



Universität  
Rostock



Traditio et Innovatio

# Programmiersprache C

## Einführungskurs

Prof. Dr.-Ing. habil. Gero Mühl

Architektur von Anwendungssystemen (AVA)  
Fakultät für Informatik und Elektrotechnik (IEF)  
Universität Rostock

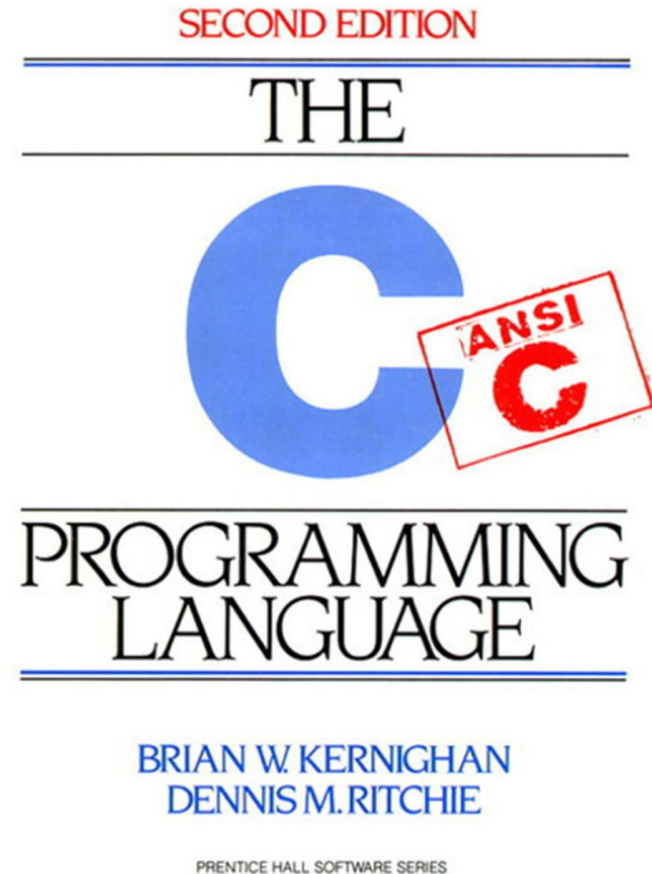
# Übersicht

- > Einführung
  - > Programmiersprache C
  - > Programmentwicklung
- > Grundlegende Konstrukte
  - > Programm, Bezeichner, Variable, Typ
  - > Anweisung, Zuweisung, Ausdruck
- > Kontrollstrukturen
  - > Bedingte Anweisung (if, if-else) und Anweisungsblock
  - > Mehrfachauswahl (switch)
- > Schleifen
  - > for, while, do-while
- > Zeiger
- > Prozeduren / Funktionen
  - > Deklaration, Aufruf, Parameterübergabe
- > Felder (Arrays)
  - > Eindimensionale Felder
  - > Mehrdimensionale Felder
- > Structs
- > Rekursion

# Einführung

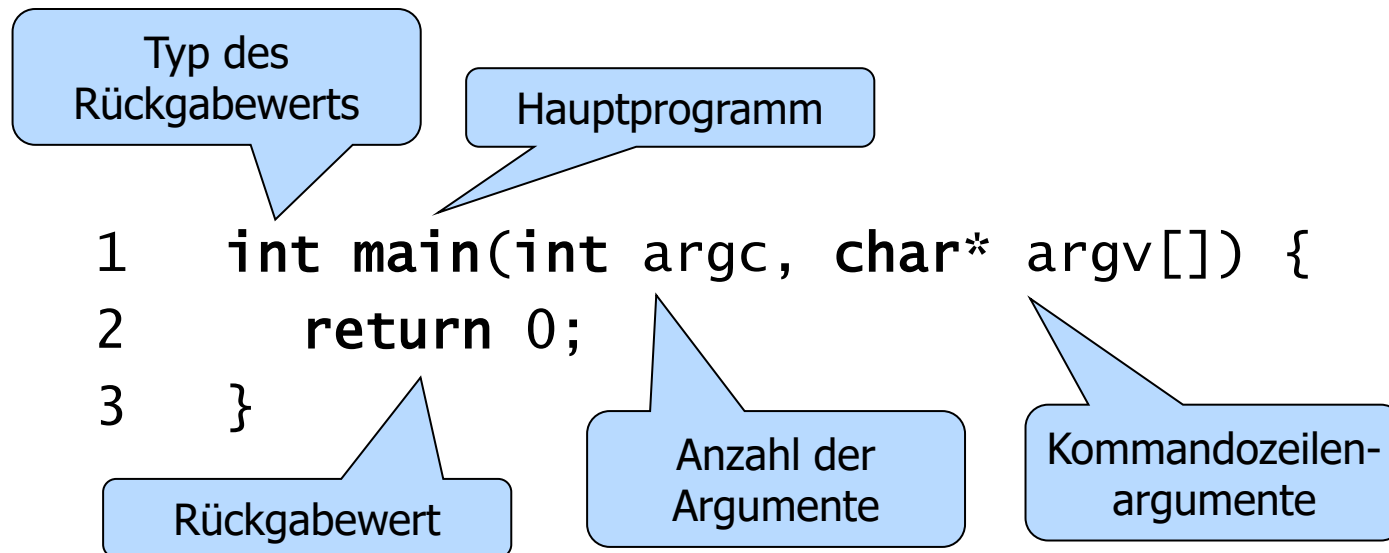
# Programmiersprache C

- > Imperativ und maschinennah
- > Kompakt, klein und effizient
- > Portabel und weit verbreitet  
(auf jedem Betriebssystem)
- > Keine automatische  
Speicherverwaltung
- > Echte Zeiger
- > Keine Objektorientierung
- > Syntaktische Grundlage vieler  
anderer Programmiersprachen  
(Objective C, C++, Java, C#, Awk, Perl, PHP, ...)



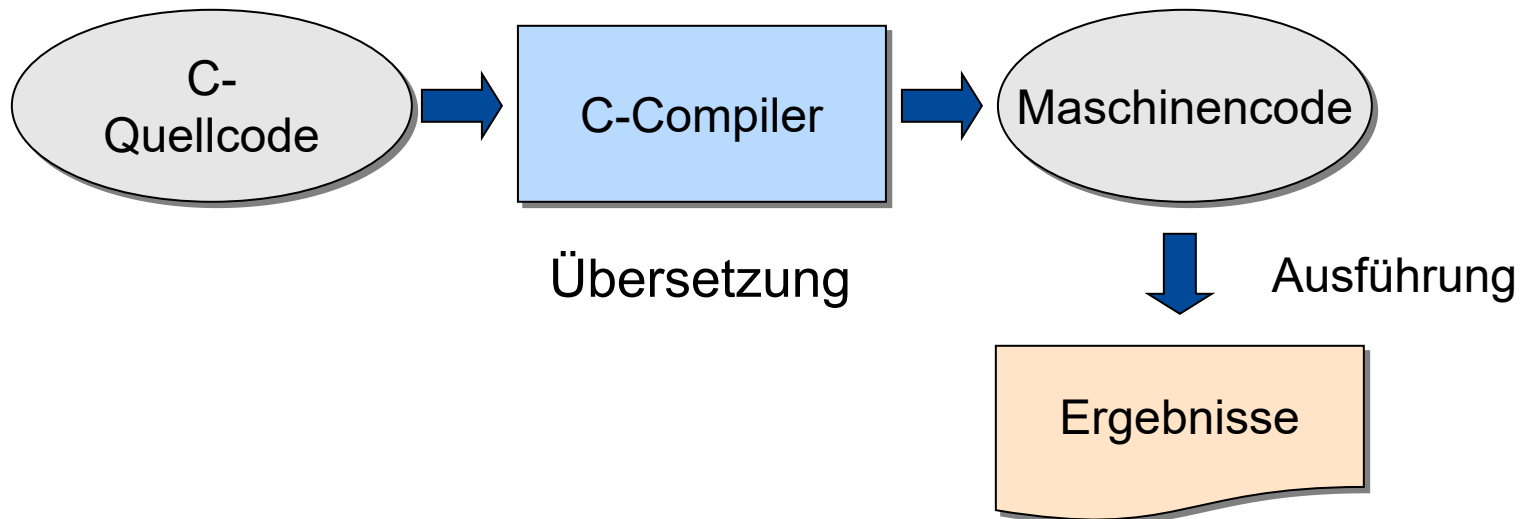
# Einfaches C-Programm

- > Das einfachste C-Programm besteht lediglich aus einer `main` Prozedur.
- > Das untenstehende Programm ist korrekt, tut aber nichts.
- > Reservierte **Schlüsselwörter** sind **fettgedruckt**.
- > Schlüsselwörter dürfen nicht für anderes verwendet werden.

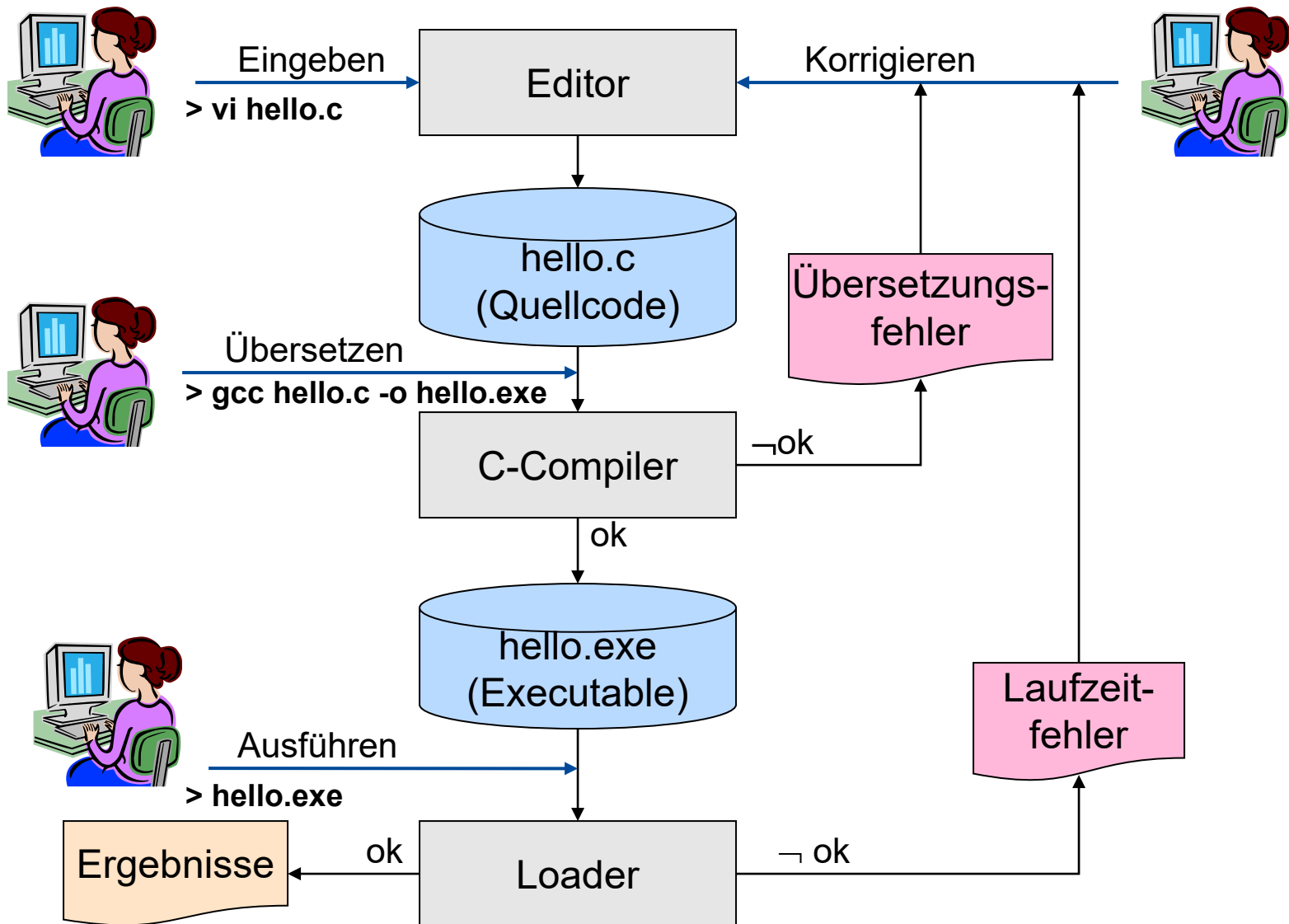


# Ausführung von C-Programmen

- > C-Programme werden in die **Maschinensprache** der **Zielformat** (Prozessor, Betriebssystem, ...) übersetzt.
- > Der **Maschinencode** kann dann auch nur auf dieser Plattform ausgeführt werden und ist nicht portabel.
- > Ausnahme sind **Cross-Compiler**, die Maschinencode für eine andere Plattform erzeugen.



# Ablauf des Programmierens



# Grundlegende Konstrukte



# Noch ein einfaches C-Programm

- > C-Programme bestehen Quellcodedateien (z.B. `hello.c`).
- > Diese enthalten **Prozeduren** (procedures), welche die „Arbeit“ erledigen, z. B. Berechnungen durchführen.
- > Eine Prozedur enthält in ihrem **Rumpf** (eingeschlossen in `{}`) eine Sequenz von **Anweisungen** (statements).
- > Die Anweisungen werden nacheinander ausgeführt.

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

Einbinden einer  
Bibliothek

```
int main(int argc, char* argv[]) {  
    printf("Hello world!");  
    return 0;  
}
```

Ausgabe auf dem  
Terminal

# Kommentare

- > Ändern den Programmablauf nicht.
- > Wichtig für andere, die das Programm verstehen wollen.
- > Auch für den Programmierer selbst, der nach wenigen Wochen nicht mehr weiß, was im Code genau geschieht.
- > Kommentiert wird direkt beim Programmieren.
- > Nur **sinnvolle Kommentare** verwenden, die eine Zusatzinformation enthalten.

```
/* multiline  
comment */  
n = 2024; // bad: n is assigned 2024  
n = 2024; // better: n is the current year
```

# Bezeichner

- > Prozeduren, Typen, Variablen etc. müssen durch eindeutige **Bezeichner** (identifizier) benannt werden.
- > Bezeichner sind in C mit Einschränkungen frei wählbar
  - > Keine Schlüsselwörter
  - > Erstes Zeichen aus: a-z, A-Z, \_
  - > Weitere Zeichen aus: 0-9, a-z, A-Z, \_
- > Groß-/Kleinschreibung wird unterschieden
- > C-Konvention für Bezeichner
  - > Prozeduren: klein                      `main, list_init`
  - > Konstanten: alle Zeichen groß        `MAX_BUFFER_SIZE`
  - > Variablen: klein                      `anzahl_elemente`

# Variablen

- > Während der Programmausführung entstehen neue Werte, die in Variablen gespeichert werden können.

```
....  
printf("%i\n", 2000 + 22 );  
....
```

Ausdruck (expression)

Literale (literals)

binärer Operator

- > Stattdessen kann auch geschrieben werden

```
....  
int year;           // declaration of variable year  
year = 2000 + 24;   // assignment of value to year  
printf("%i\n", year); // using value in print  
....
```

# Ausgabe mit printf

%i	Integer
%6i	Integer, mindestens 6 Zeichen breit
%f	Fließkommazahl
%7f	Fließkommazahl, mindestens 7 Zeichen breit
%.2f	Fließkommazahl, 2 Zeichen nach dem Dezimalpunkt
%7.2f	Fließkommazahl, mindestens 7 Zeichen breit und 2 Zeichen nach dem Dezimalpunkt

```
double x = 444.24;
double y = 234.22;
char* f = "%7.2f * %7.1f = %12.3f\n";
printf(f, x, y, x * y); // 444.24 * 234.2 = 104049.893
printf(f, x, y, x + y); // 444.24 + 234.2 = 678.460
printf(f, x, y, x - y); // 444.24 - 234.2 = 210.020
printf(f, x, y, x / y); // 444.24 / 234.2 = 1.897
                        // 1234567 1234567 123456789012
                        //      12      1      123
```

# Eingabe mit scanf

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    int n;
    while (scanf("%d", &n) == 1)
        printf("%3d\n", n);
    return 0;
}
```

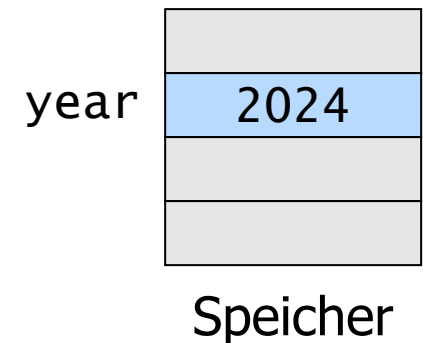
> Eingabe: 456 123 789 456 12<return>

> Ausgabe: 456  
123  
789  
456  
12

# Variablen und Typen

- > Jede Variable ist von einem bestimmten **Typ** und muss vor ihrer Verwendung **deklariert** werden.
- > Der Typ legt fest, welche **Werte** die Variable annehmen kann.
- > `int year` bedeutet, dass die Variable `year` nur **ganzzahlige Werte** (integer) annehmen kann.
- > Im Prinzip gibt eine Variable eine **Speicherzelle** an, die einen Wert enthält.

