technische universität dortmund

Einführung in die Programmierung

Wintersemester 2018/19

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

Wir kennen bisher:

- Datentypen zur Modellierung von Daten (inkl. Zeiger)
- Kontrollstrukturen zur Gestaltung des internen Informationsflusses

⇒ Damit lassen sich – im Prinzip – alle Programmieraufgaben lösen!

Wenn man aber

mehrfach das gleiche nur mit verschiedenen Daten tun muss,

dann müsste man

den **gleichen Quellcode mehrfach** im Programm stehen haben!

⇒ unwirtschaftlich, schlecht wartbar und deshalb fehleranfällig!

Funktion in der Mathematik:

f: $\mathbb{R} \rightarrow \mathbb{R}$
f(x) = sin(x)

y = f(0.5) führt zur

- Berechnung von sin(0.5),
- Rückgabe des Ergebnisses,
- Zuweisung des Ergebnisses an Variable y.

z = f(0.2) an anderer Stelle führt zur

- Berechnung von sin(0.2),
- Rückgabe des Ergebnisses,
- Zuweisung des Ergebnisses an Variable z.

Funktionen in C++

```
int main() {
  double x = 0.5, y, z;
  y = sin(x);
  z = sin(0.2);
  std::cout << y << " " << z << std::endl;
  return 0;
}
```

Achtung!
main() ist Funktion!
Nur 1x verwendbar!

Die Funktion **sin(·)** ist eine Standardfunktion.

Standardfunktionen werden vom Hersteller bereitgestellt und sind in Bibliotheken abgelegt. Bereitstellung durch **#include** – Anweisung: **#include <cmath>**

Programmierer kann eigene, benutzerdefinierte Funktionen schreiben.

Welche Arten von Funktionen gibt es?

a) Funktionen ohne Parameter und ohne Rückgabewert: **clearscreen();**
b) Funktionen mit Parameter aber ohne Rückgabewert: **background(blue);**
c) Funktionen ohne Parameter aber mit Rückgabewert: **uhrzeit = time();**
d) Funktionen mit Parameter und mit Rückgabewert: **y = sin(x);**

Konstruktionsregeln für

- Standardfunktionen und
- benutzerdefinierte Funktionen

sind gleich

(a) Funktionen ohne Parameter und ohne Rückgabewert

- Funktionsdeklaration:

void Bezeichner **();**

Prototyp der Funktion

Nichts zwischen Klammern ⇒ keine Parameter

Name der Funktion

void (= leer) zeigt an, dass kein Wert zurückgegeben wird

(a) Funktionen ohne Parameter und ohne Rückgabewert

- Funktionsdefinition:

```
void Bezeichner () {
  // Anweisungen
}
```

```
// Beispiel:
void zeichne_sterne() {
  int k = 10;
  while (k--) std::cout << '*';
  std::cout << std::endl;
}
```

Achtung:
Variable, die in einer Funktion definiert werden, sind nur innerhalb der Funktion gültig.

Nach Verlassen der Funktion sind diese Variablen ungültig!

(a) Funktionen ohne Parameter und ohne Rückgabewert

- Funktionsaufruf:

  Bezeichner `();`

```
// Beispiel:
#include <iostream>
int main() {
  zeichne_sterne();
  zeichne_sterne();
  zeichne_sterne();
  return 0;
}
```

Achtung:
Die Funktionsdefinition muss vor dem 1. Funktionsaufruf stehen!

Alternativ:
Die Funktionsdeklaration muss vor dem 1. Funktionsaufruf stehen. Dann kann die Funktionsdefinition später, also auch nach dem ersten Funktionsaufruf, erfolgen.

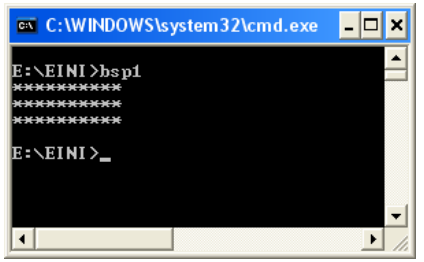
(a) Funktionen ohne Parameter und ohne Rückgabewert

```
// Komplettes Beispiel: bsp1.exe
#include <iostream>

void zeichne_sterne() {
  int k = 10;
  while (k--) std::cout << '*';
  std::cout << std::endl;
}

int main() {
  zeichne_sterne();
  zeichne_sterne();
  zeichne_sterne();
  return 0;
}
```

Zuerst Funktionsdefinition.

Dann Funktionsaufrufe.

Ausgabe:



(a) Funktionen ohne Parameter und ohne Rückgabewert

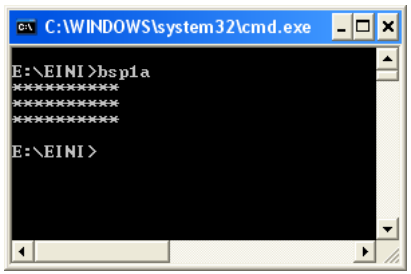
```
// Komplettes Beispiel: bsp1a.exe
#include <iostream>
void zeichne_sterne();
int main() {
  zeichne_sterne();
  zeichne_sterne();
  zeichne_sterne();
  return 0;
}
void zeichne_sterne() {
  int k = 10;
  while (k--) std::cout << '*';
  std::cout << std::endl;
}
```

Zuerst Funktionsdeklaration.

Dann Funktionsaufrufe.

Später Funktionsdefinition.

