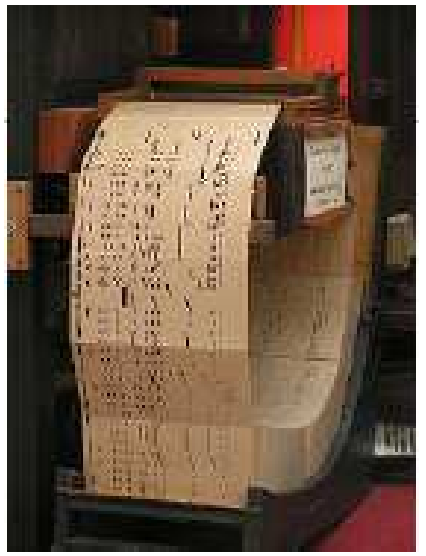
Die Programmiersprache C

Programmiersprachen

- Programm
  Folge von Anweisungen, die auf einem Computer ausgeführt werden können
  - realisieren Algorithmen
  - muss vom Prozessor verarbeitet werden
  - Binärfolgen
    z.B. Folgen von 32-Bit-Sequenzen
    `1000 1010 0001 0100 1000 1010 1100 1101`
  - für Menschen kaum lesbar
    Lösung: Programmiersprachen



Abstraktionsebenen von Programmiersprachen

Maschinensprachen
Binärcodes
abgestimmt auf die
Architektur eines
Prozessors (Register)

Assemblersprachen
an Prozessorbefehle
angelehnt;
für Menschen lesbar

Hochsprachen
Befehlssatz an
menschliche Denkweise
angepasst

Abstraktionsgrad nimmt zu

Übersetzung durch Compiler oder Interpreter

Die Programmiersprache C

- höhere Programmiersprache (mit einigen Assembler-ähnlichen Konstrukten)
  - imperative Sprache: definiert Rechenwege (*Wie* wird gerechnet?)
  - unterstützt den prozeduralen Programmierstil
  - klare, relativ einfache Syntax (wenige, assoziative Schlüsselwörter)
  - Compilersprache (Übersetzung **vor** der Ausführung)
- 1970/71 aus dem Vorläufer B entwickelt (Dennis Ritchie)
  zur Programmierung des Betriebssystems UNIX
- UNIX ist in C geschrieben (Kern und die meisten Systemkommandos)
- universell, weit verbreitet
- viele moderne Sprachen eng an C angelehnt (z.B. C++, Java, C#)

C-Compiler

- Linux/Unix `gcc`, `cc`
  iOS `gcc`, `clang`
  Windows `VC++`, `Cygwin`

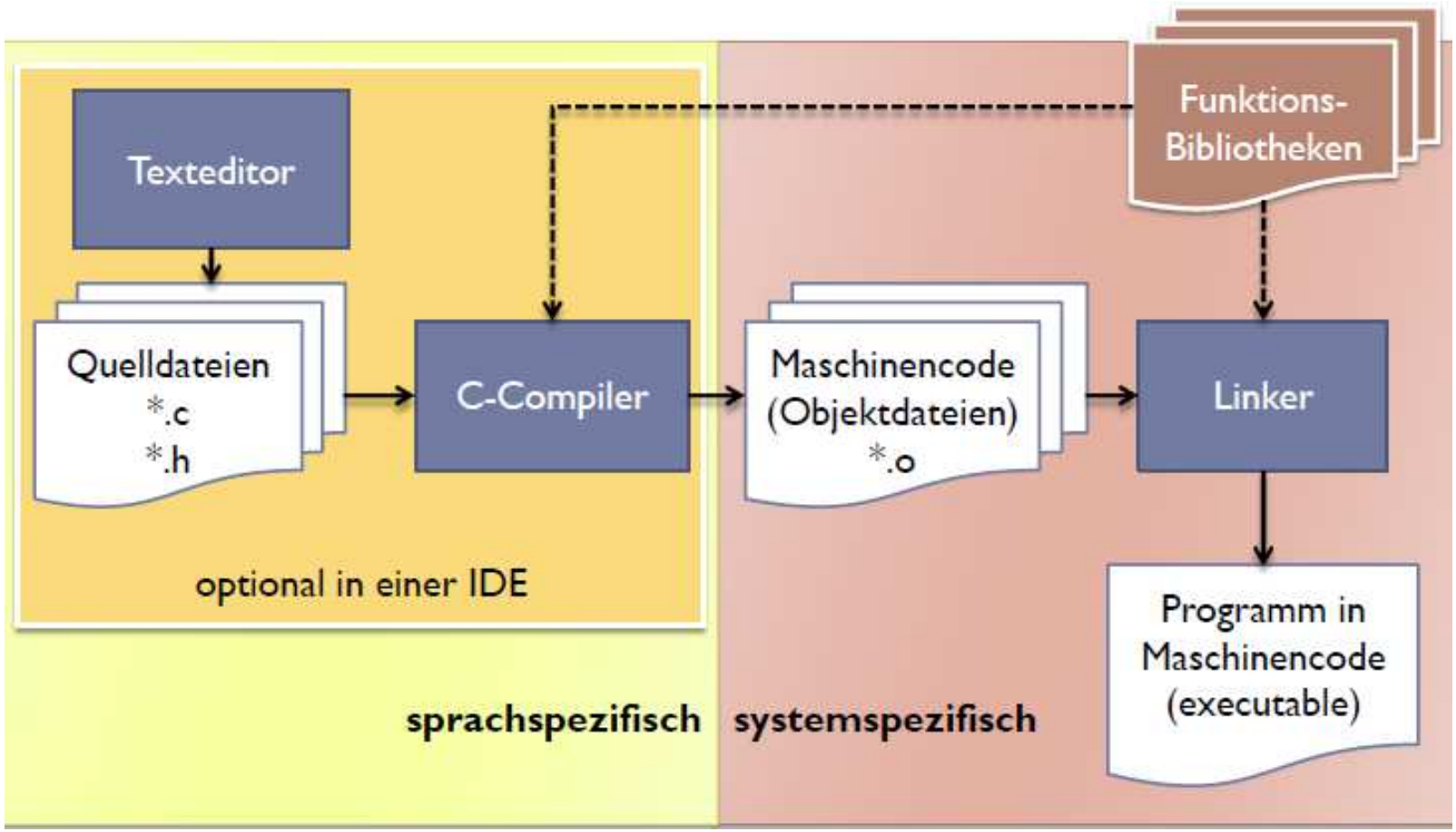
- übersetzen C-Quellcode in Maschinencode
  Quellcode (C-Code) ist portabel (unabhängig vom OS)

- Quellcode in (oft mehreren) (Text-)Dateien
  typischerweise mit Endungen `.c` *und* `.h`

$\rightsquigarrow$ Programm-Entwicklung mit Texteditor + Compiler

$\rightsquigarrow$ Integrierte Entwicklungsumgebungen (IDE)
(z.B. Eclipse CDT, Visual Studio, ...)

Entwicklung mit C



Aufbau eines C-Programms

Programm (Konzept)

- ein Text (Code), der einen *Algorithmus* formuliert, so dass er auf einer Rechenanlage ausgeführt werden kann

- Ein Algorithmus ist eine Folge von Anweisungen, die Eingabedaten in Ausgabedaten überführt (intuitiver Algorithmenbegriff).

  Dabei muss bei jeder Eingabe eindeutig sein:

  – Welche Anweisung wird zuerst ausgeführt?
  – Welche Anweisung folgt auf eine gerade ausgeführte Anweisung?
  – In welchen Situationen ist der Algorithmus beendet?

Umsetzung des Programm-Konzepts in C

1. Ein C-Programm berechnet eine **Funktion**.

algorithmisch:	Eingabedaten $\longrightarrow$ Ausgabedaten
mathematisch:	Argumente $\longrightarrow$ Funktionswert
in C:	(aktuelle) Parameter $\longrightarrow$ Rückgabewert

2. Berechnung von Funktionen durch Abarbeitung einer Folge von **Anweisungen**.

   $\rightsquigarrow$ C ist eine *imperativ-prozedurale Programmiersprche*

Funktionen und C-Programme

- Funktionen können weitere Funktionen aufrufen, z.B.:

  $f(x) = \sin(\ln x)$ vordefinierte Funktionen in richtiger Folge aufrufen

  $g(x) = \sqrt{f(x)}$ selbstdefinierte und Standardfunktion aufrufen

- Aufruf g mit Argument x $\rightarrow$ g: Aufruf f mit Argument x
  $\rightsquigarrow$ f: Rückgabewert $f(x)$
  $\rightarrow$ g: Aufruf $\sqrt{\ }$ mit $f(x)$
  $\rightsquigarrow$ $\sqrt{\ }$: Rückgabewert $\sqrt{f(x)}$
  $\rightsquigarrow$ g: Rückgabewert $g(x) = \sqrt{f(x)}$

- Den Rückgabewert einer aufgerufenen Funktion erhält die aufrufende Funktion.

Struktur von C-Programmen

- C-Programm: Definition einer oder mehrerer Funktionen
  - vom Programm realisierte Funktion: `main()`
    $\rightsquigarrow$ wird stets zuerst aufgerufen
  - ggf. weitere, aufzurufende Funktionen

- Häufig zu benutzende Funktionen (**Standardfunktionen**) sind in *Bibliotheksdateien* vordefiniert.
  $\rightsquigarrow$ können eingebunden und dann aufgerufen werden

- Besonderheit: Den Rückgabewert von `main()` erhält das Programm, das das C-Programm aufruft
  $\rightsquigarrow$ kann als Exit-Status interpretiert werden
  $\rightsquigarrow$ `main()` gibt ganzzahligen Wert zurück

Ein erstes Programm

```
/* hello.c
 *
 * Ausgabe einer Zeichenkette auf stdout
 */

#include <stdio.h>          // Bibliotheksdatei einbinden


int main() {
  printf("Hello world!\n");
  return 0;                 // Rueckgabewert 0 (alles o.k.)
}
```

Erläuterungen zum ersten Programm

- Zeichen hinter `//` und zwischen `/*` und `*/` sind *Kommentar*

- `int main()`:
  - `()` zeigen (stets) an, dass es sich um eine Funktion handelt
  - `int` zeigt an, dass der Rückgabewert ganzzahlig ist

- `printf()`:
  - Aufruf einer Funktion zur formatierten Ausgabe auf `stdout`
  - ist Standardfunktion, die in der Bibliotheksdatei `stdio.h` deklariert ist
  - Parameter von `printf()` zwischen `()`:
    Anführungszeichen $\rightsquigarrow$ Zeichenkette; `\n` $\rightsquigarrow$ Zeilenvorschub (**n**ewline)

- Kommandos und Funktionsaufrufe müssen mit `;` abgeschlossen werden

Präprozessor-Anweisungen

- beginnen mit #
- enden *nicht* mit Semikolon
- Beispiel: `#include` *datei*
  - bindet *datei* für die Arbeit des Compilers in den Quellcode ein
  - Funktionen, die in *datei* deklariert sind, werden verfügbar
  - *datei* in Anführungszeichen: *datei* aus aktuellem Verzeichnis
  - *datei* in spitzen Klammern: *datei* aus Verzeichnis mit C-Bibliotheken (z.B. `/usr/include`)

Vom Quellcode zum ausfürbaren Code

- Aufruf des Compilers:

```
gcc [-Wall] beispiel.c [-o beispiel]
```

  1. **Präprozessor** bereitet den Quellcode zur Übersetzung vor
     - kopiert Bibliotheksdateien (für den Übersetzungslauf) in den Quellcode,
     - erstzt „Aliasnamen“ im Quellcode u.ä.
  2. **Compiler** übersetzt in *Objektcode*: Befehlsfolgen für den Prozessor
  3. **Linker** verbindet mehrere Objektcode-Dateien zu einer Datei

- Option `-Wall`: alle Warnungen ausgeben (*empfohlen!*)

- Option `-o`: Name der Ausgabedatei festlegen (default: `a.out`)

Variablen und Datentypen in C

Variablen

- dienen zum Speichern von Werten (Parameter, (Zwischen-)Ergebnisse etc.)
- Werte werden im Arbeitsspeicher abgelegt
  (an eindeutiger, zum Variablennamen gehörender Speicherstelle)
- Werte werden über den Variablennamen aufgefunden
- Werte können verändert werden
- haben einen eindeutigen, unveränderlichen Datentyp
  $\rightsquigarrow$ C ist *typisiert*

Variablennamen in C

- Zeichenketten aus ASCII-Buchstaben, Ziffern und "underline" _, die mit einem Buchstaben oder _ beginnen
- Groß- und Kleinschreibung wird unterschieden!!!
- maximale Länge (systemabhängig) zwischen 63 und 255 Zeichen

Lebenszyklus von Variablen

Variablen

- müssen **definiert** werden, z.B. `int x;` *oder* `float f1, f2;`
  - *Datentyp Variablenname;*
  - Reservierung genügend vieler Speicherzellen im Arbeitsspeicher (abhängig vom *Datentyp*)
- müssen vor dem ersten Lesezugriff durch eine erste Wertzuweisung **initialisiert** werden, z.B. `x = 3;`
  gleichzeitige Definition und Initialisierung: `int x = 3;`
- **Anweisungen** ändern Werte der Variablen,
  z.B. Überschreiben durch Wertzuweisung, z.B. `x = y - x;`

$\rightsquigarrow$ = ist Zuweisungsoperator, nicht symmetrisch:
`3 = x` ist *keine* gültige Anweisung

Datentypen

- Datentyp einer Variablen bestimmt
  - Darstellung (Repräsentation) der Werte im Arbeitsspeicher
    - Anzahl der Speicherzellen (Bytes) $\rightsquigarrow$ Wertebereich/Genauigkeit
    - Bedeutung der einzelnen Bits
  - erlaubte Operationen und deren Wirkung
- **einfache/elementare Datentypen**:
  - Ganzzahltypen `char`, `int`, `short`, `long`, `long long` und deren `unsigned` Typen (z.B. `unsigned int`)
  - Gleitpunktypen `float`, `double`, `long double`
- **abgeleitete Datentypen**: setzen sich aus anderen Datentypen zusammen
- sind als *Standardtypen* vordefiniert (z.B. alle elementaren Typen) oder *selbst definierte Typen*

Elementare Ganzzahltypen

Datentyp	**Bytes** (z.B.)	**Wertebereich (dezimal)**
`[signed] char`	1	-128 ... +127
`unsigned char`	1	0 ... 255 (erweiterter ASCII-Satz)
`[signed] short [int]`	2	-32.768 ... +32.767
`unsigned short [int]`	2	0 ... +65.535
`[signed] int`	4	-2.147.483.648 ... +2.147.483.647
`unsigned int`	4	0 ... +4.294.967.295
`[signed] long [int]`	4	-2.147.483.648 ... +2.147.483.647
`unsigned long [int]`	4	0 ... +4.294.967.295
`[signed] long long`	8	-2^{63} ... $+2^{63}-1$
`unsigned long long`	8	0 ... $+2^{64}-1$

Größe vom Compiler abhängig, aber stets:

$$|\texttt{char}| < 2 \leq |\texttt{short}| \leq |\texttt{int}| \leq 4 \leq |\texttt{long}| \leq |\texttt{long long}|$$

Elementare Gleitpunktypen

Datentyp	**Bytes** (z.B.)	**Wertebereich (dezimal)**
`float`	4	$-3,4 \cdot 10^{38}$ ... $+3.4 \cdot 10^{38}$
`double`	8	$-1.7 \cdot 10^{308}$ ... $+1.7 \cdot 10^{308}$
`long double`	10	$-1.1 \cdot 10^{4932}$ ... $+1.1 \cdot 10^{4932}$

- Nutzen der Exponentialschreibweise zum "Sparen von Bits", z.B.:

  - 0,0000356 = 3.56 * 10^(-5)
  - 356 000 000 = 3.56 * 10^(8)
  - 3,1416 = 3.1416 * 10^0

