

## Software-Technik: Vom Programmierer zur erfolgreichen ...

1. Von der Idee zur Software
2. Funktionen und Datenstrukturen
3. Organisation des Quellcodes
4. **Werte- und Referenzsemantik**
5. Entwurf von Algorithmen
6. Fehlersuche und –behandlung
7. Software-Entwicklung im Team
8. Abstrakte Datentypen: Einheit von Daten und Funktionalität
9. Vielgestaltigkeit (Polymorphie)
10. Entwurfsprinzipien für Software

Anhang A: Die Familie der C-Sprachen

Anhang B: Grundlagen der C++ und der Java-Programmierung



# Wichtige Ergänzungen zu Zeigern

## Werte- und Zeigersemantik mit Strukturen in C++

```
struct Cmpl {  
    double re, im;  
};  
...  
Cmpl cx, cy;
```

### Wertesemantik

```
cx.re = 1.5; cx.im = 3.7;
```

```
cy = cx;
```

```
...
```

```
struct Cmpl {  
    double re, im;  
};  
...  
Cmpl *cx, *cy;
```

### Zeigersemantik

```
...  
cx = new Cmpl;  
cy = new Cmpl;
```

```
...  
cx->re = 1.5; cx->im = 3.7;
```

```
//(*cx).re = 1.5; (*cx).im = 3.7;
```

```
*cy = *cx;
```

```
...
```

```
delete cx; delete cy;
```

## Clicker-“Abstimmung“

```
/* Rückgabe von Summe und Differenz der Argumente */  
int* returnArray(int i1, int i2){  
    int arr[2];  
    arr[0] = i1 + i2; arr[1] = i1 - i2;  
    return arr;  
}
```

Was wird auf dem Bildschirm ausgegeben?

1. 3 -1 17 9
2. 17 9 17 9
3. 17 9 18510766 18510776
4. Keine Ahnung, mal so mal so

```
/* Ausgabe von Summe und Differenz  
von 1 und 2 sowie 13 und 4*/  
void main() {  
    int* retValues1 = returnArray(1, 2);  
    cout << "retValues1[0, 1] : " << retValues1[0]  
        << " " << retValues1[1];  
  
    int* retValues2 = returnArray(13, 4);  
    cout << "retValues2[0, 1] : " << retValues2[0]  
        << " " << retValues2[1];  
}
```

## Clicker-“Abstimmung“

```
int main(){  
    int N=5;  
    int arr[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    arr[-1] = 12;  
    for (int i=1; i < N; ++i) {  
        cout << arr[i] << " ";  
    }  
}
```

Was wird auf dem Bildschirm ausgegeben?

1. 2 3 4 5
2. 1 2 3 4 5
3. 2 3 4 5 6
4. 2 3 4 5 6 7 8 9 10 0 0

Ergebnis:

\_\_\_ 1 \_\_\_ 2 \_\_\_ 3 \_\_\_ 4; je nachdem wo N im Speicher abgelegt ist

## Wertesemantik bei der Funktionsrückgabe

```
typedef struct { double re, im; } Cmpl;  
/* bzw. struct Cmpl { double re, im; }; */  
  
...  
Cmpl Add(Cmpl a, Cmpl b) {  
    Cmpl temp;  
    temp.re = a.re + b.re; temp.im = a.im + b.im;  
    return temp;  
}  
  
...  
Cmpl cx, cy, cz;  
cx.re = 1.5; cx.im = 3.7; cy = cx;  
cz = Add(cx, cy);  
...;
```

Der im Stackbereich der Funktion *Add* reservierte Speicherplatz für die Variable *temp* wird beim Beenden der Funktion automatisch wieder freigegeben.

## Zeigersemantik bei der Funktionsrückgabe in C++

```
Cmpl* Add(Cmpl* a, Cmpl* b) {  
    Cmpl* temp = new Cmpl;  
    temp->re = a->re + b->re; temp->im = a->im + b->im;  
    return temp;  
}
```

```
...  
Cmpl *cx, *cy, *cz;  
cx = new Cmpl;  
cy = new Cmpl;  
cz = new Cmpl;  
cx->re = 1.5; cx->im = 3.7; *cy = *cx;  
cz = Add(cx, cy);  
...;  
delete cx; delete cy; delete cz;
```

