

Vorlesung Informatik 3  
Sommersemester 2012

## Foliensatz

Stand: 29.02.2012

Hochschule für Technik  
Fakultät Informationstechnik  
Prof. Dr.-Ing. Nikolaus Kappen

## Überblick über die Vorlesungs-Bausteine

- B1 Einführung
- B2 Standard-Ein-/Ausgabe
- B3 Voreingestellte Funktionen
- B4 Namensräume
- B5 Inline-Funktionen
- B6 Schlüsselwort const
- B7 Überladen von Funktionen
- B8 Referenzen
- B9 Freispeicherverwaltung
- B10 Klassen
- B11 Konstruktoren
- B12 const-Objekte
- B13 Objekte als Klassenelemente
- B14 Friends
- B15 Klassen mit Zeigerelementen
- B16 Überladen des Zuweisungsoperators
- B17 Zeiger this
- B18 Kopierkonstruktor
- B19 Funktionen und Instanzen
- B20 Static-Attribute und static-Methoden
- B21 Verwandte Datentypen in C
- B22 Vererbung
- B23 Polymorphismus
- B24 Exceptions
- B25 STL
- B26 Advanced Topics

## Literaturverzeichnis

- /1/ Microsoft Corporation Richtig einsteigen in C++  
Microsoft Press 1991 (vergriffen, unser Textbook)**
- /2/ Bruce Eckel In C++ denken  
Prentice Hall Verlag GmbH 1996
- /3/ Herbert Schildt C++ from the GROUND UP  
Osborn McGraw-Hill Verlag 1998
- /4/ Herbert Schildt C++ The Complete Reference  
Osborn McGraw-Hill Verlag 1998
- /5/ André Willms GoTo C++ Programmierung  
Addison Wesley Longman Verlag GmbH 1999
- /6/ Richard Kaiser C++ mit MS Visual C++ 2008  
Springer Verlag 2009**
- /7/ Rolf Isernhagen Softwaretechnik in C und C++  
Hartmut Helmke Carl Hanser Verlag 2004
- /8/ John R. Hubbard C++ - Programmierung  
mitp-Verlag 2003

## Einführung

Die Unterlagen zur Vorlesung Informatik 3 gliedern sich in:

- Textbook
- Foliensatz
- begleitende Übungsaufgaben
- Übungen am Rechner (Block 1 .. 5)

Die Unterlagen bis auf das Textbook können von meiner WebSite bezogen werden:

[www.fht-esslingen.de/~kappen](http://www.fht-esslingen.de/~kappen)

Wegen dem Textbook sprechen Sie mich an.

In der Vorlesung werden die einzelnen Bausteine an Hand der Folien erläutert. Die Folien stellen einen Extrakt aus dem Textbook dar und erklären sich ohne das Textbook nicht immer von selbst. In der Vorlesung werden die Folien durch weitere Erläuterungen und Programmbeispiele ergänzt.

Zur Vorbereitung auf die jeweils nächste Vorlesungseinheit ist das Studium des Textbooks zu empfehlen. Die entsprechenden Seiten im Textbook sind auf den Folien oben rechts angegeben (TB-S = Textbook-Seite).

Für die Übungen am Rechner stehen 5 Blöcke zur Verfügung. Die Übungstermine sind im Terminplan festgelegt (siehe WebSite).

## Einführung von C++ auf der Basis von C

### **Schwerpunkten:**

- OOP-unspezifische Änderungen von C**
- Datenkapselung**
- Vererbung**
- Polymorphismus**
- Exceptions**
- STL**

## Standard-Ein-/Ausgabe

Die Standard-Ein-/Ausgabe wird in C durch die Bibliotheksfunktionen **printf** und **scanf** erreicht.

Nachteil bei diesen Bibliotheksfunktionen u.a.: Der Typ der Variable muss explizit angegeben werden.

In C++ wird ein **Klassen**konzept verfolgt. Zur Standard-Ausgabe wird das **Objekt cout** der Klasse **ostream** mit dem **überladenen Operator <<** verwendet. Zur Standard-Eingabe das **Objekt cin** der Klasse **istream** mit dem **überladenen Operator >>**.

(Die Begriffe **cout**, **cin**, etc. werden erst im Verlauf der Vorlesung erläutert. Daher wird hier die neue Standard-Ein-/Ausgabe mit **cout** und **cin** nur als "Black-Box" benutzt und ausschliesslich durch Beispiele eingeübt.)