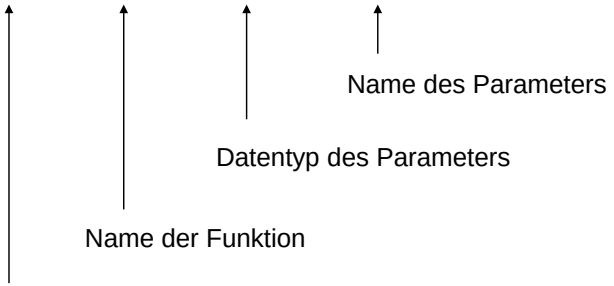
Ausgabe:



(b) Funktionen mit Parameter aber ohne Rückgabewert

- Funktionsdeklaration:

  `void` Bezeichner `(`Datentyp Bezeichner`);`



`void` (= leer) zeigt an, dass kein Wert zurückgegeben wird

(b) Funktionen mit Parameter aber ohne Rückgabewert

- Funktionsdefinition:

```
void Bezeichner (Datentyp Bezeichner) {

  // Anweisungen

}
```

```
// Beispiel:
void zeichne_sterne(int k) {
  while (k--) std::cout << '*';
  std::cout << std::endl;
}
```


(b) Funktionen mit Parameter aber ohne Rückgabewert

- Funktionsaufruf:

```
Bezeichner (Parameter);
```

```
// Beispiel:
#include <iostream>
int main() {
  zeichne_sterne(10);
  zeichne_sterne( 2);
  zeichne_sterne( 5);
  return 0;
}
```

Achtung:
Parameter muss dem Datentyp entsprechen, der in Funktions-deklaration bzw. Funktions-definition angegeben ist.

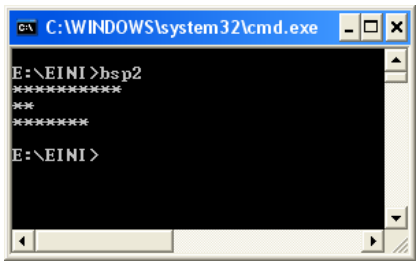
Hier: `int`

Kann Konstante oder Variable sein.

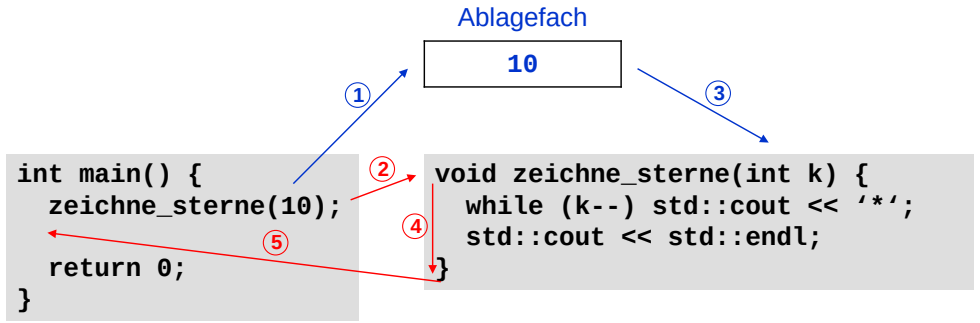
(b) Funktionen mit Parameter aber ohne Rückgabewert

```
// Komplettes Beispiel: bsp2.exe
#include <iostream>

void zeichne_sterne(int k) {

  while (k--) std::cout << '*';
  std::cout << std::endl;
}
int main() {
  zeichne_sterne(10);
  zeichne_sterne(2);
  zeichne_sterne(7);
  return 0;
}
```

Ausgabe:



Wie wird die Parameterübergabe technisch realisiert?

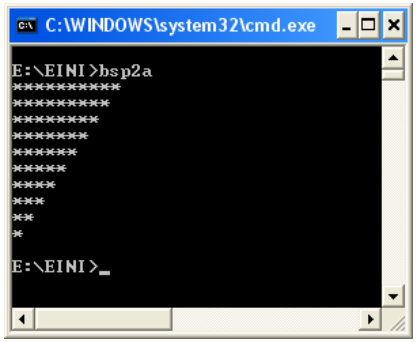


1. bei Aufruf `zeichne_sterne(10)` wird Parameter `10` ins Ablagefach gelegt
2. der Rechner springt an die Stelle, wo Funktionsanweisungen anfangen
3. der Wert `10` wird aus dem Ablagefach geholt und `k` zugewiesen
4. die Funktionsanweisungen werden ausgeführt
5. nach Beendigung der Funktionsanweisungen Rücksprung hinter Aufruf

(b) Funktionen mit Parameter aber ohne Rückgabewert

```
// Komplettes Beispiel: bsp2a.exe
#include <iostream>

void zeichne_sterne(int k) {
  while (k--) std::cout << '*';
  std::cout << std::endl;
}
int main() {
  int i;
  for (i = 10; i > 0; i--)
    zeichne_sterne(i);
  return 0;
}
```

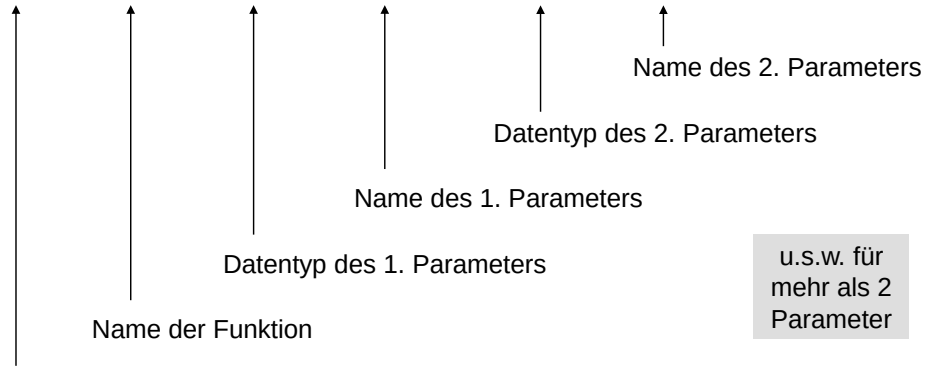
Ausgabe:



(b) Funktionen mit Parametern aber ohne Rückgabewert

- Funktionsdeklaration:

void Bezeichner **(**Datentyp1 Bezeichner1, Datentyp2 Bezeichner2**);**



void (= leer) zeigt an, dass kein Wert zurückgegeben wird

(b) Funktionen mit Parametern aber ohne Rückgabewert

- Funktionsdefinition:

void Bezeichner **(**Datentyp1 Bezeichner1, Datentyp2 Bezeichner2**) {**

```
  // Anweisungen
}
```

```
// Beispiel:
void zeichne_zeichen(int k, char c) {
  // zeichne k Zeichen der Sorte c
  while (k--) std::cout << c;
  std::cout << std::endl;
}
```


(b) Funktionen mit Parametern aber ohne Rückgabewert

- Funktionsaufruf:

Bezeichner **(**Parameter1, Parameter2**);**

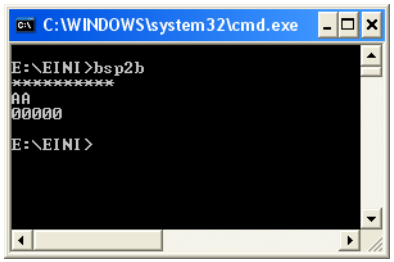
```
// Beispiel:
#include <iostream>
int main() {
  zeichne_zeichen(10,'*');
  zeichne_zeichen( 2,'A');
  zeichne_zeichen( 5,'0');
  return 0;
}
```

Natürlich:
Bei mehr als 2 Parametern wird die Parameterliste länger!

(b) Funktionen mit Parametern aber ohne Rückgabewert

```
// Komplettes Beispiel: Bsp2b.exe

#include <iostream>
void zeichne_zeichen(int k, char c)
{
  // zeichne k Zeichen der Sorte c
  while (k--) std::cout << c;
  std::cout << std::endl;
}

int main() {

  zeichne_zeichen(10,'*');
  zeichne_zeichen( 2,'A');
  zeichne_zeichen( 5,'0');

  return 0;

}
```

Ausgabe:



(b) Funktionen mit Parametern aber ohne Rückgabewert

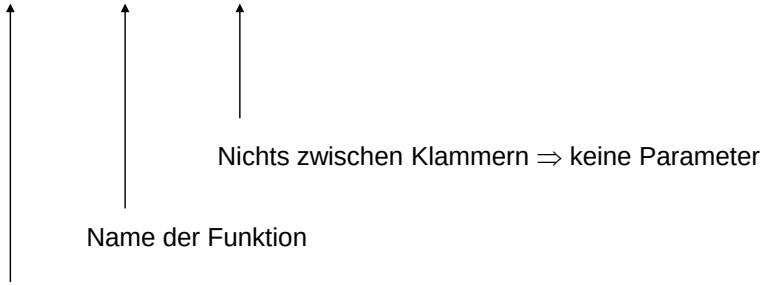
```
// Komplettes Beispiel: Bsp2c.exe

#include <iostream>
void zeichne_zeichen(int k, char c)
{
  // zeichne k Zeichen der Sorte c
  while (k--) std::cout << c;
  std::cout << std::endl;
}

int main() {

  int i;
  for (i = 0; i < 26; i++)
    zeichne_zeichen(i + 1,'A' + i);

  return 0;

}
```

Ausgabe:

(c) Funktionen ohne Parameter aber mit Rückgabewert

- Funktionsdeklaration:

  Datentyp Bezeichner **();**



(c) Funktionen ohne Parameter aber mit Rückgabewert

- Funktionsdefinition:

  Datentyp Bezeichner **() {**

  // Anweisungen

  return Rückgabewert;

  }

Achtung!
Datentyp des Rückgabewertes muss mit dem in der Funktionsdefinition angegebenen Datentyp übereinstimmen.

```
// Beispiel:
bool Fortsetzen() {
  char c;
  do {
    cout << "Fortsetzen (j/n)? ";
    cin >> c;
  } while (c != 'j' && c != 'n');

  return (c == 'j');
}
```

(c) Funktionen ohne Parameter aber mit Rückgabewert

- Funktionsaufruf:

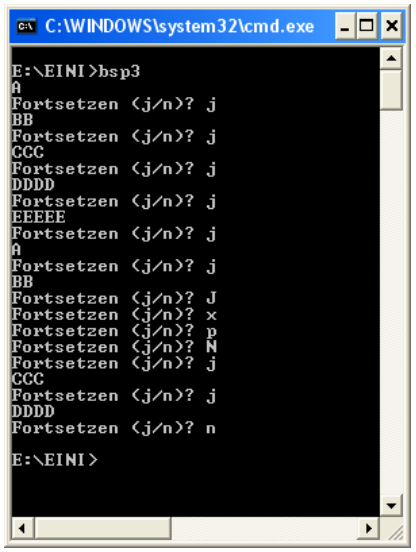
  Variable = Bezeichner **();**

oder: Rückgabewert ohne Speicherung verwerten

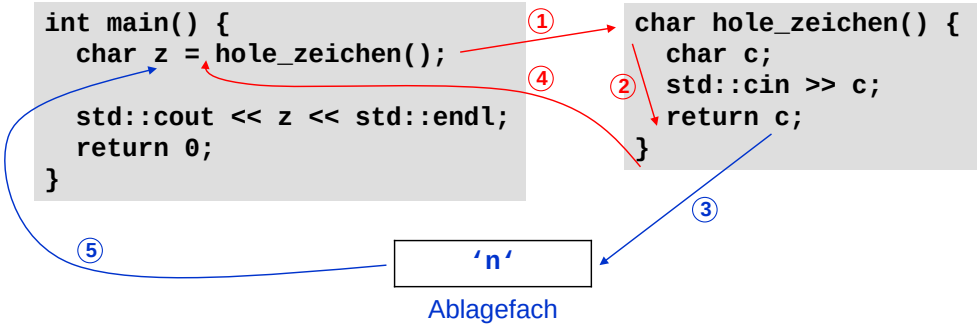
```
// Beispiel:
#include <iostream>
int main() {
  int i = 0;
  do {
    zeichne_zeichen(i + 1, 'A' + i);
    i = (i + 1) % 5;
  } while (fortsetzen());
  return 0;
}
```


(c) Funktionen ohne Parameter aber mit Rückgabewert

```
// Komplettes Beispiel: bsp3.exe
#include <iostream>

void zeichne_zeichen(int k, char c) {
  while (k--) std::cout << c;
  std::cout << std::endl;
}

bool fortsetzen() {
  char c;
  do {
    std::cout << "Fortsetzen (j/n)? ";
    std::cin >> c;
  } while (c != 'j' && c != 'n');
  return (c == 'j');
}

int main() {
  int i = 0;
  do {
    zeichne_zeichen(i + 1, 'A' + i);
    i = (i + 1) % 5;
  } while (fortsetzen());
  return 0;
}
```



Wie wird die Funktionswertrückgabe realisiert?

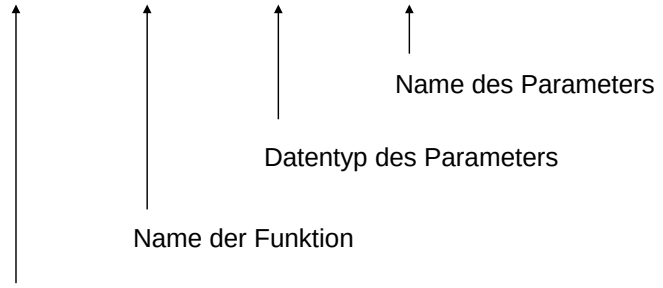


1. Rechner springt bei Aufruf **hole_zeichen()** zu den Funktionsanweisungen
2. Die Funktionsanweisungen werden ausgeführt
3. Bei return c wird der aktuelle Wert von c ins Ablagefach gelegt
4. Rücksprung zur aufrufenden Stelle
5. Der zuzuweisende Wert wird aus dem Ablagefach geholt und zugewiesen

(d) Funktionen mit Parameter und mit Rückgabewert

- Funktionsdeklaration:

  Datentyp Bezeichner **(**Datentyp Bezeichner**);**



(d) Funktionen mit Parameter und mit Rückgabewert

- Funktionsdefinition:

Datentyp Bezeichner **(**Datentyp Bezeichner**){**

```
  // Anweisungen
  return Rückgabewert;
}
```

```
// Beispiel:
double polynom(double x) {
  return 3 * x * x * x - 2 * x * x + x - 1;
}
```

Offensichtlich wird hier für einen Eingabewert x das Polynom

$$p(x) = 3\,x^3 - 2\,x^2 + x - 1$$

berechnet und dessen Wert per **`return`** zurückgeliefert.

(d) Funktionen mit Parameter und mit Rückgabewert

- Funktionsaufruf:

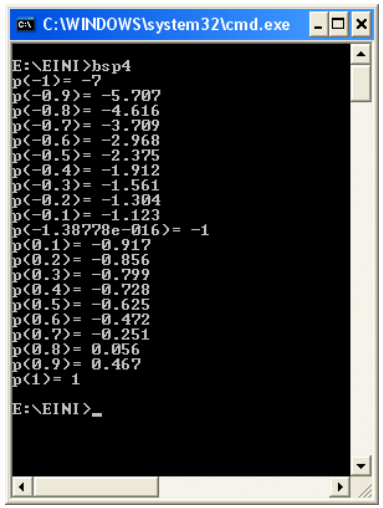
Variable = Bezeichner **(**Parameter**);**

oder: Rückgabewert ohne Speicherung verwerten

```
// Beispiel:
#include <iostream>
#using namespace std;

int main() {
  double x;
  for (x = -1.0; x <= 1.0; x += 0.1)
    cout << "p(" << x << ")= "
         << polynom(x) << endl;
  return 0;
}
```

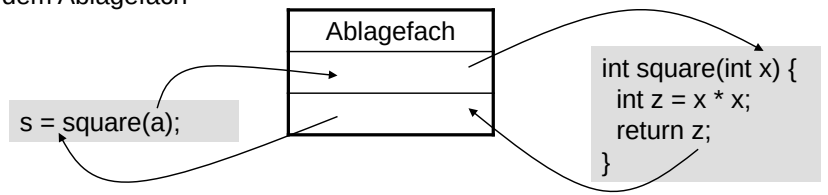

(d) Funktionen mit Parameter und mit Rückgabewert

```
// Komplettes Beispiel: Bsp4.exe
#include <iostream>
using namespace std;

double polynom(double x) {
  return 3 * x * x * x -
         2 * x * x + x - 1;
}

int main() {
  double x;
  for (x = -1.0; x <= 1.0; x += 0.1)
    cout << "p(" << x << ")= "
         << polynom(x) << endl;
  return 0;
}
```