Warum/ Wo tritt in diesem Programmfragment ein Problem auf?

```
/* Der ursprünglich für cz reservierte Speicherplatz wird  
nicht freigegeben, es entsteht ein Speicherleck !!! */
```

## Zeiger als Parameter in Funktionen

```
void funk(int i, int j)
{
    i= 67;
    j=j+4;
}

int main()
{
    int par1 = 22;
    int par2 = 88;

    funk(par1, par2);
    // par1, par2 haben hier immer
    // noch den Wert 22 bzw. 88.
}
```

```
void funk(int i, int *pj) {
    i= 67;
    *pj = *pj + 4;
}

int main() {
    int par1 = 22;
    int par2 = 88;
    funk(par1, &par2);
    // par1 ist immer noch 22
    // par2 ist nun aber 92.
}
```

```
// mit Referenzen
void funk(int i, int &pj) {
    i= 67;    pj = pj + 4;
}

int main() {
    int par1 = 22, par2 = 88;
    funk(par1, par2);
}
```



## Zeiger als Parameter in Funktionen (2)

Statt der Übergabe der Adresse von par2 kann auch direkt ein Zeiger übergeben werden:

```
int main()
{
int par1 = 22;
int par2 = 88;
int * ppar2 = &par2;

funk(par1, ppar2);
}
```

```
int main()
{
int par1 = 22;
int par2 = 88;

funk(par1, &par2);
}
```

## Clicker

```
void funk(int i, int *pj) {  
    i= 67;  
    *pj = *pj + 4;  
}  
int main() {  
    int par1 = 22;  
    int par2 = 88;  
    funk(par1, &par2);  
    cout << par1 << " " << par2;  
}
```

Bildschirmausgabe ?

1. 22 88
2. 67 71
3. 22 92
4. 67 92

## Clicker

```
void funk(int i, int pj) {  
    i= 67;  
    pj = pj + 4;  
}  
int main() {  
    int par1 = 22;  
    int par2 = 88;  
    funk(par1, par2);  
    cout << par1 << " " << par2;  
}
```

BildschirmAusgabe ?

1. 22 88
2. 67 71
3. 22 92
4. 67 92

überspringen

## Clicker

```
void funk(int& i, int pj) {  
    i= 67;  
    pj = pj + 4;  
}  
int main() {  
    int par1 = 22;  
    int par2 = 88;  
    funk(par1, par2);  
    cout << par1 << " " << par2;  
}
```

Bildschirmausgabe ?

1. 22 88
2. 67 88
3. 22 92
4. 67 92

überspringen

## Clicker

```
void funk(int* i, int pj) {  
    *i= 67;  
    pj = pj + 4;  
}  
int main() {  
    int par1 = 22;  
    int par2 = 88;  
    funk(&par1, par2);  
    cout << par1 << " " << par2;  
}
```

BildschirmAusgabe ?

1. 22 88
2. 67 88
3. 22 92
4. 67 92

## Nicht typisierte Zeiger

```
double* px;           // typisierter Zeiger
void* p;              // nicht typisierter Zeiger

p = px;              // richtig
px = p;              // in C++ falsch, in C schlecht
px = (double*) p;    // in Ordnung, C-Stil
px = reinterpret_cast<double*>(p); // in Ordnung, C++-Stil
```

- Nicht typisierte Zeiger sind mit allen typisierten Zeigern kompatibel !
- Sie können nicht dereferenziert werden.
- Sie können verwendet werden, um generisch zu programmieren!
- Was aber häufig eine unsichere Sache ist.



## Beispiel für das Verwenden nicht typisierter Zeiger

```
void copy(void* ziel, const void * quelle, int count) {  
    char* y = reinterpret_cast<char*>(ziel);  
    char* x = reinterpret_cast<char*>(quelle);  
    for (i=0; i < count; ++i) {  
        *y = *x; ++y; ++x; }  
}  
...  
Somewhat a, b;  
int i, j;  
...  
copy(&b, &a, sizeof(Somewhat));  
copy(&j, &i, sizeof(int));           // entspricht j = i
```

## Weiteres zu Zeigern



## **const\_cast-Operator**

Der Operator ***const\_cast*** erlaubt einen Zeiger auf ein konstantes Objekt in einen Zeiger auf ein nicht konstantes Objekt zu konvertieren:

```
const int size = 100;  
int* ptrSize  = const_cast<int*>(& size);  
*ptrSize      = 150;                    // d.h. size erhält den Wert 150
```

