technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 5: Funktionen

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

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`**-Direktive: **`#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

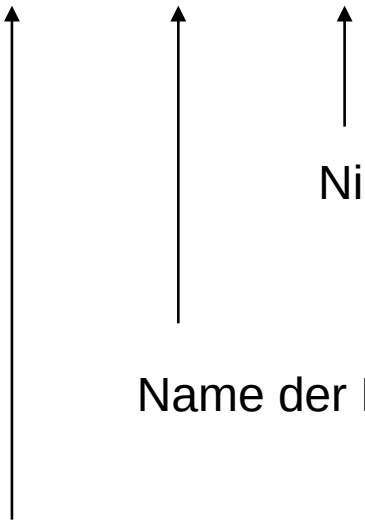
sind gleich.

(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 ersten Funktionsaufruf stehen.

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

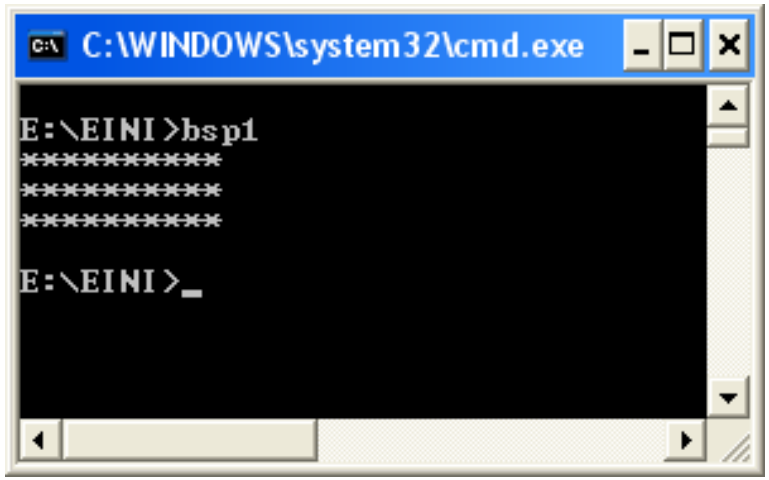
(a) Funktionen ohne Parameter und ohne Rückgabewert

```
// Komplettes Beispiel (v1)
#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

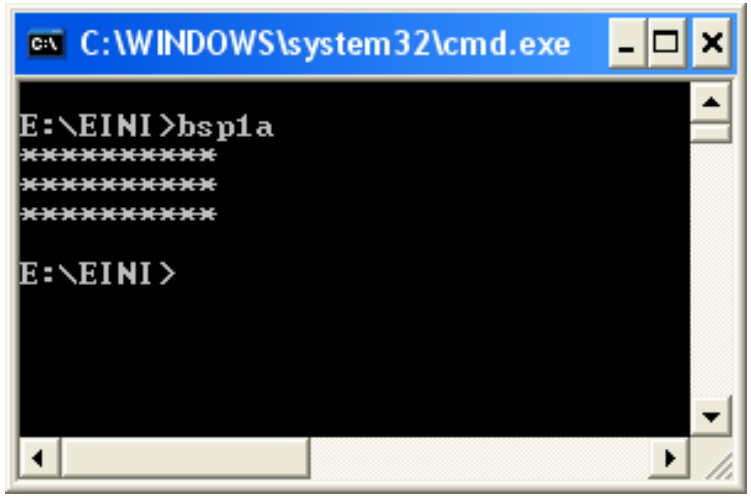
```
// Komplettes Beispiel (v2)
#include <iostream>

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 Funktions**deklaration**.

Dann Funktions**aufrufe**.

Später Funktions**definition**.

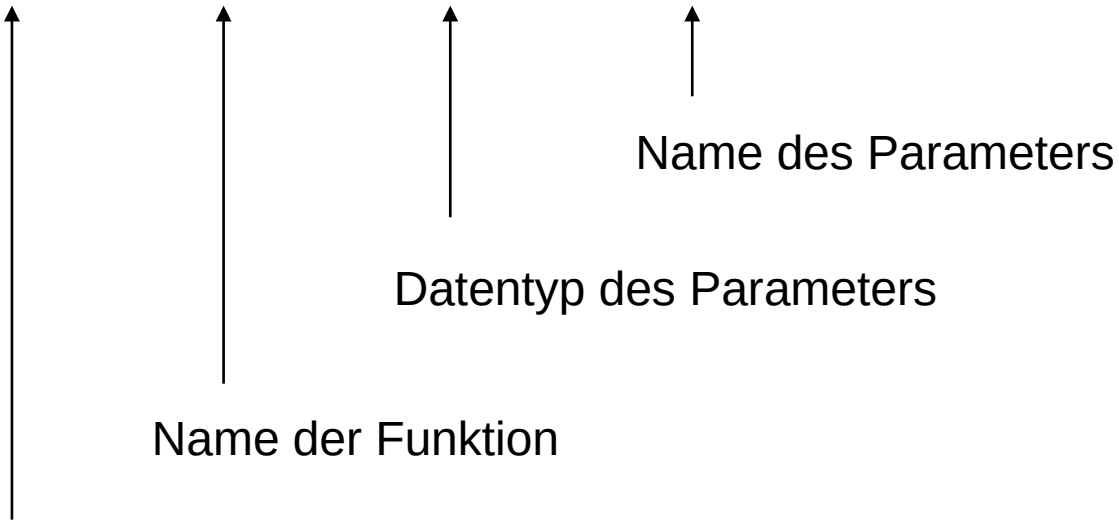
Ausgabe:



(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.