Wir kennen bisher:

- Funktionen mit/ohne Parameter sowie mit/ohne Rückgabewert:
- Parameter und Rückgabewerte kamen als Kopie ins Ablagefach (Stack)
- Funktion holt Kopie des Parameters aus dem Ablagefach
- Wertzuweisung an neue, nur lokale gültige Variable
- Rückgabewert der Funktion kommt als Kopie ins Ablagefach
- Beim Verlassen der Funktion werden lokal gültige Variable ungültig
- Rücksprung zum Funktionsaufruf und Abholen des Rückgabewertes aus dem Ablagefach



Übergabe eines Wertes:

```
double x = 0.123, a = 2.71, b = .35, z;
z = sin(0.717);            // Konstante
z = cos(x);                // Variable
z = sqrt(3 * a + 4 * b);   // Ausdruck, der Wert ergibt
z = cos( sqrt( x ) );      // Argument ist Fkt.,
                           // die Wert ergibt
z = exp(b * log( a ) );    // Argument ist Ausdruck aus Fkt.
                           // und Variable, der Wert ergibt
```

Wert kann Konstante, Variable und wertrückgebende Funktion sowie eine Kombination daraus in einem Ausdruck sein!

Bevor Kopie des Wertes ins Ablagefach kommt, wird Argument ausgewertet!

Übergabe eines Wertes:

```
struct KundeT {
  char   name[20];
  int    knr;
  double umsatz;
};
enum StatusT { gut, mittel, schlecht };
StatusT KundenStatus(KundeT kunde) {
  if (kunde.umsatz > 100000.0) return gut;
  if (kunde.umsatz <  20000.0) return schlecht;
  return mittel;
}
```

Übergabe und Rückgabe als Wert funktioniert mit allen Datentypen …

Ausnahme: Array! ⟶ später!

Übergabe eines Wertes:

```
void tausche_w(int a, int b) {
  int h = a;
  a = b;
  b = h;
  cout << "Fkt.: " << a << " " << b << endl;
}
int main() {
  int a = 3, b = 11;
  cout << "main: " << a << " " << b << endl;
  tausche_w(a, b);
  cout << "main: " << a << " " << b << endl;
}
```

Ausgabe:
```
main: 3 11
Fkt.: 11 3
main: 3 11
```
⇒ funktioniert so nicht, da Übergabe von Kopien!

Übergabe eines Zeigers: (als Wert)

```
void tausche_p(int* pu, int* pv) {
  int h = *pu;
  *pu = *pv;
  *pv = h;
  std::cout << "Fkt.: " << *pu << " " << *pv << std::endl;
}
int main() {
  int a = 3, b = 11;
  std::cout << "main: " << a << " " << b << std::endl;
  tausche_p(&a, &b);
  std::cout << "main: " << a << " " << b << std::endl;
}
```

Ausgabe:
```
main: 3 11
Fkt.: 11 3
main: 11 3
```
⇒ funktioniert, da Übergabe von Zeigern!

Übergabe eines Zeigers:

Man übergibt einen Zeiger auf ein Objekt (als Wert).

```
// Beispiel:
void square(int* px) {
  int y = *px * *px;
  *px = y;
}
```

```
int main() {
  int a = 5;
  square(&a);
  cout << a << '\n';
  return 0;
}
```

```
int main() {
  int a = 5, *pa;
  pa = &a;
  square(pa);
  cout << a << '\n';
  return 0;
}
```


Übergabe eines Zeigers

Funktionsaufruf:

Funktionsname(&Variablenname) ;

Variable = Funktionsname(&Variablenname) ;

```
int x = 5;
square(&x);
```

oder:

Funktionsname(Zeiger-auf-Variable) ;

Variable = Funktionsname(Zeiger-auf-Variable) ;

```
int x = 5, *px;
px = &x;
square(px);
```

Achtung!

Im Argument dürfen nur solche zusammengesetzten Ausdrücke stehen, die legale Zeigerarithmetik darstellen: z.B. **`(px + 4)`**

Zeigerparameter

```
void reset(int *ip) {
  *ip = 0; // ändert Wert des Objektes, auf den ip zeigt
  ip = 0;  // ändert lokalen Wert von ip, Argument unverändert
}
```

```
int main() {
  int i = 10;
  int *p = &i;
  cout << &i << ": " << *p << endl;
  reset(p);
  cout << &i << ": " << *p << endl;
  return 0;
}
```

Ausgabe:

```
0012FEDC: 10
0012FEDC: 0
```

Also:

Zeiger werden als Kopie übergeben (als Wert)

Rückgabe eines Zeigers

```
struct KontoT {
  char Name[20];
  float Saldo;
};
```

```
KontoT const* reicher(KontoT const* k1, KontoT const* k2) {
  if (k1->Saldo > k2->Saldo) return k1;
  return k2;
}
```

```
// ...
KontoT anton = {"Anton", 64.0 }, berta = {"Berta", 100.0};
cout << reicher(&anton, &berta)->Name << " hat mehr Geld.\n";
// ...
```

Ausgabe:

```
Berta hat mehr Geld.
```

Rückgabe eines Zeigers

ACHTUNG:

Niemals Zeiger auf lokales Objekt zurückgeben!

```
KontoT const* verdoppeln(KontoT const* konto) {
  KontoT lokalesKonto = *konto;
  lokalesKonto.Saldo += konto->Saldo;
  return &lokalesKonto;
}
```

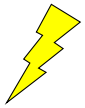
Gute Compiler sollten warnen!

