

[LV-Nr. 23001]

# Imperative Programmierung

## Einführung in die Programmiersprache C

Prof. Thomas Kirste

Universität Rostock

Lehrstuhl Mobile Multimediale Informationssysteme

# Was ist Programmierung?

## Eine intuitive Annäherung

- Programmierung ist die Entwicklung eines systematischen Verfahrens, mit dem eine Maschine eine bestimmte Aufgabenstellung erfüllen kann.
  - Zum Beispiel: Aufgabenstellung: ein Auto von Rostock nach Wismar lenken.
  - Verfahren: Forschungsgegenstand ...
- Ein solches Verfahren nennt man auch *Algorithmus*
- Wir konzentrieren uns auf Fragestellungen, bei denen man Eingabe (Problemstellung) und Ausgabe (Ergebnis) als Zahlen darstellen kann.
  - Jede denkbare Information, auch Bilder, Videos, Musik, ..., lässt sich als Zahl codieren.
- Wir konzentrieren uns auf Maschinen, die in der Lage sind *alle* Probleme zu lösen, die sich als Berechnung mit Zahlen repräsentieren lassen
  - Dies sind die "universellen" Maschinen
  - Computer sind solche universellen Maschinen

# Was ist Programmierung?

## Eine intuitive Annäherung

- Programmierung ist der Gewinn von *Erkenntnis*
  - Vorher: Problem der unstrukturierten Wirklichkeit.
  - Nachher: Repräsentation der Wirklichkeit durch eine Struktur, die eindeutig das Verhalten der Wirklichkeit wiedergibt.
  - Also: Die Entwicklung einer mathematischen Theorie (eines Teils) der Wirklichkeit.
- Aufgabe: Versuchen Sie zu beschreiben, wie man Kupplung und Gaspedal so bedient, dass man den Motor nicht abwürgt – und zwar *ohne* unklare Begriffe wie „langsam“, „mit Gefühl“ usw. zu verwenden ...
- Programmierer haben die Fähigkeit, einer Maschine die Wirklichkeit zu erklären. Dazu müssen sie die Wirklichkeit besser verstehen als Nichtinformatiker.
- „Informatiker“, die nicht programmieren können, haben nicht verstanden, was der Beruf eines Informatikers ist.

# Was ist imperative Programmierung?

- Ein imperatives Programm ist eine Folge von elementaren Befehlen, die der Computer sequentiell abarbeitet.
- Im Kontrast dazu: deklarative Programmierung:
  - Funktionale Programmierung: ein Programm ist eine Menge von Gleichungen zwischen Ausdrücken, Berechnung erfolgt durch Ersetzen von Ausdrücke durch gleichwertige Ausdrücke.
  - Logische Programmierung: ein Programm ist eine Menge von logischen Klauseln, Berechnung erfolgt durch konstruktiven Beweis mit Hilfe der Programmklauseln.
- Sowie: Objektorientierte Programmierung: ein Programm ist eine Menge von Objekten (einfachen Prozessen), die sich gegenseitig Nachrichten senden und darauf reagieren.
  - Das interne Verhalten der Objekte wird üblicherweise imperativ definiert.
- Die imperative Programmierung bildet die technische Grundlage aller anderen Programmierungsformen. Alle Computer werden letztendlich in einer sehr einfachen imperativen Sprache – der Maschinensprache – programmiert.

# Was ist „Programmieren Lernen“?

## Versuch einer Analogie

- Programmiersprache = Auto
- Programmieren = Auto fahren
- Programmieren lernen = Auto fahren lernen
  - Theorie: Erklärt die Zusammenhänge
  - Praxis: das *Entscheidende*. Trainiert die Fähigkeit, tatsächlich Auto zu fahren

So wenig, wie man Autofahren aus einem Buch lernen kann, so wenig geht das beim Programmieren.

- Ihre erste Programmiersprache = Das Fahrschulauto
  - Nicht unbedingt das schönste, schnellste, teuerste, modernste, ...
  - Eher *kein* Automatikfahrzeug (nicht übersimplifizieren)
  - Eher *kein* Formel-1 Auto (nicht überfordern)
  - Ziel der Fahrschule ist nicht, Ihnen ein *bestimmtes Auto* beizubringen, sondern das *Autofahren* – mit dem Ziel, nachher *beliebige* Autos fahren zu können.

# Unsere Programmiersprache: C

- Imperativ
- Relativ Maschinennah
- Kompakt, klein, effizient
- Portable und weit verbreitet (auf jedem Betriebssystem)
- *keine* automatische Speicherverwaltung und *echte* Zeiger (= Handschaltung!)
- *keine* Objektorientierung (= keine S-Klasse!)
- Syntaktische Grundlage von Objective C, C++, Java, C#, Awk, Perl, PHP, ...
- (Leider fehlen Sicherheitsgurte und Airbag ...)

# Literatur

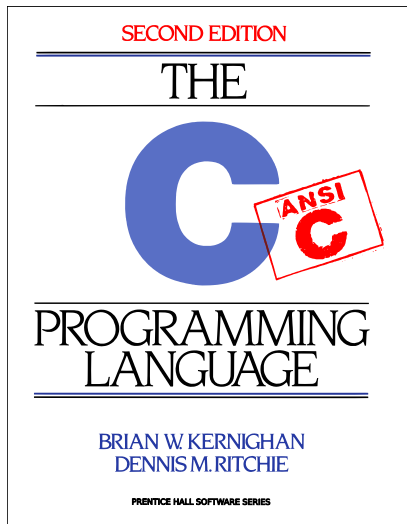
- ▶ Brian W. Kernighan und Dennis M. Ritchie.

*The C Programming Language*,  
2. Ausgabe (ANSI C)

Prentice Hall, 1988.

## Anmerkungen:

- Kernighan & Ritchie sind die Entwickler der Sprache C.
- Das Buch setzt Grundkenntnisse in der Programmierung voraus. („Was ist eine Variable?“ „Wie rufe ich den C-Compiler auf“?)



# Programmierungsumgebung

## ■ Editor

- z. B. Emacs, vi, pico oder nano unter Linux / Mac OSX
- z. B. Notepad++ unter Windows

Mit dem Editor erstellen Sie die Quelltexte

## ■ Compiler

- GNU C-Compiler (gcc)
- Linux: Standardmäßig installiert
- Mac OSX: Mit XCode (Command-Line Tools)
- Windows: MinGW ([www.mingw.org](http://www.mingw.org))

Der Compiler übersetzt Quelltexte in die binäre Maschinensprache und liefert eine ausführbare Programmdatei.

Empfehlung: Wenn Sie noch *keine* Programmiererfahrung haben, verwenden Sie *keine* Entwicklungsumgebung (VisualStudio, Eclipse, ...) – Nutzen Sie die Kommandozeile.

Warum? Wenn Sie noch *keine* Ahnung vom Autofahren haben, verwenden Sie keinen Formel-1 Wagen für Ihre erste Autofahrt ...

Die Konfigurations- und Interaktionsmöglichkeiten dieser Umgebungen werden Sie überfordern.



# Hello World

Inhalt der Datei `hw.c`:

Quelltext: `hw.c`

```
#include <stdio.h>

int main() {
    printf("hello, world\n");
    return 0;
}
```

## ■ Übersetzen mit dem Shell-Kommando:

```
$ gcc -o hw hw.c
```

## ■ Das Kommando `gcc` ist das C-Compiler-System.

- „`gcc -o hw hw.c`“ heißt: „Kompiliere die Quelltextdatei `hw.c` in ein ausführbares Programm und speichere den Maschinencode in der Datei `hw`.“

## ■ Ausführen mit dem Shell-Kommando:

```
$ ./hw
```

- „`./hw`“ heißt: führe die Programmdatei `hw` aus, die sich im aktuellen Verzeichnis befindet.

# Berechne die Summe von zwei Zahlen

Quelltext: *sum.c*

```
#include <stdio.h>

main() {
    int a, b, s;

    printf("First number? ");
    scanf("%d",&a);
    printf("Second number? ");
    scanf("%d",&b);
    s = a+b;
    printf("%d + %d = %d.\n",a,b,s);
}
```

# Imperative Programme

- Bestehen aus *Anweisungen*:  
Befehle an den Computer, etwas zu tun:
  - z. B.: `printf("First number? ");`  
Ist der Befehl „Gebe den Text ‚First number?‘ auf der Standardausgabe aus“.
  - Am Ende einer Anweisung steht immer ein Semikolon.
- Besitzen *Variablen*:  
Speicherplätze für Werte (z. B. Zahlen)
  - z. B.: `int a;`  
Besagt: „a“ soll der Name eines Speicherplatzes sein, in dem Werte des Typs `int` gespeichert werden können.
- Die Werte von Variablen können in Anweisungen genutzt und verändert werden:
  - `s = a+b;`  
bedeutet: „Nimm den Wert aus Speicherplatz `a` und den Wert aus Speicherplatz `b`, zähle beide Werte zusammen und speichere das Ergebnis in Speicherplatz `s`.“

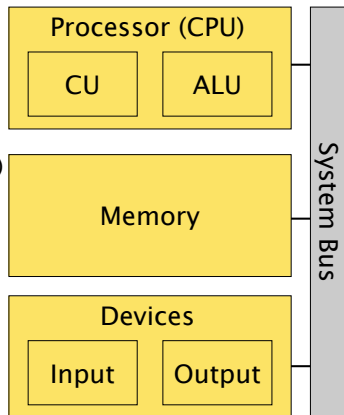
# Die Von-Neumann-Architektur

Programme laufen typischerweise auf Maschinen mit *von-Neumann-Architektur*.

- **Komponenten:**

- Prozessor: Rechenwerk und Steuerwerk
- Eingabe (z. B. Maus, Tastatur)
- Ausgabe (z. B. Bildschirm, Drucker)
- Speicher (z. B. Plattenspeicher, DRAM, SRAM, CD)
- Bus: Datenkommunikation

- Jedes Bauteil eines früheren oder heutigen Computers fällt in eine dieser 5 Kategorien.



- In von-Neumann-Maschinen liegen Programme und Daten im selben Speicher. D. h., Programme *sind* Daten!

# Sprachebenen

Digitale von-Neumann-Maschinen verstehen nur „1“ und „0“:

- Programmierung und Programmausführung über elektrische Signale: An/Aus, 0/1, binary digit = *bit*
- *Maschinensprache* basierend auf Zahlen im Binärsystem  
Computer führt programmierte Anweisungen „sklavisch“ aus
- Anweisungen in Maschinensprache = Sequenzen von bits,  
Beispiel: 1000110010100000  
Bedeutung: Addition zweier Zahlen
- Einfachere Programmierung in *Assemblersprache*  
Beispiel: `add A, B`  
Durch Assembler Übersetzung in Binärsprache
- Mächtiger: Programmierung in *Hochsprache*  
Beispiel: `A+B`  
Durch Compiler Übersetzung in Assemblersprache

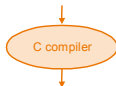
# Hochsprachen

## Abstraktion von spezifischen Maschinengegebenheiten

High-level  
language  
program  
(in C)

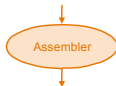
```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

← Hardware  
unabhängig



Assembly  
language  
program  
(for MIPS)

```
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
```

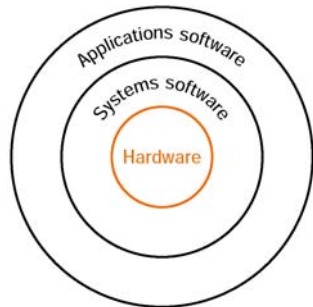


Binary machine  
language  
program  
(for MIPS)

```
000000001010000100000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

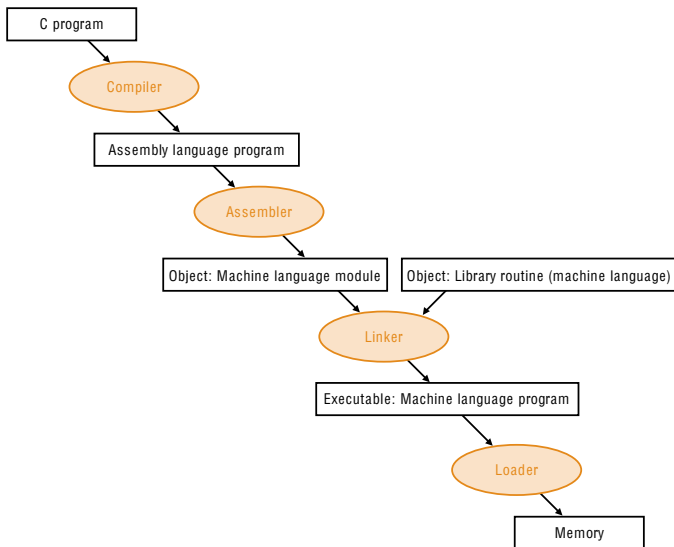
← Hardware  
abhängig

hierarchische  
HW-/SW-Schichten



# Hochsprachen

## Vom Quelltext zum laufenden Programm



# Maschinensprachen

## Tief unten: Gatter & Chips

1. AND gate ( $c = a \cdot b$ )



a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ( $c = a + b$ )

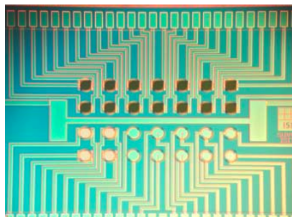


a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

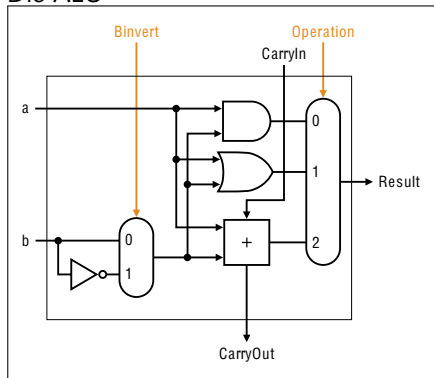
3. Inverter ( $c = \bar{a}$ )



a	$c = \bar{a}$
0	1
1	0



## Ein Stück von-Neumann Maschine: Die ALU





# Statische Typisierung von Variablen

- In C besitzen Variablen *immer* einen Typ.
- Der Typ einer Variablen legt fest, was für Arten von Werten in ihr gespeichert werden können.  
Ganz praktisch: Der Typ bestimmt die Größe des Speicherplatzes, der für die Variable reserviert wird.
- Der Typ einer Variablen wird bei der *Deklaration* der Variablen festgelegt.
  - `int a;`  
ist die Deklaration der Variablen `a` vom Typ `int`
  - Werte vom Typ `int` (kurz für „Integer“) sind ganze Zahlen.  
Am Rande: Können das *alle* ganzen Zahlen sein?
- Der Typ einer Variablen kann nicht geändert werden.
- Dies bezeichnet man als *statische* Typisierung.
- Am Rande: Was könnte *dynamische* Typisierung bedeuten?

# Grundlegende Datentypen in C

Quelltext: *types.c*

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

```
main() {
```

```
    char a; short b; int c; long d;
```

```
    float e; double f;
```

```
    printf("Size(char)      = %d\n", sizeof a);
```

```
    printf("Size(short)     = %d\n", sizeof b);
```

```
    printf("Size(int)        = %d\n", sizeof c);
```

```
    printf("Size(long)        = %d\n", sizeof d);
```

```
    printf("Size(float)       = %d\n", sizeof e);
```

```
    printf("Size(double)      = %d\n", sizeof f);
```

```
}
```

# Grundlegende Datentypen in C

## Typische Wertebereiche

Type	Size	Max	Min
char	1	127	-128
short	2	32.767	-32.768
int	4	2.147.483.647	-2.147.483.648
long	8	9.223.372.036.854.775.807	-9.223.372.036.854.775.808
float	4	$3,40282 \times 10^{38}$	$1,17549 \times 10^{-38}$
double	8	$1,79769 \times 10^{308}$	$2,22507 \times 10^{-308}$

# Blöcke

- Aus einer Folge von hintereinander geschriebenen Anweisungen kann durch Einklammern mit geschweiften Klammern ein *Block* erstellt werden.
  - `{ a=7; b=3; c=5; }`  
ist ein Block von drei Anweisungen
  - Zwischen den Anweisungen kann beliebiger Zwischenraum („white space“) gesetzt werden, um den Block besser lesbar zu machen:

```
{  
    a  =  7;  
    b  =  3;  
    c  =  5;  
}
```

- Am *Anfang* eines Blocks können Variablen deklariert werden.
- Diese Variablen sind nur für Anweisungen *innerhalb des Blocks* sichtbar. Man nennt sie *lokale* Variablen.
- Überall wo eine Anweisung stehen darf, darf auch ein Block stehen.

# Blöcke mit lokalen Variablen

Konsequenz: Blöcke können geschachtelt werden:

```
{ /* Outer Block */
    int a, b, c;

    a = 7;
    b = 3;
    { /* Inner Block */
        int b;

        b = 5;
        c = a+b;
    }
    printf("b=%d, c=%d\n", b, c);
}
```

Was für eine Ausgabe erhalten wir?

# Blöcke mit lokalen Variablen

- In C können die Anweisungen eines Blocks nur diejenigen Variablen sehen, die am Anfang des Blocks deklariert wurden, oder in äußeren, umgebenden Blöcken.
- Welche Variablen eine Anweisung sieht, hängt also davon ab, wo die Anweisung (und die Variablendeklaration) *im Programmtext* steht.
- Man nennt dies auch *lexikalische Gültigkeitsbereiche*.
- Manche Sprachen bieten (auch) *dynamische Gültigkeitsbereiche*:
  - Wenn eine Variable während des *Programmablaufs* erzeugt wird, ist sie von allen *zeitlich* später ausgeführten Anweisungen sichtbar – egal, wo diese im Programmtext stehen.
  - Aufgabe: Überlegen Sie, welche Form der Gültigkeitsbereiche für den Programmierer einfacher zu beherrschen ist.

# Prozeduren: Benannte Blöcke

Blöcke auf der obersten Ebene können benannt werden.

```
foo () {  
    int a, b, c;  
  
    a = 7;  
    b = 3;  
    {  
        int b;  
  
        b = 5;  
        c = a+b;  
    }  
    printf("b=%d, c=%d\n", b, c);  
}
```

Ein benannter Block wird auch als *Prozedur* bezeichnet.