```
// declaration and initialization  
int year = 2024;
```



# Programm mit Variablen

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    int a, b, s;
    printf("First number? ");
    scanf("%d", &a);        // read number from stdin
    printf("Second number? ");
    scanf("%d", &b);        // read number from stdin
    s = a + b;
    printf("%d + %d = %d.\n", a, b, s);
}
```



# Basistypen

Datentyp	Speicherplatz	Wertemenge
char	8 bits	-128 ... 127
short	16 bits	-32.768 ... 32.767
int	32 bits	-2.147.483.648 ... 2.147.483.647
float	32 bits	$\pm 1.4\text{E}-45$ ... $\pm 3.4028235\text{E}+38$
double	64 bits	$\pm 4.9\text{E}-324$ ... $\pm 1.7977\text{E}+308$
unsigned char	8 bits	0 ... 255
unsigned short	16 bits	0 ... 65.535
unsigned int	32 bits	0 ... 4.294.967.295

> Auch bei Literalen kann der Typ angegeben werden

```
3.14           // default is double
3.14f          // f or F indicates float
0xff           // 0x indicates hexadecimal number
1e-9           // double with mantissa & decimal exponent
```

# Grenzen der Wertebereiche

- > Die Header-Datei `float.h` enthält die Grenzwerte der Zahlentypen.

```
#define DBL_DIG 15 // # of decimal digits of precision
#define DBL_MAX 1.7976931348623158e+308 // max value
#define DBL_MAX_10_EXP 308 // max decimal exponent
#define DBL_MIN 2.2250738585072014e-308 // min positive value
#define DBL_MIN_10_EXP (-307) // min decimal exponent
#define FLT_DIG 6 /* # of decimal digits of precision
#define FLT_MAX 3.402823466e+38F // max value
#define FLT_MAX_10_EXP 38 // max decimal exponent
#define FLT_MIN 1.175494351e-38F // min positive value
#define FLT_MIN_10_EXP (-37) // min decimal exponent
#define SHRT_MIN (-32768) // min (signed) short value
#define SHRT_MAX 32767 // max (signed) short value
#define INT_MIN (-2147483647 - 1) // min (signed) int value
#define INT_MAX 2147483647 // max (signed) int value
#define LONG_MIN (-2147483647L - 1) // min (signed) long value
#define LONG_MAX 2147483647L // max (signed) long value
```

# Basistypen

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    char a;
    short b;
    int c;
    float e;
    double f;
    printf("size(char)      = %d\n", sizeof(a));      // 1
    printf("size(short)     = %d\n", sizeof(b));      // 2
    printf("size(int)        = %d\n", sizeof(c));      // 4
    printf("size(float)       = %d\n", sizeof(e));      // 4
    printf("size(double)    = %d\n", sizeof(f));      // 8
}
```

# Operatoren

+	Addition	}
-	Subtraktion	
*	Multiplikation	}
/	Division	
%	Modulo-Operator	}
-	Vorzeicheninvertierung (unär)	}

Vorrang



Bei gleichrangigen Operatoren wird von links nach rechts ausgewertet.

**Signaturen** der binären Operatoren:

`short x short -> short`

`int x int -> int`

`float x float -> float`

`double x double -> double`

# Ausdrücke (expressions)

- > Neue Werte entstehen durch Auswertung von **Ausdrücken**.
- > Ausdrücke werden (ähnlich mathematischen Ausdrücken) nach einer vorgegebenen Syntax gebildet.
- > Ein Ausdruck wird durch Einsetzen der aktuellen Variablenwerte ausgewertet:

```
double celsius = 0.0;  
double fahrenheit = 94.1;
```

<b>Zustand vorher</b>	fahrenheit	94.1	celsius	0.0
-----------------------	------------	------	---------	-----

```
celsius = ((5.0 / 9.0) * (fahrenheit - 32.0));
```

<b>Zustand nachher</b>	fahrenheit	94.1	celsius	34.5
------------------------	------------	------	---------	------

# Mathematische Ausdrücke in C

## Mathematische Schreibweise

$$y = mx + c$$

$$x = (a - b)(a + b)$$

$$y = 3[(a - b)(a + b)] - x$$

$$y = 1 - \frac{2a}{3b}$$

$$b = -a$$

## C-Schreibweise

$$y = m * x + c;$$

$$x = (a - b) * (a + b);$$

$$y = 3 * ((a - b) * (a + b)) - x;$$

$$y = 1 - (2 * a) / (3 * b);$$

$$b = -a;$$

# Typkonvertierung

- > Was passiert, wenn in einem Ausdruck Variablen unterschiedlichen Typs verknüpft werden?

```
int    n = 4;  
double x = 3.14 + n;    // 7.14
```

- > Es erfolgt eine implizite (automatische) Typkonvertierung.
- > Auch möglich, aber problematisch

```
double d = 3.14;  
int    n = 4 + d;    // 7
```

- > Um den Verlust von Genauigkeit und falsche Ergebnisse zu vermeiden, müssen Über- und Unterläufe durch den richtigen Zieltyp vermieden werden

```
float f = 1e9;  
int   x = 4 * f    // wrong -2147483648
```

# Typkonvertierung

```
int c = 128000;  
printf("%i\n", c);           // 128000  
printf("%i\n", (short)c);    // wrong -30727
```

```
double x1 = (3.0 + 7.0)  
           / (4.0 * 5.0 + 80.0);    // = 0.1
```

```
double x2 = (3.0 + 7)  
           / (4 * 5 + 80);           // = 0.1
```

```
double x3 = (double)(3 + 7)  
           / (4 * 5 + 80);           // = 0.1
```

```
double x4 = (3 + 7)  
           / (4 * 5 + 80);           // = 0
```



# Probleme mit Zahlen ohne Vorzeichen

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    int i = -1;
    unsigned int u = 1;

    int x = i * u;
    printf("%i\n", x);                // -1 OK

    double d = i * u;
    printf("%.1f\n", d);              // 4294967295.0 wrong
}
```

# Probleme mit Fließkommazahlen

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    double x = 81.0;
    while (x != 0.0) {
        printf("%.14f\n", x); // 0.0000000000000006
        if (x <= -3.0)        // -2.9999999999999994
            break;
        x -= 1.0 / 3.0;
    }
}
```

# Anweisungen (statements)

- > C-Programme bestehen aus Anweisungen, die nacheinander ausgeführt werden.
- > C wird daher auch als **imperative Sprache** bezeichnet
- > Anweisungen werden durch ein **Semikolon** getrennt.
- > Arten von Anweisungen
  - > Zuweisung                      ändert den Wert einer Variablen
  - > Bedingte Anweisung            führt eine Anweisung nur aus, wenn eine Bedingung erfüllt ist
  - > Schleife                         wiederholt eine Anweisung oder einen Anweisungsblock
  - > Prozeduraufruf                 ruft eine Prozedur auf
  - > ...

# Zuweisung (assignment)

- > Eine Zuweisung weist einer Variablen einen neuen Wert zu.
- > Beispiel:  $x = 3 + y;$
- > Allgemeine Form:  
 $\langle \text{assignment} \rangle ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle$
- > Diese Schreibweise heißt **Backus-Naur-Form (BNF)** und wird zur Syntax-Definition von Programmiersprachen benutzt.
- > Sie beschreibt die Syntax in rekursiver Weise  
 $\langle \text{komplexes Konstrukt} \rangle ::= \langle \text{einfachere Konstrukte} \rangle$
- > Alternativ werden auch **Syntax-Diagramme** benutzt:



# Zuweisung

- > Eine Zuweisung besteht aus zwei Teilen
  1. der Auswertung des **Ausdrucks** auf der **rechten Seite**
  2. der Speicherung des Ergebnisses der Auswertung in der Variablen auf der **linken Seite**

```
int x;  
  ↓  
x = 5;  
  ↓  
int y;  
  ↓  
y = 10 - 4 * 2;  
  ↓  
y = x + 2 * (x + y);
```

Zustand    x   

Zustand    x   

Zustand    x        y   

Zustand    x        y   

Zustand    x        y

# Zuweisungs- vs. Gleichheitsoperator

- > In C ist = der Zuweisungsoperator  
(und nicht der Gleichheitsoperator ==):

`x = 5;`                      `x` wird der Wert 5 zugewiesen

`x = x + 1;`                Wert von `x` wird um 1 erhöht

- > In C kann also eine Variable nacheinander verschiedene Werte annehmen.
- > Dies ist ein wichtiger Unterschied zur funktionalen Programmierung!

# Beispiel: Swap

- > Die Werte zweier Variablen sollen vertauscht werden (swap):

```
// swap values of x and y
```

```
x = y;
```

```
y = x;
```

Zustand	x	5	y	7
---------	---	---	---	---

Zustand	x	7	y	7
---------	---	---	---	---

Zustand	x	7	y	7
---------	---	---	---	---

- > Es wird eine Hilfsvariable zum Zwischenspeichern benötigt

**Falsch!**

```
// swap values of x and y
```

```
int z;
```

Zustand	x	5	y	7	z	?
---------	---	---	---	---	---	---

```
z = x;
```

Zustand	x	5	y	7	z	5
---------	---	---	---	---	---	---

```
x = y;
```

Zustand	x	7	y	7	z	5
---------	---	---	---	---	---	---

```
y = z;
```

Zustand	x	7	y	5	z	5
---------	---	---	---	---	---	---

# Inkrementieren und Dekrementieren

- > Die Operatoren ++ und -- gibt es in einer **vorangestellten** und in einer **nachgestellten** Form.
- > Sie erhöhen, bzw. erniedrigen, den Wert einer Variablen um 1.
- > Bei ++n **vor der Verwendung** des Werts von n, bei n++ **danach**.
- > Diese Operatoren können auch als Ausdrücke verwendet werden

```
int i = 1;
int j = i++; // increment i after assignment
printf("i = %i, j = %i", i, j);
// prints i = 2, j = 1
```

- > Es gibt weitere ähnliche Operatoren (+=, -=, \*=, /=)

```
int i = 1;
i += 2;
i *= 3;
printf("%i\n", i); // prints 9
```



# Kontrollstrukturen

# Bedingte Anweisung

- > Oft soll eine Anweisung (oder ein Anweisungsblock) nur ausgeführt werden, wenn eine gegebene **Bedingung** erfüllt ist.

- > Syntax der bedingten Anweisung

```
if (<integer expression>) <statement>
```

- > Beispiel: Berechnung des Absolutwerts einer Variable

```
if (x < 0)
```

```
    x = -x;
```

- > Ablauf der bedingten Anweisung

1. Werte Ausdruck aus.

2. Falls Ergebnis ungleich 0, führe Anweisung (oder Block) aus.

- > Meist wird aber ein **boolescher Ausdruck** verwendet, dessen Ergebnis nur 1 oder 0 (wahr oder falsch) sein kann:

```
int x = 5 < 7           // 1
```

```
int y = 5 > 7           // 0
```

# Bedingte Anweisung mit Alternative

- > Allgemeine Form der if-Anweisung mit Alternative

```
if (<integer expression>
    <statement>
else
    <statement>
```

- > Abhängig von der Auswertung wird entweder die eine (bei != 0) oder die andere Anweisung (bei == 0) ausgeführt.
- > Beispiel: Bildung des Maximums zweier Werte

```
// set z to maximum of x and y
if (x > y)           // condition
    z = x;           // then part
else
    z = y;           // else part
```

Kurzform mit ?-Operator:  
`z = x > y ? x : y;`

# Dangling else

```
z = 0;  
if (x == 2)  
    if (y == 2)  
        z++;  
else  
    z--;
```

- > Falsche Einrückung des else-Zweiges
- > Unübersichtlich durch fehlende Klammerung

```
z = 0;  
if (x == 2) {  
    if (y == 2)  
        z++;  
    else  
        z--;  
}
```

```
z = 0;  
if (x == 2) {  
    if (y == 2)  
        z++;  
} else  
    z--;
```

# Mehrfachauswahl mittels `if-else`

- > Das `if-else`-Konstrukt wird verwendet, um abhängig von Bedingungen zwischen verschiedenen Blöcken zu wählen

```
if (n == 1) { // execute block #1
    ...
} else if (n == 2) { // execute block #2
    ...
} else if (n == 3) { // execute block #3
    ...
} else { // if all fails, execute block #4
    ...
}
```

# Mehrfachauswahl mittels if-else

```
#include <stdio.h>
int main() {
    int n;
    printf("Number?\n");
    scanf("%d", &n); // read integer number from stdin
    if (n < 0) {
        printf("%d is a negative number.\n", n);
    } else if (n > 0) {
        printf("%d is a positive number.\n", n);
    } else {
        printf("%d has a value of zero.\n", n);
    }
}
```

# Mehrfachauswahl mittels `switch`

- > Eine Mehrfachauswahl kann auch mit der `switch`-Anweisung realisiert werden:

```
switch (n) {  
    case 1:                // execute block #1  
        ...  
        break;            // stop here  
    case 2:                // execute block #2  
        ...  
        break;            // stop here  
    case 3:                // execute block #3  
        ...  
        break;            // stop here  
    default:               // if all fails,  
        ...                // execute block #4  
        break;            // stop here  
}
```

# Logische Ausdrücke (boolean expressions)

## > Vergleichsoperatoren geben Werte

vom Typ `int` (0 oder 1) zurück

<code>==</code>	(gleich)	<code>!=</code>	(ungleich)
<code>&lt;</code>	(kleiner)	<code>&gt;</code>	(größer)
<code>&lt;=</code>	(kleiner gleich)	<code>&gt;=</code>	(größer gleich)

## > Logische Operatoren geben auch Werte

vom Typ `int` (0 oder 1) zurück

> <code>&amp;&amp;</code>	(und)
> <code>  </code>	(oder)
> <code>!</code>	(nicht)



# Logische Operatoren

<b>&amp;&amp;</b>	<b>1</b>	<b>0</b>
<b>1</b>	1	0
<b>0</b>	0	0

<b>  </b>	<b>1</b>	<b>0</b>
<b>1</b>	1	1
<b>0</b>	1	0

<b>!</b>	
<b>1</b>	0
<b>0</b>	1

# Logische Operatoren

```
int a(int x) {  
    printf("%i\n", x);  
    return x > 4;  
}
```

Zweite Bedingung wird  
nicht ausgewertet

```
if (a(2) && a(6))  
    printf("Yes!\n");  
else  
    printf("No!\n");
```

2  
No!

```
if (a(5) && a(6))  
    printf("Yes!\n");  
else  
    printf("No!\n");
```

5  
6  
Yes!

# Logische Operatoren

```
int a(int x) {  
    printf("%i\n", x);  
    return x > 4;  
}
```

```
if (a(2) || a(3))  
    printf("Yes!\n");  
else  
    printf("No!\n");
```

2  
3  
No!

Zweite Bedingung wird  
nicht ausgewertet

```
if (a(5) || a(6))  
    printf("Yes!\n");  
else  
    printf("No!\n");
```

5  
Yes!

# Blöcke

- > Es soll sichergestellt werden, dass  $x \leq y$  gilt.

```
// make sure  $x \leq y$ . Swap, if necessary.  
if (x > y)  
    z = x;  
x = y;  
y = z;
```

- > Problem: nur die erste Anweisung wird bedingt ausgeführt; die anderen beiden Anweisungen werden immer ausgeführt.
- > Es ist daher ein Konstrukt notwendig, mit dem mehrere Anweisungen zu einer zusammengefasst werden können.

# Blöcke

- > Blöcke werden in C durch geschweifte Klammern { } realisiert.
- > Ein **Block** ist eine Zusammenfassung einer Folge von Anweisungen.

```
// Make sure x <= y. Swap, if necessary
if (x > y) {           // begin of block
    z = x;
    x = y;
    y = z;
}                     // end of block
```

- > Hilfreich für die Lesbarkeit des Programms und für die Fehlersuche ist eine an der Blockstruktur orientierte **Einrückung** (indentation).
- > Blöcke können **geschachtelt** werden.
- > Allgemeine Form eines Blocks  
    <block> ::= { <statements> }  
    <statements> ::= <statement> | <statements> <statement>

# Schleifen

# Wiederholungen (Iteration)

- > Häufig sollen einige Anweisungen mehrmals (z.B. mit jeweils anderen Werten) durchlaufen werden.
- > Für diesen Zweck gibt es **Wiederholungsanweisungen / Schleifen**.
- > Sofern vorab bekannt ist, wie oft wiederholt werden soll, wird meist die **for-Schleife** verwendet.
- > Ist die Zahl der Wiederholungen unbekannt und hängt sie von anderen Kriterien ab, wird meist die **while-Schleife** verwendet.
- > Abhängig davon, ob die Wiederholungsbedingung zu Beginn oder am Ende des Durchlaufs der **while-Schleife** geprüft wird, gibt es zwei Varianten
  - > **while-do-Schleife**
  - > **do-while-Schleife**

# For-Schleife

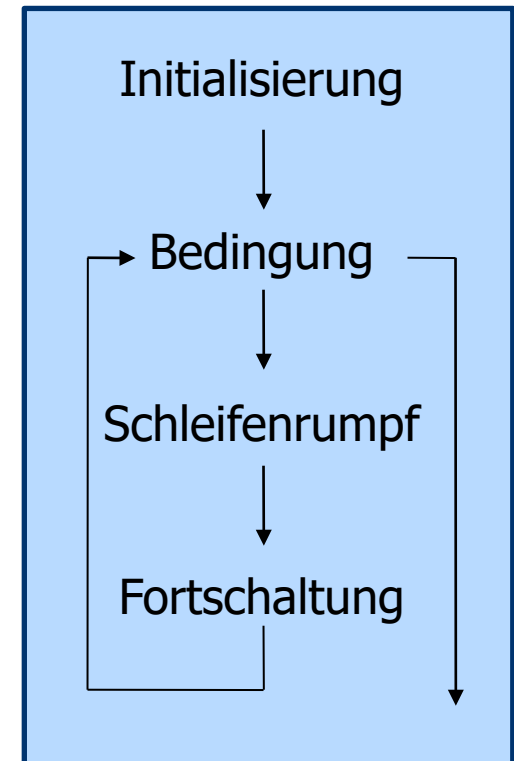
- > Folgende Schleife zählt von 0 bis **einschließlich** 10.

```
> for (int i = 0; i <= 10; i++)  
    printf("%i\n", i);
```

- > Syntax

```
> <for statement> ::= for (<for init>;  
    <conditional expression>;  
    <for update> ) <statement>
```

- > **Initialisierung**: Deklaration und Zuweisung der Schleifenvariable
- > **Bedingung**: Logischer Ausdruck, der vor jeder Ausführung des Schleifenrumpfs getestet wird.
- > **Schleifenrumpf**: Anweisung(en), die wiederholt ausgeführt werden.
- > **Fortschaltung**: Anweisungen, die den Wert der Schleifenvariablen nach jeder Ausführung des Schleifenrumpfs ändern.





# For-Schleife

- > Was bewirkt die folgende Schleife?