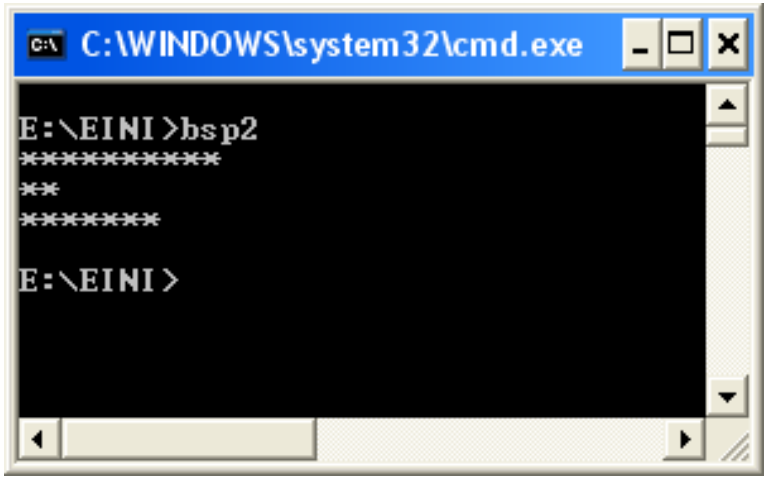
Hier: **`int`**

Kann Konstante oder Variable sein.

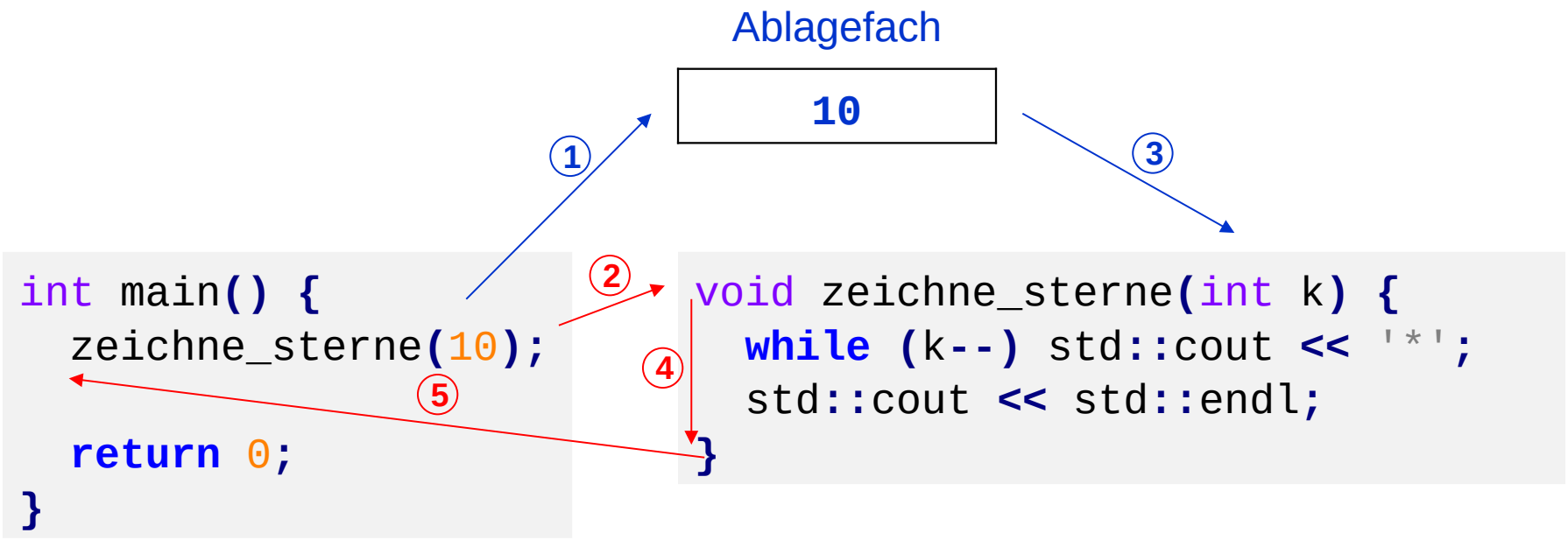
(b) Funktionen mit Parameter aber ohne Rückgabewert

```
// Komplettes Beispiel
#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?

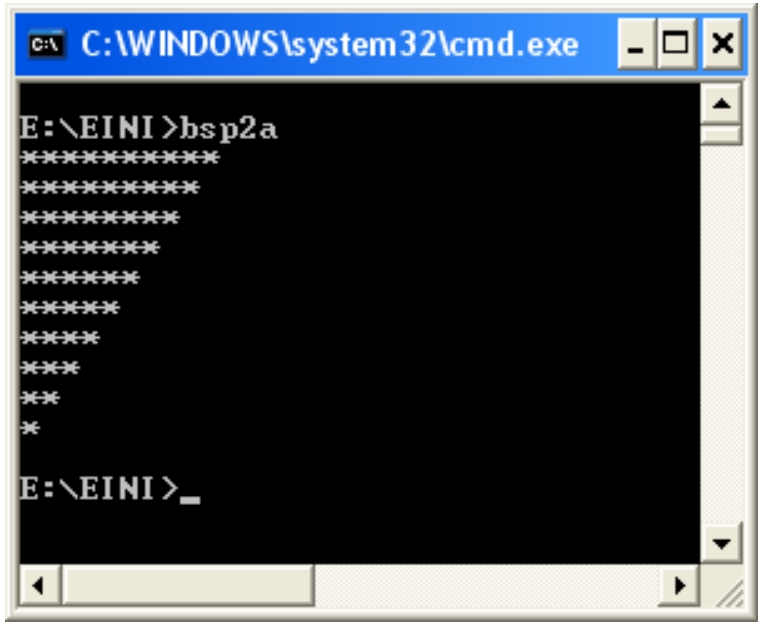


1. bei Aufruf **`zeichne_sterne(10)`** wird Parameter **`10`** ins Ablagefach gelegt
2. der Rechner springt an die Stelle, wo Funktionsanweisungen anfangen
3. der Wert **`10`** wird aus dem Ablagefach geholt und **`k`** zugewiesen
4. die Funktionsanweisungen werden ausgeführt
5. nach Beendigung der Funktionsanweisungen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

(b) Funktionen mit Parameter aber ohne Rückgabewert

