen
- Neben Platzhaltern kann das Zeichenkettenliteral auch zusätzlichen Text enthalten. Dieser Text wird dann auch in der Eingabe erwartet
- Es gilt
  - Der erforderliche Parameter muss die Adresse einer Variablen sein
  - Der Datentyp der Variablen muss zum Platzhalter passen

```
long zahl;
scanf("%ld", &zahl);
```

Was passiert bei der Eingabe 12hallo?

Datentyp	Platzhalter	Eingabeformat
`int`	`%d`	dezimal (Basis 10)
	`%i`	dezimal (Basis 8, 10 , 16)
`long`	`%ld`	dezimal
	`%lo`	oktal
	`%lx`	hexdezimal
	`%li`	beliebig
`float`	`%f`	Gleitpunktzahl
`double`	`%lf`	Gleitpunktzahl
`char`	`%c`	Zeichen
`char *`	`%s`	Zeichenkette

– Einlesen eines einzelnen Zeichens

```
char c;
c = getchar();
```

Zuweisung

- Mit der Zuweisung wird einer Variablen ein Wert zugewiesen
- Binärer Operator
  - linke Seite : Variable
  - rechte Seite : Ausdruck ← wird vor der Zuweisung ausgewertet

```
a = 2;
wert = 5 * (4 - a);
y = sin(x)/2;
```

← Typische Ausdrücke

- Eine Zuweisung ist selber auch ein Ausdruck und besitzt damit einen Wert (der Wert, der der Variablen zugewiesen wird)

```
int a, b, c, wert;
a = b = c = 1;
wert = a * (4 - (a = 2));
```

Welchen Wert enthält die Variable wert?

Logische Operatoren

- Logische Operatoren liefern stets den Wert 1 für „wahr“ und den Wert 0 für „falsch“

inkl. char

- Vergleichsoperatoren (für numerische Datentypen)

Operator	Erklärung
==	Gleich
<	Kleiner
>	Größer
!=	Ungleich
<=	Kleiner Gleich
>=	Größer Gleich

Übungsaufgabe

– Gegeben seien die folgenden Definitionen:

```
int x = 5, y = 4, z = 2;
```

Geben Sie zu den folgenden Ausdrücken den Wert an:

a) `((y < (z+3)) && (z < x))`

b) `!(z <= x)`

c) `(z != (y-2))`

d) `(z == 2)`

e) `!(x != 1)`

– Logische Verknüpfungsoperatoren

UND

x	y	x && y
0	0	0
0	1	0
1	0	0
1	1	1

statt 1 sind für die Argumente auch andere Werte ≠0 möglich

ODER

x	y	x || y
0	0	0
0	1	1
1	0	1
1	1	1

NEGATION

x	!x
0	1
1	0

Übungsaufgabe

– Der logische Verknüpfungsoperator **o** sei wie folgt definiert:

x	*y*	*x* **o** *y*
0	0	1
0	1	0
1	0	0
1	1	1

Geben Sie für *x* **o** *y* einen äquivalenten Ausdruck an, der nur die logischen Verknüpfungsoperatoren &&, || und ! verwendet!

- Bit-Operatoren
  - arbeiten auf der Binärcodierung von ganzen Zahlen
  - werden auf alle Bits der Zahl angewendet

Operator	Erklärung
`&`	Bitweises Und
`|`	Bitweises Oder
`^`	Bitweises Exklusiv-Oder
`~`	Bitweises Komplement
`<<`	Linksschieben
`>>`	Rechtsschieben

```
long a = 64, b = 32, c;
c = a | b;
b = a << 1;
```

Welche Werte haben c und b?

Inkrement und Dekrement

- Um eine Variable um den Wert 1 zu erhöhen, kann der folgende Ausdruck verwendet werden

```
a = a + 1;
```

- Analog kann das Postinkrement (ein unärer Operator) verwendet werden

```
a++;
```

a muss eine Variable sein

- Das Postinkrement wird nach der Auswertung des Ausdrucks ausgewertet
- Das Präinkrement wird dagegen vor der Auswertung des Ausdrucks durchgeführt

```
a = 0;
b = ++a;
```

Welchen Wert hat b?