Vorteile von **cout** und **cin**:

- selbstständige Einstellung auf den Variablentyp
- Ein-/Ausgabe von benutzerdefinierten Datentypen

## Manipulatoren

<b>oct</b>	Ganzzahlen oktal ausgeben
<b>hex</b>	Ganzzahlen hexadezimal ausgeben
<b>dec</b>	Ganzzahlen dezimal ausgeben
<b>endl</b>	ein "\n" anfügen und flush ausführen
<b>left</b>	nach Zahlenwert füllen
<b>right</b>	vor Zahlenwert füllen
<b>uppercase</b>	Ausgabe in Großbuchstaben
<b>nouppercase</b>	Ausgabe in Kleinbuchstaben
<b>fixed</b>	Gleitpunktzahlen als ddd.dd
<b>scientific</b>	Gleitpunktzahlen als d.ddddEdd
<b>ends</b>	ein "\0" anfügen und flush ausführen
<b>flush</b>	Ausgabepuffer entleeren
<b>ws</b>	whitespace von der Eingabe entfernen

## Beispiele zur Standardausgabe cout

```
#include <iostream.h>
int main(void)
{cout << "Hello world\n";
}
```

```
#include <iostream.h>
int main(void)
{double z = 13.4;
  cout << z << endl;
  cout << 13.4 << endl;
  cout << 10 << endl;
  cout << "Wert von z=" << z << endl;
}
```

## Beispiele zur Standardeingabe cin

```
#include <iostream.h>
int main(void)
{int z;
  cin >> z;
  cout << "\nZahl z=" << z;
}
```

```
#include <iostream.h>
int main(void)
{char name[20];
  cin >> name;
  cout << "\nName" << name;
}
```

## Voreingestellte Funktionen (1)

In C++ ist es möglich, bei der Funktionsdefinition den formalen Übergabeparametern einen Default-Wert zu geben.

Parameter mit Default-Werten können beim Funktionsaufruf weggelassen werden.

Die Funktion

```
void fName (double num = 0.0, char ch = 'X');
```

kann auf 3 verschiedene Arten aufgerufen werden:

```
fName(19.8, 'A');    //num = 19.8; ch = 'A'  
fName(19.8);        //num = 19.8; ch = 'X'  
fName();            //num = 0.0;  ch = 'X'
```

## Voreingestellte Funktionen (2)

Parameter, die nur in seltenen Fällen ihren Wert ändern, werden am Besten als Default-Parameter notiert. Beim Aufruf der Funktion kann dann auf die aktuelle Notation dieses Parameter verzichtet werden.

Um Eindeutigkeit zu erreichen, können Default-Parameter nur von rechts nach links weggelassen werden. Daraus folgt, dass bei der Definition der Funktion Default-Parameter nicht links von Parametern ohne Default-Vorgabe stehen dürfen.

Werte von Default-Parametern dürfen nur einmal angegeben werden, also entweder in der Deklaration oder in der Definition, nicht in beiden. Gibt es eine Deklaration und eine Definition, dann in der Deklaration.

## Beispiel 1

```
void myFunc (int = 5, double = 1.23);
```

```
myFunc(12,3.45);    //alle Parameter angeben  
myFunc (3);        //entspricht myFunc (3, 1.23)  
myFunc ();         //entspricht myFunc (5, 1.23)  
myFunc (1.5);      //nicht erlaubt  
myFunc (,1.5);     //nicht erlaubt
```

```
void myFunc (int, double = 10.); //erlaubt
```

```
void myFunc (int = 5, double);    //nicht erlaubt
```

## Beispiel 2

```
#include<iostream.h>
void show(int = 1,double = 2.3,long = 4L);

int main(void)
{show();
 show(5);
 show(5, 7.8);
 show(5,9.9,12L);
}

void show(int first,double sec,long third)
{cout << "1. Wert: " << first << "  ";
 cout << "2. Wert: " << sec << "  ";
 cout << "3. Wert: " << third << endl;
}
```

## Teste dein Wissen: Aufgabenstellung

1.) Gegeben ist folgender Programmausschnitt:

```
void myFunc(int x,int y);  
...  
void myFunc(int x=0,int y=0)  
{...}
```

Was meint der Compiler dazu?