```
// Komplettes Beispiel
#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;
}
```

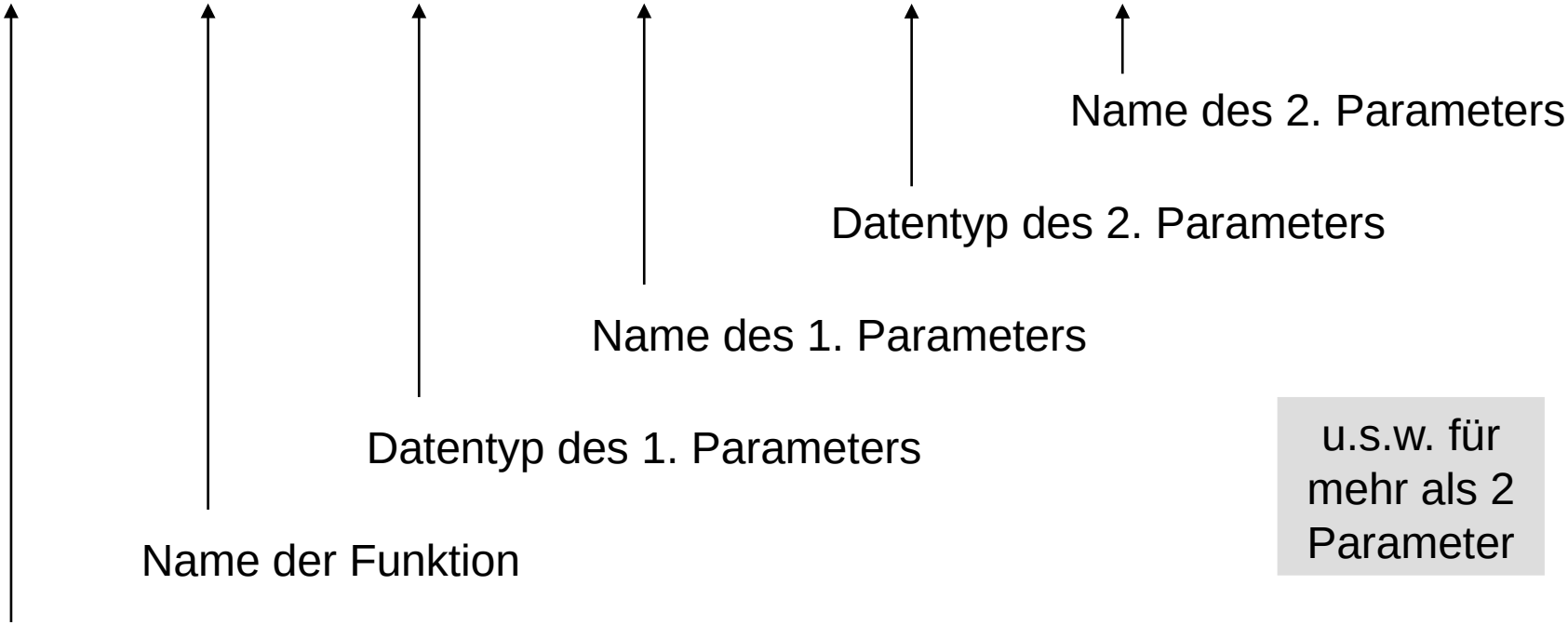
Ausgabe:



(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);

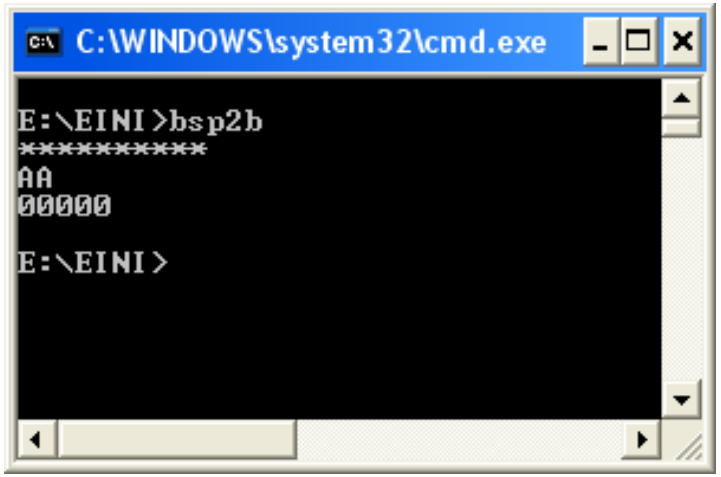
```
// Beispiel:
#include <iostream>
int main() {
  zeichne_zeichen(10, '*');
  zeichne_zeichen( 2, 'A');
  zeichne_zeichen( 5, '0');
  return 0;
}
```

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

(b) Funktionen mit Parametern aber ohne Rückgabewert