  Mantisse Exponent

- interne Darstellung: $Mantisse * 2^{Exponent}$ (nach IEEE 754)
- von `float` zu `long double` wächst die Genauigkeit (Dezimalstellen)

Literale

- bezeichnen eine Konstante, die durch ihren Wert dargestellt wird
- **ganzzahlige Literale:**
  - `int`: dezimal `[1-9][0-9]*` oder `0` z.B.: `26`
    oktal `0[0-7]*` z.B.: `032`
    hexadezimal `0x[0-9a-f]+` oder `0X[0-9A-F]+` z.B.: `0x1a`

    $\rightsquigarrow$ stets positiv; ggf. Minus-Operator anwenden
  - Suffix `u` oder `U` $\rightsquigarrow$ `unsigned`
  - Suffix `l` oder `L` $\rightsquigarrow$ `long`

Gleitkomma-Literale

• `double`:	Dezimalbruch	mit Dezimalpunkt	z.B.: `300.0`
	Exponentialdarstellungen	$Mantisse$`e`$Exponent$	z.B.: 3e2
	(zur Basis 10)	$Mantisse$`E`$Exponent$	z.B.: .3E3

- Suffix `f` oder `F` $\rightsquigarrow$ `float`
- Suffix `l` oder `L` $\rightsquigarrow$ `long double`
- Beispiele erlaubter Werte im Quellcode:

```
3.1416e-4          // Exponentialschreibweise (e=10)
3.14159265e+300    // ein sehr großer Gleitkommawert
17.42              // auch ohne Exponent
-35                // auch ganze Zahlen möglich
```

Character-Literale

- **Character-Literale:**
  - Zeichen in einfachen Hochkommata, z.B. `'0'`
  - Ersatzdarstellungen nicht druckbarer Zeichen in einfachen Hochkommata, z.B. `'\n'`, `'\t'`, aber auch `'\''`
  - Oktal- oder Hexadezimaldarstellung des Zeichens: `'\`*Oktalziffern*`'` (`'\060'`) bzw. `'\x`*Hexadezimalziffern*`'` (`'\x30'`)

Arithmetik in C

Operatoren (1)

- arithmetische Operatoren: `+`, `-`, `*`, `/`, `%`
  (% (Modulo) nur für den Ganzzahltyp)
- Vergleichsoperatoren: `==`, `!=`, `<`, `>`, `<=`, `>=`
- bei geschachtelten Operatoren:
  - vordefinierte *Prioritäten* (z.B. * vor +)
    Bsp.: 3+5*2 ergibt 13
  - Auswertung bei gleicher Priorität von links nach rechts (*Linksassoziatvität*)
    Bsp.: 5-3-2 ergibt 0
  - explizite Reienfolge mit Klammern ausdrücken!

Operatoren (2)

- Ausdrücke haben einen Rückgabewert $\rightsquigarrow$ können Teil eines Ausdrucks sein
- Inkrement und Dekrement in Präfix- und Suffixnotation:
  - **Präfixnotation**: `++A` bzw. `--A`
    $\rightsquigarrow$ Rückgabewert ist Inkrement bzw. Dekrement von `A`
    Nebeneffekt: Wert von `A` ist in- bzw. dekrementiert
  - **Postfixnotation**: `A++` bzw. `A--`
    $\rightsquigarrow$ Rückgabewert ist der Wert von `A`
    Nebeneffekt: Wert von `A` ist in- bzw. dekrementiert

zum Vergleich:
`A + B` gibt die Summe der Werte von `A` und `B` zurück; keine Nebeneffekte

Operatoren (3)

- Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=`, `%=` geben den Wert der Operation zurück und weisen als Nebeneffekt diesen Wert dem linken Ausdruck zu, z.B.:

  `x += 8` realisiert `x = x + 8`

- bitweise Operatoren, sonstige Operatoren, Assoziativitäten und Prioritäten s. Literatur, z.B.:

  M. Dausmann, U. Bröckl, J. Goll: C als erste Programmiersprache. Teubner Verlag/GWV Fachverlage, Wiesbaden, 2008.

- Funktionsaufrufe können eingebunden werden (z.B. `3+sin(1.2)`)

- vordefinierte mathematische Funktionen und Konstanten
  `#include <math.h>`

Beispiele vordefinierter Funktionen (math.h)

Signatur	Beschreibung
`double sin(double x)`	Sinus von `x`
`double asin(double x)`	Arcussinus von `x`
`double exp(double x)`	e^x
`double log(double x)`	$\ln(x)$
`double log10(double x)`	$\log_{10}(x)$
`double ceil(double x)`	Aufrunden von `x` auf nächste Ganzzahl
`double floor(double x)`	Abrunden von `x` auf nächste Ganzzahl
`double pow(double x, double y)`	x^y
`double fabs(double x)`	Absoluter Betrag von `x`
`double sqrt(double x)`	Quadratwurzel aus `x`

Typumwandlung

- bei Kombination verschiedener Typen in einem Ausdruck oder Zuweisung eines Ausdrucks an eine Variable eines anderen Typs

- explizite Typumwandlung durch den Programmierer:
  (*Zieltyp*) *Ausdruck*

```
int x = 3;
float pi = 3.14;
x = x * (int) pi;   // x ist 9
```

- implizite Typumwandlung durch den Compiler:
  wenn immer es nötig und möglich ist, mit möglichst geringem Genauigkeitsverlust

Implizite Typumwandlung bei einfachen Typen

- Wenn Operatoren verschiedene Typen verknüpfen:
  - kleinere Ganzzahltypen werden immer nach `int` umgewandelt.
    (`unsigned short` in `unsigned int`, falls `short` und `int` äquivalent sind)
  - Umwandlung aller Operanden in den höchsten Typ des Ausdrucks gemäß

    `int` → `unsigned int` → `long` → `unsigned long` → `long long`
    → `unsigned long long` → `float` → `double` → `long double`

- Bei Wertzuweisungen, z.B. `float i = 5;`
  - Umwandlung des rechten Ausdrucks in Typ der linken Variablen

Implizite Typumwandlung (Beispiele)

Datentyp von `x`	Datentyp von `y`	Datentyp von `x*y` und `y*x`
double	float	double
double	int	double
float	int	float
long	int	long

etc.

Verhalten der Werte bei Typumwandlungen

- großer $\rightarrow$ kleinerer Ganzzahltyp $\rightsquigarrow$ *Abschneiden der oberen Bits*
- Ganzzahltyp $\rightarrow$ Gleitpunkttyp $\rightsquigarrow$ (meist) nächster darstellbarer Wert
- Gleitpunkttyp $\rightarrow$ Ganzzahltyp $\rightsquigarrow$ Abschneiden der Nachkommastellen
- Gleitpunkttyp $\rightarrow$ Typ mit zu kleinem Wertebereich $\rightsquigarrow$ *unbestimmt*

Ein- und Ausgabe mit C-Programmen

Formatierte Ausgabe mit `printf()`

- variable Anzahl von Parametern (Argumenten)
- erstes Argument wird ausgegeben (s. `printf("Hello world!\n")`)
- Argumente durch $Komma$ voneinander getrennt
- erstes Argument kann auf Werte der weiteren Argumente zugreifen, z.B.:

```
int x = 42;
printf("%d\t%d\n", 1, x); ⇝   1       42
```

  - `%d` ist ein Formatelement:
    der nächste, noch nicht verwendete Parameter wird an Stelle des `%d`
    als dezimale ganze Zahl ausgegeben
  - `\t` ⇝ Tabulatorschritt

Formatelemente von `printf()`

`%d`	dezimale ganze Zahl
`%`m`d`	dezimale ganze Zahl, mindestens m Zeichen breit
`%f`	Gleitpunktzahl (`double`)
`%`m`f`	Gleitpunktzahl, mindestens m Zeichen breit
`%.`n`f`	Gleitpunktzahl, n Nachkommastellen
`%`m`.`n`f`	Gleitpunktzahl, mind. m Zeichen inkl. n Nachkommastellen
`%o`	oktale ganze Zahl
`%x`	hexadezimale ganze Zahl
`%c`	einzelnes Zeichen (Datentyp `char`)

(mehr auf der Manpage `man 3 printf`; meist wie in Python)

Benutzereingaben mit `scanf()`

- formatiertes Einlesen eines Wertes mit `scanf`
- benutzt auch Formatelemente (s. Manualseite von `scanf`)
- Beispiele: Einlesen einer ganzen Zahl und Speichern auf `int n`:
  `scanf("%d", &n);`

  Einlesen einer Gleitkommazahl und Speichern auf `double x`:
  `scanf("%lf", &x);` (`%f`: `float`)

  Hinweis: Das Zeichen `&` ist nötig, da die Speicheradresse der Variablen angegeben werden muss (und kein Zugriff auf den Wert der Variablen erfolgt; s. Vorlesung zu "Pointern")

Kontrollstrukturen

Sequenzen

1. **Block**: $\{Anweisung\_1$
   $Anweisung\_2$
   $\vdots$
   $Anweisung\_n$
   $\}$

   - Block fasst eine Folge von Anweisungen zusammen $\rightsquigarrow$ „$Sequenz$“
   - Block kann überall auftreten, wo Anweisungen stehen dürfen
     $\rightsquigarrow$ Blöcke können geschachtelt werden
   - kein Semikolon nach einem Block

   $\rightsquigarrow$ Anweisungen der `main()`-Funktion bilden einen Block

Iterationen (1)

2. **while**-Schleife: `while (`*Ausdruck*`)`
   Anweisung

   - Wiederholung der *Anweisung* solange, bis der *Ausdruck* den Wert `0` hat
     - **arithmetischer** *Ausdruck*
     - **boolescher** *Ausdruck* mit Werten $\neq$`0` für `true` oder `0` für `false`
       $\rightsquigarrow$ Vergleichsoperatoren: `==, !=, <=, >=, <, >`
       und logische Operatoren `&&, ||, !`
   - kann auch gar nicht ausgeführt werden $\rightsquigarrow$ „*abweisende Schleife*“

3. **do while**-Schleife: `do {`
   Anweisung
   `} while (`*Ausdruck*`)`

   - wird mindestens einmal durchlaufen $\rightsquigarrow$ „*annehmende Schleife*“

Iterationen (2)

4. **for**-Schleife: `for (`*init*`;` *test*`;` *update*`)`
   Anweisung

init:	Anweisung	$\rightsquigarrow$	Initialisierung des Schleifenzählers
test:	Ausdruck	$\rightsquigarrow$	„`while`-Bedingung“
update:	Anweisung	$\rightsquigarrow$	Überschreiben des Schleifenzählers

Beispiel: `for (i = 0; i < 10; i++)`

- Schleifenzähler muss als Ganzahltyp definiert sein
- *init*, *test* oder *update* können leer sein, z.B. `for(;;)`
  $\rightsquigarrow$ ggf. Anweisung(en) in der Schleife zur Steuerung nutzen
- dynamische, abweisende Schleife

Selektionen (1)

5. **einfache Selektion** `if` (*Ausdruck*)
   Anweisung_1
   `else`
   Anweisung_2

   - Der `else`-Zweig ist optional.
   - Bei Schachtelung von `if`-Anweisungen bezieht sich ein `else`-Zweig immer auf die letzte `if`-Anweisung ohne `else`.

```
if (x >= 0)
  if (x > 0)
    printf("groesser als Null\n");
  else;
else
  printf("kleiner als Null\n");
```

Selektionen (2)

6. **Mehrfachselektion** – `else if`

- Auswahl unter mehreren Alternativen
- `if`-Anweisung als Anweisung im `else`-Zweig

```
if (Ausdruck_1)
  Anweisung_1
else if (Ausdruck_2)
  Anweisung_2
      ⋮
else if (Ausdruck_n)
  Anweisung_n
else                          /* optional ...
  Anweisung_else                        ... optional /*
```

Selektionen (3)

7. Wert-gesteuerte, direkte Verzweigung mit `switch`
   (nur für Variablen mit ganzzahligem Datentyp)

```
int a = ...;
switch(a) {
  case 10:
    Anweisung für den Fall a ist 10
    break;
  case 20:
    Anweisung für den Fall a ist 20
    break;
  default:
    Anweisung für alle anderen Fälle
    break;
}
```

Sprunganweisungen

1. `break;` $\rightsquigarrow$ Abbruch der Schleife
   $\rightsquigarrow$ zur ersten Anweisung nach der Schleife

2. `continue;` $\rightsquigarrow$ Abbruch des Schleifendurchlaufs
   $\rightsquigarrow$ **for:** zum $Update$ der Schleife
   $\rightsquigarrow$ **while:** zur $Bedingung$ der Schleife
   $\rightsquigarrow$ **do-while:** zur ersten Anweisung der Schleife

3. `goto` $Marke$`;` Sprung in Anweisung hinter $Marke:$
   nur zum Abfangen von Laufzeitfehlern einsetzen

Blöcke und Variablen

- In Blöcken deklarierte Variablen sind nur innerhalb dieses Blockes sichtbar.
  - $\rightsquigarrow$ auch in enthaltenen Blöcken
  - $\rightsquigarrow$ aber **nicht** in umfassenden Blöcken
- In Blöcken deklarierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken deklarierte Variablen werden beim Verlassen des Blockes wieder ungültig.
  - $\rightsquigarrow$ überdeckte Variablen werden wieder sichtbar

Praxis der Programmierung

Funktionen, Fehlerbehandlung, Pointer

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

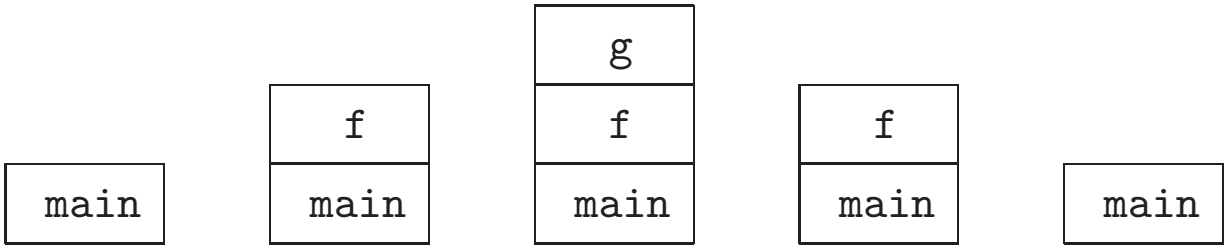
Funktionen in C

Funktionsaufrufe und -definitionen

- **Aufruf** (z.B. in `main`): `f(g(3),1.5);`

  1. Aufruf von `f` mit Argumenten/Parametern `g(3)` und `1.5`
  2. Aufruf von `g` mit Argument/Parameter `3`
     $\rightsquigarrow$ `f` pausiert, während `g` rechnet
  3. Rückgabe von $n = g(3)$ an `f`
  4. `f` rechnet weiter mit Argumentwerten n und `1.5`
  5. Rückgabe des Funktionswertes $f(n, 1.5)$ an den Aufrufer von f

- Jede Funktion rechnet in einem eigenen Speicherbereich (**Stackframe**).