***const\_cast*** ist der einzige C++-Konvertierungsoperator, der ein *const*-Attribut einer Variablen „weg“casten kann.



## const\_cast-Operator

Der Operator **const\_cast** erlaubt einen Zeiger auf ein konstantes Objekt in einen Zeiger auf ein nicht konstantes Objekt zu konvertieren:

```
const int size = 100;
int* ptrSize = const_cast<int*>(& size);
*ptrSize = 150;
const int* ptr2 = &size;
const_cast<int*>(*ptr2) = 499; // ohne const_cast kann über ptr2 die
// Speicherstelle, auf die ptr2 (bzw.
// ptrSize) verweist, nicht geändert werden.
```



## Sinnvolle Anwendung von `const_cast`

```
void Print(char* text)    // Nicht sauber programmiert, da
{                          // const vergessen wurde
    cout << text;        // besser: void Print(const char* text)
}
```

```
void MeinPrint(const char* vname,
               const char* nname)
{
    Print(const_cast<char*> (vname));
    cout << " ";
    Print(const_cast<char*> nname);
}
```

In diesem Fall wäre es vermutlich besser, die Schnittstelle von `Print` zu ändern. In der Praxis ruft aber `Print` auch wieder eine Funktion auf, und dort wird wieder eine Funktion aufgerufen usw.



## Verschiedene Funktionsparameterarten in C und C++

### In C und C++

- Übergabe per Wert: `void f (T x)`
- Übergabe per Zeiger: `void f (T* x)`
- Übergabe per konstantem Zeiger: `void f (const T* x)`

### Nur in C++:

- Übergabe per Referenz: `void f (T& x)`
- Übergabe per konstanter Referenz: `void f (const T& x)`

### Redundant für den Aufrufer:

- Übergabe per konstantem Wert: `void f (const T x)`
- Übergabe per konstantem Zeiger: `void f (T *const x)`
- Übergabe per konstantem Zeiger: `void f (const T *const x)`

Die verschiedenen Programmierparadigmen von C++

## Const Zeiger

## Verschiedene Rückgabearten für Funktionswerte

### In C und C++:

- Rückgabe eines Werts: **T f(...)**
- Rückgabe eines Zeiger: **T\* f(...)**
- Rückgabe eines Zeigers auf einen konstanten Wert: **const T\* f(...)**

### Nur in C++

- Rückgabe einer Referenz: **T& f(...)**
- Rückgabe einer konstanten Referenz : **const T& f(...)**

### Keine Information für den Aufrufer

- **const T f(...)**
- **T\* const f(...)**
- **const T\* const f(...)**

## Verwendung des Attributs const bei Zeigern

Bei der Deklaration von Zeigern kann sich die Konstanz auf den Zeiger oder auf den Inhalt oder auf beides beziehen:

```
char Name1[20] = "Isernhagen";
char Name2[20] = "Meyer";

char* s1          = Name1;          /* var. Zeiger, var. Inhalt */
const char* s2    = Name1;          /* var. Zeiger, konst. Inhalt */
char* const s3    = Name1;          /* konst. Zeiger, var. Inhalt */
const char* const s4 = Name1;       /* konst. Zeiger, konst. Inhalt */

s1[0] = 'A';          /* i.O. */ /* entspricht *s1 = 'A'; */
strcpy(s1, "xxxx");  /* i.O. */
s1 = Name2;          /* i.O. */

s2[0] = 'A';          /* Fehler: L-Wert gibt ein konstantes Objekt an */
strcpy(s2, "xxxx");  /* Fehler: versch. Typen fuer form. & akt. Param. */
s2 = Name2;          /* i.O. */
```

## Verwendung des Attributs const bei Zeigern (2)

```
char* s1           = Name1;    /* var. Zeiger, var. Inhalt */
const char* s2     = Name1;    /* var. Zeiger, konst. Inhalt */
char* const s3     = Name1;    /* konst. Zeiger, var. Inhalt */
const char* const s4 = Name1;  /* konst. Zeiger, konst. Inhalt */

s3[0] = 'A';        /* i.O. */
strcpy(s3, "xxxx"); /* i.O. */
s3 = Name2;        /* Fehler: L-Wert gibt ein konstantes Objekt an */

s4[0] = 'A';        /* Fehler: L-Wert gibt ein konstantes Objekt an */
strcpy(s4, "xxxx"); /* Fehler: versch. Typen fuer form. & akt. Param. */
s4 = Name2;        /* Fehler: L-Wert gibt ein konstantes Objekt an */
```



## Zeiger - Speicherbedarf von Zeigern

Der Speicherplatz, der für einen Zeiger reserviert wird, muss eine Speicheradresse aufnehmen können.

