

### 3 Module in C

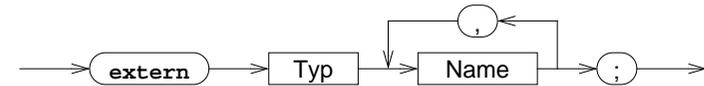
#### !!! .c-Quelldateien auf keinen Fall mit Hilfe der #include-Anweisung in andere Quelldateien einkopieren

- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
  - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefaßt
  - ◆ Header-Dateien werden mit der #include-Anweisung des Preprozessors in C-Quelldateien einkopiert
  - ◆ der Name einer Header-Datei endet immer auf **.h**

### 5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden (**extern-Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;
extern char c;
```

### 4 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
  1. Global im gesamten Programm (über Modul- und Funktionsgrenzen hinweg)
  2. Global in einem Modul (auch über Funktionsgrenzen hinweg)
  3. Lokal innerhalb einer Funktion
  4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
  - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
  - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

### 5 Globale Variablen (2)

- Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne daß der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➔ **globale Variablen möglichst vermeiden!!!**

## 5 Globale Funktionen

- Funktionen sind generell global (es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden (= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort `extern` ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:
 

```
double sinus(double);
float power(float, int);
```
- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
  - "vertragliche" Zusicherung an den Benutzer des Moduls

## 7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluß auf die Zugreifbarkeit von Variablen

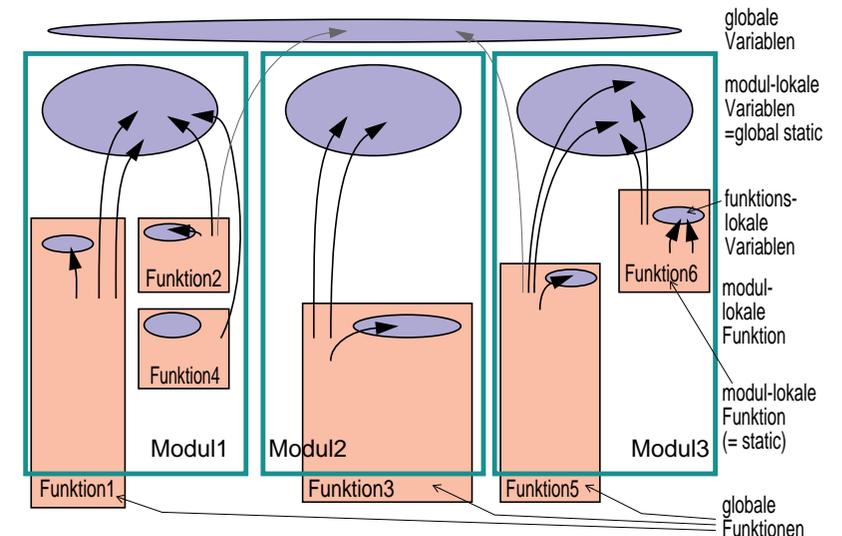
## 6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
    - Schlüsselwort `static` vor die Definition setzen
- ```

  → (static) → Variablen-/Funktionsdefinition →
  
```
- `extern`-Deklarationen in anderen Modulen sind nicht möglich
  - Die `static`-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
  - Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer `static` definiert werden
    - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)

- !!! das Schlüsselwort `static` gibt es auch bei lokalen Variablen (mit anderer Bedeutung!)

## 8 Gültigkeitsbereiche — Übersicht



## 9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
    - statische (**static**) Variablen
  - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
    - dynamische (**automatic**) Variablen

## 9 Lebensdauer von Variablen (2)

### static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort **static** eine **Lebensdauer über die gesamte Programmausführung** hinweg
  - der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- !!! Das Schlüsselwort **static** hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
  - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
  - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

## 9 Lebensdauer von Variablen (2)

### auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
- !!! **wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

## 10 Wertaustausch zwischen Funktionen

| Mechanismus       | Aufrufer → Funktion | Funktion → Aufrufer   |
|-------------------|---------------------|-----------------------|
| Parameter         | ja                  | mit Hilfe von Zeigern |
| Funktionswert     | nein                | ja                    |
| globale Variablen | ja                  | ja                    |

- Verwendung globaler Variablen?
  - ◆ Variablen, die von vielen Funktionen verwendet werden und/oder oft als Parameter übergeben werden müßten
    - Menge der Funktionen muß überschaubar bleiben
    - ➔ Zugriff auf Modul begrenzen (globale static-Variablen)
    - **sonst sehr schlechter Programmierstil**
  - ◆ Variablen, die keiner Funktion als Variable oder Parameter fest zugeordnet werden können
    - Modul suchen, dem die Variable zugeordnet werden kann!!!
  - ◆ Variablen, deren Lebensdauer nicht beschränkt sein darf, die aber nicht in **main()** deklariert werden sollen
    - in zugehöriger Funktion lokal-static definieren

## 11 Getrennte Übersetzung von Programmteilen — Beispiel

### ■ Hauptprogramm (Datei fplot.c)