Funktionsaufrufe und -definitionen

- **Aufruf:** `f(g(3),1.5);`
- setzt **Definition** der Funktionen `f` und `g` voraus, *z.B.*:

```
double f(int x, double y) { ... }
int g(int z) { ... }
```

Funktionskopf *Funktionsrumpf*

- Definition von Funktionen als *weitere* Funktionen *neben* `main`

Funktionskopf (Signatur der Funktion)

- $R\ddot{u}ckgabetyp\ Bezeichner\ (Typ\_1\ formaler\_Parameter\_1,$
  $Typ\_2\ formaler\_Parameter\_2,$
  $\vdots$
  $Typ\_n\ formaler\_Parameter\_n)$

- legt die **Schnittstelle** der Funktion fest:
  - legt Namen der Funktion fest: $Bezeichner$
  - legt Definitionsbereich der Funktion fest: Liste $formaler\ Parameter$
  - legt Wertebereich der Funktion fest: $R\ddot{u}ckgabetyp$

- legt fest, wie die Funktion aufgerufen wird

Funktionsrumpf

- ist ein Block (Sequenz von Anweisungen)
- definiert das **Verhalten**:
  legt fest, wie die Eingabewerte verarbeitet werden
- `return`-Anweisung $\rightsquigarrow$ Rückkehr zu aufrufender Funktion
  $\rightsquigarrow$ Rückgabewert übergeben

Beispiel:

```
int quadrat (int n) {
  return n * n;
}
```

$\rightsquigarrow$ nach `return` kann Ausdruck stehen

Rückgabetyp `void`

- Funktionen ohne Rückgabewert
  $\rightsquigarrow$ reine Prozeduren
- dienen z.B. zur bloßen Datenausgabe
- keine `return`-Anweisung *oder* `return;`
- `void prozedur_funktion ( ... ) { ... }`

Funktionsaufruf

Funktionsname (*aktueller_Parameter_1*, ..., *aktueller_Parameter_n*);

1. automatisches Anlegen von lokalen Variablen für die Parameter

   $\rightsquigarrow$ *typ_i formaler_Parameter_i*;

2. automatische Initialisierung mit aktuellen Parametern

   $\rightsquigarrow$ *formaler_Parameter_i* = *aktueller_Parameter_i*;

   Aktuelle Parameter können **Ausdrücke** sein!

3. Abarbeitung der Anweisungen im Funktionsrumpf

Funktionsaufruf ist Ausdruck

```
int qud = quadrat(12);        // liefert qud = 144;
```

Beispiel

```
void f(int x) {
    int y = 4;
    printf("f: x=%d, y=%d", x, y);
}

int main() {
    int x = 1;
    int y = 2;
    int a = 3;

    f(a);       // x=3, y=4

    printf("x=%d, y=%d", x, y);     // x=1, y=2

    return 0;
}
```

Erinnerung: Blöcke und Variablen

- In Blöcken definierte Variablen sind nur innerhalb dieses Blockes sichtbar.
  $\rightsquigarrow$ auch in enthaltenen Blöcken
  $\rightsquigarrow$ aber **nicht** in umfassenden Blöcken
- In Blöcken definierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken definierte Variablen werden beim Verlassen des Blockes ungültig.
  $\rightsquigarrow$ verdeckte Variablen werden wieder sichtbar
- **Lokale Variablen** einer Funktion verdecken Variablen ihres Aufrufers.
  - Variablen, die im Rumpf definiert werden,
  - Parameter der Funktion

  Sie werden erst **im Stackframe** des jeweiligen Funktionsaufrufs angelegt.

Verdeckung – Beispiel

```
int main() {
    int x = 1;
    int y = 2;

    if (x==1) {
        int y = 5;
        printf("y=%d", y);   // y=5
    }

    printf("y=%d", y);       // y=2

    return 0;
}
```

Globale Variablen

- werden außerhalb jeder Funktion deklariert
- können damit von allen Funktionen der Datei verwendet werden
- gültig während des gesamten Programmlaufs

Globale Variablen – Beispiel

```
int counter;    // globale Variable

void count() {
    counter +=1;
}

int main() {
    // int counter;  Verdeckung vermeinden!
    counter = 0;
    count();

    printf("Zaehler: %d\n", counter);    // 1
    return 0;
}
```

Deklarieren *versus* Definieren

- Funktion deklarieren
  Festlegung der Schnittstelle (Signatur)

```
long cube (int n);
```

- Funktion definieren
  Festlegung der Schnittstelle (Signatur) und des Verhaltens (Implementierung)

```
long cube (int n) {
   return n*n*n;
}
```

- Anwendung: **Vorwärtsdeklaration**
  Funktionen müssen *vor* ihrem ersten Aufruf deklariert sein
  (Implementierung kann an anderer Stelle erfolgen)

Vorwärtsdeklaration

Beispiel:

```
void g(int n);                  // Deklaration von g

void f(int n) { g(n-1); }       // Definition von f
                                // setzt voraus, dass g deklariert ist

void g(int n) { f(n-1); }       // Implementierung von g
                                // setzt voraus, dass f deklariert ist
```

Fehler und Fehlerbehandlung

Fehlerarten

- **Compilerfehler**: Fehler, die der Compiler erkennt, z.B.
  - Semikolon vergessen
  - Variable benutzt aber nicht definiert etc.

  $\rightsquigarrow$ kein ausführbares Programm

- **Laufzeitfehler**: Fehler, die bei der Abarbeitung des Programms auftreten, z.B.
  - Division durch 0
  - fehlende Werte / Zugriff auf Datei, die nicht geöffnet werden kann
  - kein Speicherplatz mehr vorhanden etc.
  - Zugriff auf geschützte Ressource: **Segmentation Fault**

- **logische Fehler**: Fehler im Programmentwurf

Fehlerbehandlung

- **Compilerfehler**:
  Fehlermeldungen des Compilers beachten und Quellcode korrigieren

- **Laufzeitfehler**:
  - Programmierer ist verantwortlich, nicht der Nutzer!
  - mögliche Fehler identifizieren
  - im Programm auf speziellem Ausführungspfad abfangen
    (z.B. mit `if-else, switch,` ...)

- **logische Fehler**: vermeiden durch
  - sorgfältigen Algorithmen- und Programmentwurf
  - Code-Review; Testen, Testen, Testen

Behandlung von Laufzeitfehlern

- in dem Block, in dem das Problem entstehen kann
- "geordneter" Programmabbruch mit **aussagekräftiger Fehlermeldung**, *z.B. in der Definition einer Funktion* `f`:

```
if (Fehlerursache) {
    printf("Fehler in f: Fehlerursache ist eingetreten.\n");
    exit(EXIT_FAILURE);   // in <stdlib.h> definiert
}
// Programmstelle, die bei Fehlerursache einen Fehler ausloesen kann
```

- *oder* **Recover**-Mechanismus, wenn der Programmlauf gerettet werden soll

  $\rightsquigarrow$ *Zurückgehen zu der Stelle, in der die Fehlerursache entstanden ist*

Pointer (Zeiger)

Vier Kennzeichen einer Variablen

- Datentyp
- Variablenname
- Wert
- Adresse im Arbeitspeicher (Primärspeicher)

In **C**: Zugriff über Namen oder Adresse auf den Wert

Adressen von Variablen

- Arbeitsspeicher ist in (gleich große) *Speicherzellen* eingeteilt (z.B. je 1 Byte)
- Speicherzellen sind durchnummeriert (Hexadezimalzahlen)
- **Adresse** einer Variablen ist *Nummer* der Speicherzelle, in der ihr Speicherplatz beginnt.

Variablen und ihre Adressen

Variable `alpha`
speichert den Wert, z.B. 3,
eines bestimmten Datentyps
an einer Speicheradresse

Definition: `int alpha = 3;`

`alpha` hat die Adresse `&alpha`

`&` - **Adressoperator**
ermittelt die Adresse / die Referenz
auf `alpha`

Pointer `p`
zum Speichern der Adresse von `alpha`,
an der der Wert 3 abgelegt wird

Definition: `int *p = &alpha;`

`p` *zeigt* auf Speicherplatz von `alpha`
an dem der Wert `*p` = 3 abgelegt ist

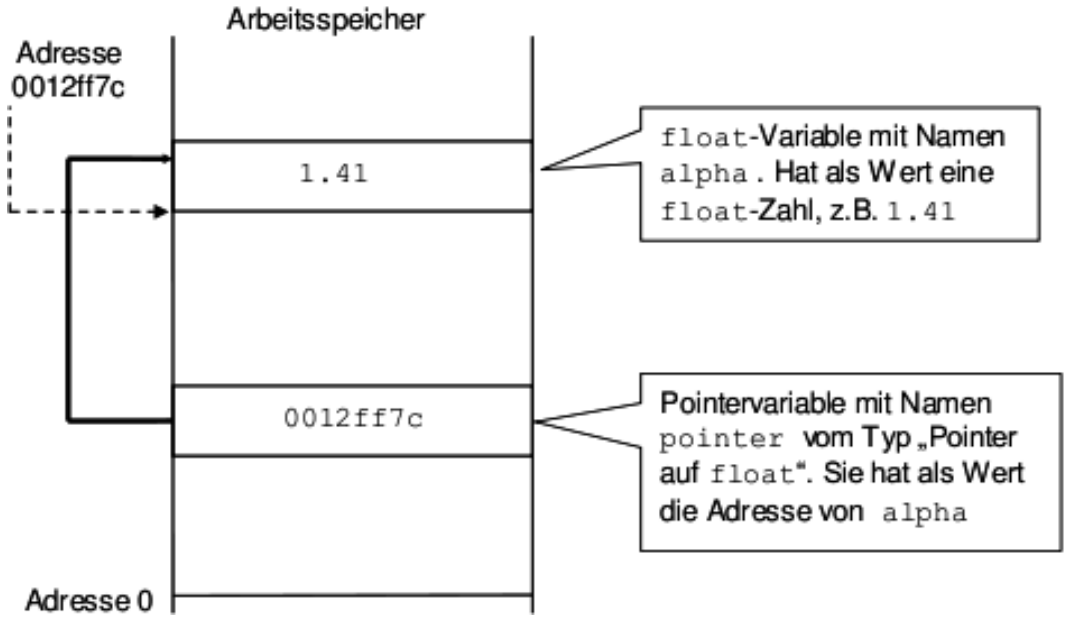
`*` - **Inhaltsoperator**
ermittelt den Wert an Speicherplatz `p`
(Dereferenzierungsoperator)

Pointer

- **Pointer:** ist Variable

  Typ: Pointer auf einen bestimmten Datentyp (z.B. float)

  Wert: *Adresse* einer Variablen




```
 alpha == 1.41
&alpha == 0012ff7c

 pointer == 0012ff7c
*pointer == 1.41
```

Pointervariablen

- Definition als Pointer auf *Datentyp*
  $\rightsquigarrow$ Pointertyp und Datentyp des Speicherobjekts sind gekoppelt!

- *Typname* `*` *Pointername*;
  $\rightsquigarrow$ Datentyp: Pointer auf *Typname*

- Beispiel: `float * pointer1;`
  $\rightsquigarrow$ Pointer auf `float` mit Namen `pointer1`

- Vorsicht:

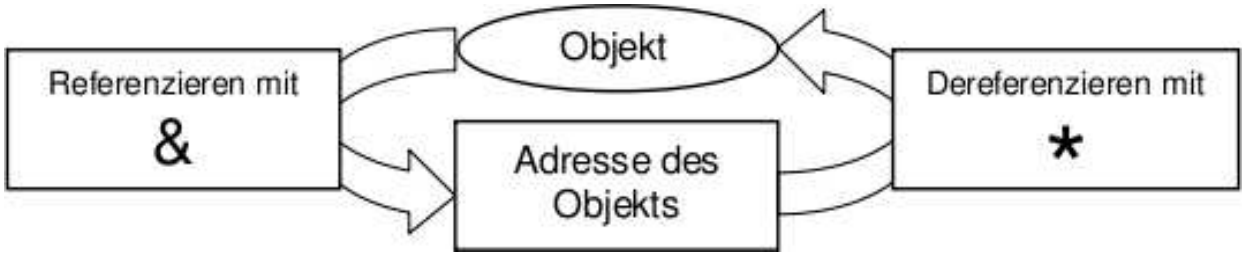
Definition	entspricht
`int * pointer, alpha;`	`int * pointer;` `int alpha;`
`int * pointer1, * pointer2;`	`int * pointer1;` `int * pointer2;`

Wertzuweisung an einen Pointer

1. **Adressoperator**
   - Adressoperator **&** liefert Pointer auf Speicherplatz einer Variablen
   - &$Variable$ $\rightsquigarrow$ Pointer auf $Datentyp$ von $Variable$
   - kann an Variable vom Typ Pointer auf $Datentyp$ zugewiesen werden:
     `pointer = &alpha;`
2. **Wert einer anderen Pointervariablen**
   - `pointer2 = pointer1;`
   - Übergabe der Adresse von einer Pointer-Variablen an eine weitere
   - nur bei Pointern auf den gleichen Typ!
3. **Konstante NULL**
   - vordefinierter Pointer, zeigt auf Adresse `0` / nie auf ein Speicherobjekt
   - darf nie dereferenziert werden ($\rightsquigarrow$ **Segmentation Fault**)

Dereferenzieren

- **Inhaltsoperator** $*$: $*Pointer$
- $*$ und & sind invers: `*&alpha` ist äquivalent zu `alpha`

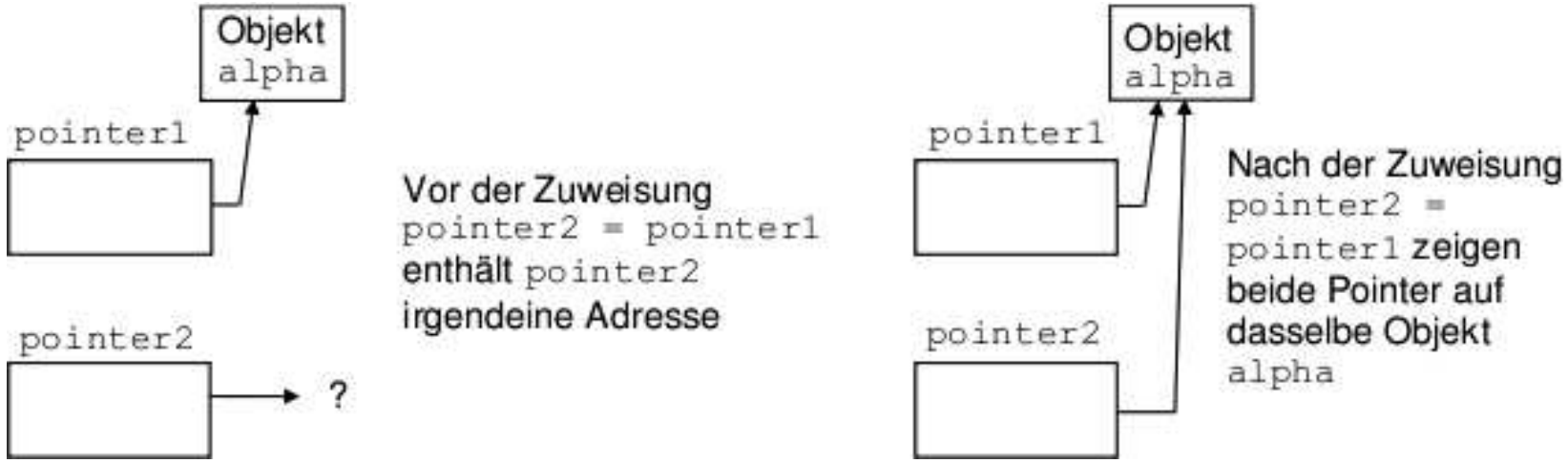


- `*pointer =` $Wert$; ist erlaubt
  $\rightsquigarrow$ nicht bevor `pointer` gültige Adresse speichert!

Arbeit mit Pointern

- nicht initialisierte Variablen haben *irgendeinen* Wert!

$\rightsquigarrow$ nicht initialisierte Pointer verweisen auf *irgendeine* Adresse!
$\hookrightarrow$ *Fehlerquelle!!!*



Rückblick: Benutzereingaben mit `scanf`

- formatiertes Einlesen eines Wertes mit `scanf`
- benutzt Formatelemente
- Beispiele: Einlesen einer ganzen Zahl und Speichern auf `int n`:
  `scanf("%d", &n);`

  Einlesen einer Gleitkommazahl und Speichern auf `double x`:
  `scanf("%lf", &x);`

  Hinweis: Das Zeichen `&` ist nötig, da die Speicheradresse der Variablen angegeben werden muss (und kein Zugriff auf den Wert der Variablen erfolgt)

  $\rightsquigarrow$ Pointer können als Parameter an Funktionen übergeben werden!!!