2.) Schreiben Sie die Funktion `clrscr`, die durch die Ausgabe von Linefeeds den Bildschirm löscht. Berücksichtigen Sie, dass in den meisten Fällen der Bildschirminhalt durch 25 Zeilen dargestellt wird.

## Teste dein Wissen: Lösung

1.)

2.) Die Funktion **clrscr** kann wie folgt aussehen (Notation mit Prototyp):

## Namensräume (1)

Umfangreiche Programme werden häufig durch ein Team von Software-Entwicklern erstellt. Das Gesamtprogramm wird durch Quellcode-Teile zusammengesetzt. Jeder Teil wird getrennt übersetzt und alle übersetzten Teile zu einem ablauffähigen Programm gelinkt. Da Bezeichnernamen (Variablen, Funktionen, etc.) einmalig sein müssen, ist für die Einhaltung ein aufwendiges Vorgehen erforderlich.

Um die Situation zu vereinfachen, wurden Gültigkeitsbereiche für Bezeichner eingeführt, sogenannte Namensräume (namespaces). Bezeichner sind dann nur in einem Namensraum gültig. In einem anderen Namensraum kann der gleiche Bezeichner in anderer Bedeutung benutzt werden.

## Namensräume (2)

Namensräume werden durch das Schlüsselwort **namespace** eingerichtet.

Syntax:

```
namespace mySpace    {int x = 20; ...}  
namespace yourSpace {int x = 10; ...}
```

Ein Zugriff über den Namensraum hinaus wird mit dem Scope-Operator **::** durchgeführt. Syntax:

```
namespace mySpace    {int x = 20;...}  
namespace yourSpace {int x = 10; cout<<mySpace::x;}
```

Wird im Namensraum **yourSpace** die Anweisung

```
using namespace mySpace;
```

notiert, so können sämtliche Bezeichner aus **mySpace** in **yourSpace** ohne den Scope-Operator verwendet werden (siehe folgende Beispiele).

## Beispiel 1

```
#include <iostream.h>

namespace green
{void druck(void){cout << "druck aus green";}
}

//Version 1
int main(void){druck();}

//Version 2
int main(void){green::druck();}

//Version 3
using namespace green;
int main(void){druck();}

//Version 4
using green::druck;
int main(void){druck();}
```

## Beispiel 2

```
#include <iostream.h>

namespace green
{void druck(void){cout << "druck aus green";}
}
//Version 1
void druck(void){cout << "druck";}
int main(void){druck();}

//Version 2
void druck(void){cout << "druck";}
int main(void){druck();green::druck();}

//Version 3
using namespace green;
void druck(void){cout << "druck";}
int main(void){druck();}
```

## Beispiel 3

Für die Nutzung der Ein-/Ausgabe durch **cout** und **cin** stellen moderne Compiler die Datei **iostream** zur Verfügung, in der sämtliche Komponenten für die Ein-/Ausgabe zur Verfügung stehen. Diese Ein-/Ausgabe-Komponenten wurden im Namensraum **std** programmiert, d.h die Notation müßte lauten **std::cout** bzw. **std::cin**, ausser man benutzt das Schlüsselwort **using**.

```
#include <iostream>
using namespace std;
int main(void)
{cout << "Hello namespace";}
```