```
#include "trig.h"
#define INTERVALL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
    char c;
    double i;

    printf("Funktion (Sin, Cos, Tan, cOt)? ");
    scanf("%x", &c);

    switch (c) {
        ...
        case 'T':
            for (i=-PI/2; i < PI/2; i+=INTERVALL)
                printf("%lf %lf\n", i, tan(i));
            break;;
        ...
    }
}
```

## 11 Getrennte Übersetzung — Beispiel (3)

### ■ Trigonometrische Funktionen (Datei trigfunc.c)

```
#include "trig.h"

double tan(double x) {
    return(sin(x)/cos(x));
}

double cot(double x) {
    return(cos(x)/sin(x));
}

double cos(double x) {
    return(sin(PI/2-x));
}

...

```

## 11 Getrennte Übersetzung — Beispiel (2)

### ■ Header-Datei (Datei trig.h)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```

## 11 Getrennte Übersetzung — Beispiel (4)

### ■ Trigonometrische Funktionen — Fortsetzung (Datei trigfunc.c)

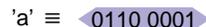
```
...
double sin (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

## 1 Einordnung

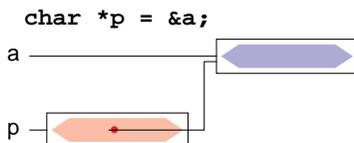
■ **Konstante:**  
Bezeichnung für einen Wert



■ **Variable:**  
Bezeichnung eines Datenobjekts



■ **Zeiger-Variable (Pointer):**  
Bezeichnung einer Referenz auf ein Datenobjekt



## 2 Überblick

■ Eine Zeigervariable (**pointer**) enthält als Wert die Adresse einer anderen Variablen

↳ der Zeiger verweist auf die Variable

■ Über diese Adresse kann man **indirekt** auf die Variable zugreifen

■ Daraus resultiert die große Bedeutung von Zeigern in C

↳ Funktionen können ihre Argumente verändern (**call-by-reference**)

↳ dynamische Speicherverwaltung

↳ effizientere Programme

■ Aber auch Nachteile!

↳ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)

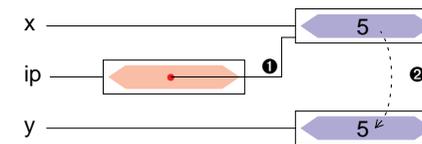
↳ häufigste Fehlerquelle bei C-Programmen

■ Syntax:

Typ \*Name ;

▲ Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ①
y = *ip; ②
```



## 4 Adreßoperatoren

▲ Adreßoperator &

&x der unäre Adreß-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) x

▲ Verweisoperator \*

\*x der unäre Verweisoperator \* ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger x verweist

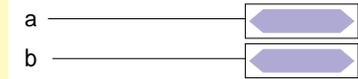
## 5 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adreßverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des *\**-Operators auf die zugehörige Variable zugreifen und sie verändern  
↳ *call-by-reference*

## 5 ... Zeiger als Funktionsargumente (2)

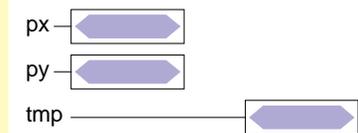
- Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b);
}
```



```
void swap (int *px, int *py)
{
  int tmp;

  tmp = *px;
  *px = *py;
  *py = tmp;
}
```