```
int i;  
for (i = 0; i <= 10; i++);  
printf("%i\n", i);
```

- > Das Semikolon vor dem Schleifenblock wird als **Leerbefehl** interpretiert, der 11-mal ausgeführt wird.
- > Danach wird die Print-Anweisung einmal mit dem Wert 11 für *i* ausgeführt.
- > Grundsätzlich sollte (bis wenige Ausnahmen) auf die Schleifenvariable *außerhalb* der Schleife nicht mehr zugegriffen werden!

# Ineinander Geschachtelte For-Schleifen

```
1  for (int i = 1; i <= 10; i++) {  
2      // print a line of the multiplication table  
3      for (int j = 1; j <= 10; j++)  
4          printf("%i ", i * j);  
5      printf("\n");  
6  }
```

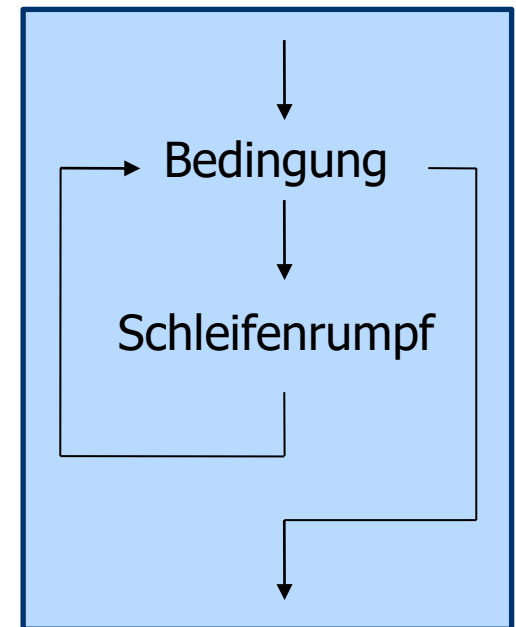
> Ausgabe

```
1 2 3 4 5 6 7 8 9 10  
2 4 6 8 10 12 14 16 18 20  
3 6 9 12 15 18 21 24 27 30  
4 8 12 16 20 24 28 32 36 40  
5 10 15 20 25 30 35 40 45 50  
6 12 18 24 30 36 42 48 54 60  
7 14 21 28 35 42 49 56 63 70  
8 16 24 32 40 48 56 64 72 80  
9 18 27 36 45 54 63 72 81 90  
10 20 30 40 50 60 70 80 90 100
```

# While-Schleifen

- > Manchmal ist nicht bekannt, wie oft die Schleife zu durchlaufen ist.
- > Das Beispiel berechnet den **größten gemeinsamen Teiler** (greatest common divisor) zweier Zahlen  $a$  und  $b$ :

```
1 // Euclids algorithm
2 int gcd(int a, int b) {
3     while (a != b) {
4         if (a > b) a -= b;
5         else b -= a;
6     } // { Assertion: a = b }
7     return a;
8 }
```



- > Allgemeine Form
  - > `<while-statement> := while ( <expression> ) <statement> ;`
- > Unter Umständen wird der Schleifenrumpf nie ausgeführt!

# Primzahltest

```
1  int is_prime(int z) {  
2      if (z <= 1) return 0;  
3          else if (z == 2) return 1;  
4  
5      int t = 3;  
6      while (z % t != 0 && t * t < z)  
7          t++;  
8      return z % t != 0;  
9  }
```

# Ägyptisches Multiplizieren

- > Das Verfahren benötigt nur Halbieren, Verdoppeln und Addieren.
- > Das kleine Einmaleins wird nicht benötigt.
- > Vorgehen entspricht der Multiplikation im Binärsystem.

```
1 int mult(int a, int b) {  
2     int result = 0;  
3     while (b > 0) {  
4         if (b % 2 == 1)  
5             result += a;  
6         a *= 2;  
7         b /= 2;  
8     }  
9     return result;  
10 }
```

Linke Spalte:  
Halbieren bis 1  
Rechte Spalte:  
Verdoppeln

17	11
<del>8</del>	22
<del>4</del>	44
<del>2</del>	88
1	176

Werte der rechten Spalte  
addieren für die der Wert der  
linken Spalte ungerade ist.

-----  
187

# Tabelle Fahrenheit in Grad Celsius

```
1  int main() {
2      double f, c;
3      printf("    F    C\n");
4      f = 0;
5      while (f <= 300) {
6          c = (5.0 / 9.0) * (f - 32);
7          printf("%3.0f %5.1f\n", f, c);
8          f += 20;
9      }
10 }
```

F	C
0	-17.8
20	-6.7
40	4.4
60	15.6
80	26.7
100	37.8
120	48.9
140	60.0
160	71.1
180	82.2
200	93.3
220	104.4
240	115.6
260	126.7
280	137.8
300	148.9

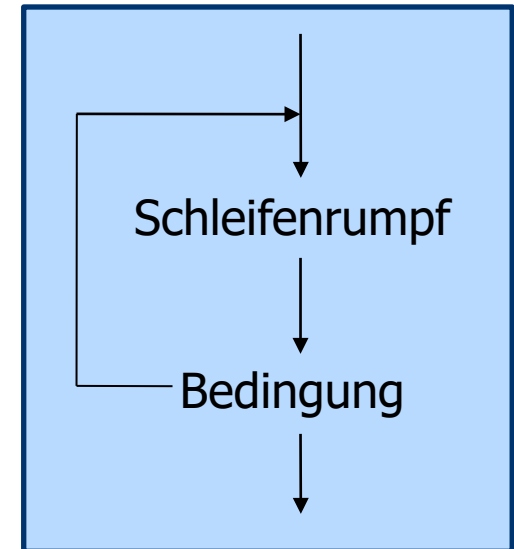
# Do-While-Schleifen

```
1  int i = 0;
2  do {
3      i++;
4      printf("%i\n", i);
5  } while (i < 10);
```

## > Allgemeine Form

```
<do-statement> ::=
    do <statement> while (<expression>;
```

- > Bei do-while-Schleifen wird der Schleifenrumpf immer *mindestens* einmal ausgeführt!



# Optimierter Primzahltest

```
1    int is_prime(int z) {
2        if (z <= 5)
3            return z != 4 && !(z <= 1);
4        if (z % 2 == 0)
5            return 0;
6
7        int t = 3;
8        do {
9            if (z % t == 0)
10                return 0;
11            t += 2;
12        } while (t * t <= z);
13
14        return 1;
15    }
```



# Zahl in Binärdarstellung umwandeln

```
1  #include <stdio.h>
2  #include <string.h>
3  void convert_to_binary() {
4      int n = 123456;
5      char digit;
6      char binary[128]="";
7      do {
8          digit = (n % 2 == 0) ? '0' : '1';
9          strncat(binary, &digit, 1);
10         n /= 2;
11     } while (n != 0);
12
13     strrev(binary);
14     printf("%s\n", binary); // 11110001001000000
15 }
```

# While-Schleifen: Unterschied

```
1 int n = 3, i = 0;
2 do {
3     i++;
4     printf("%i\n", i);
5 } while (i <= n);
```

Ausgabe:

1  
2  
3  
4

```
1 int n = 3, i = 0;
2 while (i <= n ) {
3     i++;
4     printf("%i\n", i);
5 }
```

Ausgabe:

1  
2  
3  
4

- > Ein Unterschied ergibt sich, wenn zu Beginn  $i$  auf einen Wert  $x$  größer als 3 gesetzt wird (z.B. auf den Wert 4).
- > Dann wird bei `do-while`  $x + 1$  ausgegeben, während bei `while` die Schleife nicht betreten und daher auch nichts ausgegeben wird.

# Verlassen einer Schleife: break

- > Soll eine Schleife vorzeitig verlassen werden, kann die Anweisung `break` verwendet werden, mit der die Schleife abgebrochen wird.
- > Bei geschachtelten Schleifen wird nur die (innerste) Schleife verlassen, in der sich die `break`-Anweisung befindet.

```
1 for (int i = 1; i < 5; i++) {  
2     for (int j = 1; j < 5; j++) {  
3         if (j == i) break;  
4         printf("%i ", i * j);  
5     }  
6 → printf("\n");  
7 }
```

Hier geht es  
in Zeile 6 weiter

- > Ausgabe
  - > 2
  - > 3 6
  - > 4 8 12

# Fortsetzen einer Schleife: continue

- > Mit `continue` werden die restlichen Anweisungen des Schleifenrumpfes übersprungen und dann der nächste Schleifendurchlauf begonnen.

```
1  for (int i = 1; i < 5; i++) {  
2  → for (int j = 1; j < 5; j++) {  
3      if (j == i) continue;  
4      printf("%i ", i * j);  
5  }  
6  printf("\n");  
7  }
```

Hier geht es in  
Zeile 2 weiter

- > Ausgabe
  - > 2 3 4
  - > 2 6 8
  - > 3 6 12
  - > 4 8 12

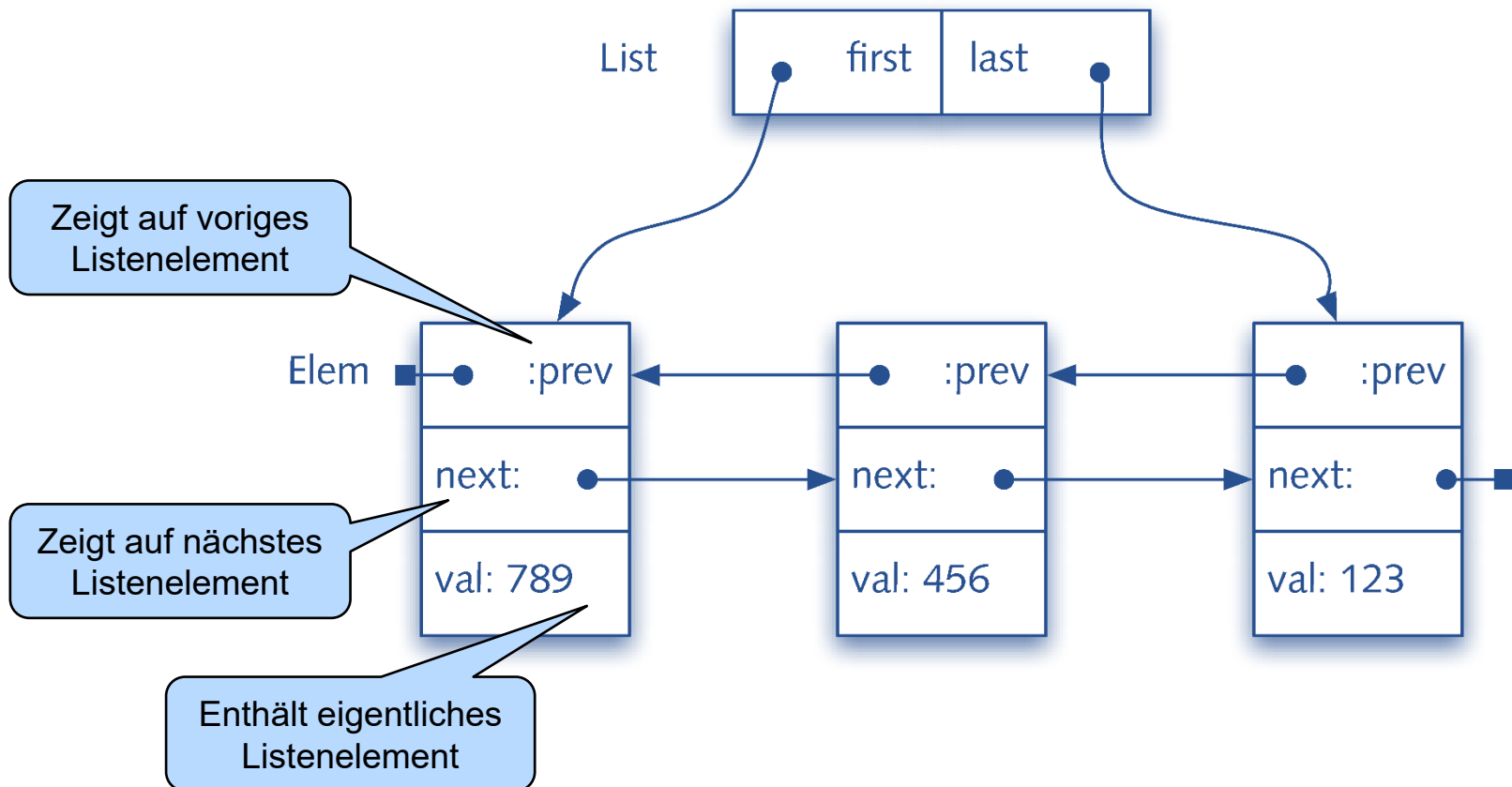
# Zeiger

# Zeiger

- > Ein **Zeiger** (pointer) ist eine Variable, welche die Adresse eines Speicherbereichs enthält.
- > Der Speicherbereich kann den Wert einer Variablen enthalten oder dynamisch allokiert sein.
- > Zeiger werden genutzt, um
  - > Speicherbereiche dynamisch zu verwalten
  - > **Adressen von Datenobjekten** an Funktionen zu übergeben
  - > **Funktionen als Argumente** an andere Funktionen zu übergeben
  - > **Dynamische Datenstrukturen** wie Listen, Bäume, Stacks oder Queues zu implementieren
- > Sie sind die häufigste Fehlerquelle in C und erfordern Disziplin vom Programmierer.

# Anwendungsbeispiel für Zeiger

- > Doppelt verkettete Liste implementiert als dynamische Datenstruktur



# Normale Variablen

- > Ermöglichen direkten Zugriff per Name der Variablen.
- > Enthalten einen Wert des deklarierten Typs.
- > Zuweisung
  - > Syntax: `<variable> = <expression>`
  - > Wert des Ausdrucks auf der rechten Seite wird in der Variable gespeichert, also im Speicher an ihrer jeweiligen Adresse.
  - > Beispiel

```
int x = 2;
```

```
int y = 2 * x;
```

	Speicher
x@0xF80C	2
y@0xF808	4



# Zeiger / Zuweisung Typ 1

- > Ermöglichen indirekten Zugriff per Adresse
- > Enthalten als Wert eine Adresse
  - > z.B. die einer anderen Variable
- > Deklaration eines Zeiger

x@0xF80C  
y@0xF808  
z@0xF800

Speicher
2
4
0xF80C

```
int* z;    // stores address of int variable
```

- > Zuweisung Typ 1
  - > Syntax: <pointer> = &<variable>
  - > Zeiger wird Adresse zugewiesen.
  - > Hier ist & der Referenzierungsoperator und liefert die Adresse einer Variablen.
  - > Beispiel

```
z = &x;    // assigns address of x to z
```

# Zeiger: Zuweisung Typ 2

- > Syntax: `*<pointer> = <expression>`
- > Das Sternchen heißt **Dereferenzierungsoperator**.
- > Der Wert der rechten Seite wird an der **Adresse** gespeichert, die im Zeiger gespeichert ist.
- > Beispiel

```
int x;  
int* z;           // declaration of integer pointer  
z = &x;           // assigns address of x to z  
*z = 4;          // assigns value of 4 to x
```

# Zeiger: Zuweisung Typ 3

- > Syntax: `<variable> = *<pointer>`
- > Wert, der an der Adresse gespeichert ist, die im Zeiger gespeichert ist, wird Variablen zugewiesen.

- > Beispiel

```
int x = 2;  
int* z;           // declaration of integer pointer  
z = &x;           // assigns address of x to z  
y = *z;           // assigns value of x to y
```

- > Auf der rechten Seite kann ein beliebiger Ausdruck mit `*<pointer>` stehen.

```
y = 2 * *z * *z + 4;    // y = 12
```

# Zeiger: Kombinierte Zuweisung

> Syntax:     \***<pointer>** = \***<pointer>**

> Beispiel

```
int x = 2;  
int y = 4;  
int* a = &x;           // assigns address of x to a  
int* b = &y;           // assigns address of y to b  
*a = *b;             // assigns value of y to x
```

> Auf der rechten Seiten kann wieder ein beliebiger Ausdruck mit \***<pointer>** stehen.

# Zeiger auf Speicherbereiche

- > Außer der Adresse von Variablen können Zeiger auch Adressen von Speicherbereichen enthalten

```
int* x = malloc(           // allocate memory
    2 * sizeof(int));      // for 2 ints
*x = 2;                    // set first int to 2
x++;                       // inc address by sizeof(int)
*x = 4;                    // set second int to 4
printf("%i\n", *x);        // print second int: 4
x--;                       // dec address by sizeof(int)
printf("%i\n", *x);        // print first int: 2
free(x);                   // deallocate memory
```

# Prozeduren

# Prozeduren

- > Ein **Unterprogramm** (auch **Prozedur**) ist ein gekapselter Programmteil, das ein geschlossenes Stück Arbeit darstellt.
- > Kommt ein Programmteil mehrmals im Programm vor, so kann er als Unterprogramm definiert und anschließend aufgerufen werden.
- > Unterprogramme stellen einen ersten Schritt dar, ein großes Programm in kleine, übersichtliche Teile zu zerlegen.
- > Das Programm ist strukturierter und leichter zu verstehen.
- > Prozeduren dienen auch der **Abstraktion**
  - > Interne Details einer Prozedur müssen nicht bekannt sein.
  - > Wichtig ist nur, wie die Prozedur aufrufen wird und welche Wirkung sie hat.

# Motivation für Prozeduren

```
int main(int argc, char* argv[]) {  
    float a, b, c;  
    ...  
    printf("*****");  
    printf("value is: %f\n" + a);  
    printf("*****");  
    ...  
    printf("*****");  
    printf("value is: %f\n" + b);  
    printf("*****");  
    ...  
    printf("*****");  
    printf("value is: %f\n" + c);  
    printf("*****");  
}
```



# Motivation für Prozeduren

```
// declaration of procedure
void print_value(float x) {
    printf("*****");
    printf("value is: %f\n", x);
    printf("*****");
}
```

```
int main(int argc, char* argv[]) {
    float a, b, c;
    ...
    print_value(a);
    ...
    print_value(b);
    ...
    print_value(c);
}
```

# Beispiel: Quadratzahlen ausgeben

```
1  int square(int n) {
2      return n * n;
3  }
4
5  int main(int argc, char* argv[]) {
6      for (int i = 1; i <= 20; i++)
7          printf("%i zum Quadrat ist %i\n",
8                i, square(i));
9  }
```

# Beispiel: Potenzieren

```
1  double power(double x, int n) {  
2      double result = 1.0;  
3      for (i = 1; i <= n; i++)  
4          result *= x;  
5      return result;  
6  }
```

# Syntax einer Prozedurdeklaration