```
// Komplettes Beispiel
#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;
}
```

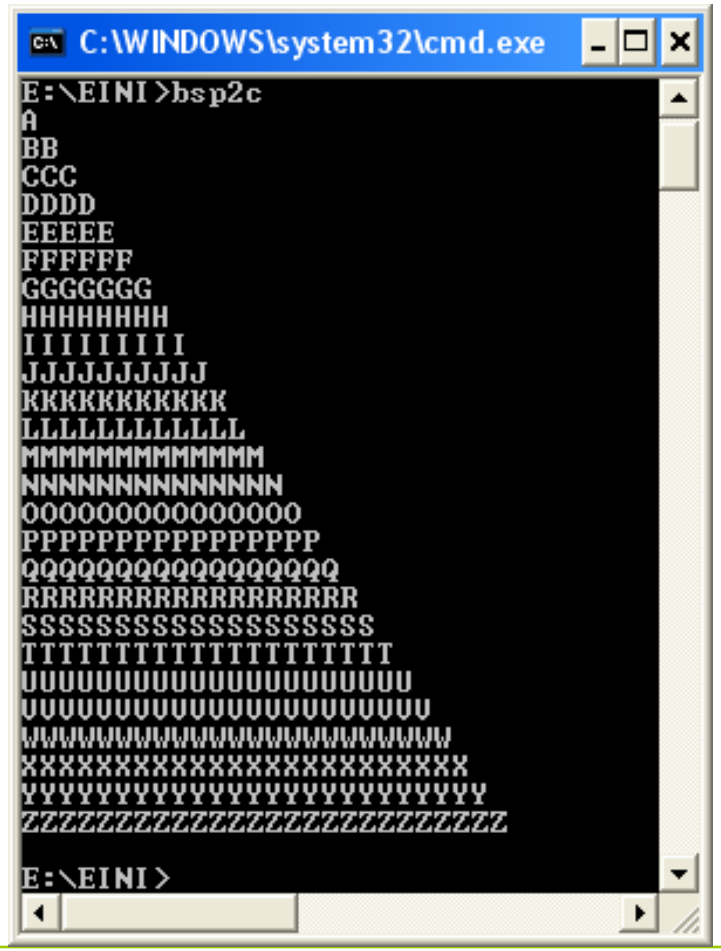
Ausgabe:



(b) Funktionen mit Parametern aber ohne Rückgabewert

```
// Komplettes Beispiel
#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;
}
```

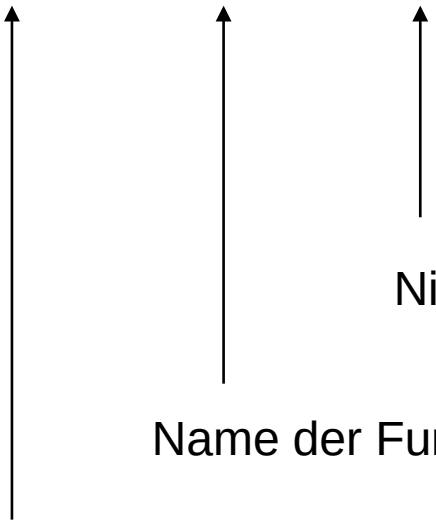
Ausgabe:



(c) Funktionen ohne Parameter aber mit Rückgabewert

- **Funktionsdeklaration:**

