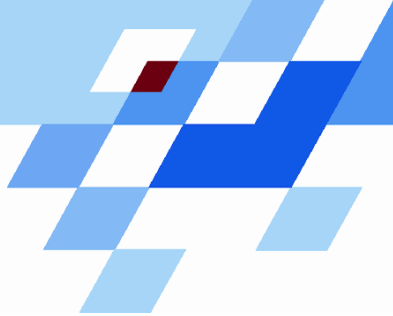


Wintersemester 2006/07

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias **Einführung in die Programmierung)**

(Vorlesung)

Prof. Dr. Günter Rudolph

Fachbereich Informatik

Lehrstuhl für Algorithm Engineering

Inhalt

- Funktionen
  - mit / ohne Parameter
  - mit / ohne Rückgabewerte

  } ☑

- Übergabemechanismen
  - Übergabe eines Wertes
  - Übergabe einer Referenz
  - Übergabe eines Zeigers

  } ☑

- Programmieren mit Funktionen → heute

  \+ static / inline / MAKROS

Übergabe von zweidimensionalen Arrays:

Im Prototypen müssen entweder **beide Indexkonstanten** oder **nur die Spaltenkonstante** abgegeben werden!

```
void inkrement(const unsigned int zeilen, int b[][4]) {
  int i, j;
  for (i = 0; i < zeilen; i++)
    for (j = 0; j < 4; j++) b[i][j]++;
}
```

```
int main() {
  int i, j, a[][4] = {{ 2, 4, 6, 8 }, { 9, 7, 5, 3 }};
  inkrement(2, a);
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 4; j++) cout << a[i][j] << " ";
    cout << endl;
  }
}
```

Übergabe von zweidimensionalen Arrays:

```
void inkrement(const unsigned int z, int b[][5]);
```

Mindestanforderung!

oder:

```
void inkrement(const unsigned int z, int b[2][5]);
```

Unnötig, wenn immer alle Zeilen bearbeitet werden:
Zeilenzahl zur Übersetzungszeit bekannt!

Wenn aber manchmal nur die erste Zeile bearbeitet wird, dann könnte das Sinn machen!

Übergabe eines zweidimensionalen Arrays

Funktionsaufruf:

Funktionsname(Arrayname) ;

Variable = Funktionsname(Arrayname) ;

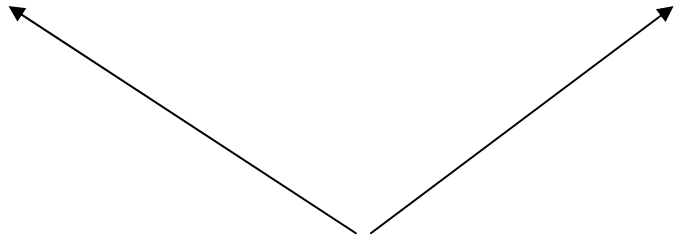
```
int a[][2] = {{1,2},{3,4}};
inkrement(2, a);
```

oder:

Funktionsname(&Arrayname[0][0]) ;

Variable = Funktionsname(&Arrayname[0][0]) ;

```
int a[][2] = {{1,2},{3,4}};
inkrement(2, &a[0][0]);
```