## 5 ... Zeiger als Funktionsargumente (2)

- Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b);
}
```



## 5 ... Zeiger als Funktionsargumente (2)

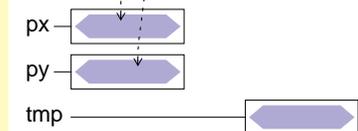
- Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b); ①
}
```



```
void swap (int *px, int *py)
{
  int tmp;

  tmp = *px;
  *px = *py;
  *py = tmp;
}
```



## 5 ... Zeiger als Funktionsargumente (2)

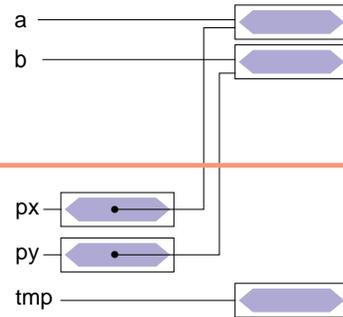
C-8 Zeiger(-Variablen)

■ Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b);
}

void swap (int *px, int *py)
{
  int tmp;

  tmp = *px;
  *px = *py;
  *py = tmp;
}
```



## 5 ... Zeiger als Funktionsargumente (2)

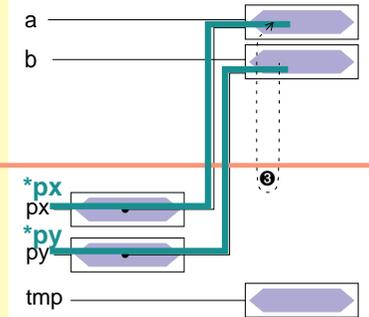
C-8 Zeiger(-Variablen)

■ Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b); ①
}

void swap (int *px, int *py)
{
  int tmp;

  tmp = *px;
  *px = *py; ②
  *py = tmp;
}
```



## 5 ... Zeiger als Funktionsargumente (2)

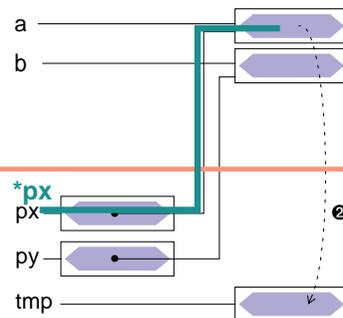
C-8 Zeiger(-Variablen)

■ Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b); ①
}

void swap (int *px, int *py)
{
  int tmp;

  tmp = *px; ②
  *px = *py;
  *py = tmp;
}
```



## 5 ... Zeiger als Funktionsargumente (2)

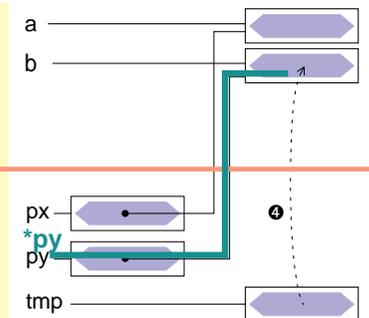
C-8 Zeiger(-Variablen)

■ Beispiel:

```
main(void) {
  int a, b;
  void swap (int *, int *);
  ...
  swap(&a, &b); ①
}

void swap (int *px, int *py)
{
  int tmp;

  tmp = *px; ②
  *px = *py; ③
  *py = tmp; ④
}
```



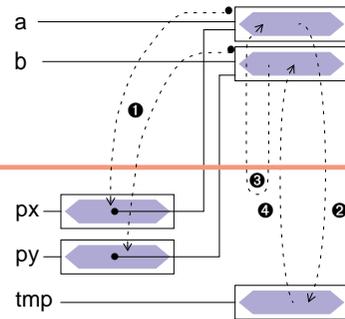
## 5 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

```
main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④
}
```



## 6 Zeiger auf Strukturen (2)

■ Zugriff auf Strukturkomponenten über einen Zeiger

■ Bekannte Vorgehensweise

- \*-Operator liefert die Struktur
- .-Operator zum Zugriff auf Komponente
- Operatorenvorrang beachten

```
➔ (*pstud).best = 'n'; unleserlich!
```

■ Syntaktische Verschönerung

- ➔ ->-Operator