Parameterübergabe

Parameterübergabe mit call-by-value

- Formaler Parameter einer Funktion = Vereinbarung einer lokalen Variablen: Name und Typ des formalen Parameters
- **Kopie** des aktuellen Parameterwertes als Wert dieser lokalen Variablen
- Im Funktionsrumpf wird immer die lokale Variable verwendet.
  $\rightsquigarrow$ *kein Einfluss auf aktuellen Parameter*
  $\rightsquigarrow$ keine Seiteneffekte
- Ausdruck des aktuellen Parameters wird nur einmal ausgewertet
- verwendet in: C, C++, Java, C#, PASCAL, ...

call-by-value - Beispiel

```
int var1 = 1;
int var2 = 2;
```

Aufgabe: Tausch der Werte von `var1` und `var2`
mit einer Funktion `swap(int m, int n)`

```
void swap(int m, int n) {
   int h = m;
   m = n;
   n = h;
}
```

$\rightsquigarrow$ Aufruf `swap(var1, var2)` ohne Effekt! (*Warum?*)

Andere Methoden der Parameterübergabe — nicht in C

- **call-by-value-result**
  - zunächst wie call-by-value:
    - lokale Variable mit Wert des aktuellen Parameters initialisiert
    - keine Änderung am aktuellen Parameter
  - am Ende (z.B. bei `return;`):

    eine weitere automatische Wertzuweisung

    $\rightsquigarrow$ Wert der lokalen Variablen $\longrightarrow$ aktueller Parameter

- **call-by-reference**
  - Vereinbarung von lokalen Variablen für alle Variablen in den aktuellen Parametern
  - referenzieren die Speicheradressen der Variablen in den aktuellen Parametern
  - Zuweisungen beeinflussen die Werte der aktuellen Parameter direkt
  - **in C**:
    Call-by-Reference-Semantik durch Übergabe von Pointern als Parameter

```
void swap(int *m, int *n) {
   int h = *m;
   *m = *n;
   *n = h;
}
```

Beispiel:

```
void f(int x, int y) {
    x = x + 1;
    y = y + 1;
}

int a = 1;
f(a,a);
```

call-by-value: `a = 1`
call-by-value-result: `a = 2 // nicht in C`
call-by-reference: `a = 3 // nicht in C`

- **call-by-name**
  - textuelle Übergabe des aktuellen Parameters
  - Im Funktionsrumpf wird jedes Vorkommen des formalen Parameters textuell durch den aktuellen Parameter ersetzt.
  - Seiteneffekt: kann aktuellen Partameter beeinflussen
  - Ausdruck des aktuellen Parameters wird mehrfach ausgewertet
  - kann zu Konflikten führen

    (falls Variablen des Funktionsrumpfes einen Bezeichner verwenden, der Teil eines aktuellen Parameters ist)

```
int foo(int a) {
  int x = 1;
  return a + x;
}
```

```
int x = 300;
foo(x)  // gibt 2 zurueck
/* egal welchen Wert x hatte */
```

Praxis der Programmierung

Arrays, Pointerarithmetik, Konstanten, Makros

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Arrays (Felder)

Arrays: Motivation

- **Gegeben:** Durchschnittstemperaturen der Monate der letzten fünf Jahre

```
double t_2019_01 = 3.7;
double t_2019_02 = 4.1;
  ...
double t_2023_12 = 4.2;
```

- **Aufgabe:** Berechnung von Durchschnittstemperaturen für Jahre, Monate, Jahreszeiten, ...
- *naiver Ansatz:*
  - `double`-Variable für jeden Monat (12 x 5 Variablen);
  - manuelle Berechnung

```
double avg_january =
        (t_2019_01 + t_2020_01 + t_2021_01 + t_2022_01 + t_2023_01) / 5.0;
```

⇝ Schleife zur Summenbildung ??? ⇝ keine gute Idee!!!

Arrays als besserer Ansatz

- Arrays fassen mehrere Variablen unter einem Namen zusammen
- Zugriff auf Werte über den gemeinsamen Namen + Nummer (Index)
- ähnlich zu Listen in Python
- Werte müssen einheitlichen Datentyp haben
- Größe des Arrays (Anzahl der Elemente) unveränderbar

$\rightsquigarrow$ Datentyp und Größe bei der Definition der Array-Variablen festlegen

Array-Definition und -Zugriff

- **Definition:** *Typname Arrayname* `[`*Größe*`];`

  Beispiele: `int ar [5];`
  `double temperaturen[12*5];`

- **Zufriff:** auf das Array-Element mit Index i: *Arrayname*`[i]`

  Beispiel: `ar[3]`

- Indizierung beginnt bei 0
  $\rightsquigarrow$ letztes Element: Länge des Arrays minus 1

`temperaturen:` 

(Werte werden im Speicher direkt hintereinander abgelegt.)

Initialisierung von Arrays (1)

Initialisierungslisten

- Elemente in geschweiften Klammern, durch Komma getrennt
- nur direkt bei der Definition

  $\rightsquigarrow$ `int ar [4] = {1, 2, 3, 4};`
- fehlende Elemente werden mit 0 aufgefüllt:

  `int ar [4] = {1, 2};` $\rightsquigarrow$ `ar[0]=1; ar[1]=2; ar[2]=0; ar[3]=0;`
- Größe in eckigen Klammern darf fehlen (**offenes Array**)

  `int ar [] = {1, 2, 3, 4};`

Initialisierung von Arrays (2)

Elemente einzeln setzen, meist in Schleifen

```
int i, ar[5];
for (i = 0; i < 5; ++i) {
   ar[i] = i+1;
}
```

$\rightsquigarrow$ `ar[0] = 1, ar[1] = 2, ar[2] = 3, ar[3] = 4 , ar[4] = 5`

Lösung des motivierenden Problems

am Beispiel: Durchschnitt aller Werte der letzten fünf Jahre:

```
double avg = 0;
int i;
for (i = 0; i < 12*5; ++i) {
   avg += temperaturen[i];
}
avg /= i;
```

Mehrdimensionale Arrays

Arrays können komplexe Daten speichern ... auch Arrays

- Verwendung mehrfacher eckiger Klammern
- Elemente sind selbst Arrays (eine Dimension niedriger)
- Beispiel: `int ar [3][4];`

`ar[0][0]`	`ar[0][1]`	`ar[0][2]`	`ar[0][3]`
`ar[1][0]`	`ar[1][1]`	`ar[1][2]`	`ar[1][3]`
`ar[2][0]`	`ar[2][1]`	`ar[2][2]`	`ar[2][3]`

- Initialisierungsliste: $\{\{\ldots\},\ldots,\{\ldots\}\}$

Was können Arrays in C nicht?

- Ändern ihrer Länge

```
int ar[10];
ar[10] = 1;        // Diese Speicherstelle gehoert nicht zum Array!!!
```

- direktes Abfragen der Länge des Arrays, z.B.

  `for(i = 0; i < len(ar); i++)     // kein len`

  ⇝ `int len = (int) sizeof(ar)/sizeof(int)`

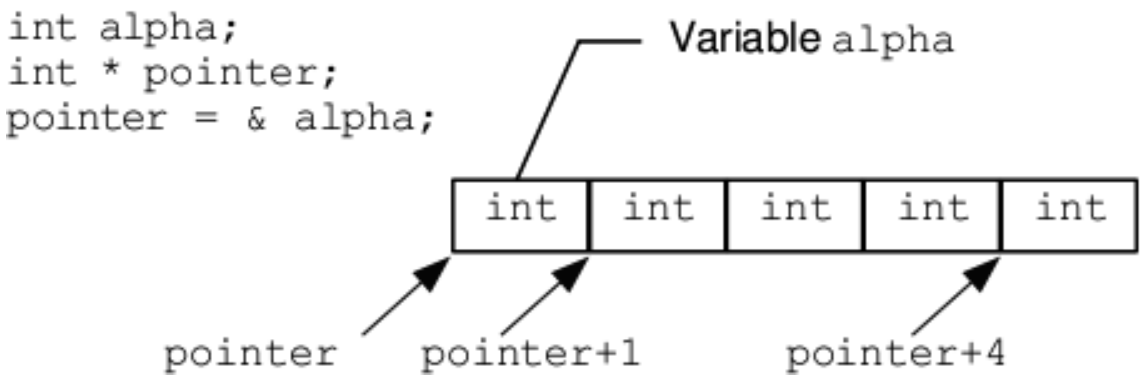
⇝ Vorlesung „Dynamische Speicherverwaltung“

Pointer und Arrays

Pointerarithmetik

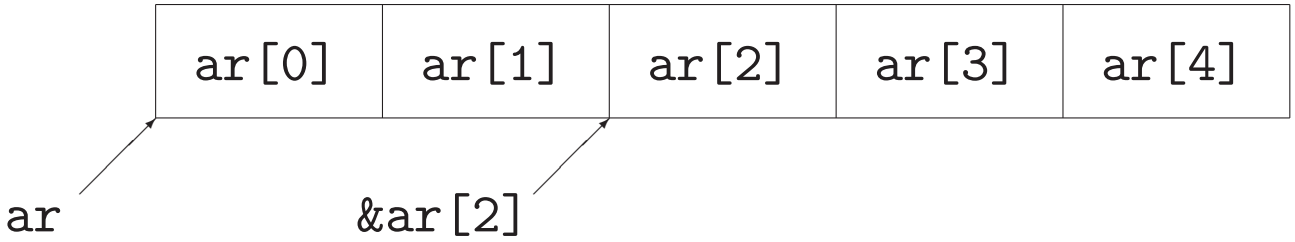
- Vergleich (== und !=) für Pointer desselben Typs
- Addition und Subtraktion (einer ganzen Zahl n)

  $\rightsquigarrow$ Verschieben des Zeigers um n Speicherobjekte des Typs, auf den der Pointer zeigt

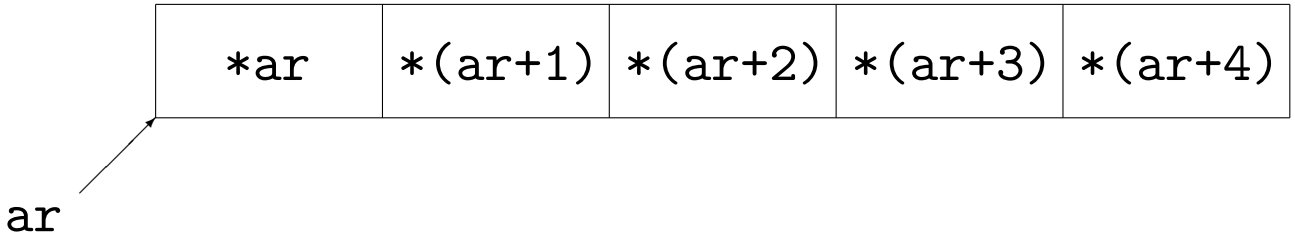


Arrays und Pointer

- Name des Arrays ist **konstanter** Zeiger auf das erste Array-Element



- `ar[i]` ist äquivalent zu `*(ar+i)`



- Sei `ptr` ein Pointer. Dann ist `*(ptr + i)` äquivalent zu `ptr[i]`.

Arrays und Pointer als Parameter

- **Problem:** Arraylänge kann nicht direkt abgefragt werden
  $\rightsquigarrow$ muss vom Programmierer übergeben werden

- **Beispiel:** Summe der Arrayelemente

```
int sum(int p[], int n) {
   int s = 0, i = 0;
   for(i=0; i<n; ++i) {
      s += p[i];
   }
   return s;
}
```

```
int sum(int *p, int n) {
   int s = 0, i = 0;
   for(i=0; i<n; ++i) {
      s += *(p++);
   }
   return s;
}
```

Pointer auf `void`

- `void *` *Pointername*`;`
- **untypisierter Pointer**: Datentyp steht nicht fest
- darf nicht dereferenziert werden (zeigt nie auf Speicherobjekte)
- kann in jeden Pointertyp umgewandelt werden (Zuweisung)
  $\rightsquigarrow$ kein Verlust an Information oder Genauigkeit
- für die Zuweisung von Pointern auf anderen Typ verwenden

Verwendung von `void`-Pointern — Beispiel

Abgreifen des ersten Bytes eines `int`-Wertes (als `unsigned char`):

```
int zahl = n;
int * pointer1 = &zahl;
unsigned char * pointer2;   // soll auf dieselbe Adresse wie pointer1 zeigen!

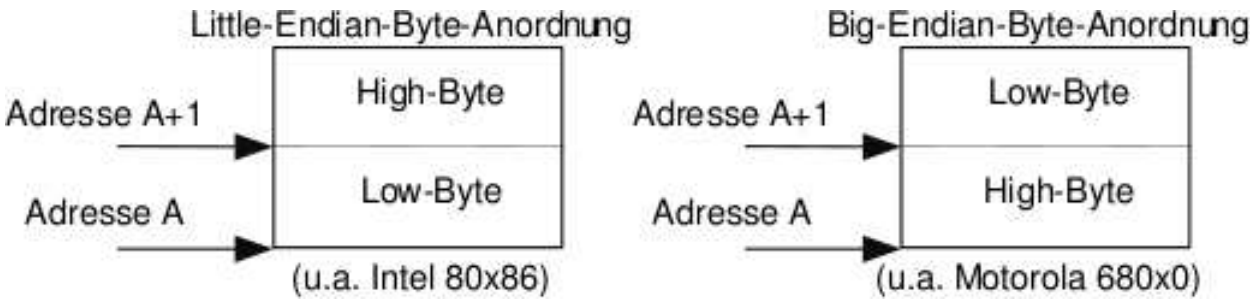
// pointer2  = pointer1;    unmoeglich (verschiedene Typen)

void * dummy;

dummy = pointer1;           // korrekt bei
pointer2 = dummy;           // Little-Endian-Byte-Anordnung
```

Byte-Anordnungen

... abhängig von der Rechnerarchitektur:



Bei Big-Endian-Byte-Anordnung müssen Pointer mit Hilfe der Pointerarithmetik zum gewünschten Byte bewegt werden.

Pointer: Pros und Cons

- Pointer erlauben hardwarenahes Programmieren
- Pointer erlauben dynamische Datenstrukturen
  $\longrightarrow$ Vorlesung „Dynamische Speicherverwaltung“
- Pointer sind häufig Quelle von schwer auffindbaren Fehlern, z.B.:

```
float v;
float *p = &v;
p[0] = 0.5;
p[1] = 3.14;  // ???
```

- Pointer können zu Datenverlust führen
  - Schreiben von Werten an nicht referenzierte Adressen
  - versehentliches Überschreiben von Werten, ...