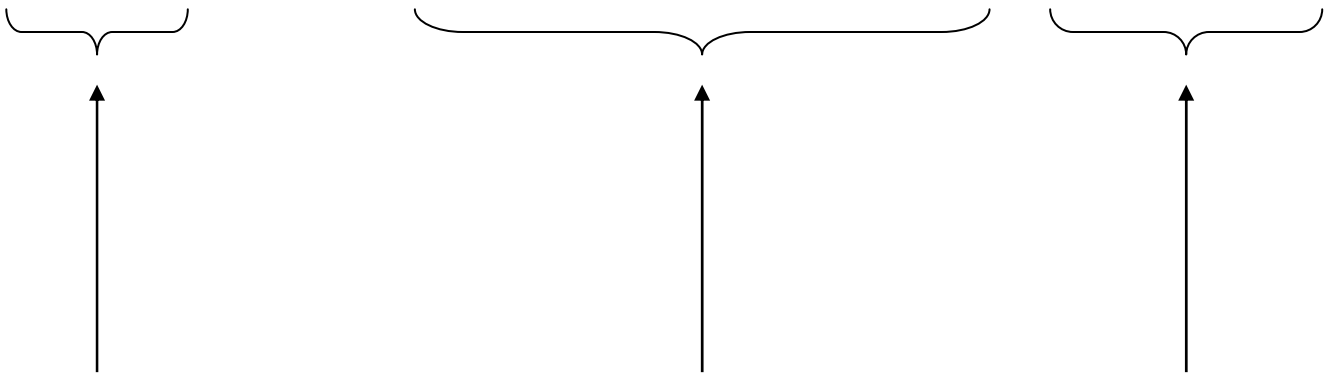
Tatsächlich: Übergabe des Arrays mit Zeigern!

Aufgabe:

Finde Minimum in einem Array von Typ **`double`**
Falls Array leer, gebe Null zurück ☹ → später: Ausnahmebehandlung

Prototyp, Schnittstelle:

```
double dblmin(const unsigned int n, double a[]);
```



Rückgabe: Wert des Minimums

max. Größe des Arrays oder Anzahl Elemente

Array vom Typ **`double`**

Aufgabe:

Finde Minimum in einem Array von Typ **`double`**
Falls Array leer, gebe Null zurück

Implementierung:

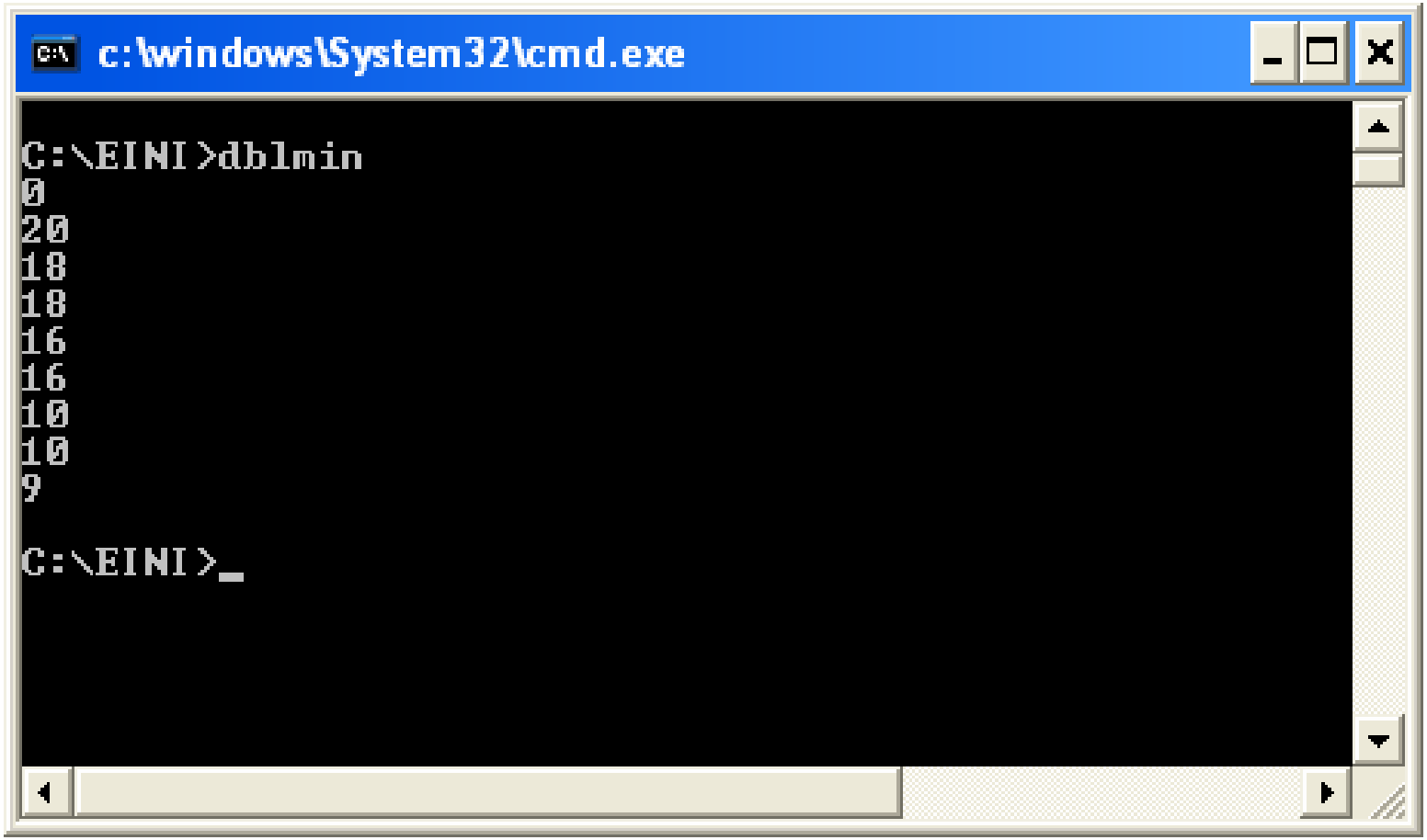
```
double dblmin(const unsigned int n, double a[]) {
  // leeres Array?
  if (n == 0) return 0.0;
  // Array hat also mindestens 1 Element!
  double min = a[0];
  int i;
  for(i = 1; i < n; i++)          // Warum i = 1 ?
    if (a[i] < min) min = a[i];
  return min;
}
```