```
pstud->best = 'n';
```

## 6 Zeiger auf Strukturen

■ Konzept analog zu "Zeiger auf Variablen"

- Adresse einer Struktur mit &-Operator zu bestimmen
- Zeigerarithmetik berücksichtigt Strukturgröße

■ Beispiele

```
struct student stud1;
struct student *pstud;
pstud = &stud1; /* => pstud -> stud1 */
```

■ Besondere Bedeutung zum Aufbau verketteter Strukturen

## 7 Zusammenfassung

■ Variable

```
int a;
a — 5
```

■ Zeiger

```
int *p = &a;
a — 5
p — [ ]
```

■ Struktur

```
struct s {int a; char c;};
struct s s1 = {2, 'a'};
s1 — [ 2 | a ]
```

■ Zeiger auf Struktur

```
struct s *sp = &s1;
s1 — [ 2 | a ]
sp — [ ]
```

## C-9 sizeof-Operator

C-9 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
  - ▶ z. B. zum Anfordern von Speicher für ein Feld (→ malloc)

### ■ Syntax:

`sizeof x` liefert die Größe des Objekts `x` in Bytes  
`sizeof (Typ)` liefert die Größe eines Objekts vom Typ `Typ` in Bytes

- Das Ergebnis ist vom Typ `size_t` (entspricht meist `int`)  
(#include <stddef.h>!)

### ■ Beispiel:

```
int a; size_t b;  
b = sizeof a; /* => b = 2 oder b = 4 */  
b = sizeof(double) /* => b = 8 */
```

SOS

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-13 13.39

C.61

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## C-10 Explizite Typumwandlung — Cast-Operator

C-10 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;  
float f = 0.2;  
double d;
```

`d = (i * f);`  
→float  
→double

- In manchen Fällen wird eine explizite Typumwandlung benötigt  
(vor allem zur Umwandlung von Zeigern)

### ◆ Syntax:

(Typ) Variable

### Beispiele:

```
(int) a (int *) a  
(float) b (char *) a
```

SOS

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-13 13.39

C.62

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## C-11 Speicherverwaltung

C-11 Speicherverwaltung

- `void *malloc(size_t size)`: Reservieren eines Speicherbereiches
- `void free(void *ptr)`: Freigeben eines reservierten Bereiches

```
struct person *p1 = (struct person *) malloc(sizeof(struct person));  
if (p1 == NULL) {  
    perror("malloc person p1");  
    ...  
}  
...  
free(p1);
```

- malloc-Prototyp ist in `stdlib.h` definiert (#include <stdlib.h>)

SOS

Softwaresysteme I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-13 13.39

C.63

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## C-12 Felder

C-12 Felder

### 1 Eindimensionale Felder

- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefaßt werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

```
int x[5];  
double f[20];
```

SOS

Softwaresysteme I

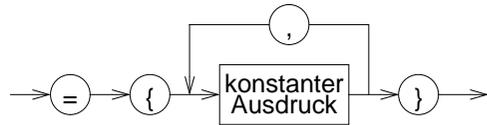
© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2005

C-Einf.fm 2005-04-13 13.39

C.64

Reproduktion jeder Art oder Verwendung dieser Unterlagen, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## 2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'o', 't', 't', 'o', '\0'};
```

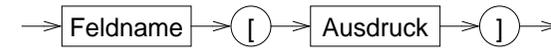
- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};
char name[] = {'o', 't', 't', 'o', '\0'};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

## 4 Zugriffe auf Feldelemente

- Indizierung:



wobei:  $0 \leq \text{Wert}(\text{Ausdruck}) < \text{Feldgröße}$

- Beispiele:

```
prim[0] == 2
prim[1] == 3
name[1] == 't'
name[4] == '\0'
```

## 3 ... Initialisierung eines Feldes (2)

- Felder des Typs **char** können auch durch String-Konstanten initialisiert werden

```
char name1[5] = "Otto";
char name2[] = "Otto";
```

## 5 Mehrdimensionale Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
- Definition eines mehrdimensionalen Feldes



- Beispiel:

```
int matrix[4][4];
```