Konstanten

Konstanten

- Literale (vordefinierte Konstanten elementarer Typen)
- Variablen, die mit Typattribut `const` definiert sind
  - `const` *Datentyp Bezeichner* `=` *Initialisierung*`;`
  - Variable nach Initialisierung schreibgeschützt
  - *Beispiel:* `const double PI = 3.1415927;`
  - *Konvention:* Bezeichner aus Großbuchstaben
  - für Variablen, Pointer, Parameter

$\rightsquigarrow$ Schutz vor unbeabsichtigten Änderungen

$\rightsquigarrow$ "Dingen einen Namen geben"
(z.B. PI, ROT, GELB, MONTAG, DIENSTAG, ...)
$\rightsquigarrow$ vereinfacht Lesbarkeit und Wartung des Quellcodes

Konstanten (2)

- Aufzählungstypen
- Definition mit Schlüsselwort `enum` und "Mengenschreibweise"
  $\rightsquigarrow$ Definition eines neuen Datentyps mit endlich vielen konstanten Werten
- Definition von Variablen dieses Typs mit Schlüsselwort `enum`

```
enum ausbildung {
       HAUPTSCHULE,
       REALSCHULE,
       ABITUR,
       BERUFSAUSBILDUNG,
       BACHELOR,
       MASTER
};
```

```
int akademiker (enum ausbildung a) {
     if (a == BACHELOR
      || a == MASTER)
               return 1;
     else
               return 0;
}
```

Aufzählungstypen

- Typ- und Variablendefinition können zusammen erfolgen:

```
enum boolean {FALSE, TRUE} b;
```

- Vereinbarungen ohne *Etikett*:

```
enum {FALSE, TRUE} b;
```

  $\rightsquigarrow$ Variablendefinition muss hier erfolgen; kein Typname vereinbart

- in **C**: keine Typprüfung durch den Compiler:

```
b = TRUE;    // o.k., typgerechte Verwendung
b = 5;       // weder Compiler- noch Laufzeitfehler !!!
```

Konstanten (3)

- symbolische Konstanten
  - `#define` $Name$ $Wert$
  - Präprozessor ersetzt $textuell$ alle Vorkommen von $Name$ durch $Wert$
  - *Beispiel:* `#define PI 3.1415927`
  - *Konvention:* $Name$ aus Großbuchstaben

$\rightsquigarrow$ keine Typprüfung durch den Compiler

Anwendung für Arraydefinitionen

```
#define MAX 60

int main() {
    int ar[MAX];
    int i;
    for(i = 0; i < MAX, i++)
          ...
    ...
    return 0;
}
```

⇝ Änderungen der Arraylänge erfordern Änderung der Programms nur an einer Stelle

Makros

- Verallgemeinerung des Vorgehens bei der Definition symbolischer Konstanten
- Textuelle Einsetzung funktioniert mit jeder Zeichenkette.
- Vermeidung lästiger Code-Wiederholungen durch Makro-Definition
- Beispiel: `printf("\n")` als Anweisung für den Zeilenvorschub

```
#include <stdio.h>
#define zv printf("\n")

zv;
printf("Hallo Praeprozessor!");
zv;
```

Makros — Beispiel

```
#include <stdio.h>
#define zv printf("\n")
#define printar for (i=0;i<4;++i) printf("%d : %d\t", i, ar[i])

int main() {
     zv;
     int i;
     int ar[4] = {10,20,30,40};
     printar;
     zv;
     for(i=1;i<4;++i)
          ar[i]++;
     printar;
     zv;
     return 0;
}
```

Soll das Ausgabeformat für das Array `ar` *verändert werden, braucht nur die Makrodefinition angepasst werden.*

Praxis der Programmierung

Zeichenketten (Strings), Ein- und Ausgabe

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Zeichenketten (Strings)

- String = Zeichenkette
- **konstante** (unveränderliche) Strings in Anführungszeichen definiert
  - bisher als Parameter von `printf`
  - `printf("Ich bin ein String.");`
- *allgemein:* Zeichenkette ist Folge von Character-Werten

$\rightsquigarrow$ kann als Array vom Typ `char` aufgefasst werden

Strings in C

- sind `char`-Arrays mit **Nullzeichen** `'\0'` als Markierung des Stringendes
- Viele String-Funktionen benötigen das Nullzeichen.
  $\rightsquigarrow$ bei Definition des Arrays einplanen!

  ```
  char vorname [6] = {'N', 'a', 'd', 'j', 'a', '\0'};
  ```

- Initialisierung mit konstanter Zeichenkette:

  ```
  char vorname [6] = "Nadja";
  ```

  $\rightsquigarrow$ automatisches Anfügen des Nullzeichens

- *Erinnerung:* Arraybezeichner liefert Pointer auf das erste Element
  $\rightsquigarrow$ alternative Definition: `char *vorname = "Nadja";`
  (ist Pointer auf `char`, nämlich auf den ersten Buchstaben)

Dynamische Strings

- Definition als char-Array: `char vorname [6] = "Nadja";`
- Definition als offenes Array ist möglich: `char str[] = "Hallo";`
- Normale Zugriffsmöglichkeiten wie bei allen Arrays
  ⇝ Überschreiben einzelner Buchstaben im String möglich

⇝ Pointer `str` ist konstant (kann nicht umgesetzt werden)

Statische Strings

- `char`-Pointer mit konstantem String initialisiert: `char *str = "Hallo";`
- Pointer kann auf andere Adresse umgesetzt werden
- Pointer auf *read-only* Speicherblock
  - im statischen Datensegment (Speicherbereich für Daten *neben* dem Stack für globale und statische Variablen) oder
  - direkt im Code-Segment

$\rightsquigarrow$ String darf nur gelesen werden

$\rightsquigarrow$ Veränderung des Strings löst **segmentation fault** aus oder kann zu schweren Fehlern führen (u.U. Änderung am Maschinencode)

Statische Strings benutzen

- Ausgabe des gesamten Strings mit `printf`:
  `printf(str);` oder `printf("... %s ...", str);`

$\rightsquigarrow$ Formatelement `%s` zum Integrieren in formatierte Ausgaben,
Übergabe eines `char`-Pointers (`str`)

Statische Strings benutzen (2)

- Zugriff auf einzelne Zeichen, z.B.:

```
char *str = "Hallo";
char c1 = str[0];   //  == 'H'  (in str nicht veraendern!!!)
char c2 = str[1];   //  == 'a'  (in str nicht veraendern!!!)
char c3 = str[5];   //  == '\0' (in str nicht veraendern!!!)
str = "String";     // erlaubt, da str nicht konstant
```

- Ergebnis:
  - in einem *read-only* Speicherbereich liegen zwei unbenannte char-Arrays (unveränderlich)
  - im Stack liegt ein Pointer `str` auf char (veränderlich)

Dynamische Strings benutzen

- lesender Zugriff wie bei statischen Strings
- außerdem schreibender Zugriff auf einzelne Buchstaben
- Definition z.B. als offene Arrays:

```
char str [] = "Hallo";  // char-Array mit 6 Komponenten
                        // liefert char-Pointer auf das 'H'

str[1] = 'e';           // wandelt den String in "Hello"

str = "String";         // verboten, da Pointer str konstant
str++;                  // verboten, da es Konstante veraendern wuerde
```

Dynamische Strings benutzen (2)

- Einlesen des Strings möglich, z.B. mit `scanf` oder `fgets`:

```
char str2[1024];
printf("Geben Sie Ihren Namen ein!");
fgets(str2, 1024, stdin);   // max. 1023 Zeichen (warum?)
                            // von stdin lesen
                            // und ab Adresse str2 speichern
```

```
char * fgets (char * s, int size, FILE * stream);
```

  - liest `size-1` Zeichen aus `stream`
    oder bis `'\n'` oder **EOF** und speichert sie ab Adresse `s`
  - `'\n'` wird mit gespeichert und `'\0'` angehängt
  - übergibt Pointer `s`

Standardfunktionen zur Eingabe aus `stdio`

- `char * fgets (char * s, int size, FILE * stream);`
  - liest `size-1` Zeichen aus `stream` oder bis `'\n'` oder **EOF** und speichert sie ab Adresse `s`
  - `'\n'` wird mit gespeichert und `'\0'` angehängt
  - übergibt Pointer `s`

- `char * gets (char * s);`
  - liest von `stdin` bis `'\n'` oder **EOF** und speichert ab Adresse `s`
  - ersetzt `'\n'` bzw. **EOF** durch das Nullzeichen
  - übergibt Pointer `s`

- *Warum ist* `gets` *im Vergleich zu* `fgets` *gefährlich?*

Standardfunktionen zur Eingabe aus `stdio` (2)

- `int scanf (const char * format, ...);`
  - Argumente nach dem Formatstring sind **Adressen von Variablen**,
  - Speichern der Werte aus `stdin` auf diesen Adressen
  - Anzahl und Typen der Formatelemente müssen zu den adressierten Variablen passen (sonst Abbruch des Einlesens)
  - Rückgabewert: Anzahl der erfolgreich eingelesenen Werte

- Beispiel:

```
int zahl;

printf ("\nEingabe: ");
scanf("%d", &zahl);

printf ("\nDer Wert %d wurde eingelesen.\n", zahl);
```

- andere Zeichen als Formatelemente im Formatstring möglich:
  - `scanf()` liest und ignoriert diese Zeichen („Wegwerfen“)
  - Whitespace-Zeichen: „Wegwerfen“ einer beliebigen Anzahl dieses Zeichens
  - nichtpassende Zeichen in `stdin` werden zurückgestellt (verbleiben)
  - verhält sich so, bis `'\n'` gelesen wird

```
float t;
printf("Temperatur im Format xx C: ");
scanf("%f C", &t);
t = (9. * t) / 5. + 32.;
printf("\nTemperatur in Fahrenheit: %f F", t);
```

$\rightsquigarrow$ Kein Newline-Zeichen `'\n'` im Formatstring von `scanf()`!

- „bewusstes“ Ignorieren von zum Formatstring passenden Eingaben durch `*` nach `%`: *Lesen ohne zu speichern*

  Datei `daten.txt` enthält
  `Artikel: Tisch Vorrat: 8 Einzelpreis: 290`

  Aufgabe: C-Programm `prog.c` soll Wert des Lagerbestands ermitteln:

```
int anzahl;
int einzelpreis;
scanf("%*s %*s %*s %d %*s %d", &anzahl, &einzelpreis);
printf("\nWert des Lagerbestands: %d", anzahl * einzelpreis);
```

  zu starten mit `prog < daten.txt`

Besonderheiten in der Signatur von `scanf`

```
int scanf (const char * format, ...);
```

- `...` – *Ellipse*
  - muss nach dem letzten expliziten formalen Parameter stehen
  - Anzahl (und Typen) weiterer Parameter offen
  - Beispiel:

```
int ellipse_func (int n, double x, ...);
...
ellipse_func(4, 5.6, "String");   // o.k.
ellipse_func(4, 5.6, 7, 8.9);     // o.k.
ellipse_func(4, 5.6);             // o.k.
ellipse_func(4);                  // Fehler!!!
```

- `const char * format` $\rightsquigarrow$ Array-Elemente (String) konstant
  `char * const format` $\rightsquigarrow$ Pointer konstant

Verwendung des Rückgabewerts von `scanf`

Abfangen von Typfehlern bei Benutzereingaben

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

int main() {
    float n;
    printf("Geben Sie eine Zahl ein: ");
    int status = scanf("%f", &n);
    if (status == 0) {
        printf("Sie haben keine Zahl eingegeben.\n\n");
        exit(EXIT_FAILURE);
    }
    printf("Die Zahl ist %f.\n", n);
    return 0;
}
```

Vermeidung von Überläufen bei Stringeingaben

- Formatelement %ns zum Einlesen eines Strings einer Länge $\leq n$

- längere Strings werden abgeschnitten
  $\rightsquigarrow$ kein Überschreiben von Speicherbereichen außerhalb eines char-Arrays

```
char str[20];
scanf("%19s", str); // bei Eingabe von mehr als 19 Zeichen
                    // verbleiben ueberzaehlige Zeichen
                    // im Buffer von scanf
```

Standardfunktionen zur Eingabe aus `stdio` (3)

- `int getchar ();`
  - liest einzelne Zeichen aus dem Eingabestrom `stdin`
  - liefert ein `unsigned char`, das in `int` konvertiert wird
  - zeilengepuffert (wartet auf `RETURN`)

```
#define LEN 40
...
char str[LEN];
int char_in;
int i = 0;

while(i < LEN && (char_in = getchar()) != '\n')
   str[i++] = (char) char_in;
```

Anwendung: Leeren des Eingabepuffers von `scanf`

Problem: Fehlerhafte Eingabe für `scanf` verbleibt im Eingabepuffer

$\rightsquigarrow$ nächster Aufruf von `scanf` beginnt dort zu lesen

```
int c, status, zahl;
status = scanf("%d", &zahl);
if (status == 0)
   do
      c = getchar();
   while (c != '\n');
```

(*Wichtig für Fehlerbehandlung mit Recovering*)

Standardfunktionen zur Ausgabe aus `stdio`

- `int printf (const char * format, ...);`
- `int puts (const char * s);`
  - schreibt übergebenen String `s` nach `stdout`
  - kopiert das Nullzeichen *nicht* mit
  - fügt ein `'\n'` an
- geben die Länge der ausgegebenen Strings zurück

Übergabe von Strings als Parameter

- **Übergabe** eindimensionaler Arrays an Funktionen:
  formale Parameter als
  - offenes Array oder
  - Pointer auf den Komponententyp
- Anwendung bei Übergabe von Zeichenketten (char-Array)

```
int lengthOfString(char * ar) {
   int i = 0;
   while (ar[i] != '\0') {
      i++;
   }
   return i;
}
```

```
int main() {
   char *s = "Hallo Du!";
   int n = lengthOfString(s);
   ...
}
```

Standardfunktionen zur Stringverarbeitung

- in Header-Datei `<string.h>`:
-