```
<proc declaration> ::= <proc header> <proc body>
<proc header> ::= <result type> <proc declarator>
<result type> ::= <type> | void
<proc declarator> ::=
    <identifier> (<formal parameters>)
<proc body> ::= <block>
```

**Ergebnistyp**  
(result type)

Typ, wenn die Prozedur ein Ergebnis liefert,  
ansonsten **void**. Default ist **int**.

**Parameterliste**  
(parameter list)

Ein oder mehrere Werte können übergeben und  
im Anweisungsblock verwendet werden.

**Prozedurrumpf**  
(procedure body)

Das Programmstück, das ausgeführt werden soll  
und aus dem die Prozedur besteht.

# Parameterübergabe

- > Bei der **Prozedurdeklaration** werden die **formalen Parameter** vereinbart
  - > `prozedurname(type1 var1, ..., typen varN)`
  - > Beispiel: `add(int x, int y)`
- > Beim **Prozeduraufruf** werden die Werte der **aktuellen Parameter** (arguments) den formalen Parametern zugewiesen
  - > `prozedurname(ausdruck1, ..., ausdruckN)`
  - > Beispiel: `add(2 + 3, 4 * 5);`
  - > Im Beispiel bekommt x also den Wert 5 und y den Wert 20 zugewiesen

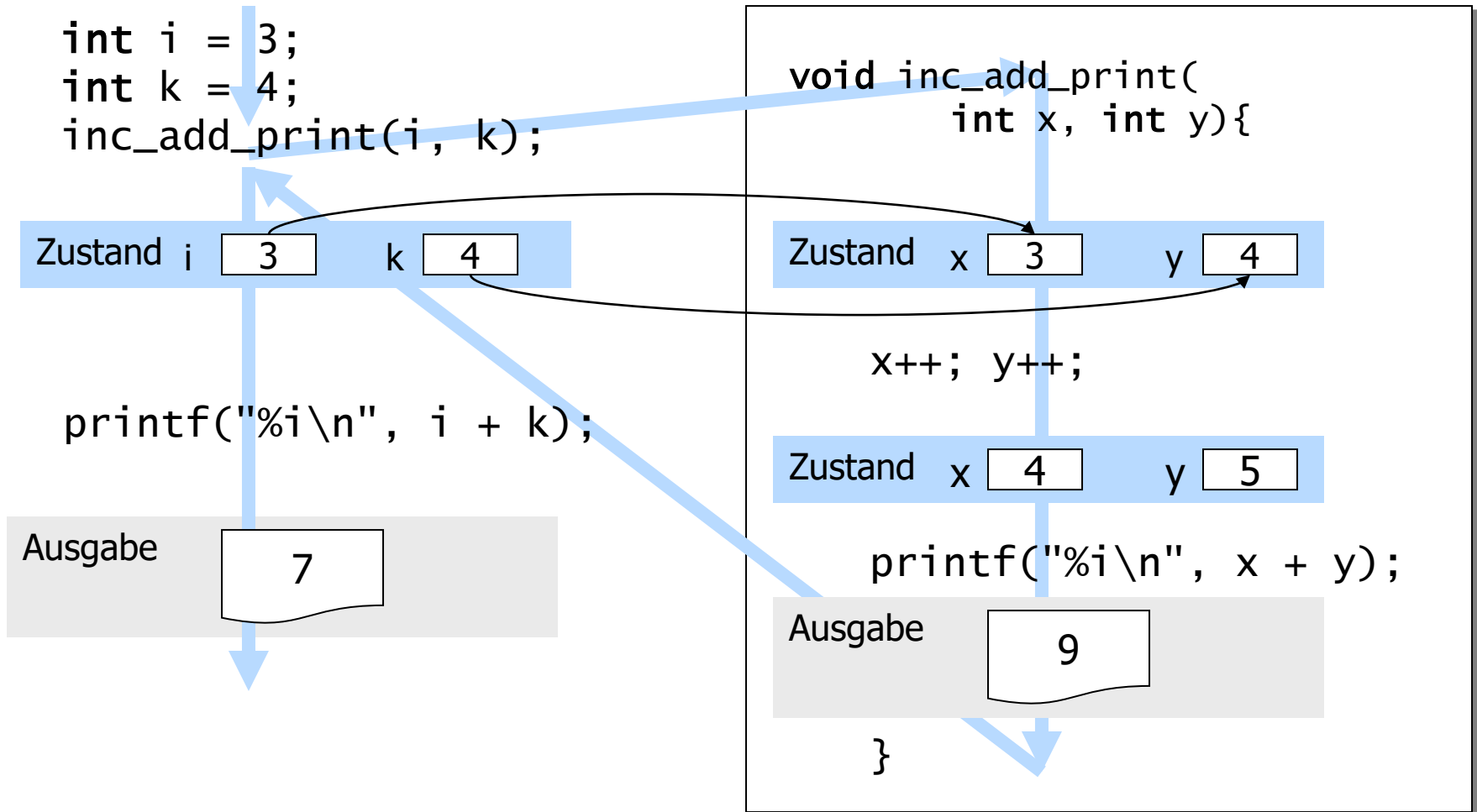
# Parameterübergabe per Wert (call by value)

```
// prints the sum of two values
// that were previously incremented
void inc_add_print(int x, int y){
    x++;
    y++;
    printf("%i + %i = %i\n", x, y, x + y); // 4 + 5 = 9
}

int main(int argc, char* argv[]) {
    int i = 3;
    int k = 4;
    inc_add_print(i, k); // i and k are not changed
    printf("%i + %i = %i\n", i, k, i + k); // 3 + 4 = 7
}
```

# Parameterübergabe per Wert (call by value)

in der Prozedur main:



# Parameterübergabe per Wert (call by value)

- > Bei der Wertübergabe werden die aktuellen Parameter (Argumente) durch den Funktionsaufruf *nicht* verändert.
- > Dies gilt selbst dann, wenn sie denselben Namen tragen, wie die formalen Parameter.

```
int sumplus(int a, int b) {  
    a++;           // increment a  
    b++;           // increment b  
    return a + b;  // return sum of a and b  
}
```

```
int a = 3;  
int b = 4;  
printf("%i\n", sumplus(a, b)); // prints 9  
printf("%i\n", a + b);        // prints 7
```



# Parameterübergabe per Referenz

- > Bei der Parameterübergabe per **Referenz** (call by reference) können die aktuellen Parameter (Argumente) durch den Funktionsaufruf verändert werden.

```
void swap(int* ptr_a, int* ptr_b){  
    int z = *ptr_a;  
    *ptr_a = *ptr_b; // lvalue = rvalue  
    *ptr_b = z;  
}
```

Die Variable auf die ptr\_a zeigt bekommt den Wert der Variablen, auf die ptr\_b zeigt, zugewiesen.

```
int x = 3; int y = 4;  
// call swap with address of x and y  
swap(&x, &y);  
printf("x = %i\n", x);           // prints x = 4  
printf("y = %i\n", y);           // prints y = 3
```

# Parameterübergabe per Referenz

```
int sumplus2(int* ptr_a, int* ptr_b) {  
    (*ptr_a)++; // increment value ptr_a points at  
    (*ptr_b)++; // increment value ptr_b points at  
    // return sum of values ptr_a and ptr_b point at  
    return *ptr_a + *ptr_b;  
}
```

```
int x = 3; int y = 4;  
printf("%i\n", sumplus2(&x, &y)); // prints 9  
printf("%i\n", x + y);           // prints 9
```

# Parameterübergabe per Referenz

```
1  void square(int x, int* s) {  
2      *s = x * x;  
3  }  
4  
5  int a;  
6  for (int i = 0; i <=10; i++) {  
7      square(i, &a); // value of a is set to i * i  
8      printf("Square of %i is %i\n.", i, a);  
9  }
```

# Ausdrücke als Parameter

- > Aktuelle Parameter sind beliebige Ausdrücke, deren Auswertung ein Resultat vom Typ des jeweiligen formalen Parameters liefert.

```
double celsius(double fahrenheit){  
    // transforms fahrenheit to celsius  
    double factor = 5.0 / 9.0;  
    return factor * (fahrenheit - 32.0);  
}
```

```
double c;  
float f = 41.0f;  
c = celsius(f);           // float variable  
c = celsius(41.0);        // double as argument  
c = celsius(6.0 * 6.0 + 5.0); // double expression  
c = celsius(25);          // integer as argument
```

# Lokale Variablen

- > Eine Prozedur hat oft **lokale Variablen**.
- > Diese werden innerhalb der Prozedur deklariert.
- > Ihr **Gültigkeitsbereich** (scope) ist die Prozedur.
- > Beispiel

```
int i; // global variable (initialized with 0)
void print_sum(int x, int y) {
    int i = x + y;           // local variable i
    printf("%i\n", i);       // print local i
}
```

```
i = i + 3;
print_sum(i, 15); // prints 18
```

- > Die Variable *i* *innerhalb* der Prozedur `print_sum` und die Variable *i* *außerhalb* haben nichts miteinander zu tun!

# Parameter als Lokale Variablen

- > Für jeden formalen Parameter wird automatisch eine Variable vom entsprechenden Typ vereinbart.
- > Beim Aufruf werden die Werte der aktuellen Parameter in die Variablen der formalen Parameter kopiert.
- > In der aufgerufenen Prozedur werden die formalen Parameter dann wie lokale Variablen behandelt.
- > Insbesondere kann auch ihr Wert geändert werden.
- > Eine Rückwirkung auf die Werte der aktuellen Parameter (in der aufrufenden Prozedur) findet aber nicht statt!

# Parameter als Lokale Variablen

> Was gibt das folgende Programmfragment aus?

```
1    void swap(int x, int y) {  
2        int z = x;  
3        x = y;  
4        y = z;  
5    }  
6  
7    int i = 3;  
8    int j = 4;  
9    swap(i, j);  
10   printf("i = %i\n", i);    // prints 3  
11   printf("j = %i\n", j);    // prints 4
```

# Verschattung

```
1  int n = 5;    // global variable
2
3  int main(int argc, char* argv[]) {
4      printf("%i\n", n); // prints global var: 5
5      { // start of block
6          int n = 3;
7          printf("%i\n", n); // prints local var: 3
8      } // end of scope of local variable n
9      printf("%i\n", n); // prints global var: 5
10 }
```



# return-Anweisung

- > Eine Prozedur kann mit **return** verlassen werden:

```
1 // prints the smallest of a, b and c
2 void print_smallest(int a, int b, int c) {
3     if (a <= b && a <= c) { // a is the smallest
4         printf("%i\n", a);
5         return;
6     }
7     // Assertion: the smallest is b or c
8     if (b <= c) {
9         printf("%i\n", b); // b is the smallest
10        return;
11    }
12    // Assertion: the smallest is c
13    printf("%i\n", c);
14 }
```

# Funktionen

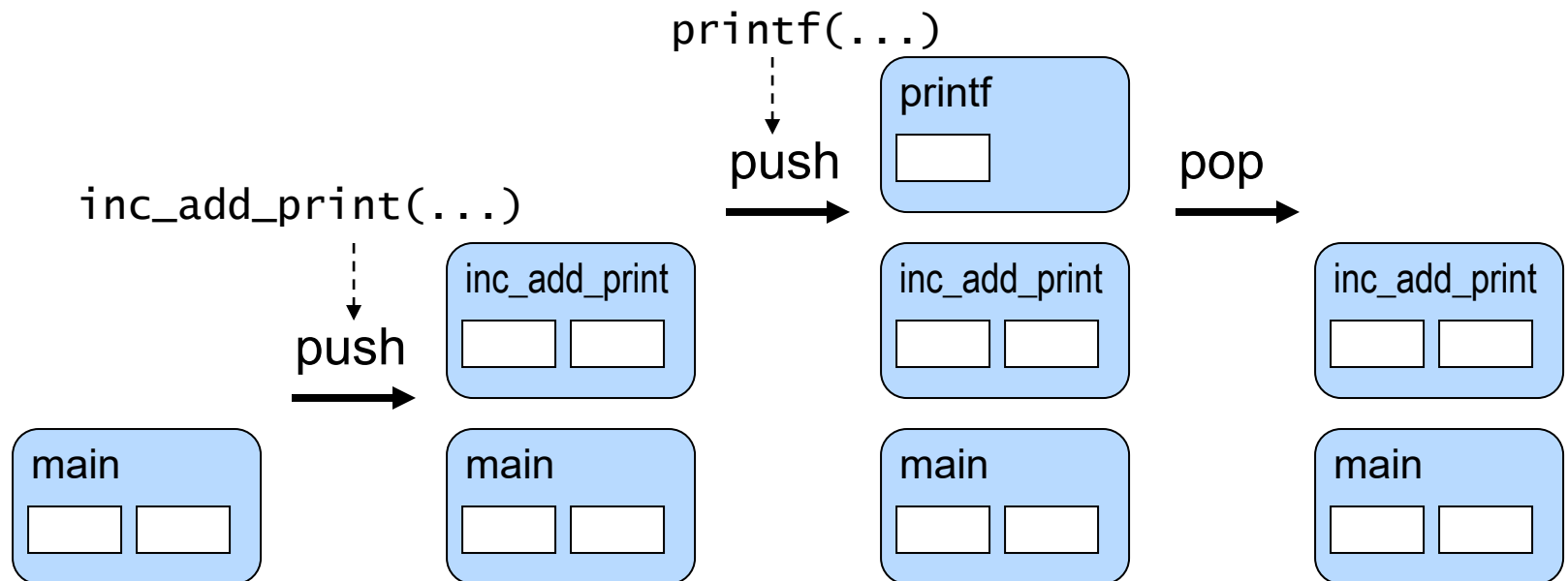
- > Prozeduren hatten bisher meist **void** als Ergebnistyp
- > Sie können aber auch einen Wert zurückliefern → Funktionen

```
1    // returns the smallest of a, b and c
2    int smallest(int a, int b, int c){
3        if (a <= b && a <= c)
4            return a;
5        // assertion: the smallest is b or c
6        if (b <= c)
7            return b;
8        // assertion: the smallest is c
9        return c;
10   } // end smallest
```

- > Prozeduraufrufe sind Anweisungen
  - > `print_smallest(2, 7, 3);`
- > Funktionsaufrufe sind Ausdrücke
  - > `int k = smallest(2, 7, 3) + 15;`

# Ablauf eines Prozeduraufrufs

- > Beim Aufruf wird ein **Aufrufrahmen** (activation record) erzeugt, der Speicherplatz für die Parameter und lokalen Variablen bereitstellt.
- > Ist die Prozedur beendet, wird der Aufrufrahmen wieder entfernt.
- > Dazu unterhält das System einen **Stapel** (stack), der die Operationen **push** (ablegen) und **pop** (herunternehmen) unterstützt.



# Ablauf eines Funktionsaufrufs

Beispiel: `x = smallest(5, x - 2, 7) + 4`

1. Argumente auswerten.
2. Einen Aufrufrahmen mit Parametern und lokalen Variablen erzeugen und auf den Stack legen (**push**).
3. Argumentwerte für die Parameter einsetzen.
4. Den Rumpf der Funktion ausführen, bis ein `return <expression>` erreicht wird.
5. `<expression>` auswerten.
6. Aufrufrahmen vom Stapel nehmen (**pop**).
7. Den Wert von `<expression>` in den obersten Aufrufrahmen (aufrufende Prozedur) einsetzen.

# Mathematische Funktionen

- > Die Datei `math.h` stellt mathematische Funktionen bereit.
  - > Trigonometrische Funktionen sowie deren Umkehrung (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`)
  - > Funktionen zum Potenzieren und Logarithmieren (`exp`, `log`, `log10`, `sqrt`, `pow`)
  - > Funktionen zum Runden einer Zahl (`round`, `ceil`, `floor`)
  - > ...

```
printf("%f\n", exp(1));    // prints value of e
printf("%f\n", pow(3, 2)); // print 9
```

# Mathematische Funktionen

- > `floor` und `ceil` runden eine Fließkommazahl zur nächsten ganzzahligen Zahl ab-, bzw. auf und liefern eine Zahl vom Typ `double`.
- > `round` rundet zur nächsten ganzen Zahl und liefert auch eine Zahl vom Type `double`.

```
printf("%f\n", round(1.5));        // prints 2
printf("%f\n", floor(1.5));        // prints 1
printf("%f\n", ceil(1.5));         // prints 2
```

```
printf("%f\n", round(-1.5));       // prints -2
printf("%f\n", floor(-1.5));       // prints -2
printf("%f\n", ceil(-1.5));        // prints -1
```

# Mathematische Funktionen

- > Der **Modulo-Operator** % ist außer für positive auch für negative Werte definiert.
- > Hierbei gilt die Regel, dass das Vorzeichen des Ergebnisses sich nach dem ersten Operanden richtet.

```
printf("%i\n", (-30) % 9);           // prints -3  
printf("%i\n", 30 % (-9));          // prints 3  
printf("%i\n", (-30) % (-9));       // prints -3
```

# Zufallszahlen

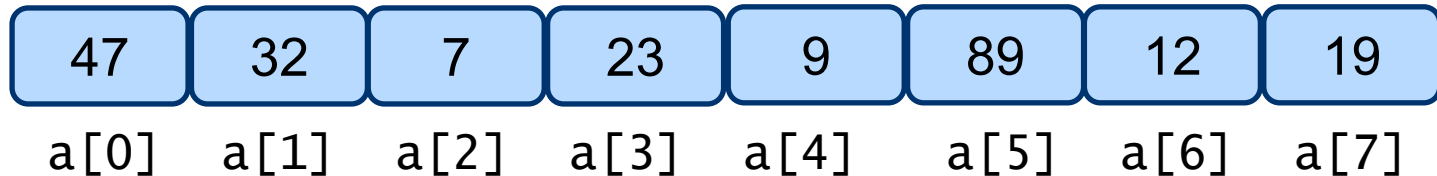
```
#include <stdlib.h>
#include <time.h>
int main() {
    // set seed for random number generator
    // using current time
    srand(time(0));
    // print five random numbers
    for (int i = 1; i <= 5; i++) {
        // derive random number from [0, RAND_MAX]
        int r = rand();
        printf("Derived[0..%d]: %d\n", RAND_MAX, r);
        // transform random number to [1, 6]
        int r_1_6 = 1 + r % 6;
        printf("Transformed [1..6]: %d\n", r_1_6);
    }
}
```



# Felder

# Felder (Arrays)

- > Mit Feldern bzw. Arrays kann eine Folge von Variablen gleichen Typs simultan erzeugt und als Einheit aufgefasst werden.



- > Auf ein einzelnes Element *i* des Arrays *a* wird durch Indizierung mittels *a[i]* zugegriffen. Arrays beginnen mit dem Index 0.

```
int scores_in_exam[] = {1, 2, 3, 3, 4, 4, 5};
int length = sizeof(scores_in_exam) / sizeof(int);
double sum = 0.0;
for (int i = 0; i < length; i++)
    sum += scores_in_exam[i];           // accumulate
double average_score = sum / length;    // derive average
```

# Felder (Arrays)

- > Array mit vorab bekannter Länge erzeugen

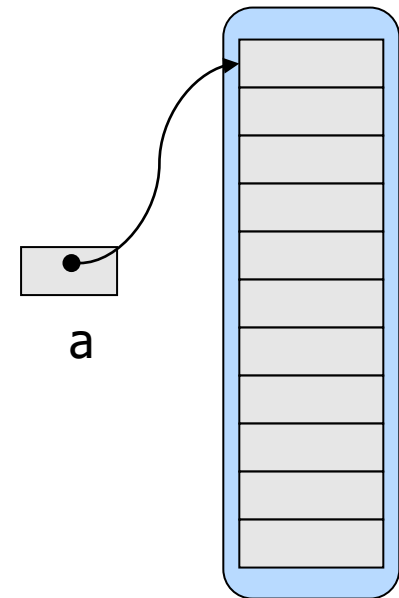
```
double a[5]; // array declaration
for (int i = 0; i < 5; i++)
    a[i] = i * i;
```

Hier liegt das Array auf dem Stack.

- > Nur die Elemente globaler Arrays werden initialisiert.
- > Array erzeugen, dessen Länge erst zur Laufzeit feststeht

```
int* a = malloc(length * sizeof(int));
for (int i = 0; i < length; i++)
    a[i] = i * i;
for (int i = 0; i < length; i++)
    printf("%i\n", a[i]);
free(a);
```

Hier liegt das Array auf dem Heap.



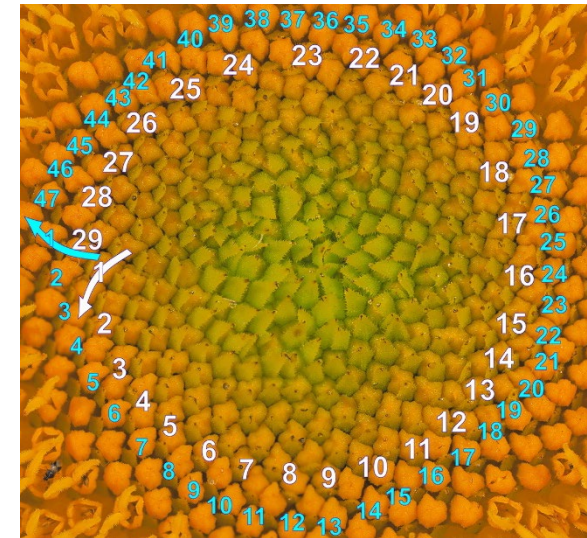
# Felder (Arrays)

- > Seit dem C99-Standard werden auch Arrays unterstützt, deren Größe keine Konstante ist.