Analoge Aussagen gelten für den Dekrementoperator a--

Operator zum Trennen von Ausdrücken

- Der binäre Komma-Operator trennt zwei Teilausdrücke und wird von links nach rechts ausgewertet
- Der Wert des Gesamtausdrucks entspricht dem Wert des rechten Teilausdrucks
- Was ist der Wert des folgenden Ausdrucks?

```
(a=1, ++a)
```

Abkürzungen bei Zuweisungen

– Einfache Zuweisungen der Form

```
zaehler1 = zaehler1 + 5;
zaehler2 = zaehler2 - 5;
zaehler3 = zaehler3 * 5;
zaehler4 = zaehler4 / 5;
```

können durch die Schreibweise

```
zaehler1 += 5;
zaehler2 -= 5;
zaehler3 *= 5;
zaehler4 /= 5;
```

abgekürzt werden

Präprozessor

- Der Präprozessor führt vor der Übersetzung des Quelltextes eine Textersetzung durch
  1. sucht im Quelltext nach Präprozessordirektiven (beginnen mit einem #)
  2. führt die Anweisungen aus
  3. entfernt die Präprozessordirektiven aus dem Quelltext
  4. entfernt alle Kommentarzeilen

- **`#include <header-datei>`**
  - Die angegebene Standard-Header-Datei wird in den Quelltext kopiert (Dateien liegen auf Unix-Systemen häufig unter `/usr/include`)

- **`#include "header-datei"`**
  - Die Header-Datei wird aus dem angegebenen Pfad geladen und in den Quelltext kopiert

- **`#define ALT neu`** Definition von (symbolischen) Konstanten
  - Jedes Vorkommen der Zeichenkette *`ALT`* im Quelltext wird durch die Zeichenkette *`neu`* ersetzt

- **`#define FEHLERMELDUNG(text) fprintf(stderr, text)`**
  - Beispiel für ein Makro
  - Die Präprozessorkonstante wird dabei um eine Liste von Parametern ergänzt

- **`#if WERT Sequenz #endif`**
  - Die Sequenz bleibt nur dann im Quelltext, wenn `WERT` ≠ 0 ist

```
#define DEBUG 1

#if DEBUG
printf("Datei %s, Zeile %d, Wert von a = %d\n",
        __FILE__, __LINE__, a);
#endif
```

für Produktion 1 durch 0 ersetzen

vom Präprozessor vordefinierte Konstanten

Übungsaufgabe

– Schreiben Sie ein Makro `min(a, b)`, welches den kleineren der beiden übergebenen Werte liefert!

Kommentare

- Kommentare werden vom Präprozessor aus dem Quelltext entfernt
- Durch `/*` werden Kommentare (über mehrere Zeilen) eingeleitet und durch `*/` abgeschlossen

```
/*
  Diese vier Kommentarzeilen
  werden vom Praeprozessor entfernt.
*/
```

- Ein Kommentar, der nur eine Zeile umfasst, wird mit `//` eingeleitet

Zusammengesetzte Datentypen

Aus elementaren Datentypen können komplexere Datentypen zusammengesetzt werden

- Feld (Array)
  - Mehrere Variablen vom gleichen Typ werden unter einem Namen zusammengefasst
  - Statt Variablen spricht man dann von Feldelementen
  - Die Adressierung der Feldelemente (Variablen) erfolgt über einen ganzzahligen Index
  - Logische Reihenfolge der Elemente entspricht der physikalischen Reihenfolge in Hauptspeicher
  - Statische Datenstruktur
  - Definition

```
Feldtyp feldname[Anzahl_Felder];
```

- Initialisierung bei der Definition möglich

```
int primzahlen[]={2,3,5,7};
```

Compiler ermittelt Anzahl

- Konstante Felder

```
const int primzahlen[4]={2,3,5,7};
```

- Mehrdimensionale Felder (hier 2 Dimensionen)

```
long tabelle[2][5] ={ {1,2,3,4,5},
                      {6,7,8,9,10}
                    };
```

- Felder als Funktionsparameter
  - werden nicht „*by value*“ sondern „*by reference*“ übergeben

Achtung:
- Feldindex wird nicht automatisch auf Gültigkeit geprüft
- Feldgröße muss separat verwaltet werden

- Zeichenketten
  - Kein eigener Datentyp, sondern Feld vom Typ `char`
  - Zeichenkette wird mit dem Null-Byte (alle 8 Bits haben den Wert 0) terminiert
  - Das Null-Byte kann im Programmtext über das Literal `'\0'` dargestellt werden

```
char text[6]="Hallo";
```