Test:

```
double dblmin(const unsigned int n, double a[]) {
  if (n == 0) return 0.0;
  double min = a[0];
  int i;
  for(i = 1; i < n; i++)
    if (a[i] < min) min = a[i];
  return min;
}
```

```
int main() {
  double a[] = {20.,18.,19.,16.,17.,10.,12.,9.};
  int k;
  for (k = 0; k <= 8; k++)
    cout << dblmin(k, a) << endl;
  return 0;
}
```

Der „Beweis“ …


c:\windows\System32\cmd.exe
C:\EINI>dblmin
0
20
18
18
16
16
10
10
9
C:\EINI>_

Variation der Aufgabe:

Finde Index des 1. Minimums in einem Array von Typ **`double`**
Falls Array leer, gebe **-1** zurück.

Entwurf mit Implementierung:

```
int imin(const unsigned int n, double a[]) {
  // leeres Array?
  if (n == 0) return -1;
  // Array hat also mindestens 1 Element!
  int i, imin = 0;
  for(i = 1; i < n; i++)
    if (a[i] < a[imin]) imin = i;
  return imin;
}
```

Neue Aufgabe:

Sortiere Elemente in einem Array vom Typ **`double`.**
Verändere dabei die Werte im Array.

Bsp:

8	44	14	81	12
8	44	14	81	12
12	44	14	81	8
12	44	14	81	8
81	44	14	12	8
81	44	14	12	8
81	44	14	12	8
81	44	14	12	8
81	44	14	12	8
81	44	14	12	8

min{ 8, 44, 14, 81 } = 8 < 12 ?

ja → tausche 8 und 12

min{ 12, 44, 14 } = 12 < 81 ?

ja → tausche 12 und 81

min{ 81, 44 } = 44 < 14 ?

nein → keine Vertauschung

min{ 81} = 81 < 44 ?

nein → keine Vertauschung

fertig!

Neue Aufgabe:

Sortiere Elemente in einem Array vom Typ **`double`**.
Verändere dabei die Werte im Array.