```
int lucas_number(int n) {  
    if (n <= 1)  
        return n == 0 ? 2 : 1;  
    int x[n + 1] = { 2, 1 };  
    for (int j = 2; j <= n; j++)  
        x[j] = x[j - 2] + x[j - 1];  
    return x[n];  
} // close relation to fib numbers
```

Hier liegt das Array  
wieder auf dem Stack.

```
int main() {  
    for (int i = 0 ; i < 20; i++)  
        printf("%i, %i\n", i, lucas_number(i));  
} // 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, ....
```



# Äquivalenz von Arrays und Zeigern

```
double get_average1(int a[], int size) {  
    int sum = 0;  
    for (int i = 0; i < size; i++)  
        sum += a[i];  
    return (double)sum / size;  
}
```

int a[] und int\* a  
sind äquivalent

```
double get_average2(int* a, int size) {  
    int sum = 0;  
    for (int i = 0; i < size; i++)  
        sum += *(a + i);  
    return (double)sum / size;  
}
```

```
int main() {  
    int balance[] = {1000, 2, 3, 17, 50};  
    printf("%f\n", get_average1(balance, 5)); // 214.4  
    printf("%f\n", get_average2(balance, 5)); // 214.4  
}
```

# Rückwirkung durch Call-by-Reference

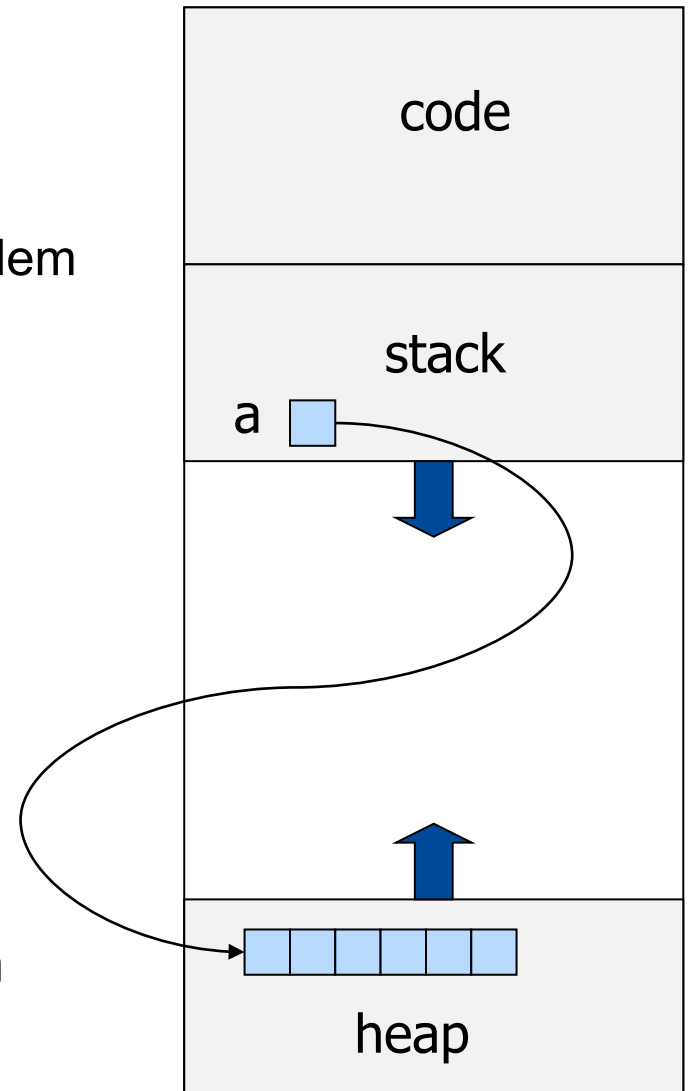
- > Arrays werden wegen der Äquivalenz zu Zeigern immer per call-by-reference übergeben.
- > Dadurch gibt es bei Änderungen eine Rückwirkung.

```
void set_to_value(int a[], int size, int value) {  
    for (int i = 0; i < size; i++) a[i] = value;  
}
```

```
int main() {  
    int a[] = {1, 2, 3, 4, 5};  
    for (int i = 0; i < 5; i++) // prints: 1 2 3 4 5  
        printf("%i ", a[i]);  
    printf("\n");  
    set_to_value(a, 5, 6); // changes values in a[]  
    for (int i = 0; i < 5; i++) // prints: 6 6 6 6 6  
        printf("%i ", a[i]);  
}
```

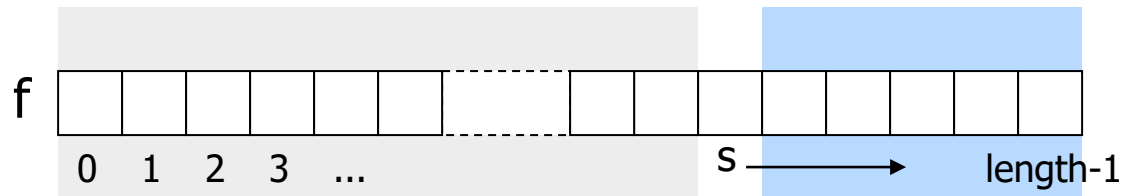
# C-Programm im Speicher

- > Ein C-Programm besteht während der Ausführung aus drei Speicherbereichen
  - > **Code**
    - > Das Programm selbst, also der aus dem Quellcode erzeugte Maschinencode.
  - > **Stack**
    - > Die Parameter und lokalen Variablen für die aktuell aktive Prozedur und alle Prozeduren weiter oben in der Aufrufhierarchie.
  - > **Heap**
    - > Die dynamisch allokierten Speicherbereiche.
- > Stack und Heap wachsen und schrumpfen während der Programmausführung.



# Lineare Suche in einem Array

```
// returns either the index
// where x was found or -1 if not
int linear_search(int f[], int length, int x) {
    int s = 0;
    while (s < length) {
        if (f[s] == x) return s;
        s++;
    }
    return -1;
}
```



```
int main() { // 0 1 2 3 4
    int a[5] = {5, 4, 1, 2, 3};
    printf("%i\n", linear_search(a, 5, 2)); // prints 3
}
```



# Sieb des Eratosthenes

```
#define N 120
int main() {
    int i, j;
    int no_prime[N] = { 0 };

    for (i = 2; i * i <= N; i++)
        if (!no_prime[i]) {
            printf("%i ", i);
            for (j = i * i; j <= N ; j += i)
                no_prime[j] = 1;
        }

    for (; i <= N; i++)
        if (!no_prime[i])
            printf("%i ", i);
}
```

	2	3	4	5	6	7	8	9	10	Primzahlen:
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

# Sortieren durch Auswählen

```
void selection_sort(int f[], int length) {  
    int min;  
    for (int i = 0; i < length; i++) {  
        // find index min for new minimal value  
        min = i;  
        for (int j = i; j < length; j++)  
            if (f[j] < f[min])  
                min = j;  
        swap(f[i], f[min]);  
    }  
}
```

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Mehrdimensionale Felder

# Mehrdimensionale Arrays

- > Beispiel: Matrix (Tabelle) mit 3 Zeilen und 4 Spalten

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

- > Diese Matrix ist ein Array mit 2 Dimensionen. Schleifen über solche Matrizen sind meist geschachtelte Schleifen

```
for (int i = 0; i < 3; i++)  
    for (int j = 0; j < 4; j++)  
        a[i][j] = i * j;
```

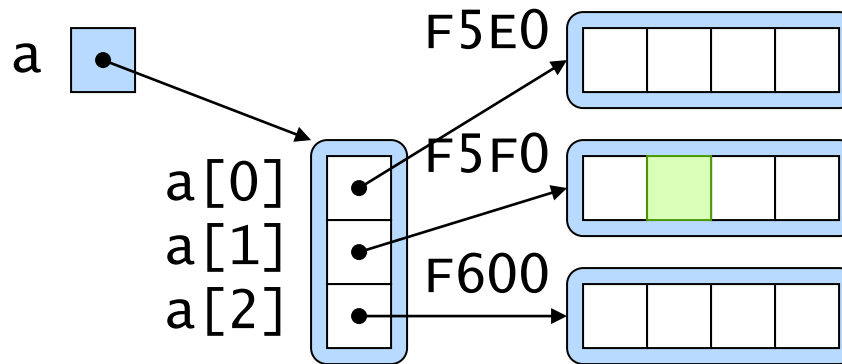
- > Folgendes Beispiel zeigt, dass a ein Array ist, das aus drei Elementen besteht, die wiederum Arrays sind:

```
int* b = a[0];  
for (int i = 0; i < 4; i++) {  
    b[i] = i;  
    printf("%i\n", a[0][i]); // prints 0 1 2 3  
}
```

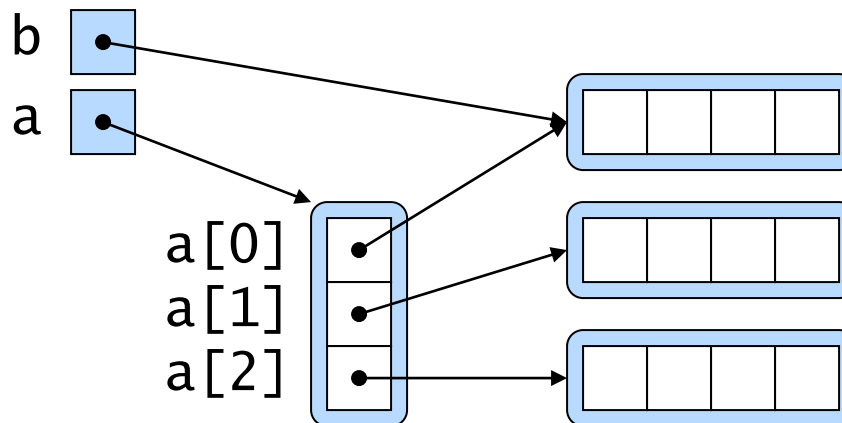
# Berechnung der Adressen

liegt auf dem Stack

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



```
int* b = a[0];
```



Adresse von Zeile  
 $= 0xF0E0 + 0x10 * \text{\#zeile}$

Adresse von Zelle  
 $= 0xF0E0 + 0x10 * \text{\#zeile} + 0x04 * \text{\#spalte}$

`&a[1][1]`  
 $= 0xF0E0 + 0x10 * 1 + 0x04 * 1$   
 $= 0xF0F4$

# Berechnung der Adressen

```
int a[3][4];  
printf("a = %p\n", a);           // a = &a = &a[0][0] = 0xF5E0  
  
printf("sizeof(a) = %i\n",  
       sizeof(a));               // sizeof(a) = 48 = 0x30  
printf("sizeof(a[0]) = %i\n",  
       sizeof(a[0]));            // sizeof(a[0]) = 16 = 0x10  
  
printf("a[0] = %p\n", a[0]);     // a[0] = 0xF5E0  
printf("a[1] = %p\n", a[1]);     // a[1] = 0xF5F0  
printf("a[2] = %p\n", a[2]);     // a[2] = 0xF600  
  
printf("&a[0][0] = %p\n", &a[0][0]); // &a[0][0] = 0xF5E0  
printf("&a[1][1] = %p\n", &a[1][1]); // &a[1][1] = 0xF5F4  
printf("&a[2][2] = %p\n", &a[2][2]); // &a[2][2] = 0xF608
```

# Initialisierung der Arrayelemente

- > Globale Arrays werden automatisch initialisiert.
- > Im Beispiel werden die fehlenden Elemente mit 0 initialisiert.

```
int z[3][4] = { {1}, {2, 1}, {3, 2, 1} };  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 4; j++)  
        printf("%i ", z[i][j]);  
    printf("\n");  
}
```

```
// 1 0 0 0  
// 2 1 0 0  
// 3 2 1 0
```

# Dynamische Mehrdimensionale Arrays

```
int set_array(int** a, int rows, int cols) {  
    for (int i = 0; i < rows; i++)  
        for (int j = 0; j < cols; j++)  
            a[i][j] = (i + 1) * (j + 1);  
}  
  
int main() { int rows = 5, cols = 5;  
    // 1. allocate row pointers  
    int** a = malloc(rows * sizeof(int*));  
    // 2. allocate rows individually  
    for (int i = 0; i < rows; i++)  
        a[i] = malloc(cols * sizeof(int));  
    set_array(a, rows, cols);  
    // 3. free rows individually  
    for (int i = 0; i < rows; i++)  
        free(a[i]);  
}
```

Die einzelnen Reihen  
können im Speicher  
verstreut sein.



# Dynamische Mehrdimensionale Arrays

```
int main() {  
    int rows = 5, cols = 5;  
    // 1. allocate row pointers  
    int** b = malloc(rows * sizeof(int*));  
    // 2. allocate all array elements at once  
    b[0] = malloc(rows * cols * sizeof(int));  
    // 3. set row pointers  
    for (int i = 1; i < rows; i++)  
        b[i] = b[0] + i * cols;  
    set_print_array(b, rows, cols);  
    // 4. free all array elements at once  
    free(b[0]);  
}
```

Die Reihen bilden  
einen durchgehenden  
Bereich im Speicher.

# Nicht-rechteckige Arrays

```
// _msize: windows only
```

```
#define N 10
```

```
int* c[N];
```

```
for (int i = 0; i <= N; i++) {
```

```
    c[i] = malloc((i + 1) * sizeof(int)); // allocate row
```

```
    for (int j = 0; j <= i; j++)
```

```
        c[i][j] = i * j;
```

```
}
```

```
for (int i = 0; i <= N; i++) {
```

```
    printf("c[%2.0i] = 0x%p %2.i ",
```

```
        i, c[i], _msize(c[i]));
```

```
    for (int j = 0; j <= i; j++)
```

```
        printf("%i ", c[i][j]);
```

```
    printf("\n");
```

```
}
```

```
for (int i = 1; i <= N; i++)
```

```
    free(c[i]); // free rows
```

Die Zeilen mehrdimensionaler Arrays können verschiedene Größen haben.

Bsp.: Dreieckiges Array

```
c[0] = 0x6A30 4 0
```

```
c[1] = 0x6A70 8 0 1
```

```
c[2] = 0x6AB0 12 0 2 4
```

```
c[3] = 0x6AF0 16 0 3 6 9
```

```
c[4] = 0xC810 20 0 4 8 12 16
```

```
c[5] = 0xC860 24 0 5 10 15 20 25
```

```
c[6] = 0xC8B0 28 0 6 12 18 24 30 36
```

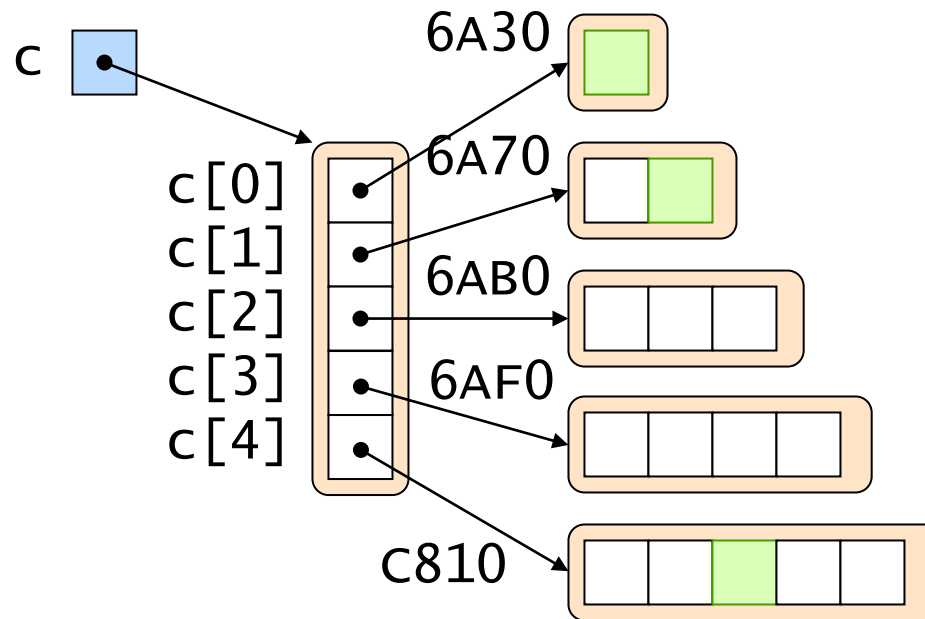
```
c[7] = 0xC900 32 0 7 14 21 28 35 42 49
```

```
c[8] = 0xC950 36 0 8 16 24 32 40 48 56 64
```

```
...
```

# Nicht-rechteckige Arrays

- > Berechnung der Adresse der Arrayelemente
  - >  $\&c[0][0] = \&c[0] = 0x6A30$
  - >  $\&c[1][1] = \&c[1] + 1 * \text{sizeof(int)} = 0x6A74$
  - >  $\&c[4][2] = \&c[4] + 2 * \text{sizeof(int)} = 0xC818$



# Abbildung auf eindimensionales Array

```
// derive index in 1d array
// from row and col of triangular array
int pos(int row, int col) {
    return row * (row + 1) / 2 + col;
}
```

```
// 0
// 1 2
// 3 4 5
// 6 7 8 9
// 10 11 12 13 14
// 15 16 17 18 19 20
// 21 22 23 24 25 26 27
// 28 29 30 31 32 33 34 35
// 36 37 38 39 40 41 42 43 44
// 45 46 47 48 49 50 51 52 53 54
// 55 56 57 58 59 60 61 62 63 64 65
...
```

Bsp.: Abbildung eines  
dreieckiges Array auf  
eindimensionales Array

$\text{pos}(3, 2) = 8$

# Abbildung auf eindimensionales Array

```
int main() {  
    int n = 10;  
    int size = (n + 1) * (n + 2) / 2;  
    int* d = malloc(size * sizeof(int));  
    for (int i = 0; i <= n; i++)  
        for (int j = 0; j <= i; j++)  
            d[pos(i, j)] = i * j;  
    for (int i = 0; i <= n; i++)  
        for (int j = 0; j <= i; j++)  
            printf("%i ", d[pos(i, j)]);  
    for (int i = 0; i < size; i++)  
        printf("%i ", d[i]);  
    free(d);  
}
```

# Angabe der Dimensionen

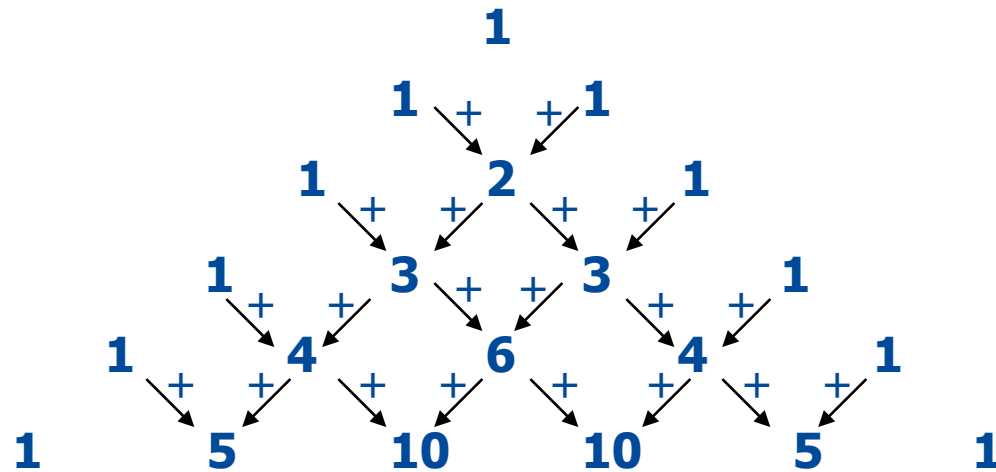
- > Bei mehrdimensionalen Arrays darf nur die **erste Dimension** unbestimmt bleiben, die **anderen** müssen angegeben werden.
- > Dies ist notwendig, um die Adresse der Array-Elemente berechnen zu können.