# Benannte Blöcke als Anweisungen

Quelltext: *blocks.c*

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

```
foo () {  
    int a, b, c;  
    a=7; b=3;  
    {  
        int b;  
        b=5; c=a+b;  
    }  
    printf ("b=%d, c=%d\n", b, c);  
}
```

```
main () {  
    foo();  
    foo();  
}
```



# Benannte Blöcke als Anweisungen

- Wir können die Anweisungen in einem benannten Block ausführen, in dem wir einfach den *Namen* des Blocks (mit Klammern) als Anweisung hinschreiben.
- Dadurch können wir Anweisungslisten, die wir an verschiedenen Stellen benötigen, einfach zusammenfassen und mehrfach verwenden.
- Man bezeichnet dies auch als *Prozeduraufruf*.
  - Was ist dann eigentlich `main` für ein Ding?
- In die Klammern eines Prozeduraufrufs können auch Werte und Ausdrücke geschrieben werden – also etwa  
`foo(13, "baz", 5+5);`
- Diese Werte sind *Argumente* des Prozeduraufrufs. Die Anweisungen im Block einer Prozedur können auf die Argumente zugreifen – wie das geht, sehen wir später.
- Was ist dann eigentlich  
`printf("%d + %d = %d\n", a, b, s);` für ein Ding?

# Die Ausgabe-prozedur `printf`

```
printf("%d + %d = %d\n", a, b, s);
```

- Das erste Argument der Ausgabe-prozedur ist der *Format-String*
  - hier: `"%d + %d = %d\n"`
- Dies ist eine *Zeichenkette*, die festlegt, wie die folgenden Argumente behandelt werden sollen.
  - Die folgenden Argumente sind hier: `a`, `b` und `s`.
- Für jedes Folgeargument muss der Format-String eine Formatierungsangabe beinhalten, in der Reihenfolge der Folgeargumente:
  - Hier also: drei Formatierungsanweisungen
- Eine Formatierungsangabe wird durch ein Prozentzeichen eingeleitet, danach folgt die eigentliche Formatanweisung. Zum Beispiel:
  - `%d` – Folgeargument ist eine ganze Zahl vom Typ `int`.
  - `%f` – Folgeargument ist eine Gleitkommazahl vom Typ `double` (oder `float`)

(Das ist nicht die ganze Wahrheit, genügt aber für's erste ...)

# Die einfachste Wiederholungsanweisung: **while**

Falls wir eine Anweisung oder Block wiederholt ausführen wollen, solange eine bestimmte Bedingung gilt, können wir die **while**-Anweisung nutzen:

- **while** (*Bedingung*) *Anweisung*

- Zum Beispiel:

- **while** ( $i < 10$ )  $i = i + 1$ ;

- Wenn am Anfang  $i$  den Wert 0 hat, welchen Wert hat  $i$  am Ende?

- Oder auch (weil ja überall wo eine Anweisung stehen darf, auch ein unbenannter Block geschrieben werden kann):

```
while ( n > 0 ) {  
    e = e * x;  
    n = n - 1;  
}
```

(Wozu könnte diese Schleife dienen?)

## Anwendung: Anlegen einer Tabelle Fahrenheit-Celsius

$$C^{\circ} = \frac{5}{9}(F^{\circ} - 32)$$

Für Temperaturen von 0  $F^{\circ}$  bis 300  $F^{\circ}$  in 20er-Schritten

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

Quelltext: *fahr1.c*

```
#include <stdio.h>

main() {
    int f, c;

    printf("      F      C\n");
    f = 0;
    while (f <= 300) {
        c = (5/9) * (f - 32);
        printf("%5d %5d\n", f, c);
        f = f + 20;
    }
}
```

(Warum funktioniert das nicht?)

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

Quelltext: *fahr2.c*

```
#include <stdio.h>

main() {
    int f, c;

    printf("      F      C\n");
    f = 0;
    while (f<=300) {
        c = 5*(f-32)/9;
        printf("%5d %5d\n", f, c);
        f = f+20;
    }
}
```

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

Quelltext: *fahr3.c*

```
#include <stdio.h>

main() {
    double f, c;

    printf("      F      C\n");
    f = 0;
    while (f<=300) {
        c = (5.0/9.0) * (f-32);
        printf("%5.0f %7.1f\n", f, c);
        f = f+20;
    }
}
```

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

Quelltext: *fahr4.c*

```
#include <stdio.h>
/* Tabelle: Fahrenheit -> Celsius */
main() {
    double f, c;

    printf("      F      C\n");
    f = 0; /* Initialisierung Laufvariable */
    while (f<=300) { /* Test Bedingung */
        c = (5.0/9.0) * (f-32);
        printf("%5.0f %7.1f\n", f, c);
        f = f+20; /* Weiterschalten Laufvar. */
    }
}
```



# Die **for**-Anweisung

Folgendes Muster ist oft zu sehen:

```
Initialisierung;  
while (Bedingung) {  
    Anweisung_1;  
    ...  
    Anweisung_n;  
    Schritt;  
}
```

Mit der **for**-Anweisung geht das kompakter:

```
for (Initialisierung; Bedingung; Schritt) {  
    Anweisung_1;  
    ...  
    Anweisung_n;  
}
```

Allgemein:

**for** (*Initialisierung; Bedingung; Schritt*) *Anweisung*

# Begrifflichkeiten

## In den Schleifenkonstruktionen

```
while (Bedingung) {  
    Anweisung_1;  
    ...  
    Anweisung_n;  
}
```

```
while (Bedingung) Anweisung;
```

```
for (Initialisierung; Bedingung; Schritt) {  
    Anweisung_1;  
    ...  
    Anweisung_n;  
}
```

```
for (Initialisierung; Bedingung; Schritt)  
    Anweisung;
```

Nennt man `Anweisung_1; ... Anweisung_n;` bzw. `Anweisung;` auch „Schleifenkörper“.

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

```
#include <stdio.h>
/* Tabelle: Fahrenheit -> Celsius */

main() {
    double f, c;

    printf("      F      C\n");
    for (f=0; f<=300; f=f+20) {
        c = (5.0/9.0) * (f-32);
        printf("%5.0f %7.1f\n", f, c);
    }
}
```

# Anwendung: Anlegen einer Tabelle Fahrenheit–Celsius

```
#include <stdio.h>
/* Tabelle: Fahrenheit -> Celsius */

main() {
    double f;

    printf("      F      C\n");
    for (f=0; f<=300; f=f+20)
        printf("%5.0f %7.1f\n",
               f, (5.0/9.0) * (f-32));
}
```

(Warum dürfen hier die Klammern ums `printf` fehlen?)

# Symbolische Konstanten

Quelltext: *fahr5.c*

```
#include <stdio.h>
#define MIN 0
#define MAX 300
#define STEP 20

/* Tabelle: Fahrenheit -> Celsius */

main() {
    double f;

    printf("      F      C\n");
    for (f=MIN; f<=MAX; f=f+STEP)
        printf("%5.0f %7.1f\n",
               f, (5.0/9.0) * (f-32));
}
```

# Symbolische Konstanten

## ■ Wozu?

- “Magische Zahlen” aus dem Quelltext entfernen.
- Falls eine Zahl sich ändert, kann dies an *einer* Stelle geschehen.
- Keine Gefahr, eine Änderung zu übersehen.
- Der symbolische Namen macht die *Bedeutung* der Konstanten klar.

## ■ Wie?

- **#define** *Name Ersetzungstext*

# Der C-Präprozessor

- **#define** und **#include**-Anweisungen gehören nicht eigentlich zur Sprache C!
- Es handelt sich um *Präprozessor*-Kommandos.
- Präprozessor-Kommandos verändern den Programmtext *bevor* er durch den Compiler übersetzt wird.
  - Der Präprozessor wird automatisch vor dem eigentlichen Compiler ausgeführt.
- Der C-Präprozessor liest eine C-Datei ein, verarbeitet die Präprozessor-Kommandos, und liefert einen modifizierten Programmtext – erst dieser wird vom C-Compiler übersetzt.
- **#define** *name text*
  - Ersetze jedes Auftreten von *name* durch *text*
- **#include** *<datei>*
  - Füge den Inhalt der Datei *datei* hier ein

# Der C-Präprozessor: Beispiel

## ■ Was im Quelltext stand:

Quelltext: *def.c*

```
#define A 7+ (7  
#define B 3) * 9
```

```
main () {  
    printf ( "A-%d \n ", A*A+B+B );  
}
```

## ■ Was der Compiler sieht:

```
main () {  
    printf ( "A-%d \n ", 7+ (7*7+ (7+3) * 9+3) * 9 );  
}
```

## ■ Wenn man sehen möchte, was der Präprozessor an den Compiler liefert:

```
■ $ gcc -E def.c
```



# Zeichenbasierte Ein- und Ausgabe

Sei die Deklaration `int c;` gegeben.

- Die Anweisung

- `putchar(c);`

gibt das Zeichen mit dem Zeichencode der in der Variablen `c` steht auf der Standardausgabedatei (z. B. dem Terminal) aus.

- Die Anweisung

- `c = getchar();`

Speichert den Zeichencode des aktuellen Zeichens in der Standardeingabedatei (z. B. das Terminal) in der Variablen `c`.

- Die symbolische Konstante `EOF` ist der Zeichencode, der geliefert wird, wenn eine Eingabedatei keine weiteren Zeichen mehr enthält.
- Soll ein Programm `p` eine Datei `f` als Standardeingabe bzw. -ausgabe verwenden, so schreibt man das Shell-Kommando
  - `„$ p <f“` bzw. `„$ p >f“`

# Zeichenkonstanten

- Ein einzelnes Zeichen wird in C mit einfachen Hochkommata geschrieben.
  - 'a', 'b' usw.
- Der Wert einer solchen *Zeichenkonstante* ist eine ganze Zahl, der Zeichencode.
- Das Zeichen 'a' hat zum Beispiel den Code 97.
- Sonderzeichen, wie etwa Zeilenvorschub oder Tabulator (oder Hochkommata) werden durch „Escape-Sequenzen“ geschrieben:

C-Notation	Code	Bedeutung
'\n'	10	Zeilenvorschub
'\t'	9	Tabulator
'\''	39	Einfaches Hochkomma
'\\'	92	Backslash

# putchar – Beispiel

Quelltext: *pc.c*

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

```
main() {  
    putchar(97); putchar(98); putchar(10);  
    putchar('a');putchar('b'); putchar('\n');  
}
```

# Anwendung: Dateien Kopieren

Wie kann man mit diesen Mechanismen Dateien kopieren?

- Schreibe ein Programm `copyio`, das nacheinander jedes Zeichen mit der Anweisung `c=getchar()`; aus der Standardeingabedatei liest und mit der Anweisung `putchar(c)`; in die Standardausgabedatei schreibt.
- Um den Inhalt einer Datei `foo.dat` in eine Datei `bar.dat` zu kopieren verwenden wir das Shell-Kommando
  - `$ ./copyio <foo.dat >bar.dat`
- Das Programm `copyio` muss folgendes tun:

Lese Zeichen `c` aus der Standardeingabe

Solange `c` nicht gleich EOF

    Schreibe `c` nach Standardausgabe

    Lese neues `c` von Standardeingabe

# Anwendung: Dateien Kopieren

Wie kann man mit diesen Mechanismen Dateien kopieren?

- Schreibe ein Programm `copyio`, das nacheinander jedes Zeichen mit der Anweisung `c=getchar()`; aus der Standardeingabedatei liest und mit der Anweisung `putchar(c)`; in die Standardausgabedatei schreibt.
- Um den Inhalt einer Datei `foo.dat` in eine Datei `bar.dat` zu kopieren verwenden wir das Shell-Kommando
  - `$ ./copyio <foo.dat >bar.dat`
- Das Programm `copyio` muss folgendes tun:

```
c = getchar();  
while (c != EOF) {  
    putchar(c);  
    c = getchar();  
}
```

fast fertig ...

# Anwendung: Dateien Kopieren: `copyio.c`

Quelltext: `copyio1.c`

```
/* Defs for EOF, getchar, putchar */  
#include <stdio.h>
```

```
main() {  
    int c;  
  
    c = getchar();  
    while (c != EOF) {  
        putchar(c);  
        c = getchar();  
    }  
}
```

Hässlich ist, dass wir zweimal „`getchar`“ verwenden müssen. In C können wir aber die Zuweisung „*Variable = Ausdruck*“ selbst wieder als Ausdruck verwenden! Der Wert von „*v = e*“ ist der Wert, der gerade in *v* gespeichert wurde. Wie hilft das?

# Anwendung: Dateien Kopieren

Quelltext: *copyio2.c*

```
/* Defs for EOF, getchar, putchar */  
#include <stdio.h>  
  
main() {  
    int c;  
  
    while ((c = getchar()) != EOF)  
        putchar(c);  
}
```

Warum muss hier `c = getchar()` in Klammern stehen? Aus dem gleichen Grund, warum man die Klammern in  $(2+3) * 4$  braucht, wenn man die Summe von 2 und 3 mit 4 multiplizieren will.

„\*“ hat eine höhere Priorität als „+“ – und in C hat der Vergleichsoperator „!=“ eine höhere Priorität als die Zuweisung „=“.  
(Was würde also `c = getchar() != EOF` bedeuten?)

# Notiz: Ausdruck vs. Anweisung

- Ein *Ausdruck* repräsentiert (zunächst einmal) einen *Wert*
  - Beispiel: „ $(2+3) * 4$ “ repräsentiert den Wert (hat den Wert) 20.
- Eine *Anweisung* befiehlt eine Zustandsveränderung.
  - Beispiel „`putchar(c);`“ ändert den Zustand der Ausgabedatei (das Zeichen mit dem Code `c` wird hinzugefügt).
  - (Eine *Anweisung* endet *immer* mit einem Semikolon oder ist ein Block in geschweiften Klammern. Ein *Ausdruck* endet *niemals* mit einem Semikolon oder einer geschweiften Klammer)
- Verwirrend: Ausdrücke in C können *Seiteneffekte* haben. Ein solcher Ausdruck hat nicht nur einen Wert, er ändert auch den Zustand.
- Der Ausdruck „ $c = (2+3) * 4$ “ ist ein Ausdruck mit Seiteneffekt.
  - Der *Wert* dieses Ausdrucks ist der Wert des Teilausdrucks rechts vom Gleichheitszeichen,  $(2+3) * 4$ .
  - Der *Seiteneffekt* ist, dass dieser Wert in der Variablen links vom Gleichheitszeichen, `c`, gespeichert wird.



# Notiz: Ausdruck vs. Anweisung

- Konsequenz: Wenn *Variable = Ausdruck* selbst ein Ausdruck ist, können wir zum Beispiel auch folgenden Ausdruck schreiben:
  - $a = b = c = 0$
  - Was bedeutet das?
- Wenn wir hinter einen Ausdruck ein Semikolon schreiben, wird daraus eine Anweisung.
- „*Ausdruck*;“ ist der Befehl:
  - Berechne den Wert von *Ausdruck*, führe dabei alle Seiteneffekte aus, und verwerfe anschließend den Wert von *Ausdruck*.
  - Offenbar ist eine Anweisung der Form „*Ausdruck*;“ nur dann interessant, wenn *Ausdruck* Seiteneffekte hat.
  - Dies ist beim Zuweisungsausdruck „*Variable = Ausdruck*“ ja offensichtlich der Fall.
- Konsequenz: „`main() { (2+3)*4; }`“ ist ein korrektes C-Programm – allerdings eines, das keinen Effekt hat. (C-Compiler können hier die Warnung „statement with no effect“ liefern.)

# Anwendung: Zeichen Zählen

Quelltext: *ccount.c*

```
#include <stdio.h>

main() {
    int nc;

    nc=0;
    while (getchar() != EOF) ++nc;
    printf("%d characters\n", nc);
}
```

„++nc“ ist eine Kurzschreibweise für „nc=nc+1“

Es gibt auch die Form „--v“ für „v=v-1“ und eine Reihe weiterer Variationen.

Offenbar ist „++nc“ wieder ein Ausdruck mit Seiteneffekt. Der Seiteneffekt: Speichere in `nc` den aktuellen Inhalt von `nc` plus 1. Der Wert des Ausdrucks: der *neue* Inhalt von `nc`.

# Anwendung: Zeichen Zählen

Statt **while** können wir kompakter **for** verwenden:

Quelltext: *ccount2.c*

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

```
main() {  
    int nc;  
  
    for (nc=0; getchar() != EOF; ++nc) ;  
    printf("%d characters\n", nc);  
}
```

- Der Schleifenkörper der **for**-Anweisung besteht nur aus einem einsamen Semikolon. Was bedeutet das?
- Ein einsames Semikolon ist die *leere Anweisung*. Sie ist der Befehl „tue nichts“.

# Anwendung: Zeilen Zählen

Quelltext: *lcount.c*

```
#include <stdio.h>

main() {
    int c, nl;

    nl = 0;
    while ( (c=getchar()) != EOF )
        nl += c == '\n';
    printf("%d lines\n", nl);
}
```

- „\n“ ist das Zeilentrennzeichen
- Der Wert eines Vergleichsausdrucks ist „1“, falls er zutrifft und sonst „0“.
- $v += w$  ist kurz für  $v = v + w$ . (Gibt es auch als  $-=$ ,  $*=$ ,  $/=$ )

# Anwendung: Zeilen Zählen

Etwas ausführlicher (und stilistisch weniger fragwürdig):

Quelltext: *lcount2.c*

```
#include <stdio.h>

main() {
    int c, nl;

    nl = 0;
    while ((c=getchar()) != EOF)
        if (c=='\n') ++nl;

    printf("%d lines\n", nl);
}
```

# Die **if**-Anweisung

- **if** (*Bedingung*)

*DannAnweisung*

**else**

*SonstAnweisung*

- Berechne *Bedingung*. Falls Ergebnis nicht 0, führe *DannAnweisung* aus. Andernfalls die *SonstAnweisung*.

- Der **else**-Fall ist optional:

**if** (*Bedingung*)

*DannAnweisung*

Berechne *Bedingung*. Falls Ergebnis nicht 0, führe *DannAnweisung* aus. Andernfalls tue nichts.

- *DannAnweisung* und *SonstAnweisung* dürfen natürlich auch unbenannte Blöcke sein.

# Spaß mit „=" und „==“

Was ist der Unterschied zwischen

```
if (a==0)
    printf("A ist Null\n");
else
    printf("A ist ungleich Null\n");
```

und

```
if (a=0)
    printf("A ist Null\n");
else
    printf("A ist ungleich Null\n");
```

? Warum akzeptiert der Compiler beide Formen?