Null-Byte wird hier automatisch ergänzt

Länge 5 reicht nicht

  - Umständliche Alternative

```
char text[6]={'H', 'a', 'l', 'l', 'o', '\0'};
```

  - Außerhalb der Definition erfolgt eine Zuweisung über die Funktion `strcpy` (aus Bibliothek `libc.a` über Header-Datei `string.h`)

```
char text[6];
strcpy(text,"Hallo");
```

- Strukturen (Records)
  - Elementare Datentypen können zu benutzerdefinierten Datentypen zusammengesetzt werden
  - Schachtelung von Strukturen ist möglich
  - Strukturen bieten eine fachliche Sicht auf die Daten
  - Definition

```
struct Strukturname
{   Typ1 Attributname1;
    Typ2 Attributname2;
};
```

  - Beispiel

```
struct adresse
{   char strasse[MAX_STRASSE];
    int hausnummer;
    char plz[MAX_PLZ];
    char stadt[MAX_STADT];
};
```

Definition der Struktur, es wird noch kein Speicher reserviert

- Definition einer Variablen vom Typ der Struktur

```
struct Strukturname Variablenliste;
```

- Beispiel

```
struct adresse kundenadresse, lieferantenadresse;
```

- Bereits bei der Struktur-Definition können Variablen definiert werden

```
struct adresse
{   char strasse[MAX_STRASSE];
    int hausnummer;
    char plz[MAX_PLZ];
    char stadt[MAX_STADT];
} kundenadresse, lieferantenadresse;
```

Strukturname darf auch entfallen

- Felder von Strukturen

```
struct adresse kundenadressen[5000];
```

- Zugriff auf die Attribute einer Strukturvariablen

```
lieferantenadresse.hausnummer = 10;

kundenadresse[0].hausnummer = 7;
```

- Strukturen können geschachtelt werden

```
struct kunde
{  char name[40];
   int kundennummer;
   struct adresse hausanschrift;
};
```

- Zugriff auf Attribute einer geschachtelten Struktur

```
struct kunde vip_kunde;

strcpy(vip_kunde.hausanschrift.strasse,"Emil-Figge-
Str.");
```

- Unions (Varianten)
  - Unions sind den Strukturen ähnlich
  - Die Attribute sind aber nicht hintereinander im Speicher angeordnet, sondern übereinander
  - Damit kann immer nur ein Attribut gespeichert werden
  - Das längste Attribut bestimmt die Gesamtlänge einer Union
  - Nutzen
    ① Einsparen von Speicherplatz
    ② Typkonvertierung
  - Beispiel: In einer Personaldatenverarbeitung wird zu jedem Mitarbeiter eine Abteilungsnummer gespeichert. Ein Vorstand ist aber keiner Abteilung zugeteilt

```
union orgaeinheit
{ int  abteilung;
  char vorstand[4];
};

union orgaeinheit orga;
```

- Beispiel: Typkonvertierung

```
union konvertierung
{  double x;
   unsigned char bytes[sizeof(double)];
};

union konvertierung zahl;
```

Nach `zahl.x = 12.1245;` kann z.B. mit `zahl.bytes[0]` direkt auf das erste Byte der Zahl vom Typ `double` zugegriffen werden

- Aufzählungen
  - Mit Aufzählungen können mehrere Konstanten zu einem neuen Typ kombiniert werden
  - Definition

```
enum Name
{  CONST1,
   CONST2,
   CONST3
};
```

Automatische Zuweisung der Werte
0
1
2

  - Beispiel

```
enum boolean
{   FALSE,
    TRUE
};

enum boolean gefunden;

gefunden = FALSE;
```

- Die Nummerierung der Konstanten muss nicht bei 0 beginnen

```
enum noten
{   sehr_gut = 1,
    gut,
    befriedigend,
    ausreichend,
    mangelhaft,
    ungenuegend
};
```

Namen für Datentypen

– Für bestehende Datentypen kann mit `typedef` ein neuer Name vergeben werden

```
typedef unsigned long NATUERLICHE_ZAHL;
```

– Der neue Name kann dann als Typ verwendet werden

```
NATUERLICHE_ZAHL n = 99;
```

– Hilfreich bei Strukturdefinitionen

```
typedef struct adresse_s
{  char strasse[MAX_STRASSE];
   int hausnummer;
   char plz[MAX_PLZ];
   char stadt[MAX_STADT];
} adresse_t;

adresse_t adressen[100];
```