⇒ nach Verlassen der Funktion wird der Speicher von **lokalesKonto** freigegeben

⇒ Adresse von **lokalesKonto** ungültig

⇒ zurückgegebener Zeiger zeigt auf ungültiges Objekt

⇒ kann funktionieren, muss aber nicht ⇒ **undefiniertes Verhalten!**

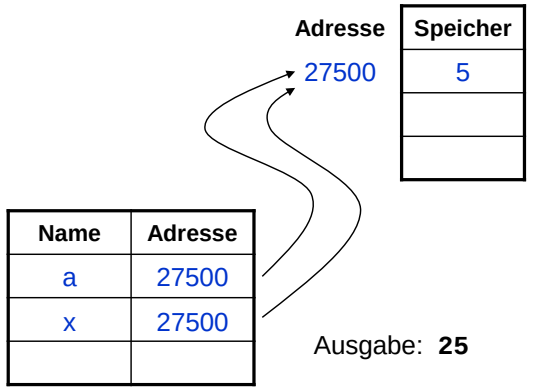


Übergabe einer Referenz: (nur in C++, nicht in C)

Referenz einer Variablen = Kopie der Adresse einer Variablen

= 2. Name der Variable

```
void square(int& x) {
  int y = x * x;
  x = y;
}
int main() {
  int a = 5;
  square(a);
  cout << a << "\n";
  return 0;
}
```



Name	Adresse
a	27500
x	27500

Ausgabe: **25**

Übergabe einer Referenz: (nur in C++, nicht in C)

Bauplan der Funktionsdeklaration:

void Funktionsname(Datentyp& Variablenname);

Datentyp Funktionsname(Datentyp& Variablenname);

zeigt Übergabe per Referenz an;
erscheint **nur im Prototypen**!

```
// Beispiele:
void square(int& x);

bool wurzel(double& radikant);
```

Durch Übergabe einer Referenz kann man den Wert der referenzierten Variable dauerhaft verändern!

Übergabe einer Referenz: (nur in C++, nicht in C)

Bauplan der Funktionsdefinition:

```
void Funktionsname(Datentyp& Variablenname) {
  // Anweisungen
}

Datentyp Funktionsname(Datentyp& Variablenname) {
  // Anweisungen
  return Rückgabewert;
}
```

```
// Beispiel:
void square(int& x) {
  int y = x * x;
  x = y;
}
```

Übergabe einer Referenz: (nur in C++, nicht in C)

Funktionsaufruf:

Funktionsname(Variablenname) ;

Variable = Funktionsname(Variablenname) ;

```
// Beispiel:
int x = 5;
square(x);
```

Achtung:

Beim Funktionsaufruf kein &-Operator!

Da Adresse geholt wird, muss Argument eine Variable sein!

→ Im obigen Beispiel würde **`square(5);`** zu einem Compilerfehler führen!

Übergabe einer Referenz: (nur in C++, nicht in C)

```
void tausche_r(int& u, int& v) {
  int h = u;
  u = v;
  v = h;
  std::cout << "Fkt.: " << u << " " << v << std::endl;
}
int main() {
  int a = 3, b = 11;
  std::cout << "main: " << a << " " << b << std::endl;
  tausche_r(a, b);
  std::cout << "main: " << a << " " << b << std::endl;
}
```

Ausgabe:
```
main: 3 11
Fkt.: 11 3
main: 11 3
```
⇒ funktioniert, da Übergabe von Referenzen!

Übergabe einer Referenz: (nur in C++, nicht in C)

Möglicher Verwendungszweck: mehr als nur einen Rückgabewert!

Bsp: Bestimmung reeller Lösungen der Gleichung $x^2 + px + q = 0$.

- Anzahl der Lösungen abhängig vom Radikand $r = (p/2)^2 - q$
- Falls $r > 0$, dann 2 Lösungen
- Falls $r = 0$, dann 1 Lösung
- Falls $r < 0$, dann keine Lösung

⇒ Wir müssen also zwischen 0 und 2 Werte zurückliefern und die Anzahl der gültigen zurückgegebenen Werte angeben können

Übergabe einer Referenz: (nur in C++, nicht in C)

Eine mögliche Lösung mit Referenzen:

```
int Nullstellen(double p, double q, double& x1, double& x2) {
  double r = p * p / 4 - q;
  if (r < 0) return 0;     // keine Lösung
  if (r == 0) {
    x1 = -p / 2;
    return 1;              // 1 Lösung
  }
  x1 = -p / 2 - sqrt(r);
  x2 = -p / 2 + sqrt(r);
  return 2;                // 2 Lösungen
}
```

Rückgabe einer Referenz

```
struct KontoT {
  char Name[20];
  float Saldo;
};
```

```
KontoT const& reicher(KontoT const& k1, KontoT const& k2) {
  if (k1.Saldo > k2.Saldo) return k1;
  return k2;
}
```

```
// ...
KontoT anton = {“Anton“, 64.0 }, berta = {“Berta“, 100.0};
cout << reicher(anton, berta).Name << “ hat mehr Geld.\n“;
// ...
```

Ausgabe:

```
Berta hat mehr Geld.
```


Rückgabe einer Referenz

ACHTUNG:

Niemals Referenz auf lokales Objekt zurückgeben!

```
KontoT const &verdoppeln(KontoT const &konto) {
  KontoT lokalesKonto = konto;
  lokalesKonto.Saldo += konto.Saldo;
  return lokalesKonto;
}
```

Gute Compiler sollten warnen!