Es ist *sehr leicht* in C etwas hinzuschreiben, was der Compiler akzeptiert, aber in Wirklichkeit ein Schreibfehler ist. Philosophie von C: *der Programmierer weiß, was er tut.*

# Anwendung: Wörter Zählen

Quelltext: *wcount.c*

```
#include <stdio.h>
#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d lines, %d words, %d characters\n", nl, nw, nc);
}
```



# Operatoren

- „| |“ ist ein *binärer Operator*.
- Mit Hilfe eines binären Operators  $\odot$  können zwei Ausdrücke  $a$  und  $b$  zu einem neuen Ausdruck  $a \odot b$  kombiniert werden (man nennt  $a$  und  $b$  auch die *Operanden* des Operators).
- Zum Beispiel können mit dem binären Operator „+“ die Ausdrücke 7 und  $3 * 9$  zu dem neuen Ausdruck  $7 + 3 * 9$  kombiniert werden.
- Der binäre Operator „| |“ ist die C-Version des „logischen Oder“: der Ausdruck „ $a \mid\mid b$ “ hat den Wert „1“ genau dann wenn der Teilausdruck  $a$  oder der Teilausdruck  $b$  einen Wert ungleich 0 hat.
- Natürlich ist auch „==“ ein binärer Operator: „ $a == b$ “ hat den Wert 1, wenn der Wert des Teilausdrucks  $a$  gleich dem Wert des Teilausdrucks  $b$  ist.

# Operatoren

- Ein weiterer interessanter binärer Operator ist das Komma „ , “. Der Wert von „ *a* , *b* “ ist der Wert von *b*

Wozu soll das denn gut sein?? Das ist praktisch, wenn man an einer Stelle, an der nur *ein* Ausdruck erlaubt ist, *mehrere* Ausdrücke mit Seiteneffekten ausführen möchte. Zum Beispiel im Kopf einer **for**-Schleife

```
for (i=0, j=n; j>0; i++, j--) { ... }
```

# Unäre Operatoren

Neben den binären Operatoren gibt es auch *unäre* Operatoren, mit einem Operanden.

- Zum Beispiel das unäre Minus: „ $-a$ “.
- aber auch Inkrement „ $++$ “ und Dekrement „ $--$ “. Diese gibt es sowohl in *Präfix*-Form „ $++a$ “ wie auch in *Postfix*-Form: „ $a++$ “.
- Allerdings muss für Inkrement/Dekrement der Ausdruck  $a$  als *Speicherplatz* interpretiert werden können (zum Beispiel, ein Variablenname).
- Ein Inkrement/Dekrement in *Präfix*-Form liefert den *neuen* Wert als Ergebnis, die *Postfix*-Form den *alten* Wert.
- Sei in `int a`; der Wert 7 gespeichert. Dann liefert `a++` den Wert 7, während `++a` den Wert 8 hat – in beiden Fällen steht anschließend der Wert 8 in der Variablen `a`.

# Operatoren

C kennt auch einen *ternären* Operator, mit drei Operanden: „?:“ in der Schreibweise „ $a ? b : c$ “

- Falls der Wert des Ausdrucks  $a$  ungleich 0 ist, liefert dieser Ausdruck das Ergebnis von  $b$ , ansonsten von  $c$ .
- Beispiel:

$p == 0.0 ? 0 : p * \log(p)$

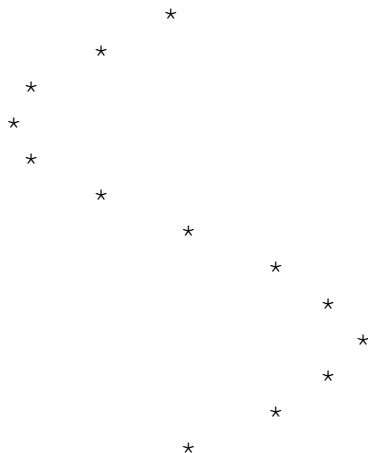
- Mit diesem Ausdruck kann man das Ergebnis von  $p \log p$  korrekt berechnen, auch für den Wert  $p = 0$ .  
Es gilt nämlich

$$\lim_{p \rightarrow 0} p \log p = 0$$

Während allerdings der C-Ausdruck  $p * \log(p)$  einen Rechenfehler liefern kann (warum?)

# Prozedurargumente

Wir wollen eine Sinuskurve plotten:



Der Ausdruck `sin(x)` liefert den Sinus des Ausdrucks  $x$ .

# Sinuskurve

Das geht im Prinzip so:

Quelltext: *sin0.c*

```
#include <stdio.h>
#include <math.h> /* for sin(x) */

main() {
    double x;
    int indent;

    for (x=-180.0; x<=180.0; x+=30.0) {
        /* compute value */
        indent = 10 + 10*sin(x/180.0*M_PI);
        /* plot star at position */
        for (; indent; --indent) putchar(' ');
        printf("*\n");
    }
}
```

# Sinuskurve

Die Anweisungen, einen Stern an einer bestimmten Stelle zu plotten sind:

```
for (; indent; --indent) putchar(' ');  
printf(" * \n");
```

Es wäre an sich praktisch, wenn wir diese Anweisungen in eine Prozedur packen könnten. Wir können dann unterschiedliche Kurven plotten, ohne dass wir diese Anweisungen jedes mal duplizieren müssen.

# Sinuskurve

Also etwa folgendermaßen:

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

star() {
    for(; indent; --indent) putchar(' ');
    printf("*\n");
}

main() {
    double x;
    int indent;

    for(x=-180.0; x<=180.0; x+=30.0)
        indent = 10 + 10*sin(x/180.0*M_PI);
    star();
}
```

Das geht so aber nicht! Warum? Die Anweisungen in `star` können die Variable `indent` in der Prozedur `main` nicht sehen.



# Sinuskurve

Die Lösung ist, ein *Prozedurargument* zu nutzen. Ein Prozedurargument ist eine lokale Variable der Prozedur, die beim Aufruf der Prozedur mit einem Wert initialisiert werden kann.

```
proc (int formal) {  
    ...  
}  
  
main() {  
    ...  
    proc(17+4);  
}
```

Die Variable `formal` ist eine *lokale Variable* der Prozedur `proc`, die als *formales Argument* der Prozedur deklariert wurde. Bei dem *Aufruf* der Prozedur in `main` wird zunächst der Ausdruck `17+4` berechnet und das Ergebnis in die Variable `formal` der Prozedur `proc` kopiert. Danach wird `proc` ausgeführt. Der Wert des Ausdrucks `17+4` ist das *Aktualargument* der Prozedur.

# Sinuskurve

Also:

Quelltext: *sin.c*

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

star(int indent) {
    for (; indent; --indent) putchar(' ');
    printf("*\n");
}

main() {
    double x;

    for (x = -180.0; x <= 180.0; x += 30.0)
        star(10 + 10 * sin(x / 180.0 * M_PI));
}
```

# Sinuskurve

Und mit etwas mehr Luxus:

Quelltext: *sin1.c*

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

star(int indent, char chr) {
    for(; indent; --indent) putchar(' ');
    printf("%c\n", chr);
}

main() {
    double x;

    for(x=-180.0; x<=180.0; x+=30.0)
        star(10+10*sin(x/180.0*M_PI), '*');
    for(x=-180.0; x<=180.0; x+=30.0)
        star(10+10*cos(x/180.0*M_PI), '+');
}
```

## Sinuskurve: Anmerkung

Die Prozedurdefinition ist stilistisch sehr fragwürdig. Warum?

```
star(int indent, char chr) {  
    for(; indent; --indent) putchar(' ');  
    printf("%c\n", chr);  
}
```

Was passiert, wenn `indent` beim Aufruf einen Wert kleiner als Null erhält?

Deutlich besser:

```
star(int indent, char chr) {  
    for(; indent > 0; --indent) putchar(' ');  
    printf("%c\n", chr);  
}
```

# Prozeduraufrufe: Call-by-Value

- Der *Wert* eines Aktualarguments wird beim Aufruf einer Prozedur in diejenige lokale Variable der Prozedur *kopiert*, die als entsprechendes formales Argument deklariert wurde.
- Man nennt dies auch *Call by Value* (Aufruf mit Wertparameter).
- Die Prozedur kann mit dem formalen Argument anstellen, was sie will. Werte im *aufrufenden Block* werden nicht beeinflusst.
- Prozedur und aufrufender Block sind somit voneinander isoliert.
- Interessant ist dies, wenn das Aktualargument eine Variable ist.

# Prozeduraufrufe: Call-by-Value

Welche Ausgabe erhalten wir also hier?

Quelltext: *vcall.c*

```
#include <stdio.h>

proc(int foo) {
    printf("proc: foo at begin: %d\n", foo);
    foo = foo + 42;
    printf("proc: foo at end: %d\n", foo);
}

main() {
    int bar;

    bar = 23;
    printf("main: bar at begin: %d\n", bar);
    proc(bar);
    printf("main: bar at end: %d\n", bar);
}
```

# Prozeduraufrufe: Call-by-Reference

- Es gibt andere Programmiersprachen, bei denen nicht der Wert des Aktualarguments in eine lokale Variable der Prozedur kopiert wird, sondern die Prozedur eine *Referenz* auf die Speicherstelle im aufrufenden Block erhält.
- Dies nennt sich *Call-by-Reference* (Aufruf mit Referenzparameter).
- Aufgabe: Was würde das Beispiel oben für Ausgaben liefern, wenn C Call-by-Reference verwenden würde?
- Wir werden später sehen, dass wir mit *Adressoperatoren* und *Zeigervariablen* auch in C Call-by-Reference realisieren können.
- Call-by-Reference ist „gefährlicher“ als Call-by-Value, da plötzlich Werte im *aufrufenden* Block von der *aufgerufenen* Prozedur verändert werden können.
- Zum Nachdenken: Könnte Call-by-Reference auch Vorteile haben?

# Prozedurdeklarationen

- Eine Prozedurdeklaration hat im Allgemeinen folgende Form:

$p(t_1\ v_1, \dots, t_n\ v_n)\ Block$

Wobei die  $t_i$  Typen sind und die  $v_i$  Variablennamen.

- Der Prozedurkopf ist der Teil „ $(t_1\ v_1, \dots, t_n\ v_n)$ “ hier werden die formalen Argumente deklariert, die als lokale Variablen der Prozedur in *Block* sichtbar sind.

- Ein Prozeduraufruf ist eine Anweisung der Form

$p(e_1, \dots, e_n);$

Hierbei sind die  $e_i$  Ausdrücke, mit denen die in der Prozedurdeklaration als formale Argumente deklarierten lokalen Variablen  $v_i$  initialisiert werden.

- Die Zuordnung von Aktualargumenten  $e_i$  zu formalen Argumenten  $v_i$  erfolgt über die Position.



# Funktionen

Wir haben Ausdrücke der Form „`sin(x)`“ gesehen – diese sehen eigentlich aus wie Prozeduraufrufe, Name und Liste von Aktualargumenten, liefern aber einen Wert zurück.

- `sin` ist ein Beispiel für eine *Funktion*.
- Funktionen sind Prozeduren – benannte Blöcke – die einen Wert zurückliefern.
- Dazu schreibt man vor den Blocknamen den Typ des zurückgelieferten Wertes. Im Block selbst sorgt die `return`-Anweisung für die eigentliche Rückgabe:

```
int x2 (int x) {  
    return x * x;  
}
```

# Funktionen

Beispiel: Potenzfunktion,  $x^n$

Es könnte zum Beispiel eine Funktion nützlich sein, die den Wert von  $x^n$  für ganzzahlige  $x$  und  $n$  berechnet.

```
int power (int x, int n) {  
    int xn;  
  
    for (xn=1; n>0; --n) xn = xn * x;  
    return xn;  
}
```

# Funktionen

Beispiel: Potenzfunktion,  $x^n$

Damit können wir jetzt eine Tabelle von 2er-Potenzen erstellen:

Quelltext: *pwr.c*

```
#include <stdio.h>
#define N 16
#define X 2

int power(int x, int n) {
    int xn;

    for(xn=1; n>0; --n) xn = xn * x;
    return xn;
}

main() {
    int i;

    for(i=0; i<=N; ++i)
        printf("%d^%d = %d\n", X, i, power(X, i));
}
```

# Funktionen

## Default-Ergebnistyp

Es wird Zeit, eine Katze aus dem Sack zu lassen:

- Wird bei der Deklaration einer Funktion der Rückgabety *nicht* angegeben, nimmt C automatisch an, dass die Funktion Werte vom Typ `int` zurückgibt. (Das hat historische Gründe.)
- Das bedeutet: Alle „Prozeduren“, die wir bisher kennengelernt haben, waren in Wirklichkeit Funktionen!
- Um zu sehen, was das bedeutet, kann man mal `vcall.c` folgendermaßen übersetzen:  

```
$ gcc -Wall vcall.c
```
- Um eine Prozedur zu definieren, also eine Funktion, die keinen Wert zurückgibt, müssen wir explizit den Rückgabety `void` verwenden.
- Außerdem ist `main` tatsächlich eine Funktion vom Typ `int`. Das Ergebnis, das `main` zurückgibt, wird vom Betriebssystem als Rückgabecode des Programms genutzt.

# Funktionen

## Korrekte Prozedur- und Funktionstypen

Quelltext: *vcall2.c*

```
#include <stdio.h>

void proc(int foo) {
    printf("proc: foo at begin: %d\n", foo);
    foo = foo + 42;
    printf("proc: foo at end: %d\n", foo);
}

int main() {
    int bar;

    bar = 23;
    printf("main: bar at begin: %d\n", bar);
    proc(bar);
    printf("main: bar at end: %d\n", bar);
    return 0;
}
```

# Prozedurdeklarationen

Korrigierte Version

- Eine Prozedurdeklaration hat im Allgemeinen folgende Form:

**void**  $p(t_1\ v_1, \dots, t_n\ v_n)\ \textit{Block}$

Wobei die  $t_i$  Typen sind und die  $v_i$  Variablennamen.

# Felder

Ein *Feld* ist eine Folge von Speicherplätzen gleicher Art, die hintereinander im Speicher stehen und auf die man mit einem *Index*, einer Nummer, zugreifen kann.

■ `int a[10];`

Besagt „a ist ein Feld mit 10 Variablen vom Typ `int`“. Die Zahl „10“ ist die *Größe* des Feldes.

- Wir können Ausdrücke der Form „`a[i]`“ verwenden, um auf die Variable Nummer *i* des Feldes *a* zuzugreifen.
- Es wird eine fortlaufende Nummerierung verwendet, die erste Nummer ist die „0“.
- Wenn ein Feld die Größe *N* hat, was ist dann die Nummer der letzten Variable im Feld? Natürlich:  $N - 1$ .
- C prüft nicht nach, ob die Nummern, die sie für den Zugriff auf ein Feld verwenden, zulässig sind. Sie können durchaus `a[-10]` oder `a[N]` angeben (natürlich auch `a[N + 42]`). Das kann dann böse Speicherfehler verursachen.

- Die Feldnummern können auch durch Ausdrücke angegeben werden. Zum Beispiel

$$a [ 2 * k + 1 ]$$

Wenn hier in  $k$  zufälligerweise der Wert 3 gespeichert ist, liefert dieser Ausdruck den Wert der Variable Nummer 7 im Feld  $a$ .

- Das heißt, im Ausdruck „ $a[i]$ “ ist  $i$  im Allgemeinen ein *Indexausdruck*.
- Da ein Feldzugriff  $a[i]$  ein Ausdruck ist, kann man ihn natürlich selbst wieder als Indexausdruck verwenden. Also etwa:

$$a [ a [ 2 * k + 1 ] - 3 ]$$

wenn in  $k$  der Wert 3 gespeichert ist, und in  $a[7]$  der Wert 11, bekommen wir also den Wert von  $a[8]$  geliefert. (Warum?)



## Anwendung: Skalarprodukt

Das Skalarprodukt  $a \cdot b$  zweier  $N$ -dimensionaler Vektoren  $a$  und  $b$  ist definiert durch

$$a \cdot b = \sum_{i=0}^{N-1} a_i b_i$$

Wie geht das in C?

# Anwendung: Skalarprodukt

Quelltext: *scal.c*

```
#include <stdio.h>
#define N 3

int main() {
    int i, a[N], b[N], ab;

    for (i=0; i<N; ++i) {
        printf("a[%d] = ", i); scanf("%d", &a[i]);
    }
    for (i=0; i<N; ++i) {
        printf("b[%d] = ", i); scanf("%d", &b[i]);
    }
    for (i=0, ab=0; i<N; ++i)
        ab += a[i]*b[i];
    printf("a.b = %d\n", ab);
    return 0;
}
```

## Anwendung: Zeichen Zählen

Wir wollen ein Programm schreiben, das eine Statistik über die Verwendung von Zeichen in einer Datei erstellt. D. h., wir wollen wissen:

- Wie oft kommt jeder Buchstabe vor?
- Wie viele Ziffern werden verwendet?
- Wie viel Leerraum (Leerzeichen, Zeilentrenner) wird verwendet?
- Wie viele sonstigen Zeichen werden verwendet?

Wie lösen wir diese Aufgabe?

- Wir erinnern uns: der Wert eines Zeichens ist der *Zeichencode*, das ist eine Zahl.
- Außerdem haben alle Großbuchstaben sowie alle Kleinbuchstaben fortlaufende Zeichencodes.
- Wir können also für das Buchstabenzählen ein Feld mit 26 Variablen verwenden und nutzen den Zeichencode als Indexausdruck

## Anwendung: Zeichen Zählen

Das Zeichen `a` hatte den Code 97. Wenn in einer Variablen `c` der Code eines beliebigen Kleinbuchstaben steht, liefert also der Ausdruck `c - 'a'` einen Wert zwischen 0 und 25. Die Idee ist also:

```
int nc[26];  
...  
int c;  
...  
c = getchar();  
...  
if (c >= 'a' && c <= 'z') ++nc[c - 'a'];  
...
```

(Der binäre Operator „&&“ ist das logische „Und“.)

# Anwendung: Zeichen Zählen

Quelltext: *cchar.c*

```
#include <stdio.h>

int main() {
    int nc[26];          /*Letters*/
    int nd, ns, np;      /*Digits, Space, Other*/
    int i, c;

    /* Initialize counters */
    nd = ns = np = 0;
    for(i=0;i<26;++i) nc[i]=0;
    /* Count characters */
    while((c=getchar())!=EOF) {
        if(c>='a' && c <='z') ++nc[c-'a'];
        else if(c>='A' && c <='Z') ++nc[c-'A'];
        else if(c>='0' && c <='9') ++nd;
        else if(c==' ' || c=='\t' || c=='\n') ++ns;
        else ++np;
    }
    /* Report statistics */
    for(i=0;i<26;++i)
        if(nc[i]) printf("%c: %d\n",i+'a',nc[i]);
    printf("%d digits, %d spaces, %d other\n",nd,ns,np);
    return 0;
}
```

# Mehrdimensionale Felder

- Was wir bereits kennen:

```
int a [ 4 ] ;
```

a ist ein Feld mit 4 Variablen vom Typ `int`

Der Ausdruck `a [ 3 ]` ist der Wert der `int`-Variablen Nummer 3 im Feld a.

- Neu: zweidimensionale Felder:

```
int a [ 3 ] [ 4 ] ;
```

- a ist ein Feld mit 3 Variablen vom Typ `int [ 4 ]`. D.h., jede Variable im Feld a ist *selbst* ein Feld.
- Der Ausdruck `a [ 1 ]` ist die Feldvariable Nummer 1 im Feld a.
- Der Ausdruck `a [ 1 ] [ 3 ]` ist die `int`-Variable Nummer 3 aus der Feldvariablen Nummer 1 des Feldes a.

# Mehrdimensionale Felder

Ein zweidimensionales Feld wie etwa

```
int a [ 3 ] [ 4 ] ;
```

heißt auch *Matrix*. Der linke Index ist der *Zeilenindex*, der rechte Index ist der *Spaltenindex*.

		column				
		0	1	2	3	
a[3][4] =	row					
	a[0]	0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
	a[1]	1	a[1][0]	a[1][1]	a[1][2]	<b>a[1][3]</b>
	a[2]	2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Analog ist

```
int a [ 5 ] [ 3 ] [ 4 ] ;
```

ein *dreidimensionales* Feld, bestehend aus fünf Matrizen des Typs

```
int [ 3 ] [ 4 ] .
```

a [ 0 ] ist die erste  $3 \times 4$  Matrix, (die Variable Nummer 0) im Feld a.

# Anwendung: Matrixmultiplikation

Die Multiplikation einer  $N \times K$ -Matrix  $A$  mit einer  $K \times M$ -Matrix  $B$  liefert eine  $N \times M$ -Matrix  $C$ , definiert durch:

$$c_{ij} = \sum_{k=0}^{K-1} a_{ik} b_{kj}$$

Lösung in C (das ist nur ein Fragment):

```
int main() {
    int A[N][K], B[K][M], C[N][M];
    int i, j, k;
    /* ToDo: Initialize A, B */
    /* Do multiplication */
    for(i=0; i<N; i++) for(j=0; j<M; j++) {
        C[i][j] = 0;
        for(k=0; k<K; k++) C[i][j] += A[i][k]*B[k][j];
    }
    /* ToDo: Output result */
    return 0;
}
```



# Globale Variablen

Variablen können auch außerhalb eines Blocks definiert werden. Dies sind dann *globale* Variablen. Diese Variablen sind *überall* sichtbar (wenn sie nicht durch eine lokale Variable überdeckt werden).

Quelltext: *gvar1.c*

```
#include <stdio.h>

int foo;

void bar() {
    printf("foo = %d\n", foo);
}

int main() {
    foo = 42;
    bar();
    return 0;
}
```

Wenn *vielen* Funktionen / Prozeduren auf dieselben Zustandsinformationen zugreifen müssen, können globale Variablen hilfreich sein.

# Initialisierung von Variablen

Welchen Wert hat eine Variable, *bevor* wir ihr zum ersten Mal etwas zuweisen?

Quelltext: *gvar2.c*

```
#include <stdio.h>

int global;

int main() {
    int local;

    printf("global=%d local=%d\n", global, local);
    return 0;
}
```

Mögliche Ausgabe:

```
$ ./gvar2
global=0 local=32767
$
```

# Initialisierung von Variablen

- *Globale* Variablen werden *einmal vor Programmstart* mit dem Wert 0 initialisiert (alle Bits werden auf 0 gesetzt).
  - Die Benutzung globaler Variablen ohne vorherige Wertzuweisung ist auf definierte Weise *möglich* (wenn auch nicht unbedingt guter Stil).
- *Lokale* Variablen haben *irgendeinen zufälligen* Wert.
  - *Lokale* Variablen kann man ohne vorherige Wertzuweisung *nicht* auf definierte Weise nutzen.
  - Das ist ein *Programmierfehler*.

# Initialisierung von Variablen

Einer Variablen kann man bereits bei der Deklaration einen Initialwert zuweisen.

Quelltext: *gvar3.c*

```
#include <stdio.h>

int global = 2 * 3;

int main() {
    int local = 5 * global;

    printf("global=%d local=%d\n", global, local);
    return 0;
}
```

# Initialisierung von Variablen

- Der Initialwert einer globalen Variable wird *zur Compilezeit* ausgerechnet und steht in der Programmdatei selbst.
- Für die Initialwerte globaler Variablen sind daher nur einfache *konstante* Ausdrücke zulässig.
- Der Initialwert einer *lokalen* Variable wird *zur Laufzeit* ausgerechnet.
- Für lokale Variablen sind daher alle Ausdrücke erlaubt.
- Tatsächlich ist

```
int local = 3 * global;
```

Nur eine Kurzform für

```
int local;  
...  
local = 3 * global;
```

# Initialisierung von Variablen

Eine Feldvariable benötigt eine Liste von Werten für die Initialisierung. Diese Liste gibt man mit geschweiften Klammern an. Bei mehrdimensionalen Feldern werden diese Listen entsprechend geschachtelt.

Quelltext: *ivar1.c*

```
#include <stdio.h>

int a[3] = {17, 18, 19};

int main() {
    int b[2][3] = {{1, 2, 3}, {4, 5, 6}};

    return 0;
}
```

Falls die Initialisierungsliste zu kurz ist, werden die restlichen Feldelemente mit 0 initialisiert.

# Initialisierung von Variablen

Wenn für ein Feld eine Initialisierungsliste angegeben wird, kann man die Feldgröße weglassen. Das Feld wird dann genau so groß, wie Elemente in der Initialisierungsliste stehen.

- Ein **int**-Feld der Größe 5:

```
int a[] = {1, 2, 3, 4, 5};
```

- Ein **char**-Feld der Größe 6:

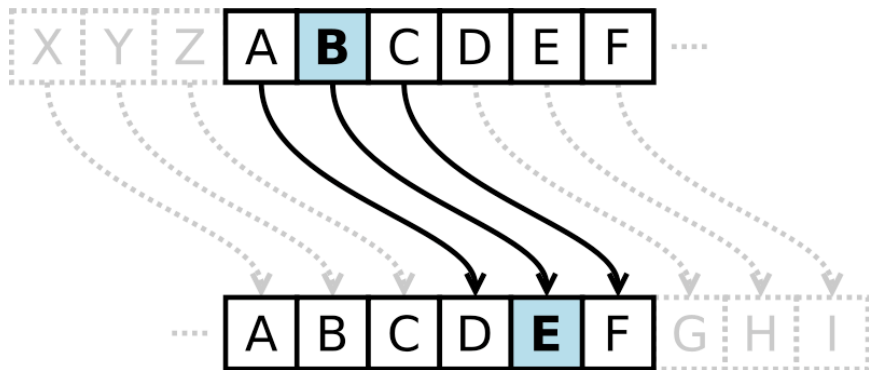
```
int b[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

- Genau das gleiche **char**-Feld:

```
int b[] = "Hello";
```

Wir werden später noch genauer den Zusammenhang zwischen Zeichenketten und **char**-Feldern kennenlernen.

# Anwendung: Der Cäsar-Chiffre



- Verschlüsselung eines Buchstabens durch die Verschiebung um  $n$  Stellen. Schlüssel =  $n$
- Zum Beispiel  $n = 3$ : 'B' (Buchstabe 1) wird 'E' (Buchstabe 4) ('A' ist Buchstabe 0)
- 'Y' (Buchstabe 24) wird Buchstabe  $(24+3) \% 26 = 1$ , also 'B' („ $\%$ “ ist der Modulo-Operator)



# Der Cäsar-Chiffre

## Verschlüsselung

```
#define MAX 100
#define N 26

char imsg[MAX]; /* plaintext */
char omsg[MAX]; /* ciphertext */

void cipher(int shift) {
    int i, c;
    for (i=0; imsg[i]; i++) {
        c = imsg[i];
        omsg[i] = (c >= 'a' && c <= 'z')
            ? ((c-'a')+shift)%N+'a' : c;
    }
    omsg[i] = '\0';
}
```

# Der Cäsar-Chiffre

## Beispiel

- Klartext:

`„c programming is fun“`

- Schlüssel:  $n = 7$

- Kryptogramm:

`„j wyvnyhttpun pz mbu“`

„Der Name der Caesar-Verschlüsselung leitet sich vom römischen Feldherrn Gaius Julius Caesar ab, der nach der Überlieferung des römischen Schriftstellers Sueton diese Art der geheimen Kommunikation für seine militärische Korrespondenz verwendet hat. Dabei benutzte Caesar eine Verschiebung des Alphabets um drei Buchstaben.“ (Wikipedia)

# Der Cäsar-Chiffre

## Entschlüsselung

- Eine mit dem Schlüssel  $n$  verschlüsselte Botschaft kann man durch Verschieben um  $-n$  Stellen entschlüsseln.
- Da wir Modulo 26 rechnen ist eine Verschiebung um  $-n$  Stellen das Gleiche wie eine Verschiebung um  $26 - n$  Stellen.

Beispiel:  $n = 3$ , Buchstabe 17, Entschlüsseln mit  $-3$ :

$$(17 - 3) \% 26 = 14 \% 26 = 14$$

$$(17 + (26-3)) \% 26 = 40 \% 26 = 14$$

- Somit kann man

`cipher(n)`

mit

`cipher(26-n)`

entschlüsseln

- Wie aber finden wir für eine *verschlüsselte* Botschaft das  $n$ , mit dem sie verschlüsselt wurde? – Das ist das Problem den Schlüssel zu *knacken*.

# Der Cäsar-Chiffre

## Knacken des Verschlüsselung

- Idee? Zählen der Häufigkeiten von Buchstaben.
- Das „e“ ist beispielsweise im Englischen der häufigste Buchstabe. Wenn wir also einen (langen) verschlüsselten Text finden, von dem wir wissen, dass der Klartext Englisch ist, müssen wir nur den häufigsten Buchstaben suchen – dieser entspricht dann dem „e“, daraus ergibt sich die Verschiebung.
- Problem? In kurzen Texten gilt das durchaus nicht:

*„c programming is fun“*

(da ist gar kein „e“ drin!)

- Lösung: vergleiche Häufigkeiten über *alle* Buchstaben.

# Der Cäsar-Chiffre

## Knacken der Verschlüsselung

Ansatz: Für jede mögliche Verschiebung des verschlüsselten Textes:

- Berechne die Häufigkeiten der Buchstaben.
- Vergleiche die beobachteten Häufigkeiten mit den erwarteten Häufigkeiten.
- Wähle die Verschiebung, bei der beobachtete Häufigkeiten am besten zu den erwarteten Häufigkeiten passen.

# Der Cäsar-Chiffre

## Knacken des Verschlüsselung

Vergleichen von Häufigkeiten: die  $\chi^2$ -Statistik:

$$\chi^2 = \sum_{i=0}^{N-1} \frac{(\text{ofreq}_i - \text{efreq}_i)^2}{\text{efreq}_i}$$

Falls der  $\chi^2$ -Wert klein ist, ist die Übereinstimmung zwischen beobachteten (*ofreq*) und erwarteten Häufigkeiten (*efreq*) gut. (hier ist *i* die Buchstabennummer und  $N = 26$ )

Erwartete Häufigkeiten für Englisch:

```
double efreq[N] = {  
    8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0,  
    6.1, 7.0, 0.2, 0.8, 4.0, 2.4, 6.7,  
    7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8,  
    1.0, 2.4, 0.2, 2.0, 0.1  
};
```

# Der Cäsar-Chiffre

Berechnen des  $\chi^2$ -Wertes

```
double ofreq[N];
double chmin=HUGE_VAL; /* + oo, in math.h */
double chisq;
int i, shift, smin;

for(shift=0; shift<N; ++shift) {
    for(chisq=0.0, i=0; i<N; ++i)
        chisq += pow(ofreq[(i+shift)%N] - efreq[i], 2.0)
                  / efreq[i];
    if(chisq < chmin) {
        chmin = chisq;
        smin = shift;
    }
}
printf("Best shift: %d\n", smin);
```

# Der Cäsar-Chiffre

Berechnen von `ofreq`

Die Häufigkeitstabelle für den Text:

```
char  imsg[MAX];
double ofreq[N], nchar;
int  i;

for (nchar=0.0, i=0; imsg[i]; ++i) {
    int c = imsg[i];
    if (c >= 'a' && c <= 'z') {
        ++ofreq[c-'a'];
        ++nchar;
    }
}

/* convert to percentages */
for (nchar/=100.0, i=0; i<N; ++i)
    ofreq[i] /= nchar;
```



# Der Cäsar-Chiffre

## Der komplette Dechiffrierer

```
void decipher() {
    double ofreq[N], nchar;
    double chmin=HUGE_VAL; /* +oo, in math.h */
    double chisq;
    int i, shift, smin;

    /* reset data */
    for (i=0; i<N; i++) ofreq[i] = 0;
    /* insert code for computing ofreq */
    /* insert code for computing smin */
    /* decode */
    cipher(N-smin);
}
```

# Der Cäsar-Chiffre

## Ein kleines Rahmenprogramm

```
int main() {  
    int shf;  
    printf("Shift = ");  
    scanf("%d",&shf);  
    fpurge(stdin); /* clear input */  
    printf("Message = ");  
    gets(msg);  
    cipher(shf);  
    printf("Coded = %s\n", msg);  
    strcpy(msg, msg);  
    decipher();  
    printf("Decoded = %s\n", msg);  
    return 0;  
}
```

Vollständiger Quelltext: `cesar.c`

# Rekursive Funktionen und Prozeduren

## Grundprinzip

- Viele Probleme besitzen eine „Problemgröße“  $n$ . Je größer  $n$ , desto schwieriger ist das Problem.
  - Beispiel: den kürzeste Weg über  $n$  Städte finden. Je größer  $n$ , desto mühsamer ist das.
- Oft ist die Lösung für  $n = 0$  einfach.
- Für manche Probleme dieser Art gilt zusätzlich, dass die Lösung für ein beliebiges  $n > 0$  einfach ist, wenn man die Lösungen für einige  $n' < n$  kennt. (Beispielsweise für  $n' = n - 1$ )
- Hier würde man folgende Lösungsprozedur nutzen können:

```
void solveP (int n) {  
    if (n==0) {  
        ... /* solve simple case */  
    } else {  
        solveP (n-1); /* compute helper */  
        ... /* solve for n using helper */  
    }  
}
```

# Rekursive Funktionen und Prozeduren

- Dies nennt man *Rekursion*.
- Eine rekursive Lösung ist dadurch gekennzeichnet, dass die Anweisungen (Ausdrücke) der Lösungsprozedur (Lösungsfunktion) selbst einen Aufruf der Prozedur (Funktion) enthalten.
  - Das kann auch indirekt erfolgen:  $f()$  ruft  $g()$  auf,  $g()$  ruft  $h()$  auf, und  $h()$  wieder  $f()$ .(Folgend sprechen wir einfach immer von „Lösungsfunktion“, auch wenn das eine Prozedur ist.)
- Damit das irgendwann anhält, muss bei jedem rekursiven Aufruf irgend ein Wert immer kleiner werden, und die Lösungsfunktion ruft sich *nicht* mehr selbst auf, wenn dieser Wert „klein genug“ ist.
  - Ob dieser „Wert“ ein Argument der Funktion ist oder in einer globalen Variablen steht, was genau „wird kleiner“ heißt, und was „klein genug“ bedeutet, hängt im Allgemeinen vom konkreten Problem ab.

# Rekursive Funktionen

## Beispiel: Fakultät

Der Ausdruck  $n!$  mit  $n \in \mathbb{N}$  ist die *Fakultät* von  $n$ .

Der Wert von  $n!$  ist definiert durch

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \times (n-1)! & \text{sonst} \end{cases}$$

Dies lässt sich unmittelbar in eine rekursive C-Funktion übersetzen:

Quelltext: *fak.c*

```
int fak (int n) {  
    return n <= 1 ? 1 : n * fak (n-1);  
}
```

Anmerkung: Die Fakultät lässt sich allerdings auch sehr gut mit einer Schleife berechnen (Übungsaufgabe!) – sie ist daher ein *Beispiel für das Prinzip* der Rekursion, aber kein guter Anwendungsfall.

Ein viel besseres Beispiel sind die *Türme von Hanoi*.

Aber vorher: Wie werden lokale Variablen in einer rekursiven Funktion verwaltet?

# Rekursive Funktionen

## Lokale Variablen

- Solange sich Funktionen nicht *rekursiv* aufrufen ist die Verwaltung lokaler Variablen einfach. Jede Funktion hat ihre lokalen Variablen, dafür wird Speicher bereitgestellt, und fertig.
- Es ist dabei nicht interessant, ob der Speicher für die lokalen Variablen einer Funktion bei Programmstart bereitgestellt wird oder erst zu dem Zeitpunkt, zu dem die Funktion tatsächlich aufgerufen wird.
- Wir bezeichnen den Aufruf einer Funktion auch *Aktivierung*. Bei der Aktivierung werden die Werte der Aktualargumente in diejenigen lokalen Funktionsvariablen kopiert, die die formalen Argumente der Funktion sind.
- Wenn die Funktion beendet wird (d. h., wenn die Ausführung bei der schließenden Klammer ankommt oder ein **return** ausgeführt wird), erfolgt die *Deaktivierung* der Funktion.
- Im Zeitraum zwischen Aktivierung und Deaktivierung ist die Funktion *aktiv*.