```
Datentyp Bezeichner();
```



Nichts zwischen Klammern ⇒ keine Parameter

Name der Funktion

Datentyp des Wertes, der zurückgegeben wird

(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:**

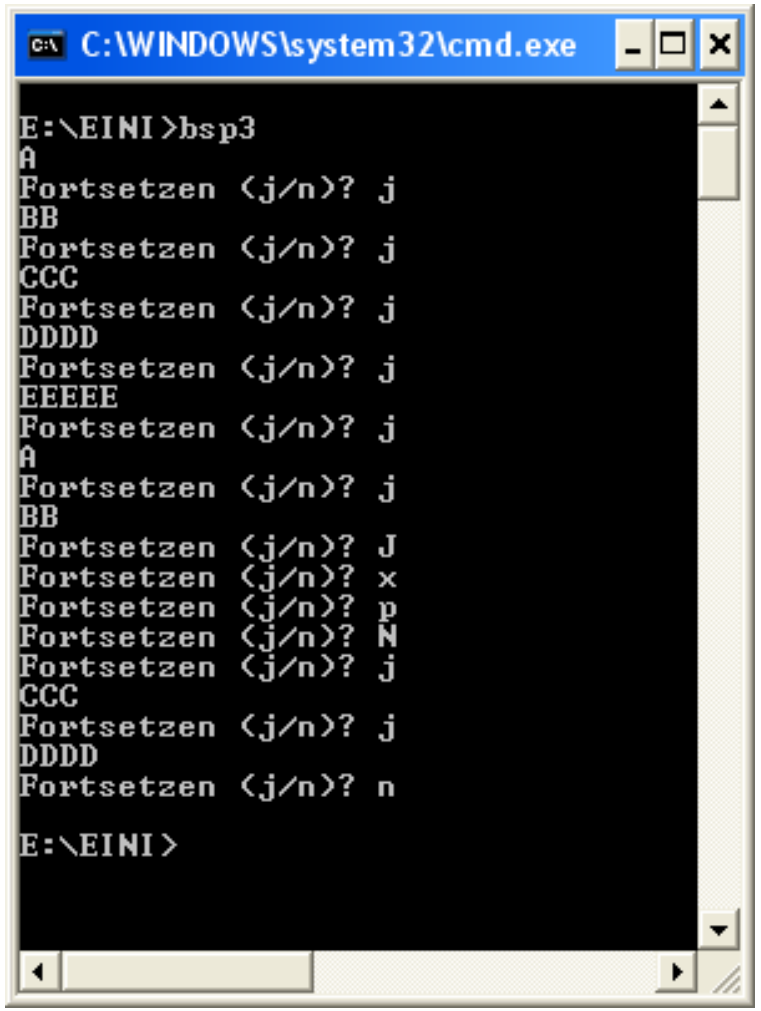
  Variable = Bezeichner **();**

  oder: Rückgabewert ohne Speicherung verwerten

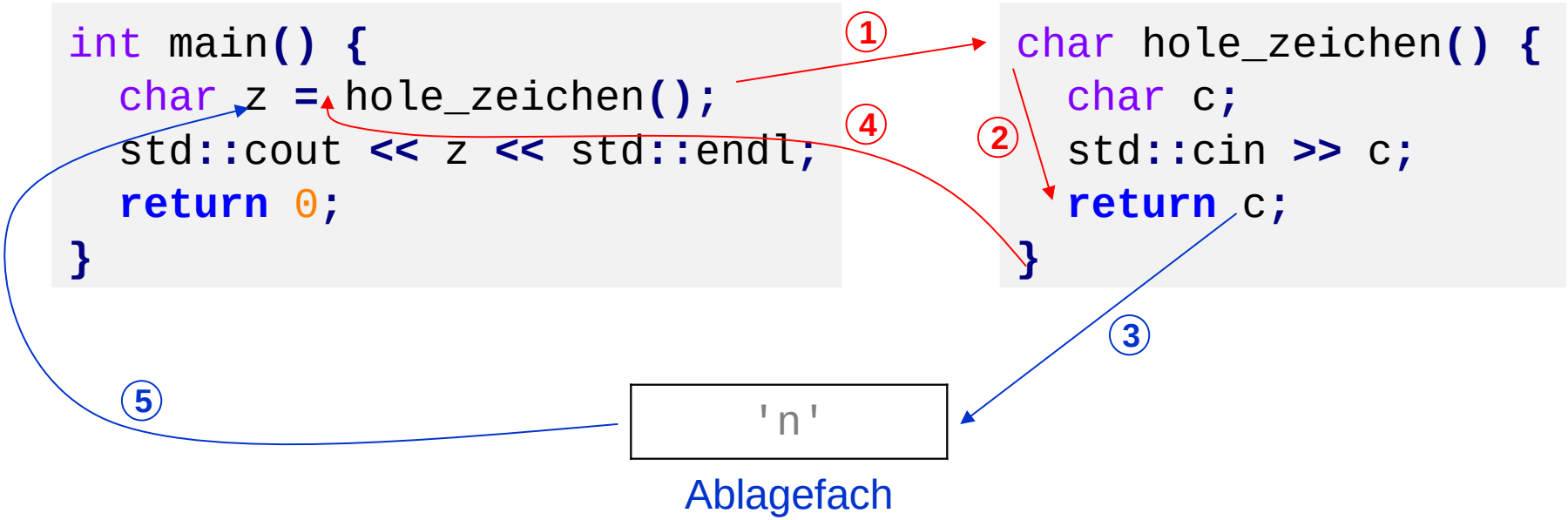
```
// Beispiel:
#include <iostream>
int main() {
  int i = 0;
  do {
    zeichne_zeichen(i + 1, 'A' + i);
    i = (i + 1) % 5;
  } while (fortsetzen());
  return 0;
}
```

(c) Funktionen ohne Parameter aber mit Rückgabewert

```
// Komplettes Beispiel
#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?

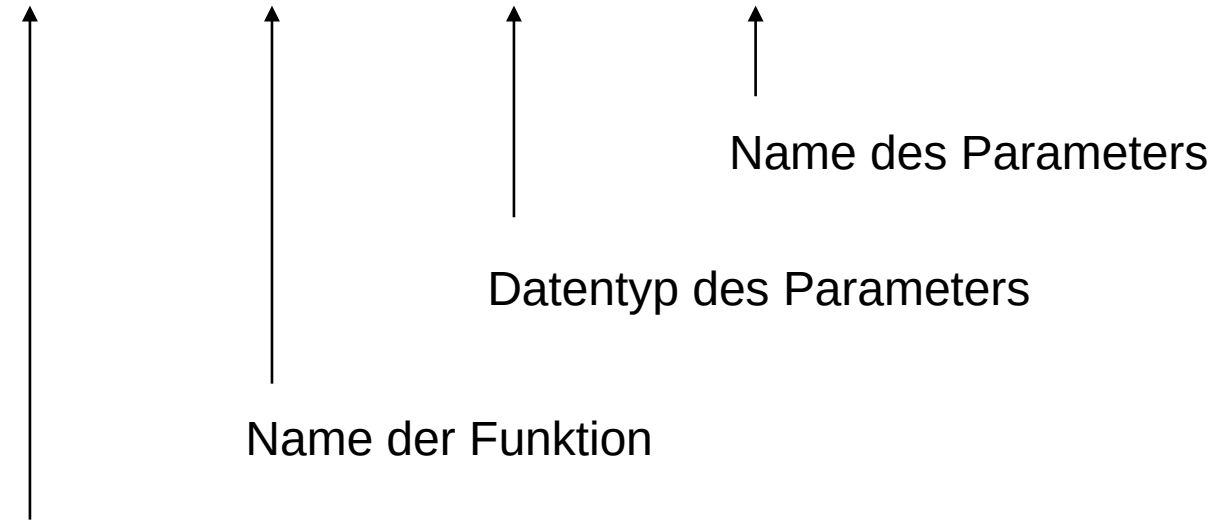


1. Rechner springt bei Aufruf **`hole_zeichen()`** zu den Funktionsanweisungen
2. Die Funktionsanweisungen werden ausgeführt
3. Bei return c wird der aktuelle Wert von c ins Ablagefach gelegt
4. Rücksprung zur aufrufenden Stelle
5. Der zuzuweisende Wert wird aus dem Ablagefach geholt und zugewiesen

