technische universität dortmund

Einführung in die Programmierung

Wintersemester 2012/13

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Inhalt

- Funktionen
  - mit / ohne Parameter
  - mit / ohne Rückgabewerte
- Übergabemechanismen
  - Übergabe eines Wertes
  - Übergabe einer Referenz
  - Übergabe eines Zeigers
- Funktionsschablonen (Übergabe von Typen)
- Programmieren mit Funktionen

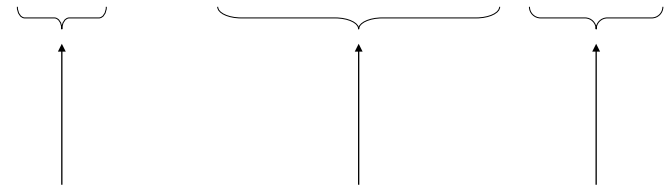
  + Exkurs: Endliche Automaten
  + static / inline / MAKROS

Aufgabe:

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

Prototyp, Schnittstelle:

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



Rückgabe: Wert des Minimums

max. Größe des Arrays oder Anzahl Elemente

Array vom Typ **`double`**

1. Aufgabe:

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

Implementierung:

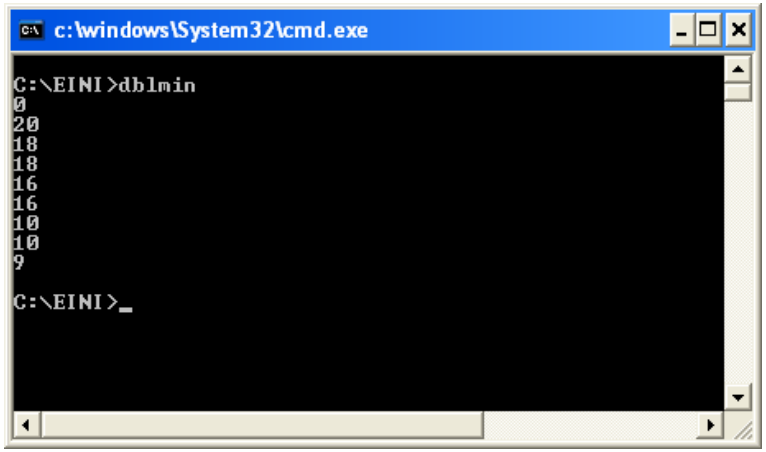
```
double dblmin(unsigned int const n, double a[]) {
  // leeres Array?
  if (n == 0) return 0.0;
  // Array hat also mindestens 1 Element!
  double min = a[0];
  int i;
  for(i = 1; i < n; i++)       // Warum i = 1 ?
    if (a[i] < min) min = a[i];
  return min;
}
```

Test:

```
double dblmin(unsigned int const 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“ …



Variation der 1. Aufgabe:

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

Implementierung:

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

Beobachtung: Programmtext fast identisch, nur Datentyp verändert auf `short`

Beobachtung: Programmtext fast identisch, nur Datentyp verändert

⇒ man müsste auch den Datentyp wie einen Parameter übergeben können!

Implementierung durch **Schablonen**:

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

Test:

```
template <typename T>
T arrayMin(unsigned int const n, T a[]) {
  if (n == 0) return 0.0;
  T 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.};
  short  b[] = {4, 9, 3, 5, 2, 6, 4, 1 };
  int k;
  for (k = 0; k <= 8; k++) {
    cout << arrayMin<double>(k, a) << " - ";
    cout << arrayMin<short>(k, b)  << endl;
  return 0;
}
```

Beim Compilieren:
Automatische Codegenerierung!

Funktionsdeklaration als Schablone:

```
template<typename T> Funktionsdeklaration;
```

Achtung:
Datentypen von Parametern und ggf. des Rückgabewertes mit `T` als Platzhalter!

Mehr als ein Typparameter möglich:

```
template<typename T, typename S> Funktionsdeklaration;
```

u.s.w.

Auch Konstanten als Parameter möglich:

```
template<typename T, int const i> Funktionsdeklaration;
```


Funktionsdefinition als Schablone:

```
template<typename T> Funktionsdeklaration {

  // Anweisungen und ggf. return
  // ggf. Verwendung von Typ T als Platzhalter

};
```

Achtung:

Nicht jeder Typ gleichen Namens wird durch Platzhalter T ersetzt!

Man muss darauf achten,
für welchen Bezeichner der Datentyp parametrisiert werden soll!

2. Aufgabe:

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

Entwurf mit Implementierung:

```
int imin(unsigned int const n, int 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;
}
```

Variation der 2. Aufgabe:

Finde Index des 1. Minimums in einem Array mit numerischen Typ.
Falls Array leer, gebe **-1** zurück.

Implementierung mit **Schablonen**:

```
template <typename T>
int imin(unsigned int const n, T 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;
}
```

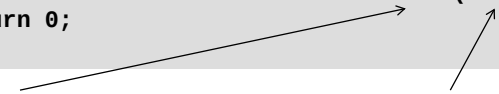

Aufruf einer Funktionsschablone: (hier mit Parameter und Rückgabewert)

```
template<typename T> T Funktionsbezeichner(T Bezeichner) {

  T result;
  // Anweisungen
  return result;

};
```

```
int main() {
  short s = Funktionsbezeichner<short>(1023);
  int   i = Funktionsbezeichner<int>(1023);
  float f = Funktionsbezeichner<float>(1023);
  return 0;
};
```



Typparameter kann entfallen, wenn Typ aus Parameter **eindeutig** erkennbar!

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`** oder **`int`** oder **...**
Verändere dabei die Werte im Array.

Mögliche Lösung mit Schablonen:

```
template <typename T>
void sortiere(unsigned int const n, T a[]) {
  int i, k;
  for (k = n – 1; k > 1; k--) {
    i = imin<T>(k - 1, a);
    if (a[i] < a[k]) swap<T>(a[i], a[k]);
  }
}
```

```
template <typename T>
void swap(T &a, T &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!
- Funktionsschablonen ermöglichen Parametrisierung des Datentyps.
  Die Funktionen werden bei Bedarf automatisch zur Übersetzungszeit erzeugt.


```
#include <cmath>
```

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 <cstdlib>
```

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 <cstdlib>

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 <cctype>
```

`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 <cctype>

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 <cctype>

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



```
#include <ctime>
```

`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 <ctime>

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

engl. FSM: finite state machine

Der DEA ist zentrales Modellierungswerkzeug in der Informatik.

Definition

Ein deterministischer endlicher Automat ist ein 5-Tupel $(S, \Sigma, \delta, F, s_0)$, wobei

- S eine endliche Menge von Zuständen,
- Σ das endliche Eingabealphabet,
- δ: S x $\Sigma \rightarrow$ S die Übergangsfunktion,
- F eine Menge von Finalzuständen mit $F \subseteq S$ und
- s_0 der Startzustand. ■

Er startet immer im Zustand s_0, verarbeitet Eingaben und wechselt dabei seinen Zustand. Er terminiert ordnungsgemäß, wenn Eingabe leer und ein Endzustand aus F erreicht.

⇒ Beschreibung eines Programms!

Grafische Darstellung

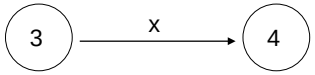
Zustände als Kreise

im Kreis der Bezeichner des Zustands (häufig durchnummeriert)

Übergänge von einem Zustand zum anderen ist abhängig von der Eingabe.

Mögliche Übergänge sind durch Pfeile zwischen den Zuständen dargestellt.

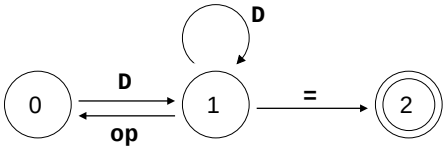
Über / unter dem Pfeil steht das Eingabesymbol, das den Übergang auslöst.



Endzustände werden durch „Doppelkreise“ dargestellt.