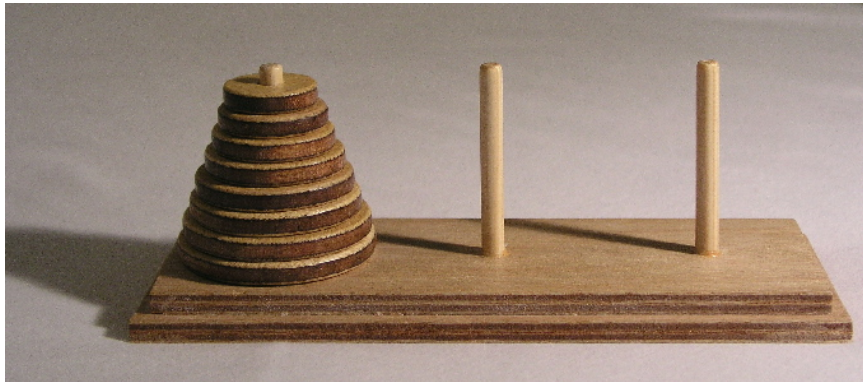
# Rekursive Funktionen

## Lokale Variablen

- Sei `int f(int n) { ... }` irgend eine rekursive Funktion.
- Wir stellen uns vor, `f(n)` wird aufgerufen – also ist jetzt `f` aktiv.
- Wenn jetzt nun innerhalb von `f` der rekursive Aufruf `f(n-1)` erfolgt, haben wir die Situation, dass wir `f` erneut aktivieren wollen (mit dem Argument `n-1`), obwohl `f` bereits aktiv ist! (mit dem Argument `n`)
- Es ist damit klar, dass beide Aktivierungen von `f` ihren *eigenen* Speicherplatz für jeweils ihre Version der lokalen Variablen benötigen.
- C erzeugt daher bei *jeder* Aktivierung der Funktion `f` eine neue *Instanz* der Funktion, die ihren eigenen Speicherplatz für die lokalen Variablen von `f` bereitgestellt bekommt.
  - Brauchen die verschiedenen Instanzen auch ihre eigenen Versionen des Programmcodes von `f`?
  - Zum Glück nicht: In C ist selbstmodifizierender Code nicht möglich, daher teilen sich alle Instanzen denselben Code.

# Türme von Hanoi

## Problemstellung



Bewege den Turm vom linken auf den rechten Platz.

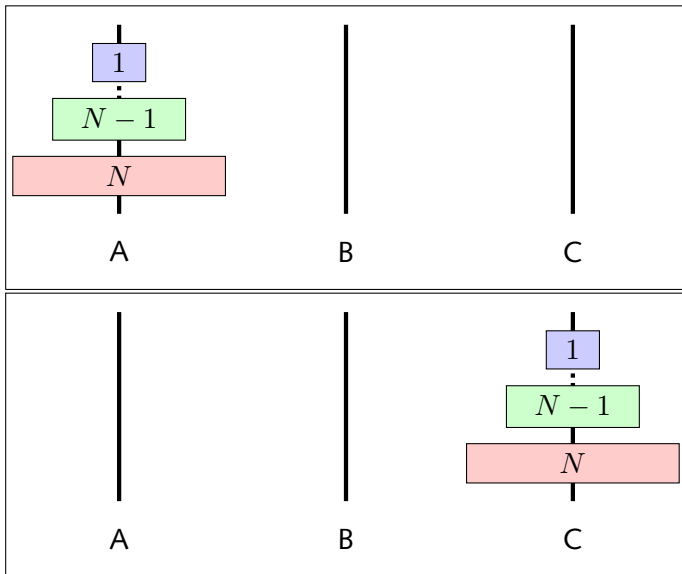
- Es darf bei jedem Zug nur eine Scheibe bewegt werden.
- Es dürfen nur kleinere Scheiben auf größere gelegt werden.
- Der mittlere Platz darf ebenfalls verwendet werden.

(Dieses Spiel wurde vermutlich 1883 vom französischen Mathematiker Édouard Lucas erfunden.)



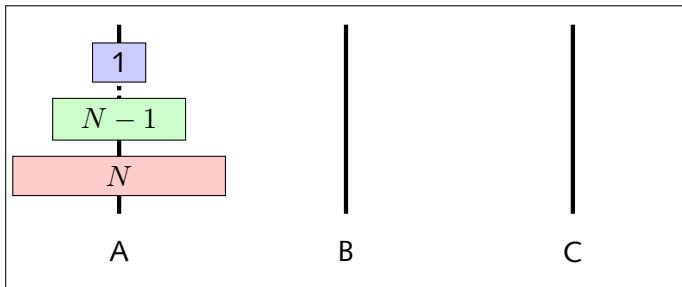
# Türme von Hanoi

## Ausgangssituation und Ziel



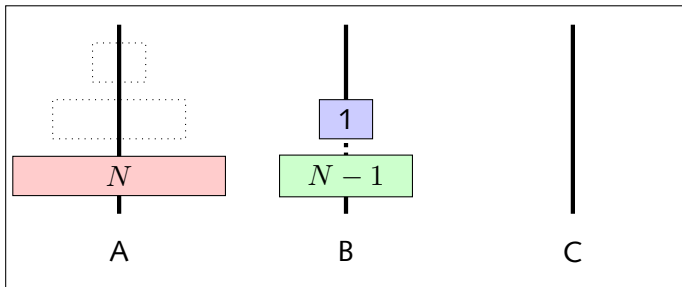
# Lösungsansatz

Ausgangssituation: Turm  $N$  auf Ausgangsort



# Lösungsansatz

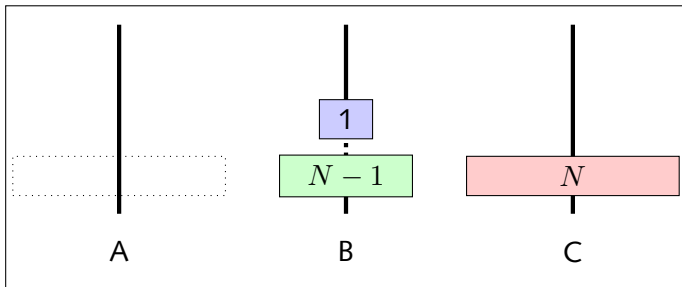
1. Bewege Turm  $N - 1$  von Ausgangsort nach Hilfsplatz



Bemerkung: Wenn wir den Turm  $N - 1$  bewegen können wir *alle* Orte beliebig nutzen: an allen Orten können nur Scheiben liegen, die größer sind als  $N - 1$ .

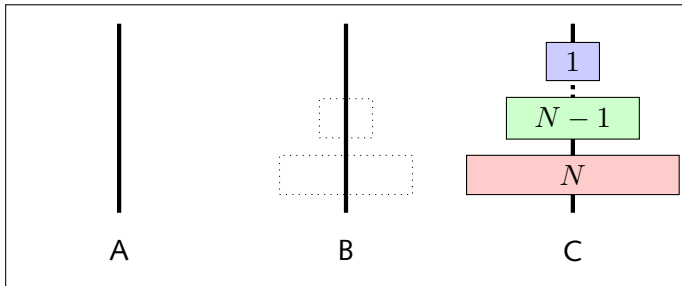
# Lösungsansatz

2. Bewege Scheibe  $N$  von Ausgangsort nach Zielort



# Lösungsansatz

3. Bewege Turm  $N - 1$  von Hilfsplatz nach Zielort



# Türme von Hanoi

## Rekursive Lösung in C

```
int nm=0; /* move counter */

void move (int n,
           char from,
           char to,
           char help) {
    if (n<=0) return;
    move (n-1, from, help, to);
    printf ("%d: D%d %c-%c\n", ++nm, n, from, to);
    move (n-1, help, to, from);
}
```

# Türme von Hanoi

## Rekursive Lösung in C

Quelltext: *hanoi.c*

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

int nm=0;

void move(int n, char from, char to, char help) {
    if (n<=0) return;
    move(n-1, from, help, to);
    printf("%d: D%d %c-%c\n", ++nm, n, from, to);
    move(n-1, help, to, from);
}

int main(int argc, char *argv[]) {
    int n;
    if (argc < 2 || (n = atoi(argv[1])) < 0) {
        printf("Usage: %s <number-of-disks>\n", argv[0]);
        return -1;
    }
    printf("Tower of hanoi with %d disks:\n", n);
    move(n, 'A', 'C', 'B');
    return 0;
}
```

# Call by Reference

## Problemstellung

- Manchmal ist es ungünstig, dass Argumente kopiert werden
- Betrachten wir beispielsweise folgende Prozedur zum Vertauschen des Inhaltes zweier Speicherplätze:

```
void swap (int a, int b) {  
    int tmp = a;  
  
    a = b;  
    b = tmp;  
}
```

- Nutzen wir diese Prozedur in irgendeinem Block

```
void block () { int x, y;  
    ...  
    swap (x, y);  
    ...  
}
```

erhalten wir natürlich nicht das gewünschte Resultat



# Call by Reference

## Problemstellung

- Die Prozedur `swap` soll den Speicher in `block` modifizieren (d.h., die Inhalte der Variablen `x` und `y` des aufrufenden Blocks `block` vertauschen)
- Aber natürlich wird durch Call-by-Value der Speicher von `block` vor Zugriffen durch `swap` isoliert.
- Damit `swap` den Speicher von `block` verändern kann, müsste es *Referenzen* auf diese Speicherstellen erhalten.
- Für diesen Zweck (und viele andere Anwendungen ...) können in C *Referenzvariablen* auch („Pointer“ oder „Zeiger“ genannt) eingesetzt werden.

# Call by Reference

## swap mit Referenzvariablen

Eine modifizierte Version von `swap` sieht wie folgt aus:

```
void swap(int *a, int *b) {  
    int tmp = *a;  
  
    *a = *b;  
    *b = tmp;  
}
```

- Hierbei besagt die Deklaration `int *a`, dass die Variable `a` eine Referenzvariable ist, die eine Referenz auf eine Variable vom Typ `int` speichern kann.
- Allgemein wird eine Deklaration „ $T *v$ “ ausgesprochen als „ $v$  ist eine Referenz auf eine Variable vom Typ  $T$ “.
- Der Ausdruck „`*a`“ ist die Variable, deren Referenz in der Referenzvariable `a` gespeichert ist.
- Wenn also in `a` eine Referenz auf die Variable `x` gespeichert ist und in `b` eine Referenz auf die Variable `y`, dann bedeutet der Ausdruck „`*a = *b`“ genau das gleiche wie „`x = y`“.

# Call by Reference

## Referenzen erzeugen

- Wenn  $x$  irgendeine Variable ist, dann erzeugt der Ausdruck  $\&x$  eine *Referenz* auf  $x$ .
- Somit können wir also folgendes schreiben:

```
int x;      /* integer variable */
int *a;     /* reference var for int var */

a = &x;     /* get ref to x into a */

*a = 17;    /* store 17 in var ref'd by a.
              that is: store 17 into x */
```

- Der Präfix-Operator „ $*$ “ ist der *Dereferenzierungsoperator*.  
Wenn  $a$  ein Ausdruck ist, der eine Referenz auf eine Variable  $x$  enthält, dann liefert der Ausdruck  $*a$  genau den Zugriff auf  $x$ .

# Call by Reference

## Die swap-Prozedur

Die neue Version von swap

```
void swap (int *a, int *b) {  
    int tmp = *a;  
  
    *a = *b;  
    *b = tmp;  
}
```

wird also folgendermaßen verwendet:

```
void block () { int x, y;  
    ...  
    swap (&x, &y);  
    ...  
}
```

Wir können somit in C den „Call-by-Reference“-Mechanismus durch die Kombination von Referenzvariablen (\*a) in der Funktion und Referenzausdrücken (&x) im aufrufenden Block realisieren.

# Call by Reference

## Die `scanf`-Funktion

Jetzt verstehen wir auch besser, was der Aufruf

```
int a;  
...  
scanf ( "%d" , &a ) ;
```

Im Programm `sum.c` bedeutet:

- Die Prozedur `scanf` liest Zeichen von der Standardeingabe, konvertiert diese in C-Werte (zum Beispiel: `int`-Werte) und speichert diese Werte dann in Variablen, die Call-by-Reference übergeben werden.
- Analog zu `printf` ist das erste Argument von `scanf` der *Format-String*, der angibt, wie die Eingabezeichen zu konvertieren sind.
  - Hier: `%d` – konvertiere Zeichenfolge in `int`-Wert
- Die Folgeargumente sind dann Referenzen auf Variablen, in denen die konvertierten Werte abgelegt werden.

# Call by Reference

## Die scanf-Funktion

Einige weitere Konvertierungen:

- %d – konvertiere Zeichenfolge in **int**-Wert
- %f – konvertiere Zeichenfolge in **float**-Wert
- %lf – konvertiere Zeichenfolge in **double**-Wert

Nutzen dieser Konvertierungen: (scanf.c)

```
int a,d,n; double b; float c;
...
n = scanf("%d %lf %f %d",&a,&b,&c,&d);
printf("%d values read: a=%d, b=%f, c=%f, d=%d\n",
       n,a,b,c,d);
```

Der Rückgabewert der scanf-Funktion ist die Anzahl der erfolgreich gelesenen Werte.

Warum muss man bei scanf zwischen **float** und **double**-Werten unterscheiden, aber nicht bei printf?

# Felder und Referenzen

In C gibt es einen sehr interessanten Zusammenhang zwischen Feldern und Referenzen:

*Wenn der Typ eines Ausdrucks „Feld mit Elementen vom Typ T“ ist, dann ist der Wert des Ausdrucks eine Referenz auf das erste Element (das Element Nummer 0) dieses Feldes.*

Beispiel:

```
int a[4] = { 314 , 159 , 265 , 358 } ;
```

deklariert `a` als Feld von vier Integer-Variablen, wobei `a[0]` mit dem Wert 314 initialisiert wird und `a[3]` mit dem Wert 358.

Damit ist der Ausdruck „`a`“ vom Typ „Feld mit Elementen vom Typ `int`“. D.h., der Wert des Ausdrucks „`a`“ ist eine Referenz auf die erste Feldvariable – also eine Referenz auf `a[0]`.

Daraus folgen eine Reihe interessanter Konsequenzen.

# Felder und Referenzen

## Konsequenzen

- Wenn `a` eine Referenz ist, dann können wir natürlich den Dereferenzierungsoperator anwenden und `*a` schreiben. Und das liefert dann eben genau die Variable `a[0]`.
- Die Ausdrücke `a` und `&a[0]` liefern *dieselbe* Referenz.
- Das folgende Programmfragment ist vollkommen in Ordnung

```
int a[4] = {314, 159, 265, 358};  
int *b;  
b = a; /* put ref to a[0] into b */  
if (*b == a[0]) { /* always true */ ... }
```

- Obendrein darf man statt `*b` genau so gut `b[0]` schreiben. Und es gilt: `b[0] == a[0]`, also auch `b[0] == 314`.
- Nach Ausführung der Anweisung `*b = 979;` (oder genauso gut: `b[0] = 979;`) gilt dann `a[0] == 979`.



# Felder und Referenzen

## Was genau passiert

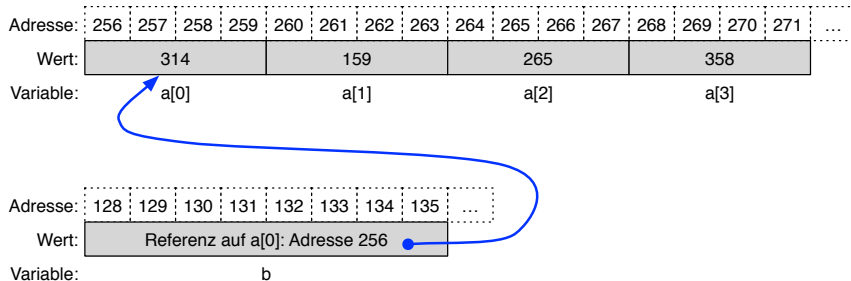
Adresse:	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	...
Wert:	314				159				265				358				
Variable:	a[0]				a[1]				a[2]				a[3]				

Adresse:	128	129	130	131	132	133	134	135	...
Wert:	undefinierte Adresse								
Variable:	b								

```
int a[4] = {314, 159, 265, 358};  
int *b;
```

# Felder und Referenzen

Was genau passiert



```
int a[4] = {314, 159, 265, 358};  
int *b;  
b = a;
```

# Felder und Referenzen

Es wird noch besser

- Wir bleiben bei dem Beispiel:

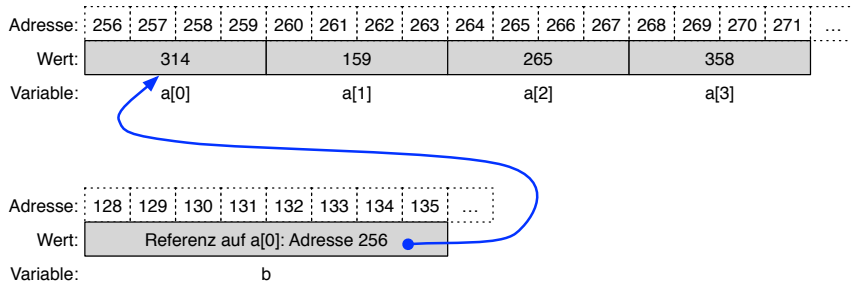
```
int a[4] = {314, 159, 265, 358};  
int *b;  
b = a; /* put ref to a[0] into b */
```

- Wenn  $b$  eine Referenz auf die erste Feldvariable des Feldes  $a$  ist, denn liefert der Ausdruck  $b+1$  eine Referenz auf die zweite Feldvariable, also auf  $a[1]$ !
- Somit gilt  $*(b+1) == a[1]$  und auch  $*(b+1) == 159$ .
- Obendrein gilt  $*(b+1) \equiv b[1]$ .
- Und auch  $*(b+3) \equiv b[3]$  und  $*(b+3) == 358$ .
- Tatsächlich gilt in C folgende Identität:

Wenn  $e_1$  und  $e_2$  zwei Ausdrücke sind, so dass einer von beiden einen ganzzahligen Wert hat und der andere einen Wert vom Typ „Referenz auf  $T$ “ dann haben die Ausdrücke „ $e_1[e_2]$ “ und „ $*(e_1+e_2)$ “ *dieselbe Bedeutung*.

# Felder und Referenzen

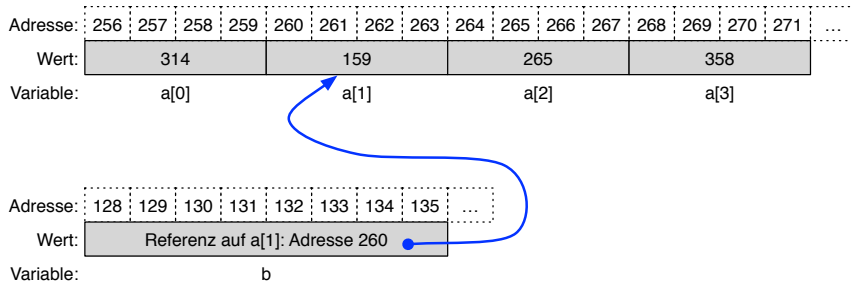
Was genau passiert



```
int a[4] = {314, 159, 265, 358};  
int *b;  
b = a;
```

# Felder und Referenzen

Was genau passiert



```
int a[4] = {314, 159, 265, 358};  
int *b;  
b = a;  
b = b + 1;
```

Woher weiß C, dass die Anweisung `b = b+1;` bedeutet, dass die Adresse, die in der Referenz steht, um 4 erhöht werden muss?

# Felder und Referenzen

## Adresse und Größe einer Referenz

Betrachten wir folgendes Codefragment, wobei  $T$  irgend ein Typ ist.