Das bedeutet, dass ein Zeiger des Typs *int* und ein Zeiger auf einen Datentyp *double* normalerweise gleich groß sind.

Der Typ, der einem Zeiger zugeordnet ist, gibt den Inhalt und damit auch die Größe des adressierten Speicherbereichs an.

```
int* pint ;           // pint belegt 4 Byte      (*)  
double* pdouble;     // pdouble belegt auch 4 Byte (*)
```

(\*) Der für Zeiger vorgesehene Speicherplatz ist natürlich implementierungsabhängig. Der Wert von 4 Byte ist der für heutige 32-Bit-Systeme in der Regel verwendete.

## **Beziehung zwischen Zeigern und Arrays**

## Beziehungen zwischen Zeigern und Arrays

Einem Zeiger kann der Name eines Arrays zugewiesen werden:

```
int arr[10];  
int* ptr = arr;    // entspricht: int* ptr = &arr[0]
```

Der Arrayname entspricht damit der Adresse des ersten Arrayelementes, d.h. dem mit dem Index 0.

Ein Zeiger kann wie ein Array benutzt werden, z.B. „*ptr [4] = 16;*“:

```
void funk(int* ptr, int arr[], int k)  
{  
    ptr [k] = arr [4];  
}
```

```
int main(){  
    int a[14], b[25];  
    funk(a, &b[0], 4);  
    funk(&a[2], b, 2);  
}
```

## Zeiger, Adressen und Vektoren, Beispiel

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

int main(void) {
    int i;
    int a[10];      /* Array fuer 10 int-Werte      */
    int b[10];      /* Array fuer 10 int-Werte      */
    int* pb;        /* ein Zeiger auf einen int-Wert */
    int c[9];       /* ein Array fuer 9 int-Werte   */
    int* pc;        /* ein Zeiger auf einen int-Wert */

    pb = &b[0];     /* oder auch pb = b             */
    pc = c;         /* oder auch pc = &c[0]        */
}
```

## Zeiger, Adressen und Vektoren, Beispiel (2)

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

```
int main(void) {
    int i;
    int a[10];
    int b[10];
    int* pb;
    int c[9];
    int* pc;
```

```
    pb = &b[0];
    pc = c;
```

```
    for (i = 0; i < 10; ++i) {
        a[i] = i * i;
        *pb = i * i; ++pb;
        *pc = i * i; ++pc;    /* Achtung: es ist nur */
    } /* Speicherplatz fuer c[0] bis c[8] reserviert !!! */

    pb = &b[0]; pc = c;
    for (i = 0; i < 10; ++i) {
        cout << setw(4) << i << setw(4) << a[i]
             << setw(4) << b[i] << setw(4) << *pb
             << setw(4) << c[i] << setw(4) << *pc
             << endl;
        ++pb; ++pc;
    }
    return 0;
}
```

## Dualität zwischen Vektoren und Zeigern

- `int a[10]` reserviert Speicher für 10 Integer: `a[0]` bis `a[9]`
- `int* pb` bedeutet: der Inhalt von `pb` ist vom Typ `int`, also `pb` ist ein Zeiger
- Mit `pb = &b[0]` wird der Zeiger `pb` auf das erste Element gesetzt
- `pb = b` bewirkt das gleiche!
- Ein Vektor (Array) wird durch seine Adresse dargestellt
- Zugriff auf das `i`-te Element: `b[i]` oder `*(b+i)` bzw. `pb[i]` oder `*(pb+i)`
- `b+i` bedeutet `b` wird *typgerecht* um `i` Schritte erhöht
- Es findet keine Bereichsüberprüfung des Index statt!
- Adressrechnungen werden ohnehin nicht überprüft
- Das obige Programm funktioniert *meistens, nicht immer !!!*  
(Es ist kein Platz für `c[9]` reserviert !!!)

## Dualität zwischen Vektoren und Zeigern (2)