```
double get_average(int a[][3], int rows, int cols) {  
    double sum = 0;  
    for (int i = 0; i < rows; i++)  
        for (int j = 0; j < cols; j++)  
            sum += a[i][j];  
    return (double)sum / (rows * cols);  
}
```

```
int main() {  
    int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
    printf("%f\n", get_average(matrix, 3, 3)); // 5.0  
}
```

# Anwendungsbeispiel

- > C-Programm, das das **Pascalsche Dreieck** ausdrückt.



- > Die Ausgabe soll linksbündig erfolgen:

```
1
1 1
1 2 1
1 3 3 1
```

# Ansatz 1: Quadratisches Array

	j=0	j=1	j=2	j=3	j=4	...
i=0	1	0	0	0	0	
i=1	1	1	0	0	0	
i=2	1	2	1	0	0	
i=3	1	3	3	1	0	
i=4	1	4	6	4	1	
...						



# Ansatz 1: Quadratisches Array

```
1 #define N 10
2 int main() {
3     int p[N][N] = { 0 };
4     int i, j;
5     for (i = 0; i < N; i++) { // initialize boundary
6         p[i][0] = 1;           // first column
7         p[i][i] = 1;           // diagonal element
8     }
9     for (i = 2; i < N; i++)    // calculate values
10        for (j = 1; j < i; j++)
11            p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
12    for (i = 0; i < N; i++) { // print result
13        for (j = 0; j <= i; j++)
14            printf("%i ", p[i][j]);
15        printf("\n");
16    }
17 }
```

# Ansatz 1: Quadratisches Array

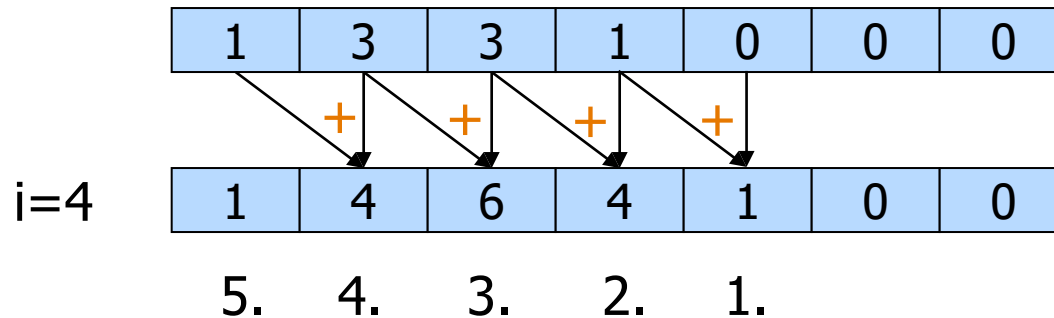
- > Ansatz 1 verschwendet Speicherplatz, da nur die linke untere Hälfte der Matrix tatsächlich benutzt wird
- > Das Programm hat eine quadratische Speicherkomplexität.
  - > Speicherplatz in der Größenordnung von  $n^2$  wird benötigt, wenn ein Pascalsches Dreieck mit  $n$  Zeilen berechnet werden soll.
- > Bei  $n = 1000$  benötigt die Matrix bereits 4MByte Speicher.
- > Ein dreieckiges zweidimensionales Array würde nur die Hälfte des Speichers benötigen.

## Ansatz 2: Dreieckiges Array

```
1 #define N 10
2 int main() {
3     int *p[N];
4     int i, j;
5     for (i = 0; i < N; i++) {
6         p[i] = malloc(sizeof(int) * (i + 1));
7         p[i][0] = 1;    // initialize 1st column
8         p[i][i] = 1;    // initialize diagonal element
9     }
10    for (i = 2; i < N; i++) // derive values "inside"
11        for (j = 1; j < i; j++)
12            p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
13    for (i = 0; i < N; i++) { // print result
14        for (j = 0; j <= i; j++)
15            printf("%i ", p[i][j]);
16        printf("\n");
17    }
18    for (i = 0; i < N; i++) free(p[i]);
19 }
```

## Ansatz 3: Eindimensionales Array

- > Allerdings wird gar kein zweidimensionales Feld benötigt, da Zeile  $i$  aus der vorangegangenen Zeile  $i - 1$  berechnet werden kann.
- > Wird die neue Zeile außerdem **von rechts nach links berechnet**, werden keine Werte überschrieben, die noch benötigt werden.
- > Bei diesem Vorgehen reicht daher ein eindimensionales Array aus, das so lang wie die letzte Zeile des Dreiecks ist.
- > In diesem Fall (Ansatz 3) braucht das Programm bei  $n = 1000$  nur 4KByte Speicherplatz.



## Ansatz 3: Eindimensionales Array

```
#define N 10
int main() {
    int p[N];
    int i, j;
    p[0] = 1;                                // initialize first element
    for (i = 1; i < N; i++) {                 // loop across lines
        for (j = 0; j < i; j++)               // print line
            printf("%i ", p[j]);
        printf("\n");
        p[i] = 1;                             // initialize diagonal
        // compute new line from previous line
        for (j = i - 1; j >= 1; j--)
            p[j] = p[j] + p[j - 1];
    }
    for (j = 0; j < i; j++)                   // print last line
        printf("%i ", p[j]);
}
```

# Kalenderumrechnung

```
char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

/* return day of year from year, month and day */
int day_of_year(int year, int month, int day) {
    int i, leap;
    leap = year % 4 == 0 && year % 100 != 0
           || year % 400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}
```

# Kalenderumrechnung

```
/* get month and day from year and day of year */
void month_day(int year, int yearday, int* pmonth, int* pday) {
    int i, leap;
    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}

int main() {
    int year = 1998;
    int month = 10;
    int day = 19;
    int x = day_of_year(year, month, day);
    printf("day of year: %i\n", x); // prints 292

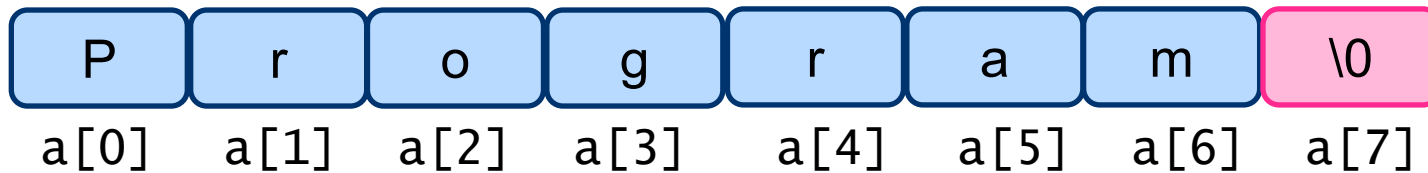
    int doy = 198;
    month_day(year, doy, &month, &day);
    printf("month = %i, day = %i\n", month, day); // prints 7, 17
}
```

# Strings



# Strings

- > Strings sind eindimensionale Arrays vom Typ char, die mit dem **Nullzeichen** abgeschlossen werden



```
char a[] =                // array initialized w/ string
    {'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
char b[] = "Program";      // array initialized w/ string
char* c = "Program";      // pointer to constant string
a[0] = 'X';                // OK
b[1] = 'X';                // OK
c[2] = 'X';                // undefined behavior
c = "Margorp";             // c points to different string
printf("%s %s %s\n", a, b, c); // Program Program Margorp
c = b;                     // c points now to string in b
```

# Suche in einem String

```
#include <string.h>
// finds first occurrence of string g in string f
int search(char f[], char g[]) {
    int found;
    // outer loop scans text for pattern
    for (int s = 0; s <= strlen(f) - strlen(g); s++) {
        found = 1;
        // inner loop compares pattern with text
        for (int i = 0; i < strlen(g); i++)
            if (f[s + i] != g[i]) { found = 0; break; }
        if (found) return s;
    }
    return -1;
}
```

# Suche in einem String

```
int main() {  
    char* a = "This is a test.";  
    char* b = "test";  
    printf("%i\n", search(a, b)); // prints 10  
}
```

# Einige Prozeduren in string.h

```
// appends the string src to dest
char* strcat(char* dest, char* src);
// appends at most n bytes of the string src to dest
char* strncat(char* dest, char* src, size_t n);
// locates byte c in string s, searching from the beginning
char* strchr(char* s, int c);
// locates byte c in string s, searching from the end
char* strrchr(char* s, int c);
// compares two strings lexicographically
int strcmp(char* s1, char* s2);
// copies a string from one location to another
char* strcpy(char* dest, char* src);
// finds the length of a string s
size_t strlen(char* s);
// finds the first occurrence of any byte in accept
char* strpbrk(char* c, char* s);
// finds the first occurrence of the string s2
// in the longer string s1
char* strstr(char* s1, char* s2);
```

# Strukturen

# Strukturen

- > Wir wollen Informationen über Studierende speichern
  - > 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.

# Strukturen

```
struct _person {  
    char v[16];           // first name  
    char n[16];           // surname  
    long m;               // registration number  
    int g;                // year of birth  
} p;
```

- > Diese Deklaration besagt: die Variable p ist eine Strukturvariable mit den Komponenten v, n, m und g.
- > Mit dem Ausdruck p.m kann auf die Komponente m der Strukturvariable p zugegriffen werden.
- > \_person ist der zugehörige Strukturtyp.
- > Mit struct \_person var\_name; können weitere Variablen dieses Strukturtyps deklariert werden.

# Arrays von Strukturen

```
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}
};

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



# Strukturen als Parameter

- > Strukturen werden mittels Call-by-Value übergeben.

```
typedef struct _person person; // define type person
```

```
void pprint(person p) { // call by value  
    printf("%s %s %ld %d\n", p.v, p.n, p.m, p.g);  
}
```

```
int main() {  
    int size = sizeof(pa) / sizeof(pa[0]);  
    for(int i = 0; i < size; i++)  
        pprint(pa[i]);  
    return 0;  
}
```

# Strukturen als Parameter

- > Eine Übergabe per Call-by-Reference ist aber auch möglich.

```
void pprint(person* p) { // call by reference
    printf("%s %s %ld %d\n", p->v, p->n, p->m, p->g);
}
```

```
int main() {
    int size = sizeof(pa) / sizeof(pa[0]);
    for(int i = 0; i < size; i++)
        pprint(&pa[i]);
    return 0;
}
```

# Geschachtelte Structs

```
struct point {  
    int x, y;  
};
```

```
struct line {  
    struct point start;  
    struct point end;  
};
```

```
int main() {  
    struct line l = {{1, 2}, {3, 4}};  
    printf("l.start.x = %i\n", l.start.x);  
    printf("l.start.y = %i\n", l.start.y);  
    printf("l.end.x   = %i\n", l.end.x);  
    printf("l.end.y   = %i\n", l.end.y);  
}
```

# Studierendenverwaltung

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int length;

struct _date {
    int day; int month; int year;
};
typedef struct _date date;

struct _student {
    char* first_name;
    char* last_name;
    int matrikel;
    date birth_date;
};
typedef struct _student student;
```

# Studierendenverwaltung

```
student students[] = {  
    "Gero",    "Müller",    111111,    {1, 9, 1989},  
    "Peter",   "Scholz",    222222,    {13, 3, 2000},  
    "Harald",  "Müller",    333333,    {12, 5, 1991},  
    "Helge",   "Schmidt",   444444,    {29, 4, 1987},  
    "Willi",   "Graf",        555555,    {27, 7, 1993},  
    "Peter",   "Schmitz",    666666,    {5, 1, 1997},  
    "Gero",    "Peters",    777777,    {24, 1, 2007},  
    "Anke",    "Wolf",      888888,    {18, 11, 1956},  
    "Anne",    "Schulze",   999999,    {11, 12, 1965},  
    "Olaf",    "Backe",     101010,    {10, 9, 2003},  
};
```

```
void print_date(date* d) {  
    printf("%i.%i.%i", d->day, d->month, d->year);  
}
```

# Studierendenverwaltung

```
void print_student(student* s) {
    if (s != NULL) {
        printf("{%s, %s, %i, ",
            s->first_name, s->last_name, s->matrikel);
        print_date(&s->birth_date);
        printf("}\n");
    } else printf("NULL!");
}

student* search_by_date(date d) {
    for (int i = 0; i < length; i++)
        if (d.day == students[i].birth_date.day &&
            d.month == students[i].birth_date.month &&
            d.year == students[i].birth_date.year)
            return &students[i];
    return NULL;
}
```

# Studierendenverwaltung

```
student* search_by_name(  
    char* first_name, char* last_name) {  
    for (int i = 0; i < length; i++)  
        if (strcmp(students[i].first_name,  
                    first_name) == 0 &&  
            strcmp(students[i].last_name,  
                    last_name) == 0)  
            return &students[i];  
    return NULL;  
}
```

```
student* search_by_matrikel(int matrikel) {  
    for (int i = 0; i < length; i++)  
        if (students[i].matrikel == matrikel)  
            return &students[i];  
    return NULL;  
}
```

# Studierendenverwaltung

```
student* search_oldest() {
    student* oldest_student = &students[0];
    date oldest_date = oldest_student->birth_date;
    date d;
    for (int i = 1; i < length; i++) {
        d = students[i].birth_date;
        if (d.year < oldest_date.year ||
            d.year == oldest_date.year &&
                d.month < oldest_date.month ||
            d.year == oldest_date.year &&
                d.month == oldest_date.month &&
                d.day < oldest_date.day) {
            oldest_student = &students[i];
            oldest_date = oldest_student->birth_date;
        }
    }
    return oldest_student;
}
```



# Studierendenverwaltung

```
int main() {  
    length = sizeof(students) / sizeof(student);  
    for (int i = 0; i < length; i++)  
        print_student(&students[i]);  
    printf("\n");  
  
    date d = {27, 7, 1993};  
    student* s = search_by_date(d);  
    printf("searched for ");  
    print_date(&d);  
    printf(": ");  
    print_student(s);  
  
    char* first_name = "Gero";  
    char* last_name  = "Peters";
```

# Studierendenverwaltung

```
s = search_by_name(first_name, last_name);  
printf("searched for ");  
printf("%s, %s: ", first_name, last_name);  
print_student(s);
```

```
int matrikel = 333333;  
s = search_by_matrikel(matrikel);  
printf("searched for ");  
printf("%i: ", matrikel);  
print_student(s);
```

```
s = search_oldest();  
printf("oldest student: ");  
print_student(s);
```

```
}
```

# Rekursion

# Rekursive Berechnung der Fakultät

```
int factorial(int n) {  
    // assertion: n >= 1  
    return n == 1 ? 1 : n * factorial(n - 1);  
}
```

```
int main() {  
    int a = 6;  
    int x = factorial(a);  
    printf("factorial of %i is %i.\n", a, x);  
}
```

# Rekursive Berechnung GGT

```
int gcd(int a, int b) {  
    printf("gcd(%i, %i)\n", a, b);  
    while (a != b)  
        return a > b ? gcd(a - b, b) : gcd(a, b - a);  
    return a;  
}
```

```
int main() {  
    int a = 30;  
    int b = 50;  
    int x = gcd(a, b);  
    printf("gcd of %i and %i is %i.\n", a, b, x);  
}
```