Speicherverwaltung

- C teilt den zur Verfügung stehenden Hauptspeicher in drei Speicherbereiche ein
  - Statischer Speicherbereich
  - Stack-Bereich
  - Heap-Bereich

- Statischer Speicherbereich
  - Globale Variablen und Konstanten werden dort gespeichert
  - Eine globale Variable wird außerhalb jeder Funktion definiert
  - Globale Variablen haben damit über die gesamte Programmlaufzeit eine feste Adresse

- Stack-Bereich
  - Lokale Variablen werden im Stack-Bereich gespeichert
  - Jeder Funktionsaufruf erzeugt einen logischen Rahmen (*frame*), der oben auf den Stack gelegt wird
  - Der Rahmen enthält neben den lokalen Variablen auch die Funktionsparameter und den Rückgabewert
  - Wenn eine Funktion beendet wird, dann wird der Rahmen vom Stack gelöscht

- Heap-Bereich
  - Der Code für die Verwaltung des statischen Speicherbereichs und des Stacks wird automatisch vom Compiler erzeugt
  - Der Heap-Bereich wird vom Programmierer verwaltet
  - Zur Laufzeit kann Speicher im Heap-Bereich dynamisch belegt und auch wieder freigegeben werden

- Sichtbarkeit
  - Globale Variablen sind im gesamten Programm sichtbar
  - Lokale Variablen können nur innerhalb der Funktion genutzt werden, in der sie definiert wurden
  - Durch Blöcke wird die Sichtbarkeit von Variablen eingeschränkt

```
int main()
{  int x=1;
   {  int x=2;
      printf("%d\n",x);
   }
   printf("%d\n",x);
   return 0;
}
```

Wie lautet die Ausgabe des Programms?

- Speicherklassen
  - Es werden drei Speicherklassen für lokale Variablen, und zwei Speicherklassen für globale Variablen unterschieden
    1. `auto`
    2. `register`
    3. `static` (lokal)
    4. `extern`
    5. `static (global)`

  - `auto`
    - Variable ist nur in dem Block sichtbar, in dem die Variable definiert wurde
    - Wird keine Speicherklasse angegeben, wird `auto` angenommen

```
auto float gehalt;
```

Abkürzung:

```
float gehalt;
```

- `register`
  - Empfehlung an den Compiler, die Variable in einem Register zu speichern

Wann verwendet man register?

Warum kann register nur eine Empfehlung sein?

- `static` (local)
  - Lokale Variablen der Speicherklasse `static` existieren über die gesamte Laufzeit

- `extern`
  - Darf nur bei der Deklaration einer globalen Variablen angegeben werden
  - Es wird kein Speicherplatz für die Variable reserviert
  - Hinweis darauf, dass die Variable in einer anderen Programmdatei definiert wird

- `static` (global)
  - Eine globale Variable der Speicherklasse `static` ist nur in der Programmdatei sichtbar, in der sie auch definiert wurde