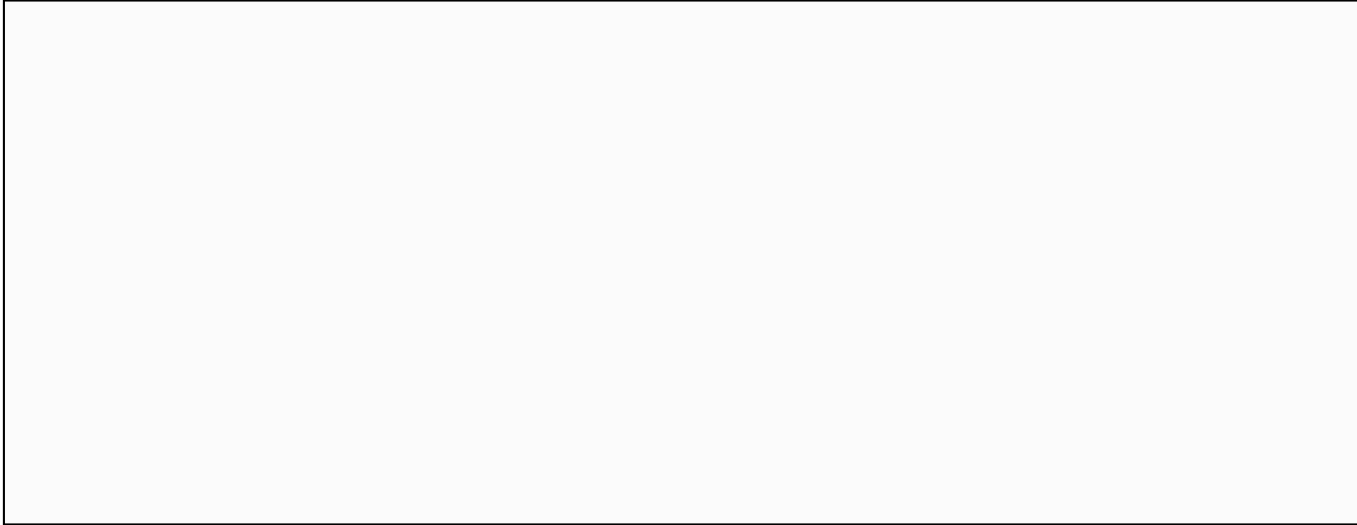
## Teste dein Wissen: Aufgabenstellung

Im folgenden Programm gibt es mehrere Namensräume (engl.: Namespaces). Das Programm lässt sich nicht fehlerfrei übersetzen. Versuche die Fehler zu finden und zu verbessern.

```
#include <iostream>
using namespace std;
namespace eins
{int wert1;

  class A
  {char vau;
  public:
    A(void)
    {vau = 'A';}
  };
}
namespace zwei
{int wert2;
  int mult(int x,int y)
  {return x*y;}
}
int main(void)
{A objA;
  wert1 = 5;
  wert2 = 10;
  cout << mult(wert1,wert2) << endl;}
```

## Teste dein Wissen: Lösung



## Inline-Funktionen (1)

Funktionen werden eingesetzt, um ein Programm zu strukturieren und um Speicherplatz zu sparen.

Bei kleinen Funktionen lohnt es sich, die Makrotechnik einzusetzen, Dadurch wird der Programmablauf beschleunigt und u.U. sogar auch der Speicherplatz reduziert, da der Overhead beim Funktionsaufruf wegfällt. Nachteilig bei der Makrotechnik ist die fehlende Typenüberprüfung sowie das Auftreten von Seiteneffekten.

In C++ gibt es nun eine weitere Möglichkeit, in der Funktionstechnik zu programmieren, aber im ausführbaren Code ohne Funktionsaufrufe auszukommen: die **inline**-Funktion, d.h. eine Funktion wird mit dem Schlüsselwort **inline** gekennzeichnet.

## Inline-Funktionen (2)

Eine Funktion kann mit dem Zusatz **inline** wie folgt gekennzeichnet werden:

```
inline void myFunction(..);
```

Dadurch wird der Compiler aufgefordert, an Stelle der Funktionsaufrufe, den Funktionskörper direkt einzusetzen. Diese Aufforderung kann der Compiler nur bei kleinen, einfachen Funktionen nachkommen. Es gibt Fälle, bei denen der Compiler eventuell die **inline**-Bildung ablehnt:

- bei Funktionen mit einer Schleife, einem **switch** oder einem **goto**
- bei rekursiven Funktionen
- bei Funktionen, die **static**-Variable enthalten

Um an Stelle des Aufrufs den Funktionskörper zu plazieren, muss der Compiler die Definition der Funktion kennen. Konsequenz: Der Prototyp alleine genügt nicht. Vor dem Funktionsaufruf muss die Funktionsdefinition notiert werden.

## Beispiel

```
//Einsatz eines Makros
#define quadrat(x) (x*x)      //Makrodefinition
int b = 10,a;

//Wie groß ist a und b nach dem Makroaufruf?
int main(void)
{a = quadrat(b++);          //Makroaufruf
  ...
}
```

```
//Einsatz einer Funktion
inline int quadrat(int x)
{return (x*x);}

//Wie groß ist a und b nach dem Funktionsaufruf?
int main(void)
{int b = 10,a;
  a = quadrat(b++);        //Funktionsaufruf
  ...
}
```

## Teste dein Wissen: Aufgabenstellung

1.) Gegeben ist folgender Programmausschnitt:

```
#include <iostream>
using namespace std;
#define square(x) x*x
int main(void)
{int a=3,b=2;
  int y = square(a-b++);
  cout << a << endl;
  cout << b << endl;
  cout << y << endl;
}
```

Welche Werte werden ausgegeben?