⇒ nach Verlassen der Funktion wird der Speicher von **`lokalesKonto`** freigegeben

⇒ Adresse von **`lokalesKonto`** ungültig

⇒ zurückgegebene Referenz auf Objekt ungültig

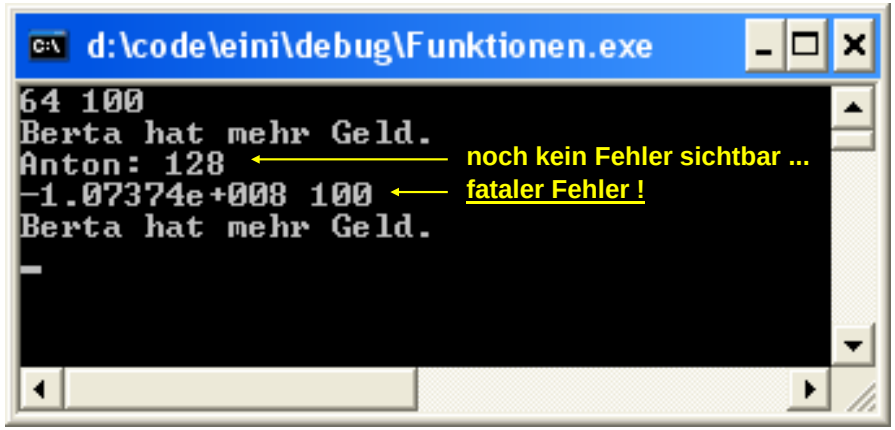
⇒ kann funktionieren, muss aber nicht ⇒ **undefiniertes Verhalten!**

Beispiel:

```
KontoT const& reicher(KontoT const& k1, KontoT const& k2) {
  cout << k1.Saldo << " " << k2.Saldo << endl;
  if (k1.Saldo > k2.Saldo) return k1;
  return k2;
}
KontoT const& verdoppeln(KontoT const& konto) {
  KontoT lokalesKonto = konto;
  lokalesKonto.Saldo += konto.Saldo;
  return lokalesKonto;
}
int main() {
  KontoT anton = {"Anton", 64.0 }, berta = {"Berta", 100.0};
  cout << reicher(anton, berta).Name << " hat mehr Geld.\n";
  cout << "Anton: " << verdoppeln(anton).Saldo << endl;
  cout << reicher(verdoppeln(anton), berta).Name
       << " hat mehr Geld.\n";
  return 0;
}
```


Rückgabe einer Referenz

Resultat:



Übergabe von Arrays:

Zur Erinnerung:

Name eines Arrays wird **wie** Zeiger auf einen festen Speicherplatz behandelt!

Schon gesehen: mit Zeigern kann man Originalwerte verändern.

Also werden **Arrays nicht als Kopien** übergeben.

```
void inkrement(int b[]) {
  int k;
  for (k = 0; k < 5; k++) b[k]++;
}
```

Vorsicht! Gefährliche Implementierung!

```
int main() {
  int i, a[] = { 2, 4, 6, 8, 10 };
  inkrement(a);
  for (i = 0; i < 5; i++) std::cout << a[i] << "\n";
}
```


Übergabe von Arrays:

Merke:

Ein Array sollte immer mit Bereichsgrenzen übergeben werden!

Sonst Gefahr der Bereichsüberschreitung

⇒ Inkonsistente Daten oder Speicherverletzung mit Absturz!

```
void inkrement(unsigned int const n, int b[]) {
  int k;
  for (k = 0; k < n; k++) b[k]++;
}
```

```
int main() {
  int i, a[] = { 2, 4, 6, 8, 10 };
  inkrement(5, a);
  for (i = 0; i < 5; i++) cout << a[i] << endl;
}
```


Programmiertes Unheil: Bereichsüberschreitung beim Array (Beispiel)

```
int main() {
  int i, b[5] = { 0 }, a[] = { 2, 4, 6, 8, 10 };
  inkrement(5, a);
  for (i = 0; i < 5; i++) cout << a[i] << " ";
  cout << endl;
  for (i = 0; i < 5; i++) cout << b[i] << " ";
  cout << endl;
  inkrement(80, a);
  for (i = 0; i < 5; i++) cout << a[i] << " ";
  cout << endl;
  for (i = 0; i < 5; i++) cout << b[i] << " ";
  cout << endl;
  return 0;
}
```

Bereichs-fehler

Ausgabe:

```
3 5 7 9 11
0 0 0 0 0
4 6 8 10 12
1 1 1 1 1
```

… auch Laufzeitfehler möglich!

Übergabe eines Arrays:

Bauplan der Funktionsdefinition:

```
void Funktionsname(Datentyp Arrayname[]) {
  // Anweisungen
}

Datentyp Funktionsname(Datentyp Arrayname[]) {
  // Anweisungen
  return Rückgabewert;
}
```

Achtung!

Angabe der eckigen Klammern `[]` ist zwingend erforderlich!