```
gcd(30, 50)  
gcd(30, 20)  
gcd(10, 20)  
gcd(10, 10)  
gcd of 30 and 50 is 10.
```

# Rekursive Berechnung der Fibonacci-Zahlen

$n$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$f_n$	0	1	1	2	3	5	8	13	21	34	55	89	144	233

Die **Fibonacci-Zahlen** (Leonardo Fibonacci, 1202)

Rekursive Definition mittels **Rekurrenzgleichung**:

$$f_0 = 0;$$

$$f_1 = 1;$$

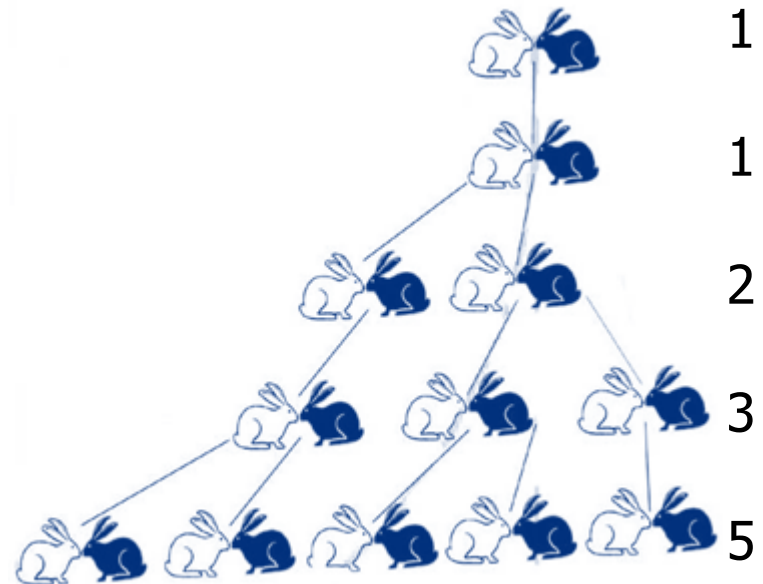
$$f_n = f_{n-1} + f_{n-2} \quad \text{für } n > 1$$

# Rekursive Berechnung der Fibonacci-Zahlen

*Wie viele Kaninchenpaare entstehen im Verlauf eines Jahres aus einem Paar?*

Ein Mann hielt ein Paar Kaninchen an einem Ort, der ringsum von einer Mauer umgeben war, um herauszufinden, wie viele Paare daraus in einem Jahr entstünden. Dabei ist es ihre Natur, jeden Monat ein neues Paar auf die Welt zu bringen, und sie gebären erstmals im zweiten Monat nach ihrer Geburt. Weil das obengenannte Paar schon im ersten Monat gebiert, kannst du es verdoppeln, so dass nach einem Monat zwei Paare da sind. Von diesen gebiert eines, d.h. das erste, im zweiten Monat wieder; und so gibt es im zweiten Monat 3 Paare. Von denen werden in einem Monat 2 wieder trüchtig, so dass im dritten Monat zwei Kaninchenpaare geboren werden; und so sind es dann in diesem Monat 5 Paare....."

Deutsche Übersetzung aus dem 12. Kapitel  
des Liber abaci nach der lateinischen Edition  
von B. Boncompagni, Rom 1857, S. 283f.



Quelle: Bibliothek ETH Zürich

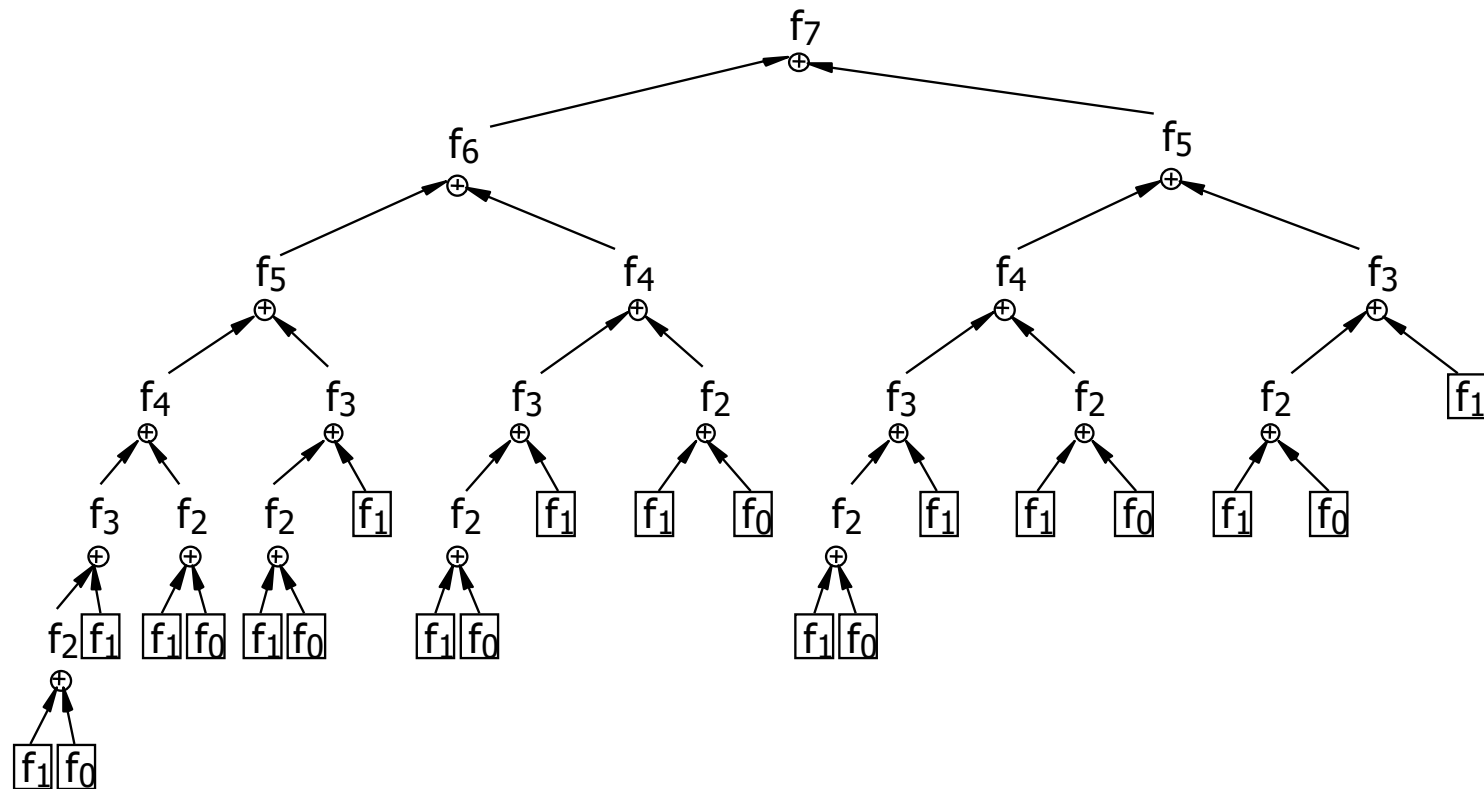
# Rekursive Berechnung der Fibonacci-Zahlen

- > Die Rekurrenzgleichung lässt sich direkt in eine rekursive Prozedur umsetzen

```
// calculates the n-th Fibonacci number
int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n - 1) + fib(n - 2);
}
```



# Rekursive Berechnung der Fibonacci-Zahlen



- > Bei  $n = 40$  sind bereits über 200 Mio. Additionen notwendig
- > Die rekursive Berechnung ist für größere  $n$  extrem aufwendig
- > Viele Fibonacci-Zahlen werden unnötig vielfach berechnet

# Optimierte Rekursive Berechnung

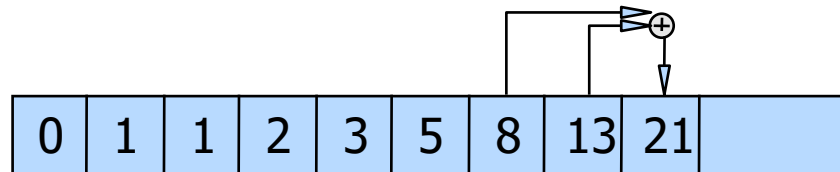
```
#define N 47
// saves and reuses fib numbers already computed
int fib(int n, int* fib_nums) {
    if (n == 0 || fib_nums[n] != 0)
        return fib_nums[n];
    fib_nums[n] =
        fib(n - 1, fib_nums) + fib(n - 2, fib_nums);
    return fib_nums[n];
}

int main() {
    int fn[N] = { 0, 1 };
    for (int i = 0; i < N; i++)
        printf("%i, %i\n", i, fib(i, fn));
}
```

# Iterative Berechnung der Fibonacci-Zahlen

- > Die folgende iterative Lösung hat nur linearen Rechenaufwand (z.B.  $n = 40 \rightarrow 39$  Additionen)

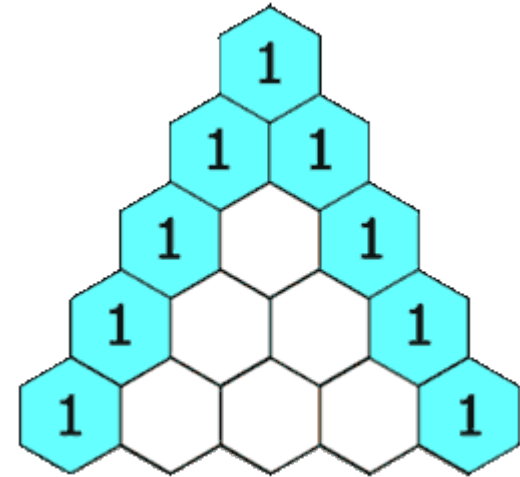
```
int fib(int n) {  
    if (n == 0) return 0;  
    else if (n == 1 || n == 2) return 1;  
    int *f = malloc(sizeof(int) * (n + 1));  
    f[0] = 0;  
    f[1] = 1;  
    for (int i = 2; i <= n; i++)  
        f[i] = f[i - 1] + f[i - 2];  
    int x = f[n];  
    free(f);  
    return x;  
}
```



# Pascalsches Dreieck

```
int pascal(int n, int k) {  
    if ((k == 0) || (k == n))  
        return 1;  
    else  
        return pascal(n - 1, k) +  
               pascal(n - 1, k - 1);  
}
```

```
int main(int argc, char* argv[]) {  
    for (int i = 0; i <= 4; i++) {  
        for (int j = 0; j <= i; j++)  
            printf("%i ", pascal(i, j));  
        printf("\n");  
    }  
}
```



# Ägyptisches Multiplizieren

## > Rekursive Variante

```
1 int mult(int a, int b) {  
3     if (b == 1)  
4         return a;  
5     else  
6         return (b % 2 ? a : 0) +  
7                 mult(2 * a, b / 2);  
8 }
```

Addiere Wert von a,  
wenn b ungerade ist.

Rufe mult mit  
doppeltem a und  
halbem b auf.

Linke Spalte:  
Halbieren bis 1  
Rechte Spalte:  
Verdoppeln

17	11
<del>8</del>	<del>22</del>
<del>4</del>	<del>44</del>
<del>2</del>	<del>88</del>
1	176

Die Werte der rechten Spalte  
addieren für die der Wert der  
linken Spalte ungerade ist.

-----  
187

# Türme von Hanoi

```
1 void move(char a, char b, char c, int n) {
2     if (n == 1)
3         printf("Move disc from %c to %c.\n", a, c);
4     else {
5         move(a, c, b, n - 1);
6         move(a, b, c, 1);
7         move(b, a, c, n - 1);
8     }
12 }
13
14 int main() {
15     move('a', 'b', 'c', 4); // 4 discs → 2**4 moves
16 }
```



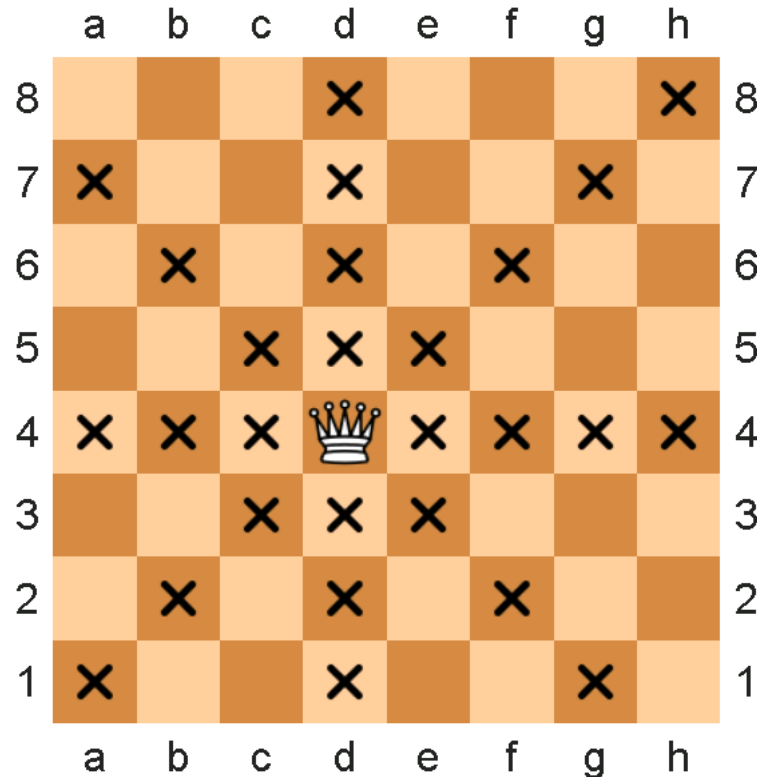
# Türme von Hanoi

1. Move disc from tower *a* to tower *b*.
2. Move disc from tower *a* to tower *c*.
3. Move disc from tower *b* to tower *c*.
4. Move disc from tower *a* to tower *b*.
5. Move disc from tower *c* to tower *a*.
6. Move disc from tower *c* to tower *b*.
7. Move disc from tower *a* to tower *b*.
8. Move disc from tower *a* to tower *c*.
9. Move disc from tower *b* to tower *c*.
10. Move disc from tower *b* to tower *a*.
11. Move disc from tower *c* to tower *a*.
12. Move disc from tower *b* to tower *c*.
13. Move disc from tower *a* to tower *b*.
14. Move disc from tower *a* to tower *c*.
15. Move disc from tower *b* to tower *c*.



# Damenproblem

- > Auf einem  $n \times n$  Schachbrett sollen  $n$  Damen so aufgestellt werden, dass keine zwei Damen sich gegenseitig bedrohen, also auf derselben Spalte, Zeile oder Diagonale stehen.





# Damenproblem

- > Eine der 92 beim 8 x 8 Schachbrett möglichen Lösungen ist
- > Dies ist die erste Lösung, die gefunden wird

	0	1	2	3	4	5	6	7
7	*	*	Q	*	*	*	*	*
6	*	*	*	*	*	Q	*	*
5	*	*	*	Q	*	*	*	*
4	*	Q	*	*	*	*	*	*
3	*	*	*	*	*	*	*	Q
2	*	*	*	*	Q	*	*	*
1	*	*	*	*	*	*	Q	*
0	Q	*	*	*	*	*	*	*

# Damenproblem


- > Ansatz: Die Damen werden spaltenweise von links nach rechts (also nacheinander in Spalte 0 bis Spalte 7) gesetzt.
- > Dann kann die **neue Dame** nur eine andere Dame in einer kleineren Spalte bedrohen bzw. von dieser bedroht werden.

	0	1	2	3	4	5	6	7
7	*	X	Q	*	*	*	*	*
6	*	*	X	*	*	*	*	*
5	X	X	X	Q	*	*	*	*
4	*	Q	X	*	*	*	*	*
3	*	X	*	*	*	*	*	*
2	X	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
0	Q	*	*	*	*	*	*	*

# Damenproblem

- > Erste Sackgasse nach der fünften Dame
- > Die sechste Dame kann nicht platziert werden
- > Daher wird jetzt die **fünfte Dame** weitergezogen


	0	1	2	3	4	5	6	7
7	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
4	*	*	Q	*	*	*	*	*
3	*	*	*	*	Q	*	*	*
2	*	Q	*	*	*	*	*	*
1	*	*	*	Q	*	*	*	*
0	Q	*	*	*	*	*	*	*



# Damenproblem

- > Zweite Sackgasse nach der fünften Dame
- > Die sechste Dame kann nicht platziert werden
- > Jetzt ist aber die fünfte Spalte zu Ende und daher wird die **vierte Dame** weitergesetzt

	0	1	2	3	4	5	6	7
7	*	*	*	*	Q	*	*	*
6	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
4	*	*	Q	*	*	*	*	*
3	*	*	*	*	*	*	*	*
2	*	Q	*	*	*	*	*	*
1	*	*	*	Q	*	*	*	*
0	Q	*	*	*	*	*	*	*



# Damenproblem

- > Dritte Sackgasse nach der siebten Dame
- > Die achte Dame kann nicht platziert werden
- > ...

	0	1	2	3	4	5	6	7
7	*	*	*	*	*	*	*	*
6	*	*	*	Q	*	*	*	*
5	*	*	*	*	*	*	Q	*
4	*	*	Q	*	*	*	*	*
3	*	*	*	*	*	Q	*	*
2	*	Q	*	*	*	*	*	*
1	*	*	*	*	Q	*	*	*
0	Q	*	*	*	*	*	*	*

# Damenproblem

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define N 8
4  int board[N][N];
5  int solutions = 0;
6  void print_board() {
7      solutions++;
8      for (int j = 0; j < N; j++)
9          printf(" %i", j % 10);
10     printf("\n");
11     for (int j = N - 1; j >= 0; j--) { // row
12         printf("%i", j % 10);
13         for (int i = 0; i < N; i++) // column
14             printf(board[i][j] ? "Q " : "* ");
15         printf("\n");
16     }
17     printf("\n");
18 }
```

# Damenproblem

```
19 void place(int i) {                                // place queen in column i
20     int at;
21     for (int x = 0; x < N; x++) { // try row x
22         at = 0;
23         for (int y = i - 1, z = 1; y >= 0; y--, z++)
24             if (x - z >= 0 && board[y][x - z]
25                 || x + z <= N - 1 && board[y][x + z]
26                 || board[y][x]) { // is new queen attacked?
27                 at = 1;           // yes, she is attacked
28                 break;
29             }
30     if (at == 0) { // no, new queen is not attacked
31         board[i][x] = 1;           // place queen
32         if (i < N - 1) place(i + 1); // place next queen
34         else print_board();        // or print solution
35         board[i][x] = 0;           // remove queen
36     } }
37 }
```

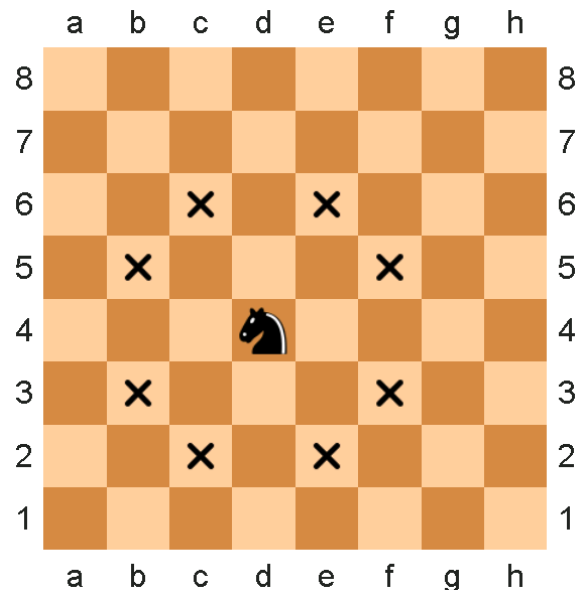
# Damenproblem

```
38 int main(int argc, char* argv[]) {  
39     place(0);  
40     printf("\nNumber of solutions: %i\n",  
41           solutions); // prints 92  
42 }
```

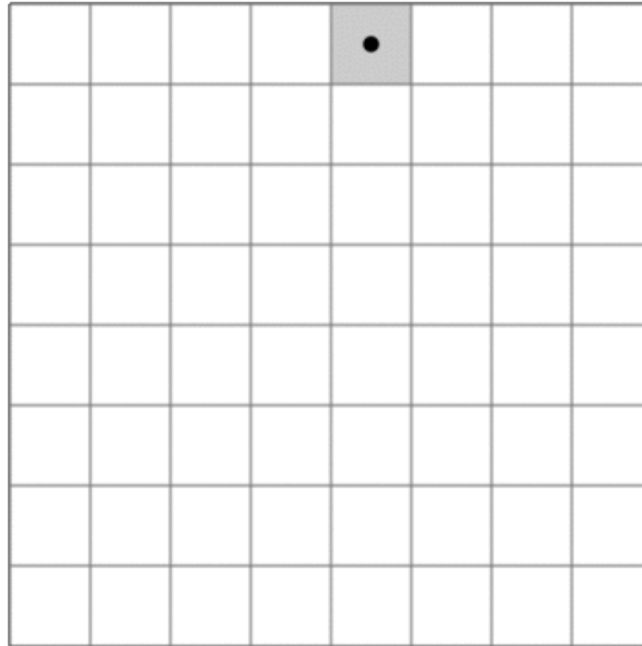


# Springerproblem

- > Ausgehend von einem Feld soll ein Springer alle Felder eines Schachbretts genau einmal besuchen → **Springertour**
- > **Geschlossene Springertour**
  - > Start- und Endfeld liegen *genau einen* Zug auseinander.
- > **Offene Springertour**
  - > Start- und Endfeld *liegen mehr als einen* Zug auseinander