```
 $T$  a[N], *r;  
r = a;
```

Dies besagt,  $r$  ist eine Referenz auf die Variable  $a[0]$ .

Die Variable  $a[0]$  hat den Typ  $T$ . Eine Variable vom Typ  $T$  benötigt  $s = \text{sizeof}(T)$  Byte Speicher.

Wenn also  $a[0]$  die Adresse  $p$  hat, dann besitzt  $a[1]$  die Adresse  $p + s$ . Und allgemein hat  $a[k]$  die Adresse  $p + k \cdot s$ .

Wenn  $r$  den Typ  $*T$  hat, und in  $r$  irgendeine Adresse  $p$  gespeichert ist, liefert der Ausdruck  $r+k$  die Adresse  $p + k \cdot s$ , wobei gilt  $s = \text{sizeof}(T)$ .

Den Wert  $s$  nennen wir manchmal auch *Objektgröße* einer Referenz.

# Felder und Referenzen

## Mehrdimensionale Felder

Das geht auch mit Feldern von Feldern:

```
int a [ 2 ] [ 3 ] [ 4 ] ;
```

sagt, dass `a` ein Feld mit zwei Elementen ist, wobei jedes Element eine  $3 \times 4$ -Matrix von `int`-Werten ist. Damit ist der Ausdruck `a` eine Referenz auf eine Variable vom Typ `int [ 3 ] [ 4 ]`: er referenziert die erste der beiden Matrizen im Feld `a`.

Die Deklaration

```
int (*b) [ 3 ] [ 4 ] ;
```

sagt, dass `b` eine Referenzvariable ist, die eine Referenz auf eine Variable vom Typ `int [ 3 ] [ 4 ]` speichern kann.

Damit ist die Zuweisung `b = a;` vollkommen in Ordnung.

Und es gilt `(* (b+1)) [ 2 ] [ 3 ] == a [ 1 ] [ 2 ] [ 3 ]`

Natürlich auch `(* (b+1)) [ 2 ] [ 3 ] == b [ 1 ] [ 2 ] [ 3 ]`

# Felder und Referenzen

## Mehrdimensionale Felder

Sei gegeben

```
int a[2][3][4];
```

Sei  $p$  die Adresse von  $a[0][0][0]$ . Welche Adressen, Typen und Objektgrößen  $s$  erhalten wir für folgende Ausdrücke?

Ausdruck :: Typ	Adresse	$s$	Referenziert
$a :: \text{int } (*)[3][4]$	$p$	48	$a[0]$
$a[0] :: \text{int } (*)[4]$	$p$	16	$a[0][0]$
$a[0][0] :: \text{int } *$	$p$	4	$a[0][0][0]$
$\&a :: \text{int } (*)[2][3][4]$	$p$	96	$a$
$a+1 :: \text{int } (*)[3][4]$	$p + 48$	48	$a[1]$
$\&a+1 :: \text{int } (*)[2][3][4]$	$p + 96$	96	erster Speicherplatz hinter $a$
$a[0][0]+1 :: \text{int } *$	$p + 4$	4	$a[0][0][1]$
$a[0][0][0]+1 :: \text{int}$	1	–	Wert von $a[0][0][0]+1$



# Felder als Funktionsargumente

Betrachten wir folgende Funktion:

```
#define N 2
#define M 3
#define K 4

int matmult(int A[N][K], int B[K][M], int C[N][M]) {
    int i, j, k;
    for(i=0; i<N; ++i)
        for(j=0; j<M; ++j)
            for(C[i][j]=0, k=0; k<K; C[i][j]+=A[i][k]*B[k][j], ++k);
}

int main() {
    int c[N][M], a[N][K], b[K][M];
    /* TODO initialize a,b */
    matmult(a,b,c);
    /* TODO output c */
}
```

Geht das? Natürlich: Im Aufruf von `matmult` werden ohnehin Referenzen übergeben (Warum?) Und wenn in der Deklaration eines *formalen* Parameters ein Feld angegeben wird, dann erwartet die Funktion hier eine *Referenz* auf das erste Feldelement.

# Felder als Funktionsargumente

Betrachten wir folgende Funktion:

```
#define N 2
#define M 3
#define K 4

int matmult(int A[N][K], int B[K][M], int C[N][M]) {
    int i, j, k;
    for(i=0; i<N; ++i)
        for(j=0; j<M; ++j)
            for(k=0; k<K; C[i][j]+=A[i][k]*B[k][j], ++k);
}

int main() {
    int c[N][M], a[N][K], b[K][M];
    /* TODO initialize a,b */
    matmult(a,b,c);
    /* TODO output c */
}
```

Mit anderen Worten: Felder werden in C nach dem *Call-by-Reference* Prinzip übergeben, nicht nach dem *Call-by-Value*-Prinzip. Daher kann die Prozedur `matmult` die Werte des Feldes `c` der Funktion `main` verändern.

# Felder als Funktionsargumente

Die äußere Felddimension (die am weitesten links stehende) kann bei formalen Argumenten auch entfallen:

```
#define N 2
#define M 3
#define K 4

int matmult(int A[][K], int B[][M], int C[][M]) {
    int i, j, k;
    for(i=0; i<N; ++i)
        for(j=0; j<M; ++j)
            for(C[i][j]=0, k=0; k<K; C[i][j]+=A[i][k]*B[k][j], ++k);
}

int main() {
    int c[N][M], a[N][K], b[K][M];
    /* TODO initialize a,b */
    matmult(a,b,c);
    /* TODO output c */
}
```

# Felder als Funktionsargumente

Ausgeschrieben passiert übrigens Folgendes:

```
#define N 2
#define M 3
#define K 4

int matmult(int (*A)[K], int (*B)[M], int (*C)[M]) {
    int i, j, k;
    for(i=0; i<N; ++i)
        for(j=0; j<M; ++j)
            for(k=0; k<K; ++k)
                (*C+i)[j] += (*A+i)[k] * (*B+k)[j];
}

int main() {
    int c[N][M], a[N][K], b[K][M];
    /* TODO initialize a,b */
    matmult(a,b,c);
    /* TODO output c */
}
```

# Felddeklarationen in formalen Argumenten

In einem formalen Argument sind die drei Deklarationen

$$T *a$$
$$T a[]$$
$$T a[n]$$

vollkommen gleichbedeutend. In jedem Fall ist  $a$  eine Referenzvariable, die eine Variable vom Typ  $T$  referenziert.

- Falls  $T$  der Typ `int` ist, hätten wir konkret: „`int *a`“, „`int a[]`“ und „`int a[42]`“

Warum 42? Warum nicht? – (die äußerste Felddimension ist in einem formalen Argument vollkommen irrelevant, da C keine Prüfung von Indexgrenzen durchführt.)

- Falls  $T$  der Typ `int [3]` ist, ergibt sich: „`int (*a)[3]`“, „`int a[][3]`“ und „`int a[42][3]`“

# Felddeklarationen in Variablen

Für eine lokale oder globale Variable sind die drei Deklarationen

$$T *a$$
$$T a []$$
$$T a [n]$$

*nicht* gleichbedeutend

- $T a [n]$  reserviert Speicherplatz für  $n$  Elemente vom Typ  $T$  und sorgt dafür, dass der Ausdruck  $a$  als Referenz auf das erste Element verwendet werden kann.
- $T *a$  reserviert *keinen* Speicherplatz für Elemente, sondern lediglich für eine Referenzvariable. Zu einem späteren Zeitpunkt muss der Wert dieser Referenzvariablen auf eine Referenz gesetzt werden.
- $T a []$  ist *nicht zulässig*. (Es würde bedeuten „reserviere Speicher für unbekannt viele Elemente vom Typ  $T$ “ – das geht natürlich nicht.)

# Felddeklarationen

## Variablen vs. formale Argumente

Warum sind unvollständige Felddeklarationen in der Form „ $T\ a[\ ]$ “ in formalen Argumenten erlaubt, aber nicht bei lokalen und globalen Variablen?

- Felder werden als Referenzen übergeben, also *Call-by-Reference*.
- Der Speicherplatz für das Feldargument existiert also bereits beim Aufruf.
- Damit wird die äußerste Felddimension nicht benötigt (man braucht sie nur um auszurechnen, wie viel Speicher für ein Feld reserviert werden muss).
- Bei der Deklaration von globalen / lokalen Variablen existiert der Speicher natürlich noch *nicht* (er wird eben erst durch die Deklaration reserviert).
- Damit ist in diesem Fall die äußerste Felddimension wichtig!

# Felddeklarationen

## Variablen vs. formale Argumente

Was auch bei Variablen erlaubt ist, sind Deklarationen der Form

$$T \ a[] = \{e_0, \dots, e_{N-1}\};$$

Wobei die  $e_i$  (konstante) Ausdrücke vom Typ  $T$  sind. Solche Deklarationen mit *Initialisierung* können vom Compiler leicht umgewandelt werden in Deklarationen der Form

$$T \ a[N] = \{e_0, \dots, e_{N-1}\};$$

bei denen klar ist, wie viel Speicher das Feld  $a$  benötigt.



# Felder und Referenzen

Referenzvariablen, die Referenzen auf Feldvariablen beinhalten, ermöglichen ziemlich spannende Konstruktionen in Zusammenhang mit den Inkrement- und Dekrement-Operatoren.

```
int a[4] = { 314, 159, 265, 358 };  
int *b, c;  
b = a+1; /* put reference to a[1]==159 into b */
```

Die Anweisung `c = *b++`; hat dann folgenden Effekt:

- Da `++` stärker bindet als `*` wird zunächst `b++` ausgewertet.
- Das Wert des Ausdrucks `b++` ist die aktuell in `b` gespeicherte Referenz (die Referenz auf `a[1]`), als *Seiteneffekt* steht in `b` danach eine Referenz auf `a[2]`.
- Der Wert des Ausdrucks `*b++` ist der Wert der Variablen, die durch den Wert des Ausdrucks `b++` referenziert wird – also der Wert der Variablen `a[1]`, das heißt: 159.
- Der Wert des Ausdrucks `*b++` wird dann `c` zugewiesen.
- Somit hat `c` anschließend den Wert 159 und `b` enthält eine Referenz auf `a[2]`.

# Felder und Referenzen

Was passiert dann in folgendem Programm?

```
#include <stdio.h>

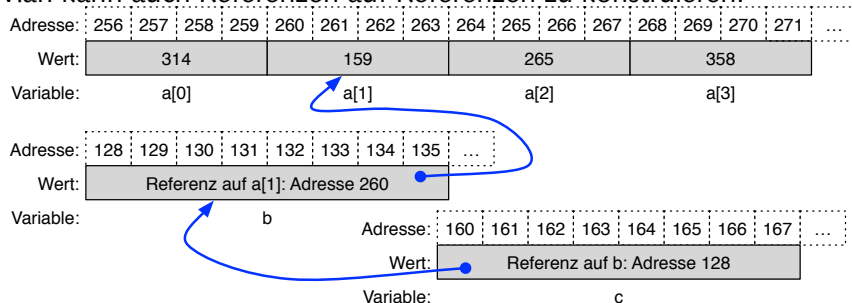
void strcpy(char *from, char *to) {
    while (*to++ = *from++)
        ;
}

char foo[] = "Hello World\n";
char baz[32];

int main() {
    strcpy(foo, baz);
    printf(baz);
    return 0;
}
```

# Indirekte Verweise

Man kann auch Referenzen auf Referenzen zu konstruieren:



```
int a[4] = {314, 159, 265, 358};
int *b;
int **c;
b = a+1;
c = &b;
printf("**c = %d\n", **c);
```

Welche Ausgabe erhalten wir? `**c = 159`

## Zeichenketten: `char` \*

Wir können auch Felder von Zeichen und Referenzen auf Zeichenvariablen definieren:

```
char a[] = { 'H', 'e', 'l', 'l', 'o', '\0' };  
char *b = a; /* Ref to a[0] */
```

`a` ist ein Feld mit sechs `char`-Variablen, das Zeichen `'\0'` markiert in C das Ende einer Zeichenkette. So muss die Länge der Zeichenkette nirgendwo gespeichert werden.

Es gibt hier eine abkürzende Definition, bei der man schreiben kann

```
char *b = "Hello";
```

Auch hier wird ein Feld mit sechs `char`-Variablen angelegt, das Feld wird mit den Zeichen in „Hello“ (sowie dem `'\0'`-Zeichen) initialisiert und eine Referenz auf das erste Feldelement in `b` abgelegt. Nur hat das Feld keinen Namen. Es ist lediglich über die Referenz in `b` erreichbar.

Dies ist die übliche Form, um vordefinierte Zeichenketten („Strings“) in C zu deklarieren.

# Zeichenketten: `char *`

Spaß mit `gcc`

Der `gcc` compiler legt via `char *` deklarierte Strings sogar tatsächlich in schreibgeschützten Speicherbereichen ab. Wenn man das nicht weiß, erlebt man Überraschungen:

Quelltext: `cconst.c`

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

```
char a[4] = "foo";
```

```
char b[] = "foo";
```

```
char *c = "bar";
```

```
char *d = "foo";
```

```
int main() {  
    a[0] = 'F';  
    printf("a = %s\n", a);  
    b[0] = 'F';  
    printf("b = %s\n", b);  
    c = d; /* can write the refs ... */  
    printf("c = %s\n", c);  
    c[0] = 'F'; /* ... but not the chars */  
    printf("c\'' = %s\n", c);  
    return 0;  
}
```

## Zeichenketten: `char` \*

Wir können auch Felder von Strings definieren:

Quelltext: *strings.c*

```
#include <stdio.h>

/* array of strings */
char *v[] = { "Dies", "ist", "ein", "kleiner", "Test" };

/* array of char arrays */
char w[][7] = { "Dies", "ist", "ein", "kleiner", "Test" };

int main() {
    /* number elements in 'c' */
    int n = sizeof v / sizeof(char *);
    int i;

    for(i=0; i<n; i++)
        printf("Word %d: %10s-%10s\n", i, v[i], w[i]);
}
```

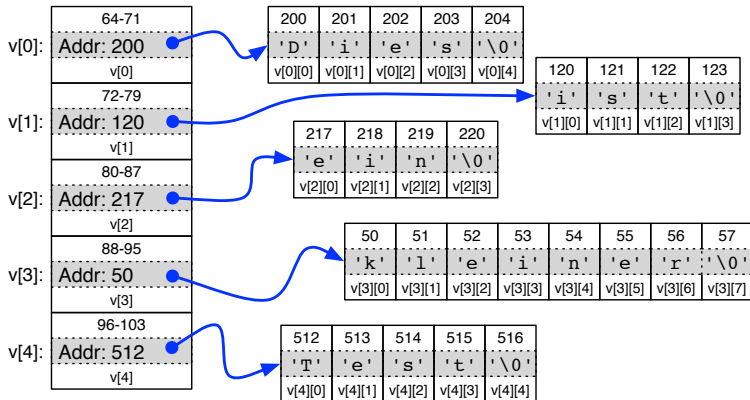
Frage: Worin unterscheiden sich `v` und `w`?

# Unterschiede zwischen `* []` und `[] [n]`

	100	101	102	103	104	105	106
w[0]:	'D'	'i'	'e'	's'	'\0'	'\0'	'\0'
	w[0][0]	w[0][1]	w[0][2]	w[0][3]	w[0][4]	w[0][5]	w[0][6]
	107	108	109	110	111	112	113
w[1]:	'i'	's'	't'	'\0'	'\0'	'\0'	'\0'
	w[1][0]	w[1][1]	w[1][2]	w[1][3]	w[1][4]	w[1][5]	w[1][6]
	114	115	116	117	118	119	120
w[2]:	'e'	'i'	'n'	'\0'	'\0'	'\0'	'\0'
	w[2][0]	w[2][1]	w[2][2]	w[2][3]	w[2][4]	w[2][5]	w[2][6]
	121	122	123	124	125	126	127
w[3]:	'k'	'l'	'e'	'i'	'n'	'e'	'r'
	w[3][0]	w[3][1]	w[3][2]	w[3][3]	w[3][4]	w[3][5]	w[3][6]
	128	129	130	131	132	133	134
w[4]:	'T'	'e'	's'	't'	'\0'	'\0'	'\0'
	w[4][0]	w[4][1]	w[4][2]	w[4][3]	w[4][4]	w[4][5]	w[4][6]

```
char w [ ] [ 7 ] = { "Dies", "ist", "ein",  
                     "kleiner", "Test" };
```

# Unterschiede zwischen \* [] und [] [n]



```
char *v[] = { "Dies", "ist", "ein",  
              "kleiner", "Test" };
```



# Zeichenketten: `char` \*

Bemerkenswert auch folgende Variante:

Quelltext: *strings2.c*

```
#include <stdio.h>

/* array of strings */
char *words[] = {
    "Dies", "ist", "ein",
    "kleiner", "Test",
    NULL
};

int main() {
    char **p;

    for(p = words; *p; ++p)
        printf("%s\n", *p);
}
```

Wie funktioniert das?

## Ergänzungen zu Schleifen: **do-while**

Die Anweisung „**do** *Anweisung* **while** (*Bedingung*) ;“ sorgt dafür, dass *Anweisung* *mindestens einmal* ausgeführt wird und dann wiederholt wird, solange *Bedingung* gilt.

Wie unterscheidet sich das von der **while**-Anweisung?

Beispiel: Ausgabe einer Zahl als Binärzahl

```
void printBinary (int n) {  
    while (n) {  
        printf ( "%d", n%2 );  
        n /= 2;  
    }  
}
```

Probleme? Die Binärziffern werden in umgekehrter Reihenfolge ausgegeben (aber das ignorieren wir) Ärgerlicher: Falls *n* den Wert 0 hat, wird nichts ausgegeben.

## Ergänzungen zu Schleifen: **do-while**

Lösung: statt **while**-Schleife,

```
void printBinary (int n) {  
    while (n) {  
        printf ( "%d", n%2 );  
        n /= 2;  
    }  
}
```

verwenden wir eine **do-while**-Schleife:

```
void printBinary (int n) {  
    do {  
        printf ( "%d", n%2 );  
        n /= 2;  
    } while (n);  
}
```

## Ergänzungen zu Schleifen: **break**

Manchmal ist es sinnvoll, wenn man eine Schleife vorzeitig beenden kann. Hierzu kann man die **break**-Anweisung verwenden.

Beispiel: terminiere Zeichenkette an erstem Leerzeichen.

Quelltext: *trim.c*