```
size_t strlen (const char * s);                          Länge
char * strcpy (char * dest, const char * src);           Kopieren
char * strcat (char * dest, const char * src);           Anhängen
int strcmp (const char * s1, const char * s2);           Vergleichen
int strncmp (const char * s1, const char * s2,
                                        size_t n);       Vergleichen
```

  (0 bei Gleichheit)
- Nullzeichen `'\0'` entscheidend für korrektes Arbeiten
- `size_t` vordefinierter Datentyp als Rückgabetyp des `sizeof`-Operator (ist meist `unsigned int` oder `unsigned long`)

Warum Stringfunktionen wie `strcpy` ?

- Aufgabe: Kopieren von String `src` in String `dest`
- naives Herangehen: `dest = src;`

  $\rightsquigarrow$ Was passiert?
- Übergabe des Pointers

  $\rightsquigarrow$ Jede Änderung an `dest` auch in `src` und umgekehrt
- `strcpy` ändert keinen Pointer,
  sondern kopiert den Inhalt von `src` an die Stelle `dest`

  $\rightsquigarrow$ Verdopplung des Strings im Speicher

Vergleichen mit `strcmp` und `strncmp`

- `int strcmp (const char * s1, const char * s2)`
  - zeichenweiser Vergleich bis Unterschied oder `'\0'`
  - Rückgabewert ist
    - `<` `0` wenn erster String lexikographisch kleiner
    - `>` `0` wenn erster String lexikographisch größer
    - `0` bei Gleichheit

- `int strncmp (const char * s1, const char * s2, size_t n)`
  - wie `strcmp` mit zusätzlichem Abbruchkriterium
  - Abbruch, wenn Unterschied, `'\0'` oder `n` Zeichen verglichen

Funktionen zur Speicherbearbeitung

- ähnliche Funktionen zu den Stringfunktionen für beliebige Speicherobjekte
- Funktionsbezeichner beginnen mit `mem` statt mit `str`
  (z.B. `memcpy`, `memcmp` etc.)
- formale Parameter `void *` statt `char *`
- verarbeiten die übergebenen Speicherobjekte byteweise
- keine Prüfung/Verwendung des `'\0'`-Zeichens
- haben Anzahl der zu bearbeitenden Bytes als weiteren Parameter
- `#include <string.h>`

Funktionen zur Speicherbearbeitung (2)

- `void * memcpy(void * dest, const void * src, size_t n);`
  kopiert `n` Bytes aus Speicherplatz `scr` in Speicherplatz `dest`
  $\rightsquigarrow$ *Vorsicht bei überlappendem Speicherbereich!*

- `void * memmove(void * dest, const void * src, size_t n);`
  wie `memcpy`, schützt vor Fehlern durch überlappenden Speicherbereich
  $\rightsquigarrow$ kopiert zuächst in Zwischenpuffer, bevor auf `dest` geschrieben wird

- `int memcmp(const void * s1, const void * s2, size_t n);`
  byteweiser Vergleich, bis Unterschied oder `n` Bytes verglichen

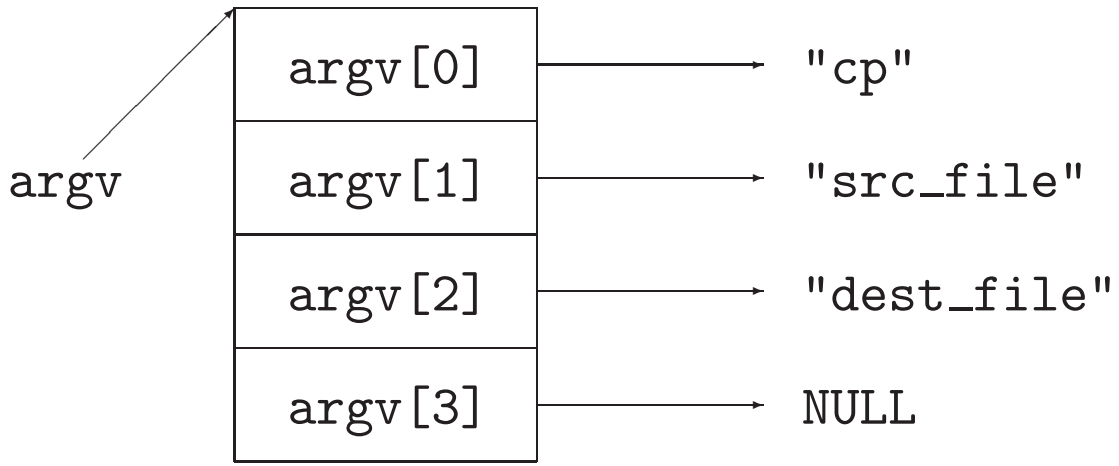
Funktionen zur Speicherbearbeitung (3)

- `void * memchr(const void * s, int c, size_t n);`
  durchsucht die ersten `n` Bytes des Speicherobjekts an `s` nach dem Wert `c` (interpretiert als `unsigned char`)
  $\rightsquigarrow$ gibt Pointer auf das erste Vorkommen von `c` oder `NULL` zurück

- `void * memset(void * s, int c, size_t n);`
  setzt die `n` Bytes ab Adresse `s` auf `c` (konvertiert in `unsigned char`)

Parameterübergabe beim Programmaufruf

- Beispiel: `cp src_file dest_file`
- zwei Varianten der `main`-Funktion:
  - `int main()` — parameterlos
  - `int main(int argc, char * argv[])` — zwei Parameter
- `argc` (**arg**ument **c**ounter): Anzahl der Argumente
- `argv` (**arg**ument **v**ector): Vektor (Array) der Argumente
  Argumente sind Strings $\rightsquigarrow$ Array von `char`-Arrays
  $\rightsquigarrow$ Array von Pointern auf `char`
- erstes Element von `argv` (`argv[0]`): Programmname ($\Longrightarrow$ `argc` ≥ 1)

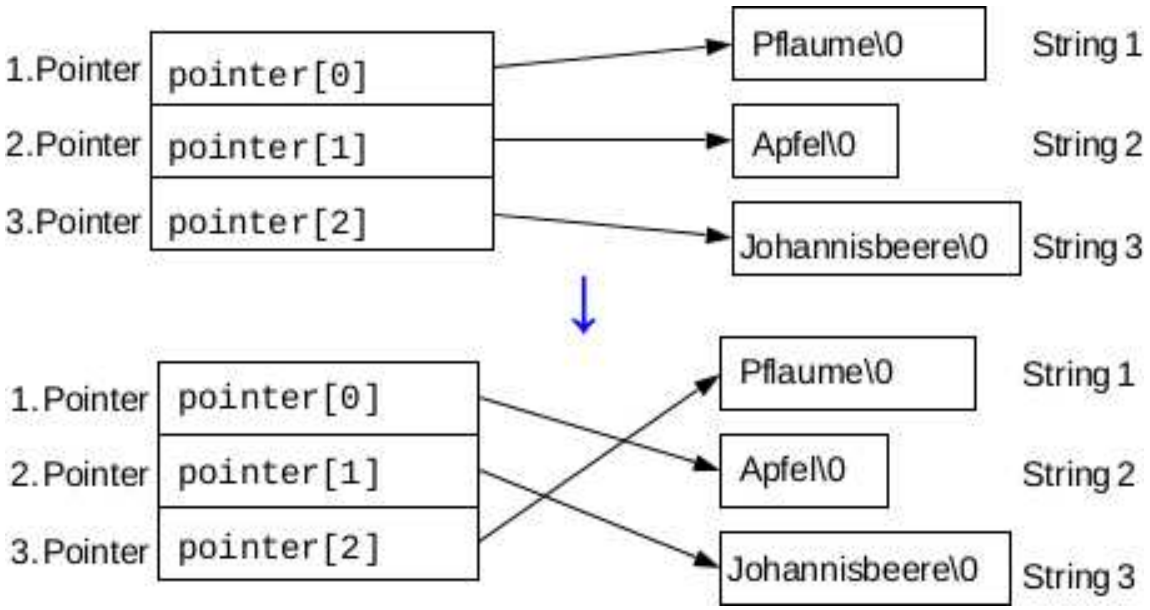


- Übergabe von Zahlen: Typumwandlung String $\longrightarrow$ Zahltyp erforderlich
- Standardfunktionen aus <stdlib.h>

```
double atof(const char * nptr);   ascii to float
int atoi(const char * nptr);      ascii to int
long atol(const char * nptr);     ascii to long
```

Arrays von Pointern

- z.B. in `int main(int argc, char * argv[])`
- erlaubt z.B. Sortieren von Strings ohne Kopieraktionen



Arrays von Pointern *versus* mehrdimensionale Arrays

- mehrdimensionale Arrays: Anzahl der Elemente für jede Dimension fest:

  `int matrix [6][10];` $\rightsquigarrow$ Array mit 60 `int`-Werten

- Array von Pointern: nur die Anzahl der Pointer ist fest:

  `int * pointer_array [6]` $\rightsquigarrow$ 6 Pointer auf `int`

  $\rightsquigarrow$ „zweite Dimension“ nicht festgelegt

- häufigste Anwendung für Datentyp `char`

  $\rightsquigarrow$ Array von Strings unterschiedlicher Länge

Pointer auf Pointer als formale Parameter

- `char * stringArray[]` ausdrückbar als `char * * stringArray`

- beim Aufruf: Übergabe eines Stringarrays
  ⇝ Übergabe der Adresse des ersten Strings im Array
  ⇝ Übergabe der Adresse des ersten Zeichens der ersten Komponente

- z.B. Ausgabe aller Strings in einem Array `ar` mit 36 Strings als Text:

```
void textausgabe(char * * stringArray, int anzahl) {
      int i;
      for (i = 0; i < anzahl; i++)
            printf("%s ", stringArray[i]);
}
```

Aufruf: `textausgabe(&ar[0],36);`

Pointer auf Pointer als formale Parameter (2)

Nach Übergabe von `&ar[0]` an `char * * stringArray`:

- `*stringArray` ist Pointer `ar[0]` (Pointer auf `char`)
- `**stringArray` ist das erste Zeichen des Strings in `ar[0]`
- `stringArray++` verschiebt `stringArray` auf `ar[1]`

```
void textausgabe(char * * stringArray, int anzahl) {
      while(anzahl-- > 0)
            printf("%s ", *stringArray++);  // dereferenzieren,
}                                           // dann Nebeneffekt (Postfix)
```

Praxis der Programmierung

Zusammengesetzte Datentypen, Dynamische Speicherverwaltung

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Zusammengesetzte Datentypen

Zusammengestzte Datentypen – Motivation

- Aufgabe: Kundenverzeichnis anlegen
  Für jeden Kunden erfassen:
  - Name,
  - Adresse,
  - Kundennummer,
  - Umsatz,
  - ...

- Wie kann man das in C umsetzen?

Zusammengesetzte Datentypen – Motivation (2)

- Array für jeden Kunden? $\rightsquigarrow$ ungünstig wegen der versch. Datentypen
- Array für jedes Merkmal?

```
short id[1024];
char* name[1024];
char* address[1024];
float b_volume[1024];
...
```

$\rightsquigarrow$ Zusammengehörgkeit der Variablen nicht sichergestellt

$\rightsquigarrow$ Übergabe an Funktionen nur einzeln

- **Strukturen (structs)** zur Definition komplexer Datentypen, die sich aus mehreren Datentypen zusammensetzen

Stukturen

- **Strukturtyp:** - selbst definierter Datentyp
  - zusammengesetzt aus Komponenten *verschiedener Typen*

- Variable von Typ Struktur (Verbund/Record) kann Datensatz speichern

Kundennr.	Nachname	Vorname	Straße	Hausnr.	PLZ	Wohnort	Umsatz
`short`	`char[64]`	`char[64]`	`char[64]`	`short`	`char[8]`	`char[64]`	`float`

- *Komponenten* haben einen eigenen Namen (statt Index) und einen Typ

Definition eines Strukturtyps

- ```
  struct name {
    komponententyp_1 komponentenname_1;
    komponententyp_2 komponentenname_2;
                          ⋮
    komponententyp_n komponentenname_n;
  };
  ```
- `struct` ist Schlüsselwort $\rightsquigarrow$ **Datentyp:** `struct name`
- Anzahl der Komponenten bei Definition festgelegt
- Semikolon nach } (kein Anweisungsblock!)
  ```

Definition eines Strukturtyps – Beispiel

```
short id[1024];
char* name[1024];
char* address[1024];
float b_volume[1024];
```

```
struct customer {
   short id;
   char name[64];
   char address[64];
   float b_volume;
};

struct customer meyer;
struct customer ctms[1024];
```

Strukturtypen – Warum?

Strukturtypen

- definieren immer einen Datentyp
- bilden die Zusammengehörigkeit von Daten ab
- bilden die Realität direkter ab

  ⇝ bessere Modellierung von Anwendungsdomänen
  ⇝ erleichtertes Code-Verständnis
  ⇝ erleichterte Wartung des Codes

Strukturvariablen

- **Definition:** `struct name variablenname;`
  $\rightsquigarrow$ Variable vom zusammengesetzten Typ `struct name`
- besteht aus mehreren Komponentenvariablen
  $\rightsquigarrow$ bei Stukturtypdefinition vereinbart
- gleichzeitige Vereinbarung von Strukturtyp und -variablen möglich:

```
struct point {          // Def. neuer Datentyp (Strukturtyp)
  float x;
  float y;              // zwei Komponenten (Member) x und y
} pt1;

struct point pt2, pt3;
```

$\rightsquigarrow$ `pt1, pt2, pt3` als (Struktur-) Variablen dieses neuen Datentyps definiert

Zugriff auf Komponentenvariablen

1. **Punktoperator:** *Strukturvariable*.*Komponentenvariable*

   - Beispiele:
     ```
     pt1.x
     meyer.name
     meyer.address
     ```

   - lesender und schreibender Zugriff:

     ```
     strcpy(meyer.name, "Meyer, Jens");
     strcpy(meyer.address, "14476 Potsdam, An der Bahn 2");
     printf("%s\n%s\n", meyer.name, meyer.address);
     ```

2. **Pfeiloperator:** *Pointer_auf_Strukturvariable*->*Komponentenvariable*

   - Beispiel: `(&pt1)->x`
     `(&meyer)->name`

   - Pfeil: Minuszeichen und Größerzeichen

   - lesender und schreibender Zugriff:
     ```
     printf("Adresse: %s\n", (&meyer)->address);
     ```

   - Wie kann die Adresse in ihre Komponenten zerlegt werden?
     $\rightsquigarrow$ Strukturen als Komponenten von Strukturen

Strukturen als Komponenten von Strukturen

```
struct address {
   char street[64];
   short number;
   char zip_code[8];
   char city[64];
};

struct customer {
   char name[64];
   struct address adr;
   ...
};
```

```
struct customer meyer;

strcpy(meyer.name, "Meyer, Jens");
strcpy(meyer.adr.street,
                    "An der Bahn");
meyer.adr.number = 2;
```

Initialisierung von Strukturvariablen

1. mit Punkt- oder Pfeiloperator
2. mit Initialisierungslisten
   - nur direkt bei der Definition der Strukturvariablen
   - wie bei Arrays (mit Ausdrücken passenden Typs), z.B.:

```
struct customer meyer = {
    "Meyer, Jens",
    { "An der Bahn", 2, "14476", "Potsdam" },
    ...
};
```

3. Zuweisung von struct-Variablen:

```
struct customer krause = meyer;  // Kopie aller WERTE !!!
```

Strukturen als Parameter und Rückgabewerte von Funktionen

- Voraussetzung: Definition des Strukturtyps *außerhalb* und *vor* den Funktionen
- Übergabe wie einfache Datentypen (call-by-value beachten!):

```
void print_customer (struct customer c) {
   printf("%s\n", c.name);
   printf("%s %d\n", c.adr.street, c.adr.number);
   printf("%s %s\n", c.adr.zip_code, c.adr.city);
}

struct point new_point () {
   struct point pt = {0, 0};
   return pt;
}
```

$\rightsquigarrow$ oft unnötiges Kopieren großer Datenmengen im Speicher

Pointer auf Strukturen als Parameter

- Simulation von call-by-reference durch Übergabe von Pointern:

```
void print_customer (struct customer * c) {
   printf("%s\n", c->name);
   printf("%s %d\n", c->adr.street, c->adr.number);
   printf("%s %s\n", c->adr.zip_code, c->adr.city);
}

int main() {
   struct customer meyer = { ... };
   print_customer(&meyer);
}
```

- Veränderung der Werte des Originals (c) möglich
- Gefahr: Parameter kann nicht oder mit `NULL` initialisiert sein (*Laufzeitfehler*)

Vereinbarung eigener Typnamen

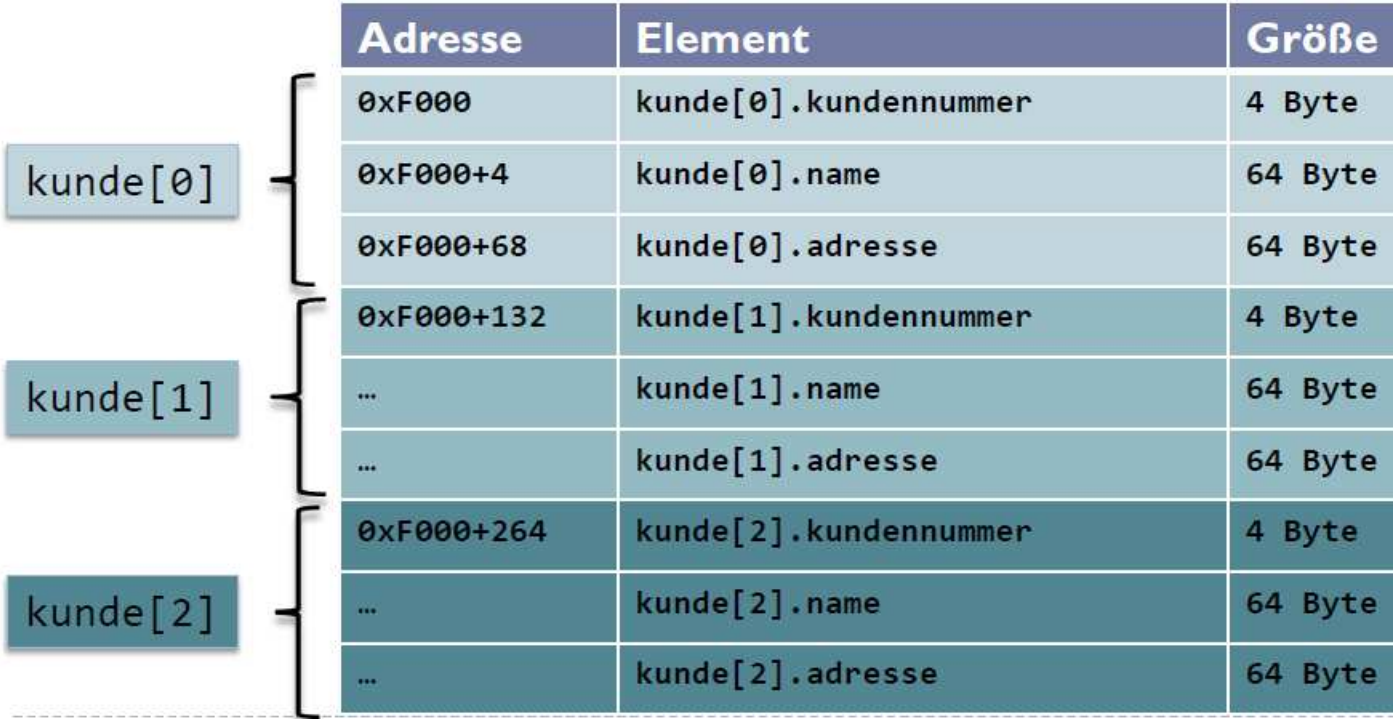
- Vereinbarung eines *Aliasnamens* für bereits definierte Datentypen
- `typedef` *Datentyp Aliasname*`;`
- `typedef int integer;`
- Anwendung:
  - Vereinfachung von Typnamen
    - `typedef struct customer Customer;`
      `Customer meyer;`
    - `typedef unsigned long long ULL`
  - Vorbereitung Portierung maschinenabhängiger Datentypen
    - `typedef int INT;` $\rightsquigarrow$ `typedef short INT;`

Strukturen im Speicher

Elemente von Strukturvariablen liegen hintereinander im Speicher, z.B.:

```
struct  kt {
    int kundennummer;
    char name[64];
    char adresse[64];
};

struct kt kunde[3];
```



Adresse	Element	Größe
0xF000	kunde[0].kundennummer	4 Byte
0xF000+4	kunde[0].name	64 Byte
0xF000+68	kunde[0].adresse	64 Byte
0xF000+132	kunde[1].kundennummer	4 Byte
…	kunde[1].name	64 Byte
…	kunde[1].adresse	64 Byte
0xF000+264	kunde[2].kundennummer	4 Byte
…	kunde[2].name	64 Byte
…	kunde[2].adresse	64 Byte

Dynamische Speicherverwaltung

Dynamische Speicherverwaltung – Motivation

Aufgabe: Kundenverzeichnis verwalten

- Zu jeder Zeit soll ein neuer Kunde hinzukommen oder entfernt werden können.
- Verzeichnis ist Liste von Kunden, deren Länge sich zur Laufzeit dynamisch ändert.
- Wie kann man das in C umsetzen?