Übergabe eines Arrays

Funktionsaufruf:

Funktionsname(Arrayname) ;

Variable = Funktionsname(Arrayname) ;

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

oder:

Funktionsname(&Arrayname[0]) ;

Variable = Funktionsname(&Arrayname[0]) ;

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

Tatsächlich: Übergabe des Arrays mit Zeigern!

Übergabe eines Arrays als Zeiger:

```
void Fkt (Datentyp *Arrayname) {
  // …
}
```

Achtung! Legale Syntax, aber irreführend:

```
void druckeWerte(int const ia[10]) {
  int i;
  for (i=0; i < 10; i++)
    cout << ia[i] << endl;
}
```

Programmierer ging davon aus, dass Array **ia** 10 Elemente hat!

Aber: fataler Irrtum!

Compiler ignoriert die Größenangabe!

Übergabe von zweidimensionalen Arrays:

Im Prototypen muss **die Spaltenkonstante** abgegeben werden!

Warum?

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

Mindestanforderung!

oder:

```
void inkrement(unsigned int const 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(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:

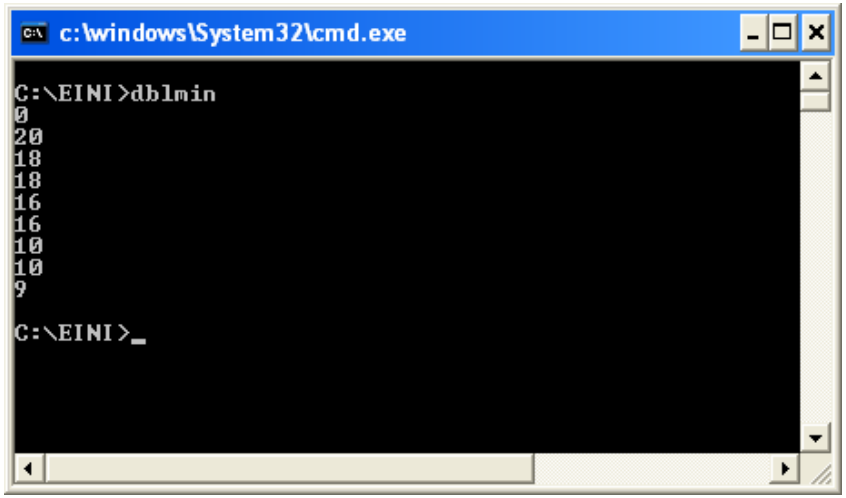
```
double dblmin(unsigned int const n, double a[]) {
  // leeres Array?
  if (n == 0) return 0.0;
  // Array hat also mindestens 1 Element!
  double min = a[0];
  int i;
  for(i = 1; i < n; i++)       // Warum i = 1 ?
    if (a[i] < min) min = a[i];
  return min;
}
```


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) {
  T 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,
- $\delta: S \times \Sigma \rightarrow S$ 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

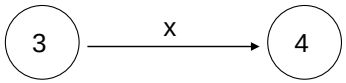
Zustände als Kreise

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.

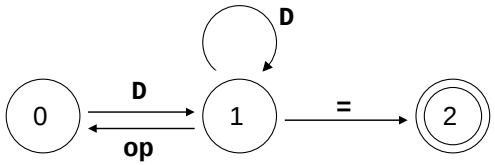
Über / unter dem Pfeil steht das Eingabesymbol, das den Übergang auslöst.



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 }

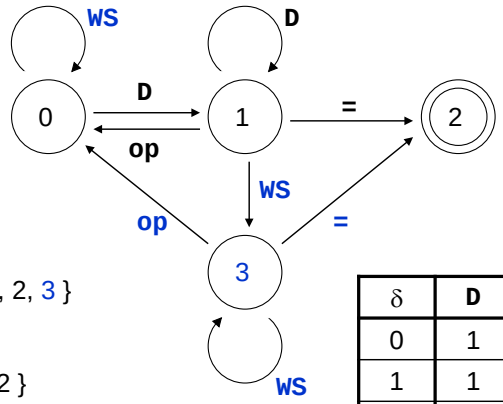
Eingabealphabet Σ = { D, op, = }

δ	D	op	=
0	1	-1	-1
1	1	0	2
2	-	-	-

-1: Fehlerzustand

Beispiel:

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



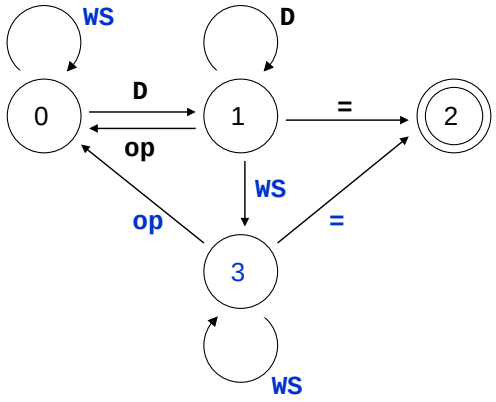
Zustände S = { 0, 1, 2, 3 }

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:

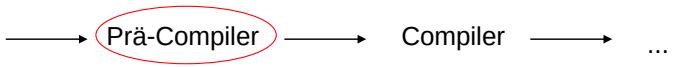
Das übersetzte Programm wird größer (benötigt mehr Hauptspeicher)

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;
}
```



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
```

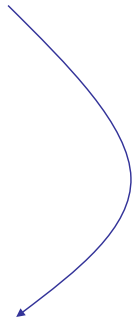
Makronamen im Programmtext werden vom Prä-Compiler durch ihre Ersetzung ersetzt


```
#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 funktionsartigen Makros besser richtige inline-Funktionen verwenden!*