```
void trim (char *s) {  
    do /* Hairy formatting ... */  
        if (*s == ' ') {  
            *s = '\0';  
            break;  
        } while (*s++);  
}
```

Bei Ausführung der **break**-Anweisung wird die innerste Schleife, die das **break** enthält, unmittelbar beendet.

## Ergänzungen zu Schleifen: `continue`

Manchmal ist es sinnvoll, wenn man einen Schleifendurchlauf vorzeitig beenden kann. Hierzu kann man die `continue`-Anweisung verwenden. Beispiel:

```
int a[N];
int s;

/* compute sum of positive elements */
for (i=0; i<N; i++) {
    if (a[i] <= 0)
        continue ; /* skip non-positives */
    s = s+a[i];
}
```

Das Beispiel hier lässt sich natürlich auch sehr einfach ohne `continue` lösen!

## Heterogene Tupel: `struct`

Wir wollen Informationen über Personen speichern, wie Vorname, Nachname, Matrikelnummer, Geburtsjahr.

Aus der Mathematik kennen wir den Begriff des Tupels. Eine Person  $p$  wäre dort ein 4-Tupel  $p = (v, n, m, g)$ , wobei  $v$  der Vorname ist,  $n$  der Nachname,  $m$  die Matrikelnummer und  $g$  das Geburtsjahr.

So etwas geht auch in C:

```
struct _person {
    char v[16];
    char n[16];
    long m;
    int g;
} p;
```

# Heterogene Tupel: `struct`

```
struct _person {  
    char v[16];  
    char n[16];  
    long m;  
    int g;  
} p;
```

- Diese Deklaration besagt: die Variable `p` ist eine Strukturvariable mit Komponenten `v`, `n`, `m` und `g`.
- Mit dem Ausdruck „`p.m`“ können wir auf die Komponente `m` der Strukturvariable `p` zugreifen.
- Der Name `_person` ist der Name des zugehörigen Strukturtyps.
- Wir können ab sofort die Deklaration `struct _person q;` verwenden, um weitere Variablen dieses Strukturtyps zu deklarieren.

# Heterogene Tupel: struct

Quelltext: *addr1.c*

```
#include <stdio.h>

struct _person {
    char v[16];
    char n[16];
    long m;
    int g;
} p = { "Erika", "Mustermann", 213092495, 1996 };

int main() {
    printf( "%s %s %ld %d\n",
            p.v,
            p.n,
            p.m,
            p.g );
    return 0;
}
```



# Heterogene Tupel: struct

	512	513	514	515	516	517	518	519		527
p.v:	'E'	'r'	'i'	'k'	'a'	'\0'	'\0'	'\0'		'\0'
	p.v[0]	p.v[1]	p.v[2]	p.v[3]	p.v[4]	p.v[5]	p.v[6]	p.v[7]		p.v[15]
	528	529	530	531	532	533	534	535		543
p.n:	'M'	'u'	's'	't'	'e'	'r'	'm'	'a'		'\0'
	p.n[0]	p.n[1]	p.n[2]	p.n[3]	p.n[4]	p.n[5]	p.n[6]	p.n[7]		p.n[15]
	544-551									
p.m:	213092495									
	p.m									
	552-555									
p.g:	1996									
	p.g									

```

struct _person {
    char v[16];
    char n[16];
    long m;
    int g;
} p = {"Erika", "Mustermann",
      213092495, 1996};

```

# Heterogene Tupel: **struct**-Felder

Es können Felder von Strukturen deklariert werden und die Initialisierung von Strukturen erfolgt auf die offensichtliche Weise (analog zur Initialisierung von Feldern).

Quelltext: *addr2.c*

```
struct _person {  
    char v[16];  
    char n[16];  
    long m;  
    int g;  
} pa[] = {  
    { "John", "Doe", 210123456, 1991 },  
    { "Fred", "Feuerstein", 211987501, 1992 },  
    { "Erika", "Mustermann", 213092495, 1996 }  
};
```

# Heterogene Tupel: struct-Felder

Quelltext: *addr2.c*

```
#include <stdio.h>
/* insert def pa */

int main() {
    int i;

    for(i = 0; i < sizeof pa / sizeof pa[0]; i++)
        printf("%s %s %ld %d\n",
                pa[i].v, pa[i].n, pa[i].m, pa[i].g);

    return 0;
}
```

# Heterogene Tupel: **struct**-Felder

Strukturen als Funktionsargumente werden Call-by-Value übergeben.

Quelltext: *addr3.c*

```
void pprint(struct _person p) {  
    printf("%s %s %ld %d\n",  
           p.v, p.n, p.m, p.g);  
}  
  
int main() {  
    int i;  
  
    for (i=0; i < sizeof pa / sizeof pa[0]; i++)  
        pprint(pa[i]);  
    return 0;  
}
```

# Heterogene Vereinigung: union

Aus der Mathematik kennen wir die Vereinigungsmenge: seien  $C$  und  $I$  zwei Mengen, dann besteht die Menge  $U = C \cup I$  aus allen Elementen, die in  $C$  oder in  $I$  enthalten sind. Auch hierfür gibt es einen Datentyp in C:

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

```
union _u {  
    char c[4];  
    int i;  
} u = { "ABCD" };
```

```
int main() {  
    int c = u.i % 256; /* get lsb */  
    printf("%4s = %d; %s endian\n", u.c, u.i,  
           c == 'A' ? "little" : "big");  
    return 0;  
}
```

	u.c[0]	u.c[1]	u.c[2]	u.c[3]
u.c:	'A'	'B'	'C'	'D'
	512	513	514	515
u.i:	1145258561			
	u.i			

# Heterogene Vereinigung: `union`

- „Heterogene Vereinigungen“ sind auch als „variante Strukturen“ bekannt.
- Man kann sie verwenden, um *dynamische Typisierung* zu realisieren.

```
struct _dt {  
    int tag ; /* 0 = int active, 1 = float */  
    union {  
        int i;  
        float f;  
    } val;  
} dt;
```

Der Wert von `dt.tag` signalisiert, welche der Varianten von `dt.val` gerade aktiv ist.

Allerdings erzwingt hier C keine konsistente Verwendung des Tags (im Gegensatz zu z. B. Pascal)

# Aufzählungstypen: `enum`

Oft haben wir Anwendungen, in denen eine Variable eine von mehreren kategorialen Werten annehmen kann – etwa eine Variable, die den Zustand einer Ampel repräsentiert, die „rot“, „rot-gelb“, „gelb“ und „grün“ leuchten kann.

Hierfür bietet C den *Aufzählungstyp* an:

Quelltext: *lights.c*

```
#include <stdio.h>

enum LightState {Red, RedYellow, Yellow, Green} ampel;
enum Reaction {Brake, Accelerate, Cruise, Prepare};

enum Reaction react(enum LightState l) {
    if(l==Red) return Brake;
    else if(l==RedYellow) return Prepare;
    else if(l==Yellow) return Accelerate;
    else return Cruise;
}

int main() {
    for(ampel=Red; ampel<=Green; ampel++)
        printf("%d => %d\n", ampel, react(ampel));
    return 0;
}
```

# Aufzählungstypen: `enum`

`enum`-Konstanten können als `int`-Werte verwendet werden. Voreinstellung ist aufsteigende Nummerierung der Konstanten, beginnend mit 0. Dies kann aber geändert werden:

Quelltext: *lights2.c*

```
#include <stdio.h>
enum LightState {Red=-7, RedYellow, Yellow=8, Green} ampel;
enum Reaction {Brake, Accelerate, Cruise, Prepare, Panic};

enum Reaction react(enum LightState l) {
    if(l==Red) return Brake;
    else if(l==RedYellow) return Prepare;
    else if(l==Yellow) return Accelerate;
    else if(l==Green) return Cruise;
    else return Panic;
}

int main() {
    for(ampel=Red; ampel<=Green; ampel++)
        printf("%d => %d\n", ampel, react(ampel));
    return 0;
}
```



# Mehrwege-Verzweigung: **switch**

Der Rumpf von `react` ist ein Beispiel für eine Mehrwege-Verzweigung: ein Ausdruck kann eine (überschaubare) Menge von konstanten Werten annehmen, und je nach tatsächlichem Wert wird eine andere Anweisung(sfolge) ausgeführt

```
if (l==Red)   return Brake;
else if (l==RedYellow) return Prepare;
else if (l==Yellow) return Accelerate;
else if (l==Green) return Cruise;
else return Panic;
```

Für diesen Fall bietet C die **switch**-Anweisung an:

```
switch (l) {
case Red: return Brake;
case RedYellow: return Prepare;
case Yellow: return Accelerate;
case Green: return Cruise;
default: return Panic;
}
```

Hierdurch werden insbesondere die Vergleiche der **else-if**-Kette vermieden: Der passende Fall kann direkt gewählt werden.

# Mehrwege-Verzweigung: **switch**

- Ein **case**-Label hat die Form  
**case** *ConstantExpression* :  
Dabei ist *ConstantExpression* ein konstanter ganzzahliger Ausdruck (z. B. vom Typ **int**, **char** oder ein **enum**).
- Eine *CaseAnweisung* ist eine Anweisung, vor der eine beliebige Anzahl von **case**-Labels steht.  
Beispiele für *CaseAnweisungen*:  

```
i=i+1;  
case 17: i=i+1;  
case 1: case 2: case 17: i=i+1;
```
- Ein *CaseBlock* ist ein unbenannter Block, der aus *CaseAnweisungen* besteht.
- Die Anweisung „**switch** (*Ausdruck*) *CaseBlock*“ berechnet den Wert von *Ausdruck* und springt zur ersten *CaseAnweisung* im *CaseBlock*, die ein Case-Label mit dem berechneten Wert hat.

# Mehrwege-Verzweigung: **switch**

- Gibt es im *CaseBlock* keine Anweisung mit einem **case**-Label, das auf den berechneten Wert passt, wird *keine* der Anweisungen im *CaseBlock* ausgeführt.
- Das **case**-Label „**default:**“ passt auf *alle* möglichen Werte.
- Es werden sequentiell *alle Anweisungen* vom passenden **case**-Label bis zum Ende des *CaseBlocks* ausgeführt.
- Soll die Ausführung *vorzeitig* beendet werden (zum Beispiel, vor der nächsten Anweisung, die ein Label trägt), kann hierfür die **break**-Anweisung verwendet werden.
  - Dies ist tatsächlich der *Regelfall*
  - Warum konnten wir im Ampel-Beispiel auf das **break** verzichten?

# Mehrwege-Verzweigung: switch

```
enum Reaction react2(enum LightState l) {  
    enum Reaction r;  
  
    switch(l) {  
        case Red:  
            r = Brake;  
            break;  
        case RedYellow:  
            r = Prepare;  
            break;  
        case Yellow:  
            r = Accelerate;  
            break;  
        case Green:  
            r = Cruise;  
            break;  
        default:  
            r = Panic;  
            break;  
    }  
    return r;  
}
```

# struct, union, enum

## Deklarationen

- Eine Deklaration der Form

**struct** *l* { ... } *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

**union** *l* { ... } *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

**enum** *l* { ... } *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

definiert einen neuen Struktur-(Varianten-, Aufzählungs-)Typ mit Namen *l* und erzeugt Variablen *v*<sub>1</sub>, ..., *v*<sub>*n*</sub> dieses Typs.

- Eine Deklaration der Form

**struct** *l* { ... };

**union** *l* { ... };

**enum** *l* { ... };

definiert lediglich den Typ.

- Für einen bereits definierten Typ *l*, erzeugt die Deklaration

**struct** *l* *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

**union** *l* *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

**enum** *l* *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>;

neue Variablen *v*<sub>1</sub>, ..., *v*<sub>*n*</sub> dieses Typs.

## Typsynonyme: typedef

In C können schnell ziemlich komplizierten Typen entstehen, zum Beispiel in der Deklaration:

```
struct _person { ... } *(*foo)[4][7];
```

(Diese Deklaration besagt, dass `foo` eine Referenz auf eine  $3 \times 4$ -Matrix von Referenzen auf `_person`-Strukturen ist – die Deklaration der Strukturkomponenten wurden aus Platzgründen weggelassen.)

Wenn wir jetzt an unterschiedlichen Stellen Variablen von diesem Typ deklarieren wollen, ist das unbequem. (Und wenn der Typ irgendwann verändert werden muss, wird das sehr aufwändig, da wir das an *allen* Programmstellen ändern müssen.)

Daher bietet C die Möglichkeit an, *Typsynonyme* zu definieren – symbolische Namen für komplizierte Typen.

## Typsynonyme: `typedef`

Wenn wir eine Variablendeklaration mit einem komplizierten Typ haben,

- `struct _person { ... } *(*foo)[4][7];`

wird daraus folgendermaßen die Definition eines Typsynonyms:

- Wir schreiben *vor* die Deklaration `typedef`,
- Wir ersetzen *in* der Deklaration den Variablennamen durch das Typsynonym, das wir definieren wollen.
- Also

- `typedef struct _person { ... } *(*FooType)[4][7];`

- Und wir können danach `FooType` wie jeden anderen Typnamen in C für die Deklaration von Variablen verwenden:

```
int i;  
FooType foo;
```

# Lvalues und Rvalues

Ausdrücke in C können *Speicherorte* oder *Werte* bedeuten.

- Ein Ausdruck, der einen *Speicherort* bedeutet, darf auf der *linken Seite* des Zuweisungsoperators auftreten („Lvalue“).
- Ein Ausdruck, der einen *Wert* bedeutet, kann nur auf der *rechten Seite* des Zuweisungsoperators auftreten („Rvalue“).
- Tritt ein Lvalue-Ausdruck in Rvalue-Position auf (z. B. ein Variablenname auf der rechten Seite einer Zuweisung), wird automatisch der im Lvalue abgelegte Wert als Rvalue verwendet.

```
int a, b;
```

```
a = b;    /* ok */
```

```
b = a;    /* ok */
```

```
a = 17;   /* ok */
```

```
17 = a;   /* NOT OK */
```

- Der Referenz-Operator „&“ kann nur auf Lvalue-Ausdrücke angewendet werden.



# Lvalues und Rvalues

```
int a, b, *c;  
a = 17;  
b = 42;  
c = &a;
```

Ausdruck	Lvalue	Rvalue Wert
a	ja	17 (Wert von a)
c	ja	Adresse von a
*c	ja	17
c+1	nein	Adresse hinter a (vielleicht Adresse von b)
*(c+1)	ja	vielleicht 42, vielleicht Speicherfehler
*c+1	nein	18

# Zeiger auf Funktionen

Wir wollen berechnen

$$a = \sum_{i=0}^{N-1} \sin(x_i^2) \quad \text{und} \quad b = \sum_{i=0}^{N-1} \log(x_i^2)$$

Was bemerken wir? Die Berechnung hat die allgemeine Form

$$\sum_{i=0}^{N-1} f(x_i^2)$$

Wobei wir sie konkret für  $f = \sin$  und  $f = \log$  auswerten. Wir könnten dann schreiben:

$$g(f) = \sum_{i=0}^{N-1} f(x_i^2)$$

und somit

$$a = g(\sin) \quad \text{und} \quad b = g(\log)$$

# Zeiger auf Funktionen

Wie geht das in C?

Quelltext: *fptr.c*

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

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

```
double x[] = {1.0, 2.0, 3.0, 4.0, 5.0, M_PI};
```

```
double g(double (*f)(double)) {
```

```
    int i;
```

```
    double s;
```

```
    for (i=0, s=0; i<sizeof x / sizeof x[0]; i++)
```

```
        s += f(x[i]*x[i]);
```

```
    return s;
```

```
}
```

```
int main() {
```

```
    printf("a = %f\n", g(sin));
```

```
    printf("b = %f\n", g(log));
```

```
    return 0;
```

```
}
```

# Zeiger auf Funktionen

Wie geht das in C?

- Eine Typdeklaration der Form

$$T_r \quad (*f) \quad (T_{a_1}, \dots, T_{a_n})$$

besagt: „Die Variable  $f$  enthält die Referenz auf eine Funktion, die  $n$  Argumente der Typen  $T_{a_1}, \dots, T_{a_n}$  akzeptiert und ein Ergebnis vom Typ  $T_r$  liefert.“

- Falls nun  $h$  der Name einer solchen Funktion ist, dann speichert die Zuweisung

$$f = \&h; \quad \text{oder auch} \quad f = h;$$

eine Referenz auf die Funktion  $h$  in  $f$ .

- Die Anweisung

$$r = (*f) (a_1, \dots, a_n); \text{ bzw. } r = f(a_1, \dots, a_n);$$

führt dann den Aufruf der Funktion  $h$  mit den Argumenten  $a_1, \dots, a_n$  aus und speichert das Ergebnis in  $r$ .

# Dynamischer Speicher

- Mit lokalen oder globalen Variablen können wir nur Speicher reservieren, dessen Größe wir zur *Compilezeit* wissen.
- In vielen Anwendungen kann man allerdings erst zur Laufzeit des Programms feststellen, wie viel Speicherplatz das Programm für die Erfüllung seiner Aufgabe benötigt.
  - Beispiel: Ein Programm, das eine Liste von Werten sortieren soll, die in einer Datei geliefert werden.
  - Natürlich weiß der Programmentwickler nicht, wie groß die Datei maximal ist ...
- Lösungsansatz: Anforderung von dynamischen Speicher:
  - „System, reserviere für mich Speicher für eine Variable vom Typ  $T$  und gib mir eine Referenz auf diesen Speicher“
  - In C wird dies durch folgenden Ausdruck bewerkstelligt:

$(T \ *)\text{malloc}(\text{sizeof}(T))$

- Eine Referenz auf ein Feld mit  $N$  Elementen vom Typ  $T$  erhält man entsprechend durch:

$(T \ *)\text{malloc}(\text{sizeof}(T) * N)$

# Dynamischer Speicher

## Einige Beispiele

```
int *a;

/* get memory for 100 ints */
a = (int *) malloc(sizeof(int) * 100);

/* use memory */
a[0]      = 42;
*(a+17)   = 23;    /* same as: a[17] = 23; */
a[99]     = 17+4;  /* access last element */
a[100]    = 0;     /* MEMORY ACCESS ERROR */

/* release memory */
free(a);    /* return memory to pool */
a[0] = 17;  /* MEMORY ACCESS ERROR */
free(a);    /* MEMORY ACCESS ERROR */
```

Das gefährliche bei Referenzen, insbesondere im Zusammenspiel mit `malloc` und `free`, ist, dass Zugriffsfehler nicht unbedingt sofort entdeckt werden. Speicherfehler sind berüchtigt schwierig zu finden.

# Dynamischer Speicher

## Verwendung

- Um `malloc` und `free` zu verwenden, muss der Header `stdlib.h` im Programm inkludiert werden.

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

- Die Funktion `malloc` liefert als Ergebnis eine Referenz vom Typ `void *`. Das heißt „Referenz auf unbekannten Typ“. Um dem C-Compiler den richtigen Referenztyp mitzuteilen, muß dieser in Klammern vor den Aufruf von `malloc` geschrieben werden. Dies wird auch als *Cast* bezeichnet.
- Kann der gewünschte Speicher nicht zur Verfügung gestellt werden, liefert `malloc` den Wert `NULL` zurück. Dieser Wert hat in Bedingungen die gleiche Bedeutung wie der Wert „0“:

```
int *a;  
if (!(a = (int *) malloc(sizeof(int)*10))) {  
    fprintf(stderr, "No memory left. Dying miserably.\n");  
    exit(-1);  
}  
/* use memory */
```

# Dynamischer Speicher

## Speicherlecks

Folgendes Programm wird, wenn es lange genug läuft, *jeden* Rechner in die Knie zwingen, auf dem `glibc` ab Version 2.7 als C Standardbibliothek eingesetzt wird.