Ausgangspunkt

zusammengesetzter Datentyp `Customer`

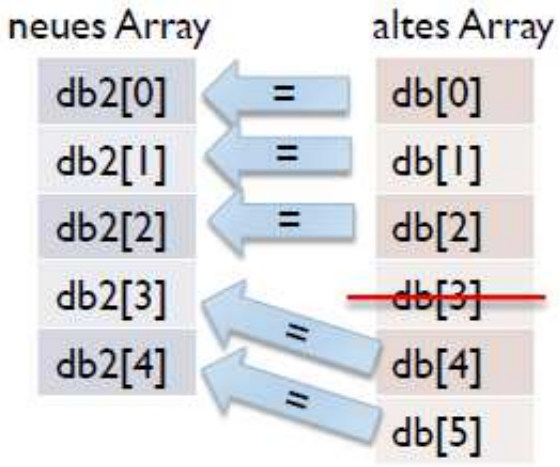
```
struct customer {
   short id;
   char name[64];
   char adress[64];
   float b_volume;
};

typedef struct customer Customer;

Customer ctms[1024];   // Groesse unveraenderlich
```

Kundenverwaltung mit Arrays

- Beispiel: Kunde löschen (aus Array???)
- erster Ansatz: – neues Array anlegen mit einem Element weniger
  – alles kopieren außer zu löschendes Element



- sehr viele Kopien entstehen; Indexzuordnung ändert sich ständig

Datentypen im Speicher

- *bisher:* Größe und Anzahl der Variablen bei Definition festlegen
  - Variablen elementarer Datentypen (auch Pointer-Variablen)
  - Strings, Arrays
  - Strukturen (structs)

- **Problem:** Größe von Arrays/Strings wird oft erst zur Laufzeit festgelegt
  $\rightsquigarrow$ immer maximale Größe annehmen ist oft kritische Speicherverschwendung

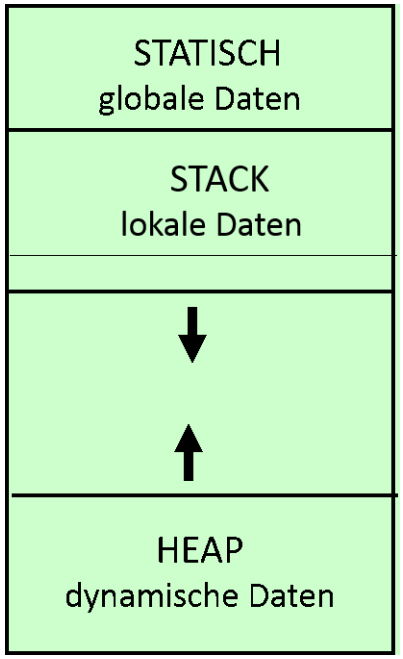
- **Lösung:** dynamische Speicherreservierung "on demand"

- **Nachteile:** komplexere Programmlogik, hohe Fehleranfälligkeit, möglicher Datenverlust

Speicherbereiche eines Programms

1. Statischer Speicherbereich
   - Codesegement für Maschinencode (kompilierte Anweisungen)
   - Datensegment für statische Daten (globale Variablen, statische Strings)
2. Programm-Stack (zur Compile-Zeit organisiert)
   - Variablen, Arrays, Funktionsparameter
   - temporäre Werte (bei Berechnungen)
   - Aufrufstack für Funktionen
3. Programm-Heap (zur Laufzeit verwaltet)
   - dynamisch als Pointer auf den Heap angefordert (in C: `malloc()`)
   - dynamisch wieder freigegeben (in C: `free()`)

Speicherbereiche eines Programms



Bibliotheksfunktionen zur dynamischen Speicherverwaltung

- in `<stdlib.h>`
- zum **Anfordern** von Heap-Speicher (liefert Pointer zurück), *z.B.:*

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

- *Beispiel:* ganze Zahlen von 1 bis N in einem Array speichern

```
int N, i;
int * nums;
scanf("%d", &N);
nums = malloc(N * sizeof(int));
for(i = 0; i < N; ++i) {
   nums[i] = i+1;
}
```

Bibliotheksfunktionen zur dynamischen Speicherverwaltung (2)

- **Freigeben** von reserviertem Speicher (Pointer auf Speicherbereich übergeben):

```
void free(void * pointer)
```

- *am Beispiel:*

```
free(nums);   // Speicherbereich zurueckgegeben
nums = NULL;  // nums zeigt nicht mehr auf den Speicherbereich
```

- keine automatische Speicherfreigabe

⇝ Zu jedem `malloc()` gehört (irgendwann) genau ein `free()`.

Probleme mit dynamischer Speicherverwaltung

- Programmierer ist voll verantwortlich
- keine Freigabe $\rightsquigarrow$ Speicher läuft voll mit Daten,
  auf die nicht mehr zugegriffen werden kann / die nicht mehr gebraucht werden
- Freigabe eines nicht (mehr) allokierten Bereichs
  $\rightsquigarrow$ Programmabsturz
- „dangling pointers“, wenn Pointer auf freigegebenen Bereich zeigen
- korrekte/geeignete Stelle zur Speicherfreigabe oft nicht klar:

```
char * create_string(int n) {  // n ist Laenge des Strings
      char * str = malloc((n+1)*sizeof(char));
      return str;
      free(str); // Problem???
}
```

Dynamisch gespeicherte Daten als Rückgabewerte

```
 char * create_string(int n) {          // n ist Laenge des Strings
        char * str;
        str = malloc((n+1)*sizeof(char));
        return str;
}

void delete_string(char * s) {
        if (s != NULL) {
                free(s);
            //  s = NULL; hier nutzlos - warum?!
        }
}
```

Zu jedem Aufruf von `create_string` gehört genau ein Aufruf von `delete_string`, gefolgt von Zuweisung des `NULL`-Pointers!

Dynamische Datentypen

... Zurück zur Problemstellung ...

Aufgabe: Kundenverzeichnis verwalten

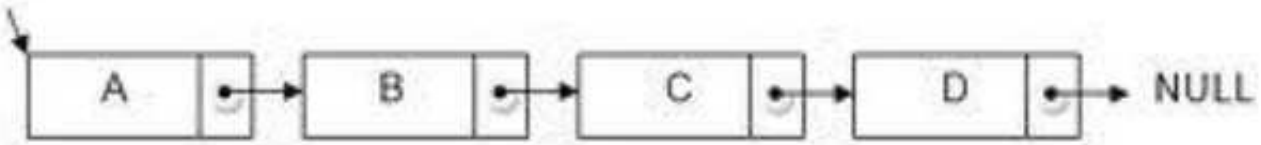
- Zu jeder Zeit soll ein neuer Kunde hinzukommen oder entfernt werden können.
- Verzeichnis ist Liste von Kunden, deren Länge sich zur Laufzeit dynamisch ändert.
- Wie kann man das in C umsetzen?

Einfach verkettete Listen

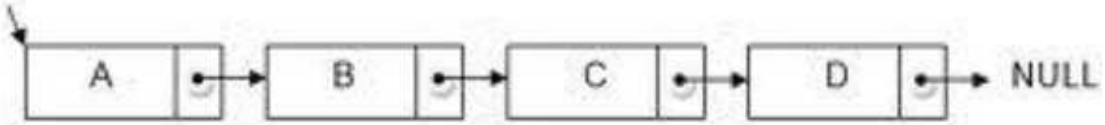
- neuer Ansatz: Nutzung dynamischer Datenstrukturen
  Beispiele: verkettete Listen, Bäume, Mengen, ...

- in fast allen modernen Sprachen vordefiniert (Standard-Bibliotheken);
  in C leider keine eingebaute verkettete Liste

- Konzept einfach verkettete Liste:
  neben den Daten wird ein Pointer auf das nächste Listenelement gespeichert



Kundenverwaltung mit einfach verketteten Listen



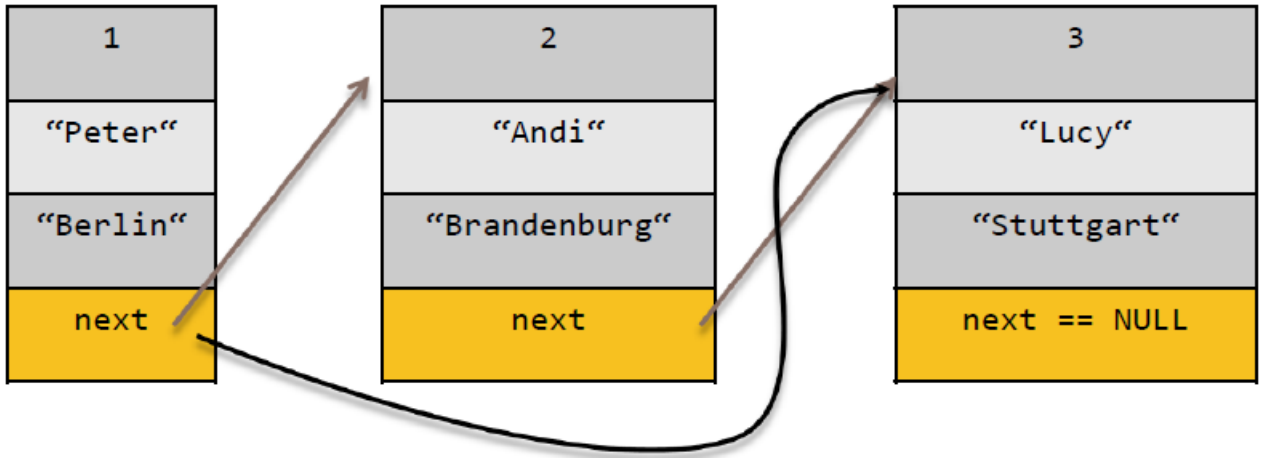
- zusätzliche Membervariable vom Typ Pointer auf Kunde (`Customer`)

```
struct customer {
    short id;
    char name[64];
    char adress[64];
    float b_volume;
    struct customer * next;
};
```

- NULL-Pointer zum Auffinden des Listenendes
- Elemente liegen *nicht* hintereinander im Speicher (*Warum nicht?*)

Arbeit mit einfach verketteten Listen

- Einfügen Löschen, Umordnen durch Umsetzen der Pointer,
  z.B. Löschen eines Listenelements ("Andi"):



kein Kopieren der Daten

- Suchen von Elementen, Bestimmen der Länge, Ausgabe aller Elemente usw.
  durch Verfolgung der Pointer

Einfach verkettete Liste ganzer Zahlen

Wir benötigen eine C-Datei mit

1. einem Typ `listelement` zum Speichern eines `int` und der Adresse des nächsten Listenelements,
2. einem Typ `list` zum Adressieren einer Liste aus solchen Elementen.

   $\rightsquigarrow$ *Wie kann dieser Typ realisiert werden?*

```
struct le {
   int value;
   struct le * next;
};
typedef struct le listelement;


typedef listelement * list;   // Pointer auf das erste Element
```

Operationen für einfach verkettete Listen

Wir benötigen je eine Funktion

1. zum Einfügen eines Listenelements als neues erstes Element,

2. zum Ausgeben aller Elemente einer Liste.

```
void insert(int v, list * l) {
   listelement * new;
   new = malloc(sizeof(listelement));
   new->value = v;
   new->next = *l;
   *l = new;
}
```

```
void print_list(list l) {
   if (l == NULL) printf("leer");
   else
      while (l != NULL) {
         printf("%d ", l->value);
         l = l->next;
      }
   }
```

Operationen für einfach verkettete Listen (2)

Wir benötigen außerdem je eine Funktion

1. zum Berechnen der Listenlänge (Anzahl der Elemente),
2. zum Entfernen des ersten Listenelements. *Denken Sie an Speicherfreigaben!*

```
int length(list l) {

    int count = 0;
    while (l != NULL) {
         count++;
         l = l->next;
    }
    return count;
}
```

```
int delete_head(list * l) {

    if (*l == NULL) return -1;
    list old = *l;
    *l = old->next;
    free(old);
    return 0;
}
```

Operationen für einfach verkettete Listen (3)

Nun geht es um eine Funktion zum Löschen der gesamten Liste.
Denken Sie wieder an die Freigabe des Speichers!

```
// nach Aufruf immer l = NULL
void delete_all(list l) {
   list next;
   while (l != NULL) {
       next = l->next;
       free(l);
       l = next;
   }
}
```

```
// alternative, sichere Impl.:
void delete_all(list * l) {
    list next;
    while (*l != NULL) {
       next = (*l)->next;
       free(*l);
       *l = next;
    }
    // *l ist jetzt NULL
}
```

Praxis der Programmierung

Präprozessor, Header-Dateien, Speicherklassen, Ausblick: Was es in C noch gibt.

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Präprozessoranweisungen

Präprozessoranweisungen (1)

- Ausführung vor der eigentlichen Kompilation
- `#Direktive Text`
- *bisher:*
  - Einfügen von Dateiinhalten mit `#include datei`

    `#include <stdio.h>`
  - Vereinbarung symbolischer Konstanten und Makros mit

    `#define Bezeichner Ersatztext`

    `#define double_inc(a,b) a++; b++;`

Präprozessoranweisungen (2)

- Makros mit Parametern
  - `#define Bezeichner(Parameter1, ...,ParameterN) Ersatztext`
  - formale Parameter ohne Typ (flexibel, aber unsicher!)
  - Einsetzung des Ersatztextes mit aktuellen Parametern

```
#define sum(n) n*(n+1)/2
...
int a = 9;
int s = sum(a);   // ersetzt zu int s = a*(a+1)/2
...               // danach hat s den Wert 45
```

- Aufheben von Makrovereinbarungen mit `#undef Bezeichner`

Präprozessoranweisungen (3)

- bedingte Kompilierung
  - Präprozessor entscheidet anhand von Ausdrücken und Symbolen, welche Codeteile übersetzt werden sollen
  - Direktiven `#if #elif #else ...`
  - Einsatz z.B. um auf Compilerversionen reagieren zu können oder Programmversionen in einer Datei zu halten (*Write once!*)
- eigene Fehlermeldungen während des Kompilierens generieren
- ...