Mögliche Lösung:

```
void sortiere(const unsigned int n, double a[]) {
  int i, k;
  for (k = n - 1; k > 1; k--) {
    i = imin(k - 1, a);
    if (a[i] < a[k]) swap_dbl(a[i], a[k]);
  }
}
```

```
void swap_dbl(double &a, double &b) {
  double h = a; a = b; b = h;
}
```

Wir halten fest:

- Arrays sind statische Datenbehälter: ihre Größe ist nicht veränderbar!
- Die Bereichsgrenzen von Arrays sollten an Funktionen übergeben werden, wenn sie nicht zur Übersetzungszeit bekannt sind.
- Die Programmierung mit Arrays ist unhandlich!
  Ist ein Relikt aus C. In C++ gibt es handlichere Datenstrukturen!
  (Kommt bald … Geduld!)
- Die Aufteilung von komplexen Aufgaben in kleine Teilaufgaben, die dann in parametrisierten Funktionen abgearbeitet werden, erleichtert die Lösung des Gesamtproblems. Beispiel: Sortieren!
- Funktionen für spezielle kleine Aufgaben sind wieder verwertbar und bei anderen Problemstellungen einsetzbar.

⇒ Deshalb gibt es viele Funktionsbibliotheken,
die die Programmierung erleichtern!

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

`exp()`	Exponentialfunktion e^x
`ldexp()`	Exponent zur Basis 2, also 2^x
`log()`	natürlicher Logarithmus $\log_e x$
`log10()`	Logarithmus zur Basis 10, also $\log_{10} x$
`pow()`	Potenz x^y
`sqrt()`	Quadratwurzel
`ceil()`	nächst größere oder gleiche Ganzzahl
`floor()`	nächst kleinere oder gleiche Ganzzahl
`fabs()`	Betrag einer Fließkommazahl
`modf()`	zerlegt Fließkommazahl in Ganzzahlteil und Bruchteil
`fmod()`	Modulo-Division für Fließkommazahlen

und zahlreiche trigonometrische Funktionen wie `sin, cosh, atan`

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

`atof()`	Zeichenkette in Fließkommazahl wandeln
`atoi()`	Zeichenkette in Ganzzahl wandeln (**A**SCII **to** **i**nteger)
`atol()`	Zeichenkette in lange Ganzzahl wandeln
`strtod()`	Zeichenkette in `double` wandeln
`strtol()`	Zeichenkette in `long` wandeln
`rand()`	Liefert eine Zufallszahl
`srand()`	Initialisiert den Zufallszahlengenerator

und viele andere …

Wofür braucht man diese Funktionen?

Funktion `main` (→ Hauptprogramm)

wir kennen:

```
int main() {
  // ...
  return 0;
}
```

allgemeiner:

```
int main(int argc, char *argv[]) {
  // ...
  return 0;
}
```

Anzahl der Elemente

Array von Zeichenketten

Programmaufruf in der Kommandozeile:

```
D:\> mein_programm 3.14 hallo 8
```

`argv[0]` `argv[1]` `argv[2]` `argv[3]`

Alle Parameter werden textuell als Zeichenkette aus der Kommandozeile übergeben!

`argc` hat Wert 4

Funktion `main` (→ Hauptprogramm)

Programmaufruf in der Kommandozeile:

```
D:\> mein_programm 3.14 hallo 8
```

Alle Parameter werden **textuell** als Zeichenkette aus der Kommandozeile übergeben!

```
#include <stdlib>

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

  if (argc != 4) {
    cerr << argv[0] << ": 3 Argumente erwartet!" << endl;
    return 1;
  }
  double dwert = atof(argv[1]);
  int iwert = atoi(argv[3]);

  // ...

}
```

```
#include <ctype.h>
```

`tolower()`	Umwandlung in Kleinbuchstaben
`toupper()`	Umwandlung in Großbuchstaben
`isalpha()`	Ist das Zeichen ein Buchstabe?
`isdigit()`	Ist das Zeichen eine Ziffer?
`isxdigit()`	Ist das Zeichen eine hexadezimale Ziffer?
`isalnum()`	Ist das Zeichen ein Buchstabe oder eine Ziffer?
`iscntrl()`	Ist das Zeichen ein Steuerzeichen?
`isprint()`	Ist das Zeichen druckbar?
`islower()`	Ist das Zeichen ein Kleinbuchstabe?
`isupper()`	Ist das Zeichen ein Großbuchstabe?
`isspace()`	Ist das Zeichen ein Leerzeichen?