```
#include <stdio.h>
char *p;
int main() {
    for (;;) {
        scanf ("%ms", &p);
        printf ("Input read: %s\n", p);
    }
}
```

Warum?



# Eine Studentenkartei

Wir haben eine Textdatei `studenten.dat`, die in jeder Zeile einen Eintrag für einen Studenten, bestehend aus Name, Vorname, Geburtsjahr und Matrikelnummer beinhaltet.

Inhalt von `studenten.dat`:

Doe John 1991 210123456

Feuerstein Fred 1992 211987501

Mustermann Erika 1996 213092495

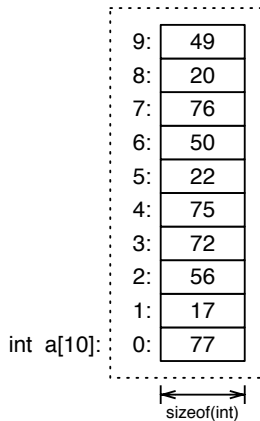
*... unbekannte Anzahl weiterer Einträge ...*

Wir wollen `studenten.dat` mit einem Programm in den Speicher einlesen und dann nach Namen sortiert ausgeben.

Für das Sortieren wollen wir die Funktion `qsort` verwenden, die in der C Standardbibliothek eingebaut ist.

# qsort

Sortieren von `int`-Feldern

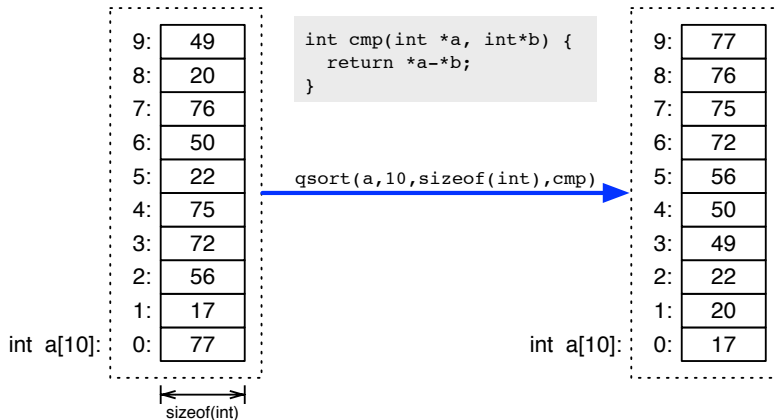


```
int cmp(int *a, int*b) {  
    return *a-*b;  
}
```

```
qsort(a,10,sizeof(int),cmp)
```

# qsort

Sortieren von `int`-Feldern



# qsort

## Naives Sortieren von großen Strukturen

S a[10]:

9:	49	??3
8:	20	??9
7:	76	??5
6:	50	??7
5:	22	??1
4:	75	??8
3:	72	??2
2:	56	??0
1:	17	??4
0:	77	??6

key    blah

sizeof(S)

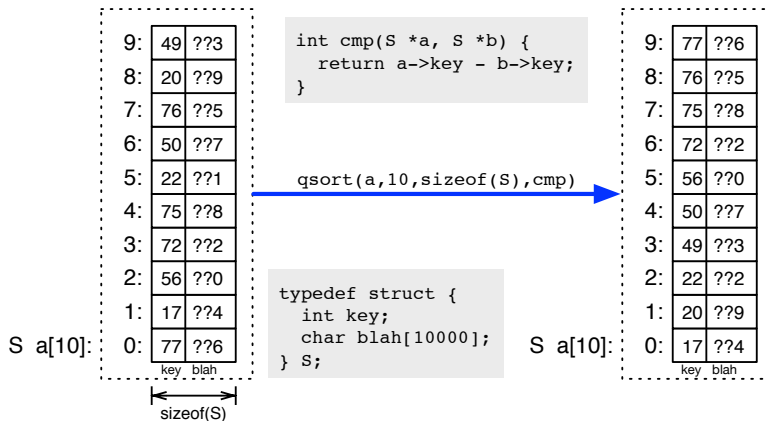
```
int cmp(S *a, S *b) {  
    return a->key - b->key;  
}
```

```
qsort(a,10,sizeof(S),cmp)
```

```
typedef struct {  
    int key;  
    char blah[10000];  
} S;
```

# qsort

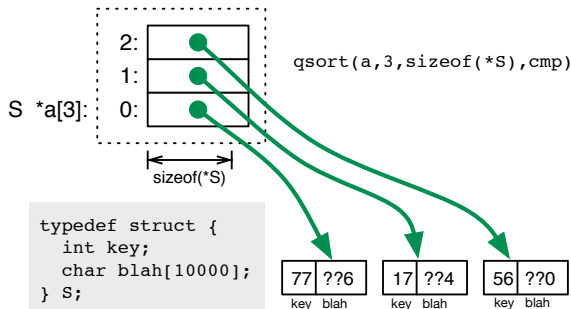
## Naives Sortieren von großen Strukturen



# qsort

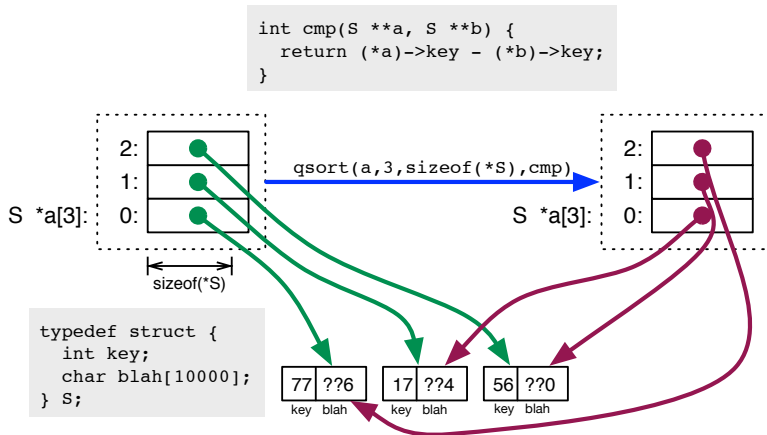
## Effizientes Sortieren von großen Strukturen: Sortieren von Zeigern

```
int cmp(S **a, S **b) {  
    return (*a)->key - (*b)->key;  
}
```



# qsort

## Effizientes Sortieren von großen Strukturen: Sortieren von Zeigerfeldern



# Studentenkartei

Lösungsidee für das Einlesen einer unbekannten Anzahl von Datensätzen.

- Zum Speichern von Datensätzen wird eine C **struct** definiert

```
typedef struct _person {  
    struct _person *next;  
    ... /* data for this record */  
} PersonRec, *Person;
```

- Es wird ein Listenzeiger `Person personList`; definiert und mit `NULL` initialisiert.
- Für jeden Datensatz wird mit Hilfe von `malloc` neuer Speicher reserviert, mit den Daten des Datensatzes gefüllt ...

```
Person new;  
new = (Person) malloc ( sizeof (PersonRec) );  
new->... = ... ;
```


- ... und durch folgende Anweisungen eingekettet:

```
new->next=personList;  
personList=new;
```



# Studentenkartei: Beispiel

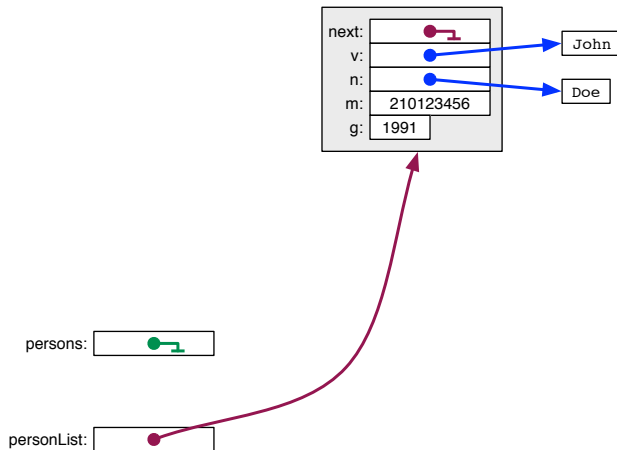
## Ausgangszustand

persons: 

personList: 

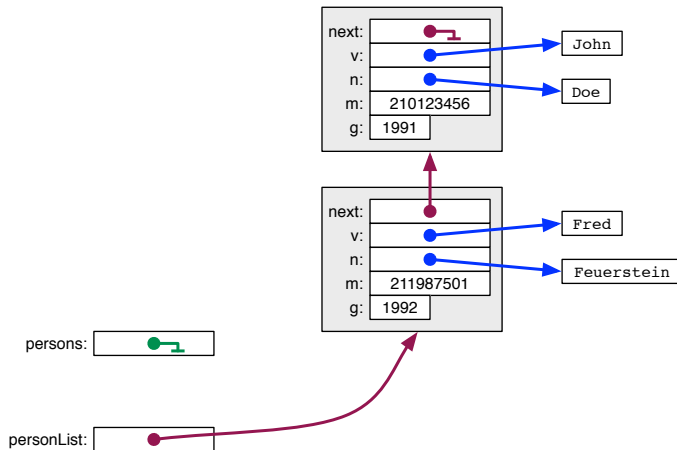
# Studentenkartei: Beispiel

Nach Einfügen von John Doe



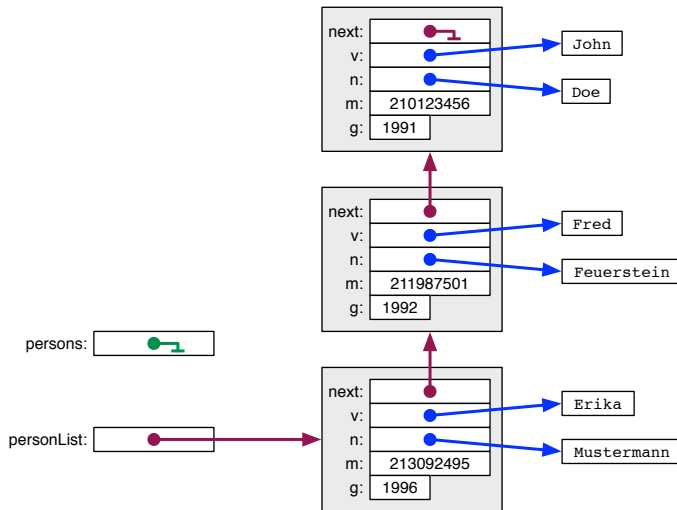
# Studentenkartei: Beispiel

Nach Einfügen von Fred Feuerstein



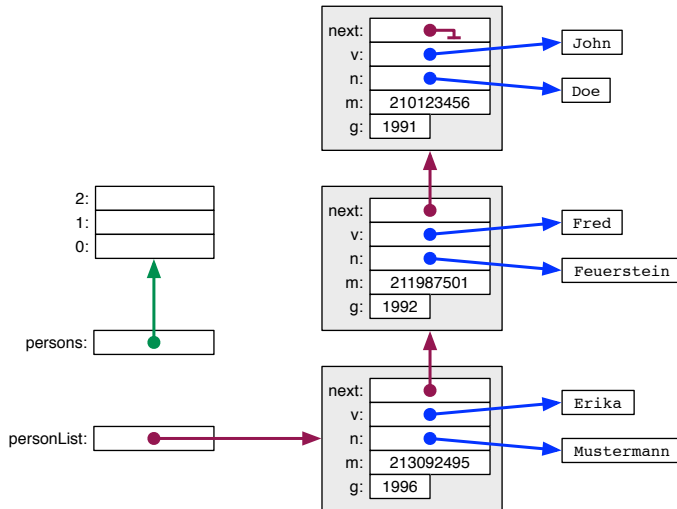
# Studentenkartei: Beispiel

Nach Einfügen von Erika Mustermann



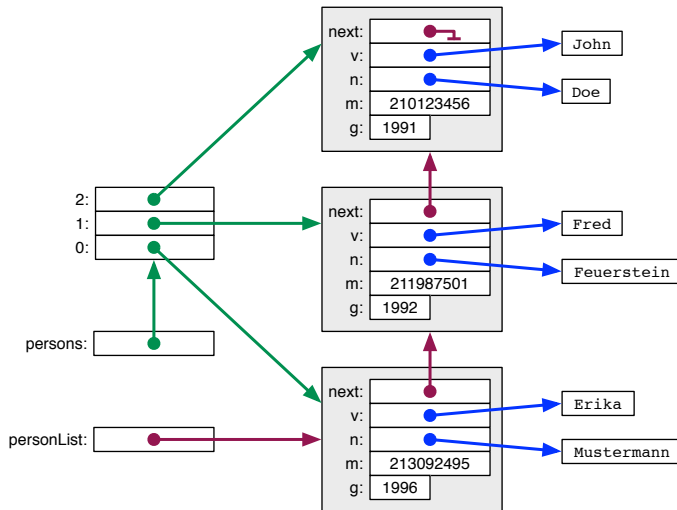
# Studentenkartei: Beispiel

Nach Allokation Zeigerfeld



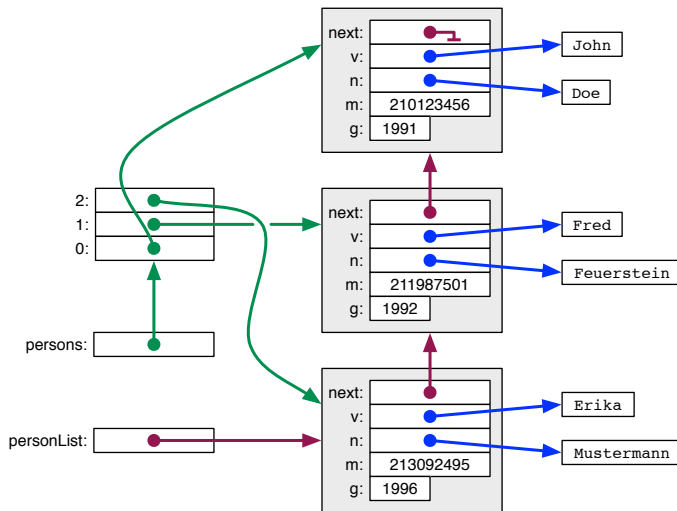
# Studentenkartei: Beispiel

Nach Auffüllen Zeigerfeld



# Studentenkartei: Beispiel

Nach Sortieren Zeigerfeld



# Studentenkartei: Quelltext

Quelltext: *addrfile2.c*

```
#include <stdio.h>
#include <stdlib.h> /* for qsort, malloc */
#include <string.h> /* for strcoll */
#include <locale.h> /* for setlocale */

typedef struct _person {
    struct _person *next;
    char *v;
    char *n;
    long m;
    int g;
} PersonRec, *Person;

Person *persons;
Person personList = NULL;
int nPersons = 0;
```



# Studentenkartei: Quelltext

```
Person makePerson(char *v, char *n, long m, int g) {  
    Person p = (Person)malloc(sizeof(PersonRec));  
  
    p->v = strdup(v);  
    p->n = strdup(n);  
    p->m = m;  
    p->g = g;  
  
    p->next = personList;  
    personList = p;  
    nPersons++;  
  
    return p;  
}
```

# Studentenkartei: Quelltext

```
int readFile() {
    char v[100], n[100];
    long m; int g;
    Person p; Person *pp;

    while (scanf("%99s %99s %ld %d", n, v, &m, &g) == 4)
        makePerson(v, n, m, g);
    /* initialize pointers */
    persons = (Person *) malloc(nPersons
                                * sizeof(Person));
    for (p=personList, pp=persons; p; p=p->next)
        *pp++ = p;

    return nPersons;
}
```

# Studentenkartei: Quelltext

```
int compare(const void *a, const void *b) {
    return strcoll ((* (Person *) a)->n,
                    (* (Person *) b)->n);
}

void pprint() {
    Person *p;

    for(p = persons; p < persons+nPersons;p++)
        printf("%s %s %ld %d\n",
               (*p)->n, (*p)->v, (*p)->m, (*p)->g);
}

int main() {
    setlocale(LC_ALL, "");
    fprintf(stderr, "read %d records\n", readFile());
    qsort(persons, nPersons, sizeof(Person), compare);
    pprint();
    return 0;
}
```

# Studentenkartei: Quelltext

## Anmerkung zu `strcoll`

- Die Funktion `int strcoll(char *, char *)` liefert den lexikographischen Vergleich zweier Zeichenketten. (Siehe ISO C Standard, ISO/IEC 9899)
- Der Aufruf `setlocale(LC_ALL, "");` sorgt dafür, dass die aktuell gültigen Landeseinstellungen verwendet werden.
- Damit sollte `strcoll` auch mit UTF-8-Zeichenketten zurechtkommen, bei denen zum Beispiel Umlaute durch mehrere aufeinanderfolgende `char`-Werte dargestellt werden („Multibyte-Characters“).
- Dies ist jedoch nicht immer der Fall: Sowohl Apple OSX wie auch Microsoft Windows 7 bieten keine korrekte Unterstützung der Sortierung von UTF-8-Zeichenketten. – Im Gegensatz zu Linux!
- Wie finden Sie das als Programmentwickler??
- Workaround: Funktionieren tut es unter Windows und OSX mit einfacheren Ein-Byte-Codierungen (z.B. ISO8859-1)

# In diesem Skript nicht diskutiert

- Weitere Deklarationen wie `static`, `const`, und `extern`
- Details zu Casts.
- Deklarationen in Header-Files; Module