Bedingte Kompilierung — Beispiel

```
#include <stdio.h>
#include <limits.h>
#define TESTVERSION 1

int main() {
   #if INT_MAX > 32767   // max. int-Wert, in limits.h definiert
      int i;
   #else
      long i;
   #endif

   #if TESTVERSION
      printf("sizeof int: %d\n", sizeof (int));
   #endif
   ...
}
```

Bedingte Kompilierung — Alternative

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

int main() {
   #if INT_MAX > 32767   // max. int-Wert, in limits.h definiert
      int i;
   #else
      long i;
   #endif

   #if defined TESTVERSION
      printf("sizeof int: %d\n", sizeof (int));
   #endif
   ...
}
```

Header-Dateien

Definition von Header-Dateien

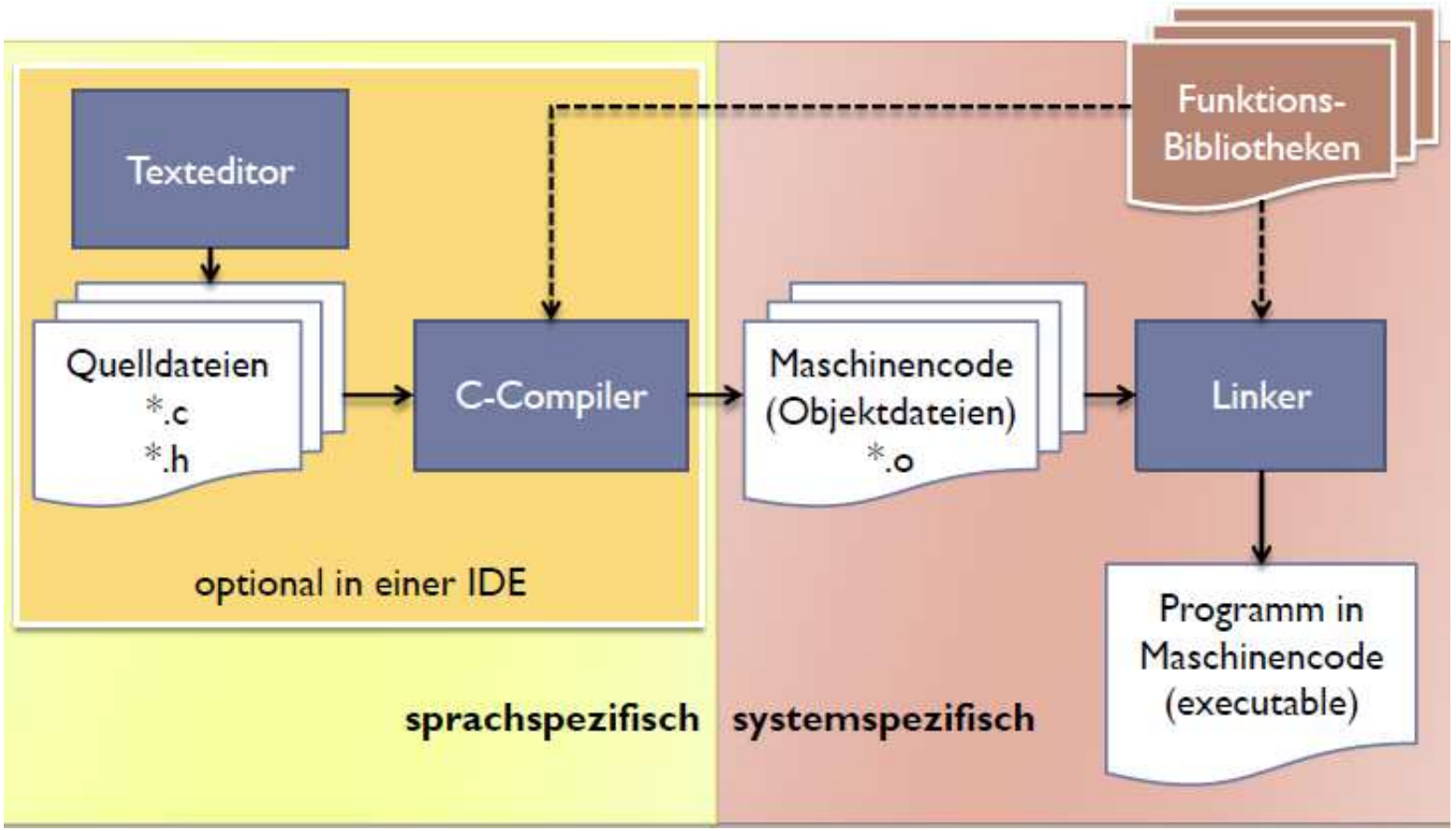
- *dateiname*`.h`
- Header-Dateien enthalten häufig Deklarationen (Signaturen) und die Definition von Makros
- Die Definitionen (Implementierungen) sind dann meist in `.c`-Dateien enthalten.
- Beispiel Vorwärtsdeklaration:

```
void g(int n);
void f(int n) { g(n-1); }
void g(int n) { f(n-1); }
```

- mit Header-Datei:

```
#include "bsp.h"              // enthaelt Deklaration void g(int n);
void f(int n) { g(n-1); }
void g(int n) { f(n-1); }
```

Einbinden der Header-Dateien (1)



Einbinden der Header-Dateien (2)

- Präprozessor setzt diese an der Stelle der `#include`-Direktive ein
  - eigene Header-Dateien: `#include "---.h"`
  - aus der C-Bibliothek: `#include <---.h>`

  $\rightsquigarrow$ Funktionen sind dem Compiler bekannt
- Implementierung erst zur Laufzeit bedeutsam, wenn die Funktion aufgerufen wird
- Linker fügt die Objektcode-Dateien zusammen
  - ggf. verschiedene C-Dateien und Bibliotheksdateien
  - Objekcode-Dateien verwenden relative/virtuelle Speicheradressen
  - Linker sorgt für einheitlichen Adressraum des ausführbaren Programms (Zuordnung eines virtuellen, überschneidungsfreien Adressraums)

Beispiel zur Quelltextaufteilung

```
// date.h

struct date {
   int day, month, year;
};
void setDate(struct date *d);
```

```
// highscore.h
#include "date.h"
struct highscore {
   int score;
   struct date d;
};
void setScore(struct highscore * h);
```

```
// date.c
#include "date.h"
 ...
```

```
// highscore.c
#include "date.h"
#include "highscore.h"
 ...
```

```
>  gcc date.c -c          // liefert date.o
>  gcc highscore.c -c     // Compilerfehler! (WARUM???)
```

Vermeiden von Doppel-Definitionen mit dem Präprozessor

```
#ifndef _DATE_h      // nur, falls das Makro _DATE_h nicht definiert ist
#define _DATE_h      // wird dieses Makro
struct date {        // und struct date definiert
   int day, month, year;
};
...
#endif               // sonst wird alles bis hier uebersprungen
```

Analog:

```
#ifndef _HIGHSCORE_h
#define _HIGHSCORE_h
struct highscore { ... };
#endif
```

Beispiel zur Quelltextaufteilung

```
// date.h, implementiert in date.c
#ifndef _DATE_H
#define _DATE_H

struct date {
   ...
#endif
```

```
// highscore.h, impl. in highscore.c
#include "date.h"
#ifndef _HIGHSCORE_H
#define _HIGHSCORE_H
struct highscore {
   ...
#endif
```

Applikation `scores.c` (mit `main`) nutzt `struct highscore` und `struct date`:

```
// scores.c
#include "date.h"
#include "highscore.h"
...
```

```
> gcc date.c highscore.c scores.c -o scores
```

Speicherklassen

Konzept der Speicherklassen

- Modifikation der Gültigkeit von Variablenvereinbarungen mit Hilfe von Schlüsselwörtern `auto`, `extern`, `static`, `register`
- ergänzen die Unterscheidung von lokalen und globalen Variablen

  lokal: Definition innerhalb eines Funktionsblocks

  – Verwaltung in den Stackframes zu Aufrufen der Funktionen
  – Verwendbarkeit nur, solange der Stackframe exisitiert

  global: Definition außerhalb aller Funktionsblöcke

  – gültig während des gesamten Programmlaufs
  – können von allen Funktionen verwendet werden

Speicherklassen `auto` und `extern`

- **auto**
  - Default-Speicherklasse für lokale Variablen ($\rightsquigarrow$ Stack)
  - Verwendung, um auf Gültigkeitsbereich im Quellcode expilizit hinzuweisen

- **extern**
  - bewirkt reines *Deklarieren* globaler Variablen:
    Name wird bekannt gegeben, aber noch kein Speicherplatz reserviert
  - Definition kann an anderer Stelle, sogar in anderer Datei erfolgen
  - `extern`-Deklaration referenziert Variablendefinition; Adresse durch Linker

Beispiel externes Deklarieren

- Schlüsselwort `extern` zeigt an, dass die Definition an anderer Stelle in derselben oder einer anderen Datei erfolgt

```
extern int value;   // Definition erfolgt an anderer Stelle
...
int value = 2014;   // muss vollstaendige Definition sein
```

- vorwärtsdeklarierte Funktionen sind implizit `extern`

```
void g(int n);              // steht fuer extern void g(int n);
void f(int n) { g(n-1); }
void g(int n) { f(n-1); }
```

Speicherklassen `register` und `static`

- **register**
  - zur Definition spezieller `auto`-Variablen
  - „Bitte“ an Laufzeitsystem, schnellen Zugriff durch Anlegen in *Registern* des Prozessors zu ermöglichen (z.B. bei sehr häufigem Zugriff)
  - bevorzugte Behandlung kann nicht garantiert werden

- **static**
  - *statische* lokale Variablen bis zum Ende des Programmlaufs gültig
  - ist aber nur in dem Block sichtbar/verwendbar, in dem sie definiert wurde
  - geeignet für Funktionen, die Informationen aus dem letzten Aufruf für den nächsten Aufruf bereitstellen wollen

Beispiel `static`

```
int globvar = 1;

void func() {
    static int statvar = 1;
    int locvar = 1;
    // alle 3 Variablen ausgeben
    statvar++;
    globvar++;
    locvar++;
}
```

```
int main() {
    func(); func(); func();
    // locvar  = 8; -> Compilerfehler!!!
    // statvar = 8; -> Compilerfehler!!!
    globvar = 8;   // o.k.
    func(); func();
}
```

1. Aufruf `func()`: statisch: 1 global: 1 lokal: 1
2. Aufruf `func()`: statisch: 2 global: 2 lokal: 1
3. Aufruf `func()`: statisch: 3 global: 3 lokal: 1
4. Aufruf `func()`: statisch: 4 global: 8 lokal: 1
5. Aufruf `func()`: statisch: 5 global: 9 lokal: 1

Weitere Operatoren

Der Bedingungsoperator

`A ? B : C`

1. Auswertung von `A`
2. Wenn Wert von `A` ungleich `0`, Auswertung von `B` (= Rückgabewert);
3. sonst Auswertung von `C` (= Rückgabewert)
4. Rückgabetyp ist der „höhere“ Typ von `B` oder `C`

entspricht

```
if (A) return B;
else return C;
```

Besipiel: `(3>4) ? 5.0 : 6  // gibt 6.0 zurueck`

Bitoperatoren

UND-Operator	&
ODER-Operator	|
Exklusives ODER	^
Negationsoperator	~
Rechtsshift-Operator	>>
Linksshift-Operator	<<

- Die logischen Operatoren beziehen sich immer auf alle Bits ihrer Operanden.
- Sind anwendbar auf ganzzahlige Operanden.
- Vorsicht bei Vorzeichen-behafteten Operanden.

Logische Bit-Operatoren

```
0 & 1        //  0 & 1 = 0
14 & 1       //  1110 & 0001 = 0000

0 | 1        //  0 | 1 = 1
14 | 1       //  1110 | 0001 = 1111

0 ^ 1        //  0 ^ 1 = 1
14 ^ 1       //  1110 ^ 0001 = 1111 / Invertieren
14 ^ 3       //  1110 ^ 0011 = 1101 / von Bits!

unsigned char a, b;
a = 9;       //  a = 0000 1001
b = ~a;      //  b = 1111 0110
```

Shift-Operatoren

`A << B`

Verschieben der Bits von A um B Bitstellen nach links;
Auffüllen der nachrückenden Bits mit Nullen (Multiplikation mit 2^B)

```
unsigned char a;
a = 8;      //  0000 1000
a = a << 3; //  0100 0000
```

`A >> B`

Verschieben der Bits von A um B Bitstellen nach rechts;
Auffüllen der vorderen Bits mit Nullen (Division durch 2^B)

```
unsigned char a;
a = 8;      //  0000 1000
a = a >> 3; //  0000 0001
```

Unionen und Bitfelder

Unionen

- **Deklaration:** wie Strukturen, mit `union` statt `struct`
- Members (Komponenten) stellen Alternativen dar;
  zu jedem Zeitpunkt ist nur eine Komponente gespeichert
- **Zugriffe** mit Punkt- bzw. Pfeilschreibweise
- **Beispiel:** Repräsentation eines Punktes entweder
  mit karthesischen oder Radial-/Polarkoordinaten

Unionen: Beispiel

```
struct cartPoint {
    double x, y;
};

struct radPoint {
    double radius, phi
};

struct point {
    enum pointType ptype;
    union coordinates c;
};
```

```
union coordinates {
    struct cartPoint cpt;
    struct radPoint rpt;
};

enum pointType {
    CART, RAD
};
```

Bitfelder

- bestehen aus vorgegebener Anzahl von Bits
- Bitmuster werden als Werte des Ganzzahltyps interpretiert
- **Deklaration** als Member von unions oder structs: `typ name:bitanzahl;`

```
unsigned a:4;   //  Werte von 0 bis 15
a = 3;          //  0011
a = 19;         //  0011 (Ueberlauf!!!)
signed b:4;     //  Werte -8 bis +7
b = 9;          //  1001 (ergibt -8 + 0 + 0 + 1 = -7)
```

- plattformabhängige Interpretation
- für hardwarenahe und Graphikprogrammierung eingesetzt

Die Standardbibliothek

Bibliotheksfunktionen (1)

- **Funktionen zur Ein- und Ausgabe (stdio.h)**
  - Schreiben von Zeichen oder Strings auf `stdout`
  - Lesen von Zeichen oder Strings von `stdin`
  - Lesen oder Schreiben von bzw. in Streams/Dateien, ...
- **String- und Speicherbearbeitung (string.h)**
  - Vergleich, Kopieren zweier Strings
  - Suchen von Zeichen oder Teilstrings in Strings
  - Bestimmen der Länge eines Strings, ...
  - analoge Funktionen für Datenblöcke im Speicher

    z.B. `strcpy()` *vs.* `memcpy()`, ...

Bibliotheksfunktionen (2)

- **Mathematische Funktionen (math.h)**

  `sin(), cos(), log(), log10(), floor(), pow(), ...`

- **Zahlenkonvertierungen, Speicherverwaltung, Zufallszahlen (stdlib.h)**

  - `abs(), atof(), atoi(), atol(), ...`
  - `malloc(), realloc(), free(), ...`
  - `rand(), ...`

- **Klassifizierung und Konvertierung von Zeichen (ctype.h)**

  - Test ob ein Character alphanumerisch, ein Buchstabe, eine Zahl, ... ist
  - `tolower(), toupper()`

Bibliotheksfunktionen (3)

- **Festlegung länderspezifischer Zeichen und Darstellungen (locale.h)**
  (Datumsformate, Währungsformate, ...)

- **Datum und Uhrzeit (time.h)**
  (Kalenderzeit in sec seit Stichtag (meist 1.1.1970),
  Kalenderzeit in Greenwich-Zeit umrechnen, Differenz zwischen zwei Zeiten,
  Zeiten in String konvertieren, ...)

- einige mehr, versionsabhängig (s. Compiler-Handbuch)