# Springerproblem



# Springerproblem

```
#define N 8
int fields = N * N;
int board[N][N];

void print_board() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] <= 9)
                printf("0");
            printf("%i ", board[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

# Springerproblem

```
void place(int r, int c, int i) {
    if (r < 0 || r >= N || c < 0 || c >= N
        || board[r][c] != 0) // pos invalid or occupied?
        return;             // yes, return

    board[r][c] = i;        // no, place knight on new field
    if (i == fields)        // all fields visited?
        print_board();      // yes, print solution
    else {                  // no, place knight on reachable field
        place(r - 1, c + 2, i + 1); place(r - 1, c - 2, i + 1);
        place(r - 2, c + 1, i + 1); place(r - 2, c - 1, i + 1);
        place(r + 1, c - 2, i + 1); place(r + 1, c + 2, i + 1);
        place(r + 2, c - 1, i + 1); place(r + 2, c + 1, i + 1);
    }
    board[r][c] = 0;        // remove knight from field
}

int main() { place(0, 0, 1); }
```

# Springerproblem für $n = 8$

> Eine der möglichen (offenen) Lösungen ist

> 01 08 23 12 03 06 17 14

> 24 11 02 07 16 13 04 19

> 09 40 25 22 05 18 15 64

> 26 49 10 39 28 63 20 57

> 41 38 27 32 21 58 29 62

> 48 35 50 59 44 31 56 53

> 37 42 33 46 51 54 61 30

> 34 47 36 43 60 45 52 55

Anfang der Tour  
Ende der Tour

> Es gibt  $13.267.364.410.532 \approx 13 \cdot 10^{12}$  geschlossene  
bzw. ca.  $1.22 \cdot 10^{15}$  offene Springertouren

# Sudoku

- > Aufgabe: Jede Zelle mit einer der Ziffern 1 – 9 füllen.
- > Dabei darf jede Ziffer nur einmal vorkommen
  - > in jeder Zeile,
  - > in jeder Spalte und
  - > in jeder 3 x 3 Box.

8	x	x	x	x	x	x	x	x
x	x	3	6	x	x	x	x	x
x	7	x	x	9	x	2	x	x
x	5	x	x	x	7	x	x	x
x	x	x	x	4	5	7	x	x
x	x	x	1	x	x	x	3	x
x	x	1	x	x	x	x	6	8
x	x	8	5	x	x	x	1	x
x	9	x	x	x	x	4	x	x

Anfangssituation

# Sudoku

```
int board[9][9] =  
    {{8, 0, 0, 0, 0, 0, 0, 0, 0},  
     {0, 0, 3, 6, 0, 0, 0, 0, 0},  
     {0, 7, 0, 0, 9, 0, 2, 0, 0},  
     {0, 5, 0, 0, 0, 7, 0, 0, 0},  
     {0, 0, 0, 0, 4, 5, 7, 0, 0},  
     {0, 0, 0, 1, 0, 0, 0, 3, 0},  
     {0, 0, 1, 0, 0, 0, 0, 6, 8},  
     {0, 0, 8, 5, 0, 0, 0, 1, 0},  
     {0, 9, 0, 0, 0, 0, 4, 0, 0}};
```

# Sudoku

```
int check_row_or_column(int check_row, int x) {  
    int roc[10] = { 0 };  
    int number;  
    for (int i = 0; i < 9; i++) {  
        number = check_row ? board[x][i] : board[i][x];  
        if (number != 0 && roc[number])  
            return 0; // second match  
        else  
            roc[number] = 1; // first match  
    }  
    return 1;  
}
```



# Sudoku

```
int check_box(int x, int y) {  
    int box[10] = { 0 };  
    int r = x * 3;  
    int s = y * 3;  
    int number;  
    for (int rr = r; rr < r + 3; rr++)  
        for (int ss = s; ss < s + 3; ss++) {  
            number = board[rr][ss];  
            if (number != 0 && box[number])  
                return 0; // second match  
            else  
                box[number] = 1; // first match  
        }  
    return 1; }
```

# Sudoku

```
void place_next(int r, int c) {  
    if (c != 8) place(r, c + 1);  
    else if (r != 8) place(r + 1, 0);  
    else print_board();  
}
```

```
void place(int r, int c) {  
    if (board[r][c] == 0) {  
        for (int n = 1; n <= 9; n++) {  
            board[r][c] = n;  
            if (check_row_or_column(1, r)  
                && check_row_or_column(0, c)  
                && check_box(r / 3, c / 3))  
                place_next(r, c);  
        }  
        board[r][c] = 0;  
    } else place_next(r, c); }  
}
```

# Sudoku

```
void print_board() {  
    for (int r = 0; r < 9; r++) {  
        for (int c = 0; c < 9; c++)  
            printf("%i ", board[r][c]);  
        printf("\n");  
    }  
    printf("\n");  
}
```

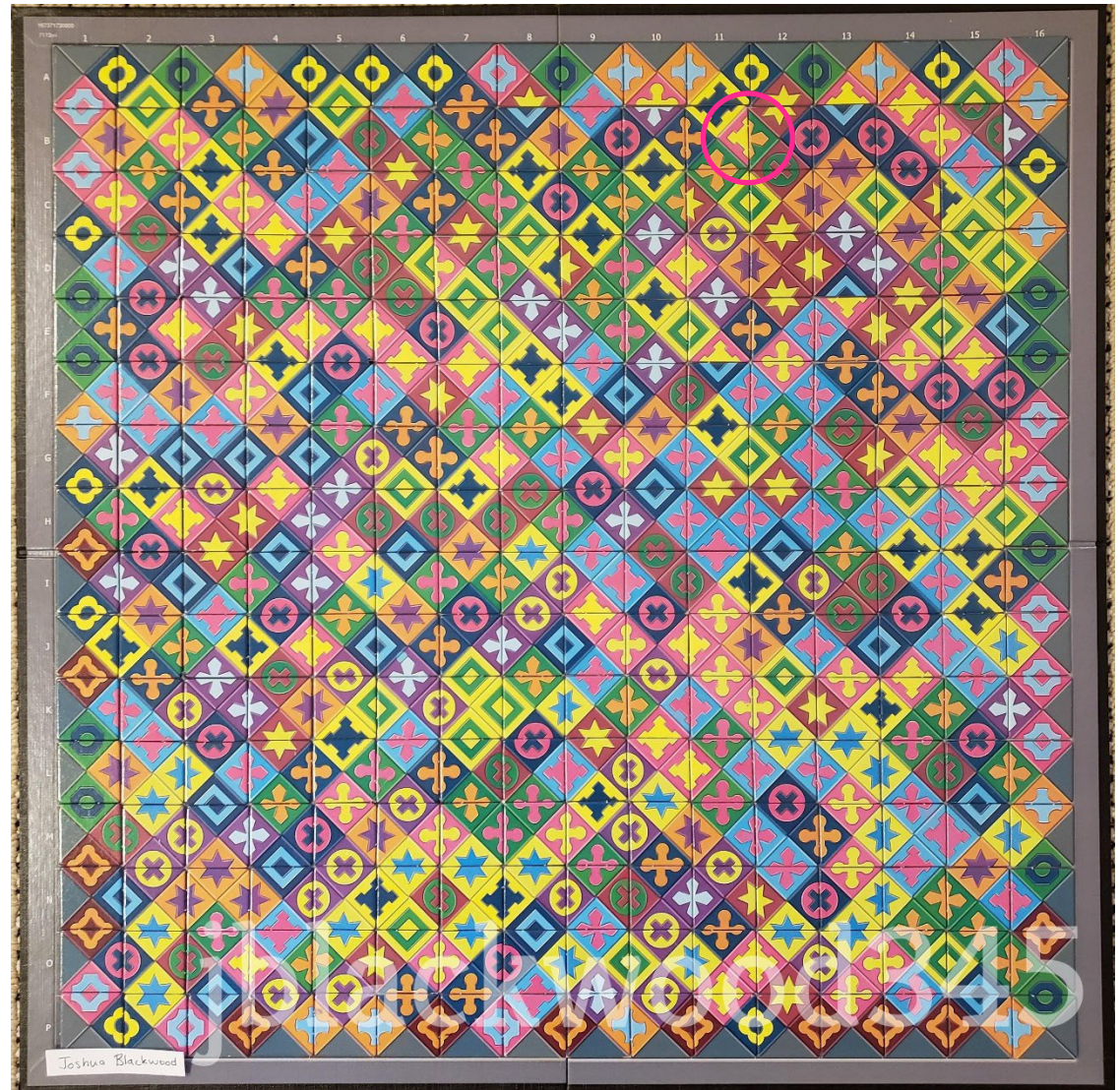
```
int main() {  
    place(0, 0);  
}
```

# Sudoku

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

# Eternity 2 Puzzle

- > 16 x 16 Puzzle von 2007
- > Lösung sehr schwer
- > Bild zeigt Teillösung mit einigen Fehlern
- > 2.000.000\$ Preisgeld für Lösung vor dem 31.12.2010 verfiel



# Schiebepuzzle

# Schiebepuzzle

- > Ziel: Puzzle durch Verschieben einzelner Zahlen in die geordnete Reihenfolge bringen
- > Ungefähr 10 Billionen gültige Startpositionen
- > 17 Startpositionen erfordern die maximale Zahl von 80 Zügen



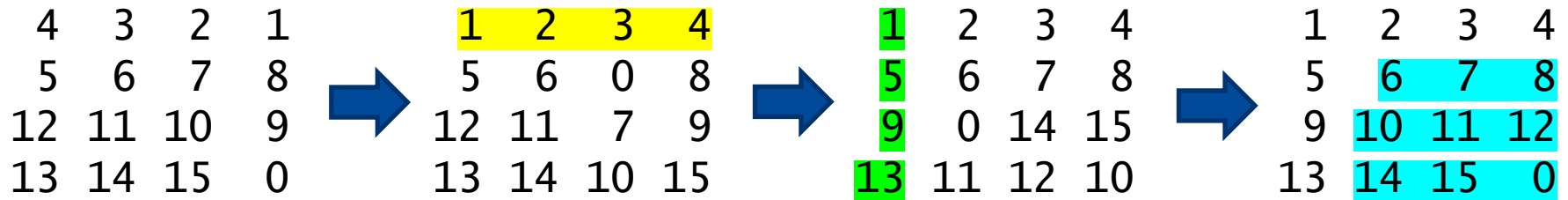
Exemplarische  
Startposition



Gewünschte  
Endposition

# Lösungsstrategie

- > Lösungsraum ist für vollständiges Durchsuchen zu groß
- > Zielgerichtete Lösungsstrategie in drei Phasen
  1. Erste Zeile ordnen und danach nicht mehr ändern
  2. Erste Spalte ordnen und danach nicht mehr ändern
  3. Lösung für verbleibendes 3x3-Puzzle suchen





# Schiebepuzzle

```
int main() {  
    while (!check()) {  
        <start new round>  
        move(max_depth);  
        restore_best();  
        print_moves();  
        print_board();  
        <start new phase if necessary>  
    }  
}
```

# Schiebepuzzle

```
void move(int x) {  
    if (x == 0)  
        return;  
  
    int sc = score(max_depth - x);  
    if (sc - x > min_moves)  
        return;  
  
    move_left(x);  
    move_up(x);  
    move_right(x);  
    move_down(x);  
}
```

# Schiebepuzzle

```
void move_left(int x) {  
    if (col == 4 - 1) return;  
    push_move(row, col + 1);  
    swap(row, col, row, col + 1);  
    col++; move(x - 1); col--;  
    swap(row, col, row, col + 1);  
    pop_move();  
}
```

```
void move_right(int x) {  
    if (col == 0 || col == 1 && phase > 2) return;  
    push_move(row, col - 1);  
    swap(row, col, row, col - 1);  
    col--; move(x - 1); col++;  
    swap(row, col, row, col - 1);  
    pop_move();  
}
```

# Schiebepuzzle

```
void move_down(int x) {  
    if (row == 0 || row == 1 && phase > 1) return;  
    push_move(row - 1, col);  
    swap(row, col, row - 1, col);  
    row--; move(x - 1); row++;  
    swap(row, col, row - 1, col);  
    pop_move();  
}
```

```
void move_up(int x) {  
    if (row == 4 - 1) return;  
    push_move(row + 1, col);  
    swap(row, col, row + 1, col);  
    row++; move(x - 1); row--;  
    swap(row, col, row + 1, col);  
    pop_move();  
}
```

# Schiebepuzzle

```
int score(int z) {  
    int x, c, r;  
    int moves = 0;  
    for (int i = 0; i < 4; i++) {  
        for (int j = 0; j < 4; j++) {  
            x = board[i][j];  
            switch (phase) {  
                case 1:  
                    if (x == 0 || x > 4) continue;  
                    break;  
                case 2:  
                    if (x == 0 || x != 1 && x != 5 && x != 9 && x != 13)  
                        continue;  
                    break;  
                case 3:  
                    if (x <= 5 || x == 9 || x == 13) continue;  
                    break;  
            }  
        }  
    }  
}
```

# Schiebepuzzle

```
        r = (x - 1) / 4;  
        c = x - r * 4 - 1;  
        moves += abs(i - r) + abs(j - c);  
    }  
}  
  
if (moves < min_moves || moves == min_moves && z < zmoves) {  
    min_moves = moves;  
    zmoves = z;  
    save_best();  
}  
  
return moves;  
}
```

# Schiebepuzzle

\*\*\* initial position with space at row = 3, col = 3

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

\*\*\* position after 100 random shuffles

1	2	0	3
7	11	6	4
5	10	14	8
9	13	15	12

# Schiebepuzzle

\*\*\* starting new round: 1

phase = 1, round = 1, min\_moves = 2, moves = 0

phase = 1, round = 1, min\_moves = 1, moves = 1

phase = 1, round = 1, min\_moves = 0, moves = 2

(0, 3) → (1, 3) →

1	2	3	4
---	---	---	---

7	11	6	0
---	----	---	---

5	10	14	8
---	----	----	---

9	13	15	12
---	----	----	----



# Schiebepuzzle

\*\*\* starting new phase: 2

\*\*\* starting new round: 1

phase = 2, round = 1, min\_moves = 3, moves = 0

phase = 2, round = 1, min\_moves = 2, moves = 13

phase = 2, round = 1, min\_moves = 1, moves = 13

phase = 2, round = 1, min\_moves = 0, moves = 12

phase = 2, round = 1, min\_moves = 0, moves = 10

phase = 2, round = 1, min\_moves = 0, moves = 8

phase = 2, round = 1, min\_moves = 0, moves = 6

→ (1, 2) → (1, 1) → (1, 0) → (2, 0) → (3, 0) → (3, 1) →

1	2	3	4
5	7	11	6
9	10	14	8
13	0	15	12

# Schiebepuzzle

\*\*\* starting new phase: 3

\*\*\* starting new round: 1

phase = 3, round = 1, min\_moves = 8, moves = 0

phase = 3, round = 1, min\_moves = 7, moves = 13

phase = 3, round = 1, min\_moves = 6, moves = 12

phase = 3, round = 1, min\_moves = 5, moves = 13

phase = 3, round = 1, min\_moves = 4, moves = 12

phase = 3, round = 1, min\_moves = 3, moves = 13

→ (2, 1) → (1, 1) → (1, 2) → (1, 3) → (2, 3) → (3, 3) →  
(3, 2) → (2, 2) → (2, 1) → (1, 1) → (1, 2) →  
(2, 2) → (2, 1) →

1	2	3	4
5	6	7	8
9	0	11	12
13	10	14	15

# Schiebepuzzle

\*\*\* starting new round: 2

phase = 3, round = 2, min\_moves = 3, moves = 0

phase = 3, round = 2, min\_moves = 2, moves = 13

phase = 3, round = 2, min\_moves = 1, moves = 12

phase = 3, round = 2, min\_moves = 0, moves = 13

phase = 3, round = 2, min\_moves = 0, moves = 3

→ (3, 1) → (3, 2) → (3, 3)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

\*\*\* solved in 24 moves, analyzed 3.975.674 boards !!! \*\*\*

# Header Files

# Header Files

- > Header-Dateien beinhalten Vereinbarungen (z.B. Deklarationen von Prozeduren und Funktionen), welche von mehreren Programmdateien benötigt werden.
- > Vor der eigentlichen Übersetzung fügt der Präprozessor die angefragte Header-Datei in die Programmdatei ein.
- > Diese wird dann vom Compiler übersetzt und die zugehörige Implementierung dazu gelinkt.
- > Header-Dateien ermöglichen somit eine gewisse Modularität und eine leichtere Wiederverwendbarkeit.

# Header File matrix.h

```
// create dim x dim matrix and return it  
int** matrix_create(int dim);
```

```
// print matrix a  
int matrix_print(int** a, int dim);
```

```
// initialize elements of matrix a with random values  
void matrix_init_randomly(int** a, int dim);
```

```
// add matrix b to matrix a and return result matrix c  
void matrix_add(int** a, int** b, int** c, int dim);
```

```
// subtract matrix b from matrix a  
// and return result matrix c  
void matrix_sub(int** a, int** b, int** c, int dim);
```

# Implementation matrix.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "matrix.h"

void matrix_print(int** a, int dim) {
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++)
            printf("%2i ", a[i][j]);
        printf("\n");
    }
    printf("\n");
}

int** matrix_create(int dim) {
    int** a = malloc(dim * sizeof(int*));
    for (int i = 0; i < dim; i++)
        a[i] = malloc(dim * sizeof(int));
    return a;
}
```

# Implementation matrix.c

```
void matrix_init_randomly(int** a, int dim) {
    srand(time(0));
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            a[i][j] = 1 + rand() % 6;
}
```

```
void matrix_add(int** a, int** b, int** c, int dim) {
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

```
void matrix_sub(int** a, int** b, int** c, int dim) {
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            c[i][j] = a[i][j] - b[i][j];
}
```



# Hauptprogramm matrix2.c

```
#include "matrix.h"
```

```
int main() {  
    int dim = 3;  
    int** x = matrix_create(dim);  
    int** y = matrix_create(dim);  
    int** z = matrix_create(dim);  
  
    matrix_init_randomly(x, dim);  
    matrix_init_randomly(y, dim);  
    matrix_init_randomly(z, dim);  
  
    matrix_add(x, y, z, dim);  
  
    matrix_print(x, dim);  
    matrix_print(y, dim);  
    matrix_print(z, dim);  
}
```

# **Vielen Dank für Ihre Aufmerksamkeit!**

**Univ.-Prof. Dr.-Ing. Gero Mühl**

`gero.muehl@uni-rostock.de`  
`https://www.ava.uni-rostock.de`