(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:**

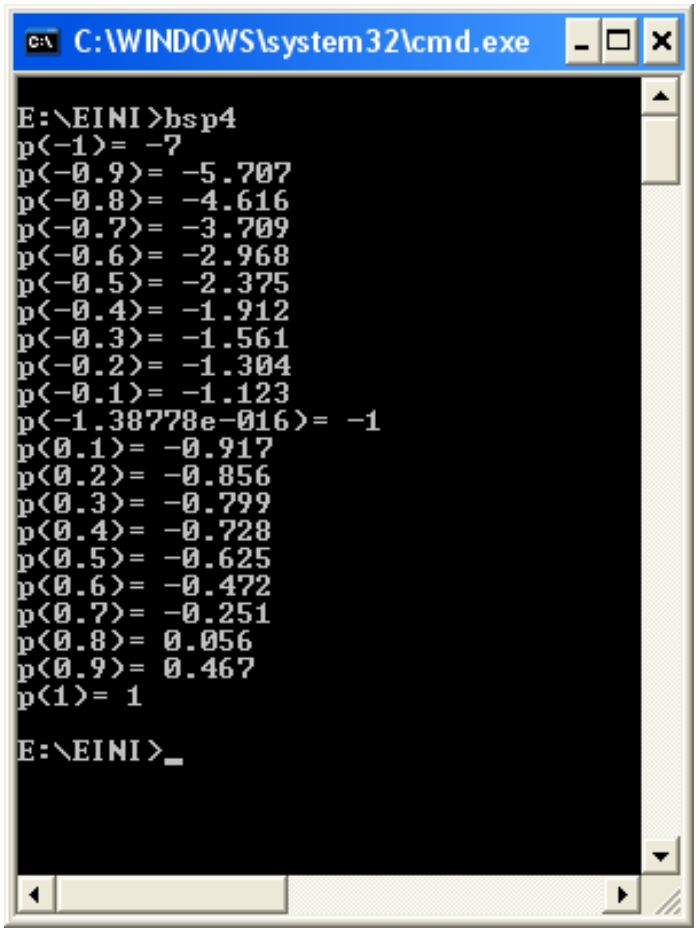
  Variable = Bezeichner**(**Parameter**);**

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;
}
```

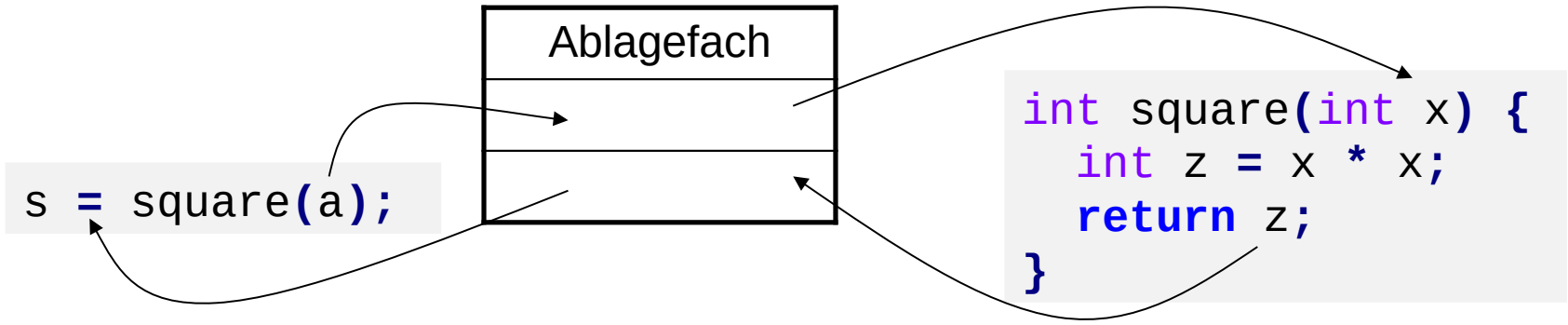
(d) Funktionen mit Parameter und mit Rückgabewert

```
// Komplettes Beispiel
#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 lokal 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 Funktion,
                          // 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;
  cout << "Fkt.: " << *pu << " " << *pv << endl;
}
int main() {
  int a = 3, b = 11;
  cout << "main: " << a << " " << b << endl;
  tausche_p(&a, &b);
  cout << "main: " << a << " " << b << 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 << endl;
  return 0;
}
```

```
int main() {
  int a = 5, *pa;
  pa = &a;
  square(pa);
  cout << a << endl;
  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;
}
```

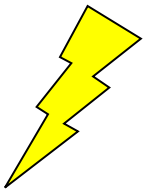
Gute Compiler sollten warnen!

⇒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!**

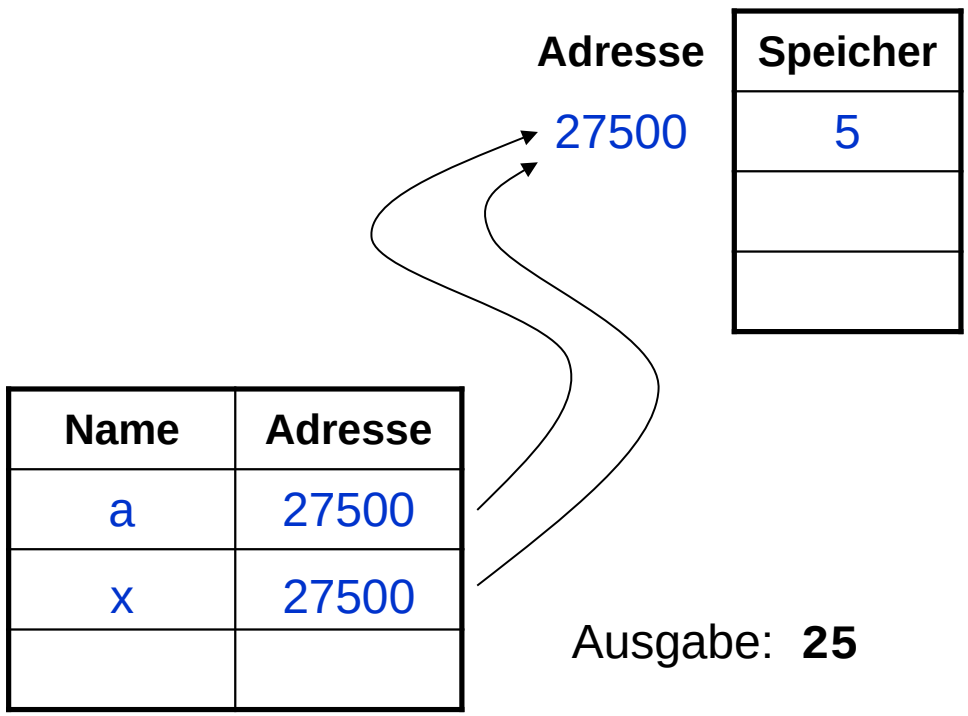


Übergabe einer Referenz (nur in C++, nicht in C)

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 << endl;
  return 0;
}
```