Beispiele für nützliche Hilfsfunktionen:

Aufgabe: Wandle alle Zeichen einer Zeichenkette in Grossbuchstaben!

```
#include <ctype.h>

char *ToUpper(char *s) {
  char *t = s;
  while (*s != 0) *s++ = toupper(*s);
  return t;
}
```

Aufgabe: Ersetze alle nicht druckbaren Zeichen durch ein Leerzeichen.

```
#include <ctype.h>

char *MakePrintable(char *s) {
  char *t = s;
  while (*s != 0) *s++ = isprint(*s) ? *s : ' ';
  return t;
}
```

```
#include <time.h>
```

`time()`	Liefert aktuelle Zeit in Sekunden seit dem 1.1.1970 UTC
`localtime()`	wandelt UTC-„Sekundenzeit“ in lokale Zeit (`struct`)
`asctime()`	wandelt Zeit in `struct` in lesbare Form als `char[]`

und viele weitere mehr …

```
#include <iostream>
#include <time.h>

int main() {
  time_t jetzt = time(0);
  char *uhrzeit = asctime(localtime(&jetzt));
  std::cout << uhrzeit << std::endl;
  return 0;
}
```

Statische Funktionen (in dieser Form: Relikt aus C)

sind Funktionen, die nur für Funktionen in derselben Datei sichtbar (aufrufbar) sind!

Funktionsdeklaration:

`static` Datentyp Funktionsname(Datentyp Bezeichner);

```
#include <iostream>
using namespace std;

static void funktion1() {
  cout << "F1" << endl;
}

void funktion2() {
  funktion1();
  cout << "F2" << endl;
}
```

Datei *Funktionen.cpp*

```
void funktion1();
void funktion2();

int main() {

  funktion1();
  funktion2();

  return 0;
}
```

Fehler! `funktion1` nicht sichtbar!

wenn entfernt, dann gelingt Compilierung:

g++ *.cpp –o test

Datei *Haupt.cpp*

Inline Funktionen

sind Funktionen, deren Anweisungsteile an der Stelle des Aufrufes eingesetzt werden

Funktionsdeklaration:

`inline` Datentyp Funktionsname(Datentyp Bezeichner);

```
#include <iostream>
using namespace std;

inline void funktion() {
  cout << "inline" << endl;
}

int main() {
  cout << "main" << endl;
  funktion();
  return 0;
}
```

→ wird zur Übersetzungszeit ersetzt zu:

```
#include <iostream>
using namespace std;

int main() {
  cout << "main" << endl;
  cout << "inline" << endl;
  return 0;
}
```

Inline Funktionen

Vorteile:

1. Man behält alle positiven Effekte von Funktionen:
   - Bessere Lesbarkeit / Verständnis des Codes.
   - Verwendung von Funktionen sichert einheitliches Verhalten.
   - Änderungen müssen einmal nur im Funktionsrumpf durchgeführt werden.
   - Funktionen können in anderen Anwendungen wieder verwendet werden.
2. Zusätzlich bekommt man schnelleren Code!
   (keine Sprünge im Programm, keine Kopien bei Parameterübergaben)

Nachteil:

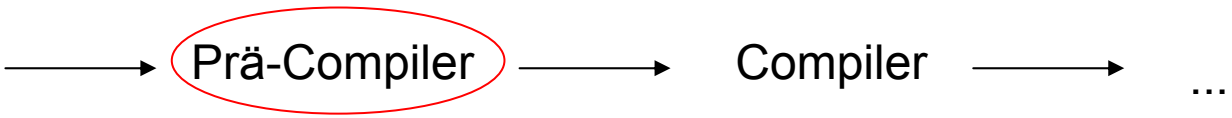
Das übersetzte Programm wird größer (benötigt mehr Hauptspeicher)

Deshalb: vorangestelltes `inline` ist nur eine Anfrage an den Compiler! Keine Pflicht!

