

TECHNISCHE UNIVERSITÄT DORTMUND

Wintersemester 2007/08

**Einführung in die Informatik für
Naturwissenschaftler und Ingenieure**
(alias Einführung in die Programmierung)
(Vorlesung)

Prof. Dr. Günter Rudolph
Fachbereich Informatik
Lehrstuhl für Algorithm Engineering

Kapitel 5: Funktionen

Inhalt

- Funktionen
 - mit / ohne Parameter
 - mit / ohne Rückgabewerte
- Übergabemechanismen
 - Übergabe eines Wertes
 - Übergabe einer Referenz
 - Übergabe eines Zeigers
- Programmieren mit Funktionen → anschließend

} ☑
} hier

Rudolph: EINI (WS 2007/08) • Kap. 5: Funktionen 2

Kapitel 5: Funktionen

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

Rudolph: EINI (WS 2007/08) • Kap. 5: Funktionen 3

Kapitel 5: Funktionen

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!

Rudolph: EINI (WS 2007/08) • Kap. 5: Funktionen 4

Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

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!

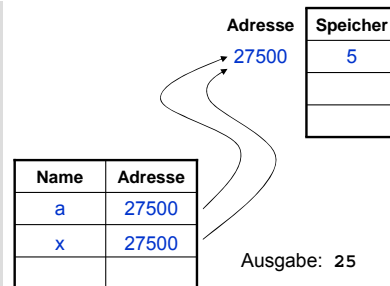
Kapitel 5: Funktionen

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;
}
```



Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

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;  
}
```

Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

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

Kapitel 5: Funktionen

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
}
```

Kapitel 5: Funktionen

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.

Kapitel 5: Funktionen

Rückgabe einer Referenz

ACHTUNG:

Niemals Referenz auf lokales Objekt zurückgeben!

```
const KontoT &verdoppeln(const KontoT &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!**



Kapitel 5: Funktionen

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:

```

C:\ d:\code\eini\debug\Funktionen.exe
64 100
Berta hat mehr Geld.
Anton: 128 ← noch kein Fehler sichtbar ...
-1.07374e+008 100 ← fataler Fehler!
Berta hat mehr Geld.
    
```

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-Variablenname);

Variable = Funktionsname(Zeiger-auf-Variablenname);

```

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`)

Übergabe eines Zeigers:

```

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!

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 von Arrays:

Zur Erinnerung:

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

Soeben gesehen: mit Zeigern verändert man Originalwerte.

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++) std::cout << a[i] << "\n";
}
```



Kapitel 5: Funktionen

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;  
}
```

Kapitel 5: Funktionen

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); ← Bereichsfehler  
    for (i = 0; i < 5; i++) cout << a[i] << " ";  
    cout << endl;  
    for (i = 0; i < 5; i++) cout << b[i] << " ";  
    cout << endl;  
    return 0;  
}
```

Bereichsfehler

Ausgabe:

```
2 4 6 8 10  
0 0 0 0 0  
3 5 7 9 11  
1 1 1 1 1
```

... auch Laufzeitfehler möglich!

Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

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!

Kapitel 5: Funktionen

Übergabe eines Arrays als Zeiger:

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

Achtung! Legale Syntax, aber irreführend:

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

Programmier ging davon aus, dass Array ia 10 Elemente hat!

Aber: fataler Irrtum!
Compiler ignoriert die Größenangabe!

Kapitel 5: Funktionen

Übergabe von zweidimensionalen Arrays:

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

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

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

Kapitel 5: Funktionen

Übergabe von zweidimensionalen Arrays:

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

Mindestanforderung!

oder:

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

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

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

Kapitel 5: Funktionen

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!

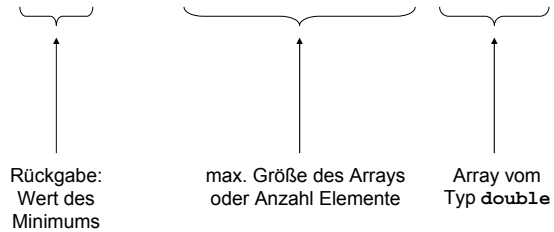
Kapitel 5: Funktionen

Aufgabe:

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

Prototyp, Schnittstelle:

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



Kapitel 5: Funktionen

Aufgabe:

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

Implementierung:

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

Kapitel 5: Funktionen

Test:

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

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

Kapitel 5: Funktionen

Der „Beweis“ ...

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