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;
  cout << "Fkt.: " << u << " " << v << endl;
}
int main() {
  int a = 3, b = 11;
  cout << "main: " << a << " " << b << endl;
  tausche_r(a, b);
  cout << "main: " << a << " " << b << 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 Diskriminante $d = (p/2)^2 – q$
- Falls $d > 0$, dann 2 Lösungen
- Falls $d = 0$, dann 1 Lösung
- Falls $d < 0$, 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 d = p * p / 4 - q;
  if (d < 0) return 0;  // keine Lösung
  if (d == 0) {
    x1 = -p / 2;
    return 1;           // 1 Lösung
  }
  x1 = -p / 2 - sqrt(d);
  x2 = -p / 2 + sqrt(d);
  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;
}
```

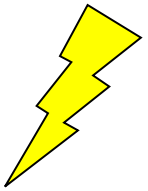
Gute Compiler sollten warnen!

⇒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!**

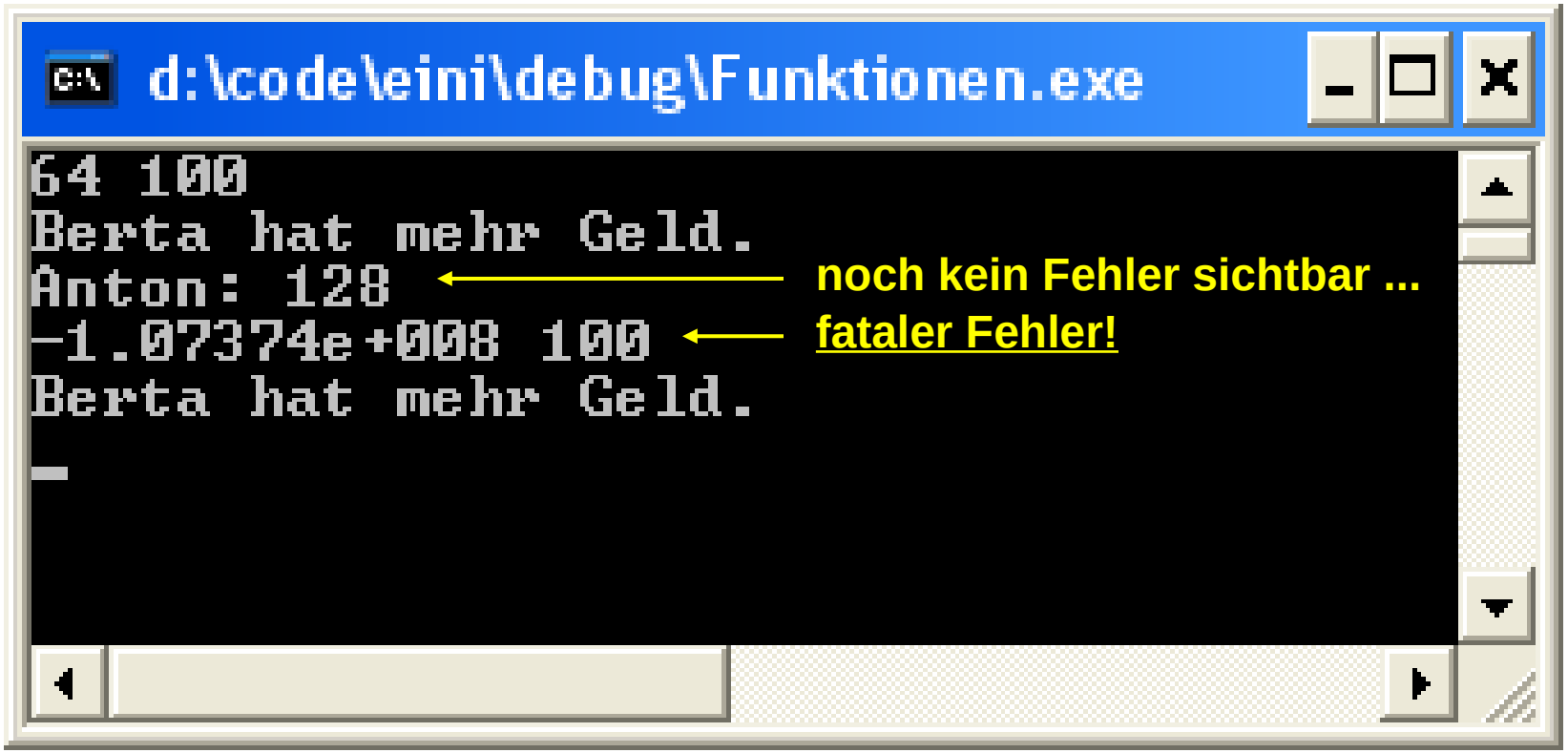


Beispiel:

```
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]++;
}

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

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:

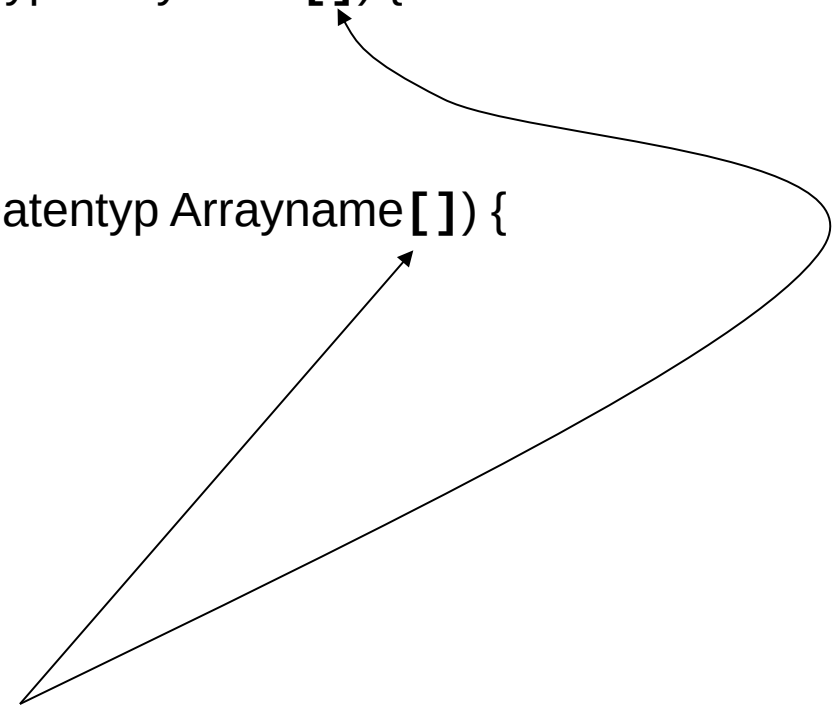
```
3 5 7 9 11
0 0 0 0 0
4 6 8 10 12
1 1 1 1 1
```

… 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) ;

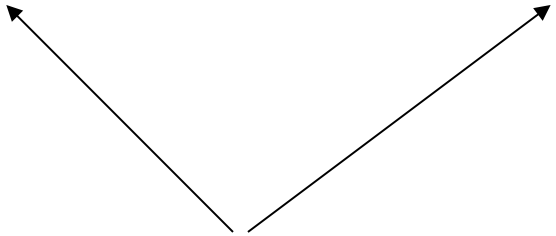
```
int a[] = { 1, 2 };
inkrement(2, a);
```

oder:

Funktionsname(&Arrayname[0]) ;

Variable = Funktionsname(&Arrayname[0]) ;

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



Tatsächlich: Übergabe des Arrays mit Zeiger!

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: irreführend!

Der Compiler **ignoriert die Größenangabe**.

Übergabe von zweidimensionalen Arrays:

Im Prototypen muss **die Spaltenkonstante** angegeben 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(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) ;

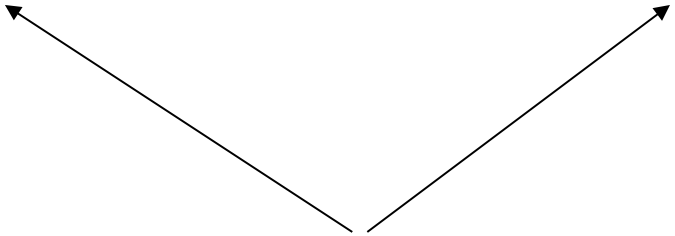
```
int a[][2] = {{1,2},{3,4}};
inkrement(2, a);
```

oder:

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

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

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



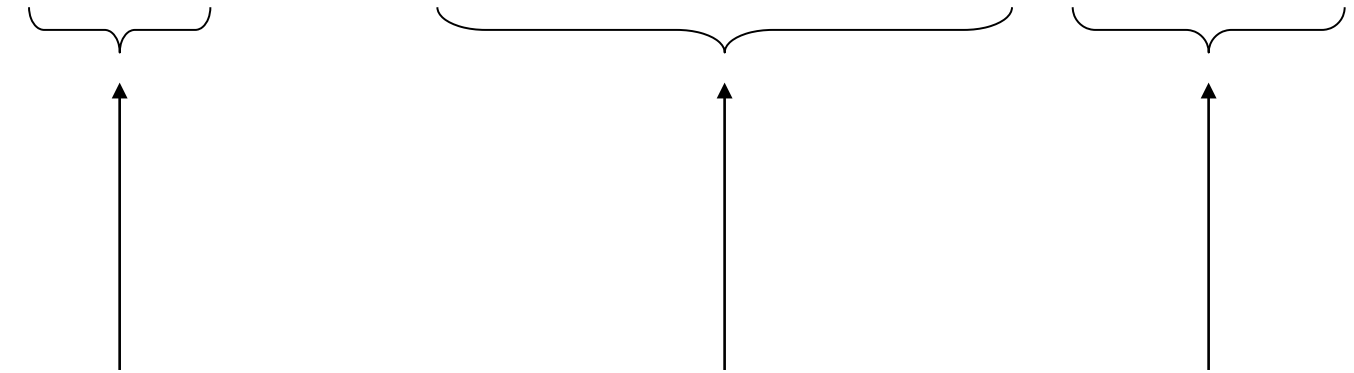
Tatsächlich: Übergabe des Arrays mit Zeiger!

1. 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`**