„Inline-Funktionsartiges“ mit Makros

Da müssen wir etwas ausholen ...

```
#include <iostream>
int main() {
  int x = 1;
  std::cout << x*x;
  return 0;
}
```



ersetzt Makros (beginnen mit #):
z.B. lädt Text aus Datei **`iostream.h`**

`#define` Makroname Ersetzung

Bsp:

```
#define MAX_SIZE 100
#define ASPECT_RATIO 1.653
```

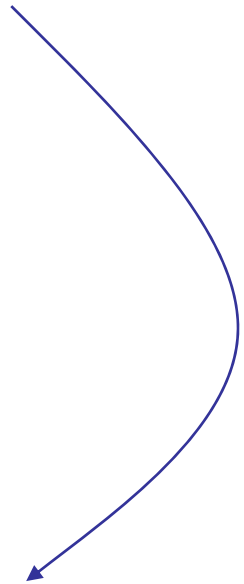
Makronamen im Programmtext werden vom Prä-Compiler durch ihre Ersetzung ersetzt

```
#define MAX_SIZE 100

void LeseSatz(char *Puffer) {

  char c = 0;
  int i = 0;
  while (i < MAX_SIZE && c != '.') {
    cin >> c;
    *Puffer++ = c;
  }
}
```

```
void LeseSatz(char *Puffer) {

  char c = 0;
  int i = 0;
  while (i < 100 && c != '.') {
    cin >> c;
    *Puffer++ = c;
  }
}
```

Makros ...

dieser Art sind Relikt aus C!



Nach Durchlauf durch den Prä-Compiler

Tip: NICHT VERWENDEN!

stattdessen:

```
const int max_size = 100;
```

„Inline-Funktionsartiges“ mit Makros

```
#define SQUARE(x) x*x
```

Vorsicht: `SQUARE(x+3)` ergibt: `x+3*x+3`

besser:

```
#define SQUARE(x) (x)*(x)
```

→ `SQUARE(x+3)` ergibt: `(x+3)*(x+3)`

noch besser:

```
#define SQUARE(x) ((x)*(x))
```

→ `SQUARE(x+3)` ergibt: `((x+3)*(x+3))`

auch mehrere Parameter möglich:

```
#define MAX(x, y) ((x)>(y)?(x):(y))
```

```
int a = 5;
int z = MAX(a+4, a+a);
```

ergibt:

```
int a = 5;
int z =((a+4)>(a+a)?(a+4):(a+a));
```

Nachteil:
ein Ausdruck wird **2x** ausgewertet!

„Inline-Funktionsartiges“ mit Makros (Relikt aus C)

Beliebiger Unsinn möglich ...

```
// rufe Funktion fkt() mit maximalem Argument auf
#define AUFRUF_MIT_MAX(x,y) fkt(MAX(x,y))
```

> „Makros wie diese haben so viele Nachteile,
> dass schon das Nachdenken über sie nicht zu ertragen ist.“
>
> Scott Meyers: Effektiv C++ programmieren, S. 32, 3. Aufl., 2006.

```
int a = 5, b = 0;
AUFRUF_MIT_MAX(++a, b);    // a wird 2x inkrementiert
AUFRUF_MIT_MAX(++a, b+10); // a wird 1x inkrementiert
```

Tipp: *statt funktionsartigen Makros besser richtige inline-Funktionen verwenden!*