Beispiel:

Entwerfe DEA, der arithmetische Ausdrücke ohne Klammern für nichtnegative Ganzzahlen auf Korrektheit prüft.



Zustände S = { 0, 1, 2 }

Startzustand s_0 = 0

Endzustände F = { 2 }

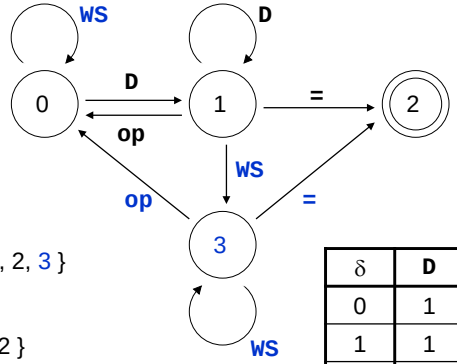
Eingabealphabet Σ = { D, op, = }

δ	D	op	=
0	1	-1	-1
1	1	0	2
2	-	-	-

-1: Fehlerzustand

Beispiel:

Erweiterung: Akzeptiere auch „white space“ zwischen Operanden und Operatoren



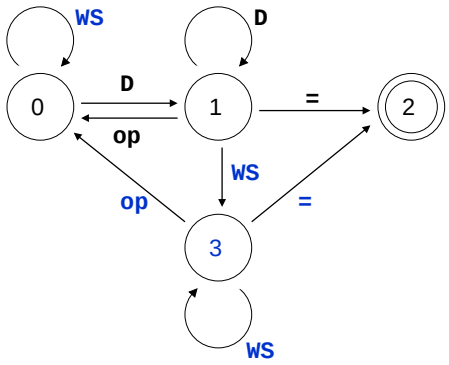
Zustände S = { 0, 1, 2, 3 }

Startzustand s_0 = 0

Endzustände F = { 2 }

Eingabealphabet Σ = { D, op, =, WS }

δ	D	op	=	WS
0	1	-1	-1	0
1	1	0	2	3
2	-	-	-	-
3	-1	0	2	3



Eingabe:

```
3+ 4 - 5=
```

Zustand 0, lese D →
Zustand 1, lese op →
Zustand 0, lese WS →
Zustand 0, lese D →
Zustand 1, lese WS →
Zustand 3, lese op →
Zustand 0, lese WS →
Zustand 0, lese D →
Zustand 1, lese = →
Zustand 2 (Endzustand)

Wenn **grafisches Modell** aufgestellt, dann Umsetzung in ein **Programm**:

- Zustände durchnummeriert: 0, 1, 2, 3
- Eingabesymbole: z.B. als `enum { D, OP, IS, WS }` (**IS** für =)
- Übergangsfunktion als Tabelle / Array:

```
int GetState[][4] = {
  {  1, -1, -1,  0 },
  {  1,  0,  2,  3 },
  {  2,  2,  2,  2 },
  { -1,  0,  2,  3 }
};
```

Array enthält die gesamte Steuerung des Automaten!

- Eingabesymbole erkennen u.a. mit: **`isdigit(), isspace()`**

```
bool isbinop(char c) {
  return c == '+' || c == '-' || c == '*' || c == '/';
}
```



```
enum TokenT { D, OP, IS, WS, ERR };
```

```
bool Akzeptor(char const* input) {
  int state = 0;
  while (*input != '\0' && state != -1) {
    char s = *input++;
    TokenT token = ERR;
    if (isdigit(s)) token = D;
    if (isbinop(s)) token = OP;
    if (s == '=')   token = IS;
    if (isspace(s)) token = WS;
    state = (token == ERR) ? -1 : GetState[state][token];
  }
  return (state == 2);
}
```


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;
}
```

Datei *Haupt.cpp*

Fehler! `funktion1` nicht sichtbar!

wenn entfernt, dann gelingt Compilierung:

g++ *.cpp –o test

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;
}
```

→ Prä-Compiler → Compiler → ...

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

Tipp: NICHT VERWENDEN!

stattdessen:

```
int const 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 funktionsartige Makros besser richtige inline-Funktionen verwenden!*