Zeiger

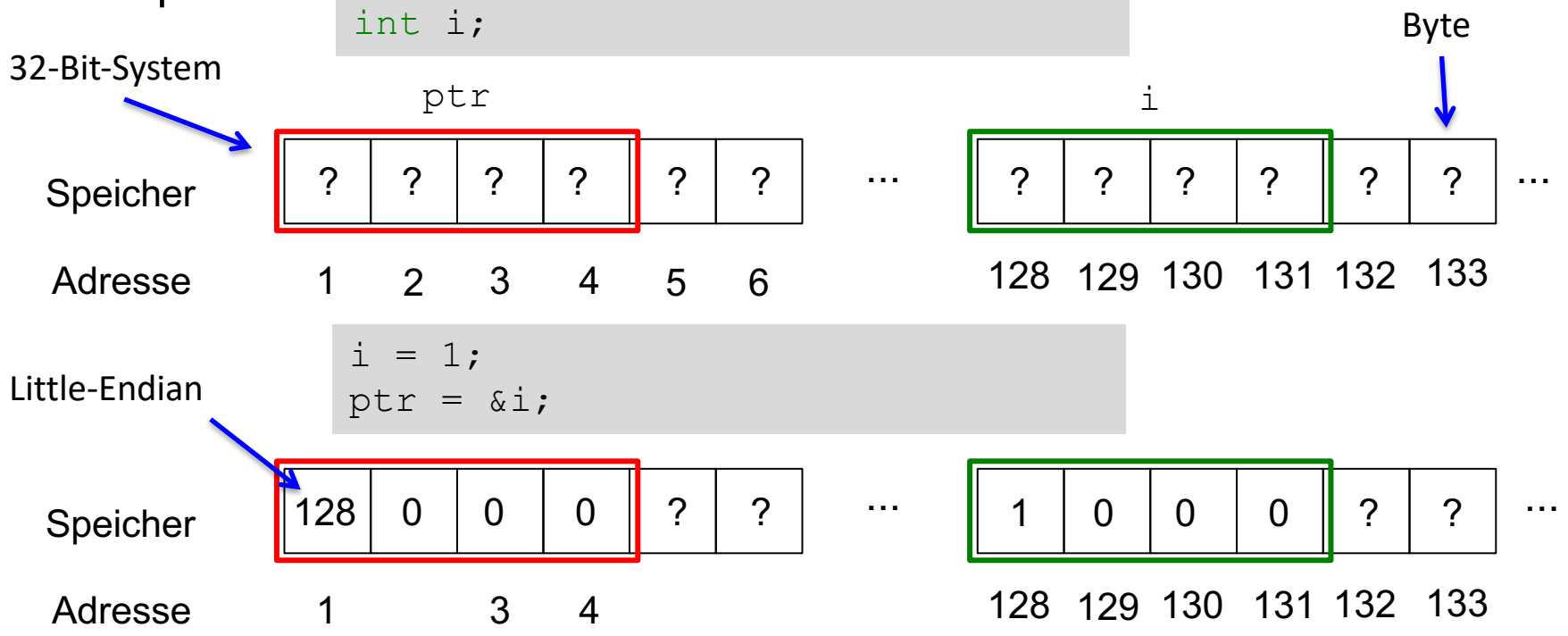
Ein Zeiger (Pointer) ist eine Variable, deren Wert eine Adresse ist

– Definition

```
Typ *Zeiger;
```

– Beispiel:

```
int *ptr;
int i;
```



- Die Speicheradresse einer Variablen kann mit dem unären Adressoperator `&` ermittelt werden

```
int *ptr;
int i;
ptr = &i;
```

- Über einen Zeiger kann der Wert einer Variablen ausgelesen werden

```
printf("i = %d\n", *ptr);
```

Dereferenzierung

- Über einen Zeiger kann der Wert einer Variablen geändert werden

```
*ptr = 2;
printf("i = %d\n", i);
```

- Es können mehrere Zeiger auf die gleiche Adresse (Variable) verweisen

```
int *zeiger;
zeiger = ptr;
```

- Zeiger als Parameter
  - Wenn Parameter als Zeiger definiert werden, dann kann direkt auf den übergebenen Argumenten gearbeitet werden (*call by reference* statt *call by value*)

    Schreiben Sie eine Funktion swap, die den Inhalt zweier Variablen vom Typ int vertauscht, die als Argumente übergeben werden

- Zeiger und Felder
  - Eine Feldvariable steht für die Speicheradresse, ab der das Feld beginnt

```
int feld[100];
int *ptr;
```

  - Eine Feldvariable kann damit als konstanter Zeiger verwendet werden. Die beiden folgenden Zuweisungen sind dann äquivalent

```
ptr = feld;
```

```
ptr = &feld[0];
```

Was ist der Unterschied zwischen sizeof(feld) und sizeof(ptr)?

- Zeigerarithmetik
  - Mit Zeigern kann gerechnet werden
  - Bei der Zeigerarithmetik wird die Länge der Variablen berücksichtigt, auf die der Zeiger verweist
  - Beispiel

```
int feld[100];
int *ptr;

ptr = &feld[0];
```

Annahme: `ptr` hat den Wert 62000

Nach

```
ptr = ptr + 1;
```

hat `ptr` den Wert 62000 + `sizeof(int)`

- Äquivalente Ausdrücke (bei `ptr = &feld[0];`)

  `&feld[i]` und `ptr+i`

  `feld[i]` und `*(ptr + i)`

– Konstante Zeiger

- Ein Zeiger kann vor Veränderung geschützt werden

```
long * const ptr = &i;
```

- Der Speicherbereich, auf den der Zeiger verweist, kann als konstant definiert werden

```
const long *ptr;
```

- Kombination beider Möglichkeiten

```
const long * const ptr = &i;
```

- Zeiger auf Strukturen
  - Zeiger können auch auf Strukturen verweisen

```
struct kunde vip;
struct kunde *ptr = &vip;
```

  - Damit kann man z.B. das Kopieren größerer Strukturen vermeiden
  - Zugriff auf ein Attribut der Struktur

```
(*ptr).kundennummer = 4711;
```

  - Kurzschreibweise

```
ptr->kundennummer = 4711;
```

Übungsaufgabe:

– Gegeben sei folgendes Programm:

```
#include <stdio.h>
main()
{ int x, y;
  int z[] = {4,2,1};
  int *ptra, *ptrb;
  ptra = &z[2];
  x = *ptra - 2;
  y = *(ptra - 1) - 2;
  ptrb = ptra - 1;
  *ptrb = 0;
  *(z+2) = 2;
  z[0] = 2;
  printf("%d, %d, %d, %d, %d",z[0],z[1],z[2],x,y);
}
```

Welche Zahlenfolge gibt `printf` aus?

- Zeiger auf Funktionen
  - Ein Zeiger kann auf eine Funktion verweisen
  - Über einen Funktionszeiger kann die Funktion aufgerufen werden
  - Der Code einer Funktion lässt sich nicht über einen Funktionszeiger verändern
  - Deklaration eines Funktionszeigers

```
void (*message)(char[]);
```

  - Definition einer Funktion und Zuweisung an Funktionszeiger

```
void error(char text[]){
  printf("Fehler: %s\n", text);
}

main(){
  message = &error;
  message("Fehlermeldung");
}
```

Parameterliste muss zum Funktionszeiger passen

auch error statt &error möglich

auch (*message) statt message möglich

Dynamische Speicherverwaltung

- Zur Laufzeit kann im Heap-Bereich Speicher angefordert und wieder freigegeben werden
- Die Funktionen zur Speicherverwaltung werden über `#include <stdlib.h>` eingebunden
- Anfordern eines Speicherblocks mit *size* Bytes über die Funktion

```
void *malloc(size_t size);
```

  - Funktion liefert einen Zeiger auf das erste Byte des reservierten Bereichs zurück
  - Falls kein Speicher reserviert werden konnte, liefert die Funktion den Wert `NULL`
  - Um mit Hilfe der Zeigerarithmetik über den Speicherbereich navigieren zu können, sollte der Zeiger vom Typ `void` in einen passenden Datentyp konvertiert werden

– Beispiel

```
double *zeiger;
zeiger = (double *) malloc(10 * sizeof(double));
```

– Freigabe von reserviertem Speicher, der nicht mehr benötigt wird

```
free(zeiger);
```

Achtung: Kein automatischer Garbage Collector

  - `free` darf nur auf Zeiger angewendet werden, die auf einen zuvor reservierten Speicherplatz verweisen
  - Zwingend erforderlich, da sonst ein „Speicherleck“ auftritt

– Initialisierung eines Speicherblocks (mit 0)

```
int *speicher = (int *) calloc(10, sizeof(int));
```

Übungsaufgabe:

– Beschreiben Sie den Datentyp der Variablen `x1,...,x6` umgangssprachlich

a) `int *const x1;`

b) `const int *x2;`

c) `char *x3[3];`

d) `char (*x4)[3];`

e) `double* x5, x6;`

Übungsaufgabe

– Eine Zeichenkette soll dupliziert werden. Führt die folgende Sequenz zum gewünschten Ergebnis? Begründen Sie die Antwort!

```
char* original = "C macht Spass.";
char* duplikat;

duplikat = original;
```

Funktion `main` mit Parametern

- Wenn die Signatur der `main`-Funktion erweitert wird, können beim Programmstart über die Kommandozeile Argumente an das Programm übergeben werden

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

- Der Parameter `argc` (*argument counter*) gibt die Länge des Feldes `argv` an
- Der Parameter `argv` (*argument value*) ist ein Feld von Zeigern auf Zeichenketten
- Die erste Zeichenkette in `argv` ist immer der Programmname

– Das folgende Programm gibt den Programmnamen und alle übergebenen Parameter aus

```
#include <stdio.h>

main(int argc, char *argv[])
{ int i;

  for(i=0; i < argc; i++){
    printf("%s\n", argv[i]);
  }
}
```