## Teste dein Wissen: Lösung

Folgende Werte werden ausgegeben:

## Schlüsselwort const (1)

Eine Alternative für die Konstantendeklaration mit **define** ist der Einsatz des Schlüsselworts **const**.

Mit **const** kann zunächst eine Variable eingeführt werden, die über die Programmdauer hinweg nicht verändert werden kann. Weitergehend wird **const** u.a. auch mit Zeigern zusammen eingesetzt, bekommt also über eine Konstante hinaus eine Bedeutung für die Programmierung.

Der Einsatz von **const** sollte intensiviert werden, da er eine stabilisierende Wirkung auf ein Programm hat.

## Schlüsselwort const (2)

Mit **const int x = 10;** wird eine **const**-Variable eingeführt, die ihren Wert nicht mehr ändern kann. Eine **const**-Variable muss initialisiert werden, da eine Wertzuweisung nicht möglich ist.

Eine **const**-Variable kann auch bei einer Felddefinition verwendet werden:

```
const int bufsize = 100;  
char buf[bufsize];
```

Bei einer **const**-Variablen nimmt der Compiler eine Code-Optimierung vor. Es existieren zwei Möglichkeiten je nach Kontext (siehe Beispiele):

- 1.) Die **const**-Variable belegt Speicherplatz wie eine "normale" Variable.
- 2.) Die **const**-Variable wird als Konstante geführt.

## Schlüsselwort const (3)

**const** im Zusammenhang mit Zeigern ist der wichtigste Einsatz von **const**.  
Es kommen 3 Fälle vor:

1.) **const char\* chptr**

Mit **chptr** kann das Objekt, auf welches **chptr** zeigt bzw. zeigen wird, nicht verändert werden.

2.) **char\* const chptr**

**chptr** kann keinen neuen Wert bekommen, d.h. **chptr** kann nicht auf ein anderes Objekt zeigen.

3.) **const char\* const chptr**

Kombination von 1.) und 2.)

## Schlüsselwort const (4)

Werden an Funktionen speicherintensive Parameter übergeben, wird die Zeigertechnik (call-by-reference) eingesetzt:

```
void myFunc(const char* chptr){...}
```

Um in diesem Fall zu verhindern, dass in der Funktion die Originalobjekte über den Zeiger verändert werden können, wird der Zeiger als **const**-Zeiger deklariert. Damit wird erreicht, dass der Funktion ein **const**-Zeiger aber auch ein nicht-**const**-Zeiger übergeben werden kann. Man gewinnt Flexibilität.

## Beispiel 1

```
//Konstante in C++
#include <iostream>
using namespace std;
int main(void)
{const int WERT;           //Compiler?
  const char C1 = 'A';     //Speicherplatz ja/nein?
  const char C2 = getchar();//Speicherplatz ja/nein?
  int y;
  .....
  const int Z = y;        //Speicherplatz ja/nein?
  const int I = 100;
  const int J = I+10;     //Speicherplatz ja/nein?
  const int SIZE = 5;     //Speicherplatz ja/nein?
  char ca[SIZE];
  cout << "Größe von ca: " << sizeof ca;
}
```

## Beispiel 2

```
char buf1[2] = {'A','B'};  
char buf2[2] = {'C','D'};  
const char* chptr = buf1;  
char* chptr2;  
*chptr = 'a';           //Compiler?  
chptr = buf2;          //Compiler?  
*chptr = 'c';          //Compiler?  
chptr2 = chptr;        //Compiler?
```

```
char* const chptr = buf1;  
*chptr = 'a';           //Compiler?  
chptr = buf2;          //Compiler?
```

```
const char* const chptr = buf1;  
*chptr = 'a';           //Compiler?  
chptr = buf2;          //Compiler?
```

## Beispiel 3

```
void onlyReading(const int i)
{
    i++;           //Compiler?
}
```

```
void onlyReading(const struct Node* nodeptr)
{
    struct Node* writeptr;
    writeptr = nodeptr;           //Compiler?
    writeptr->node_elem1 = 20;
}

int main(void)
{
    ...
    struct Node* ptr = ...;
    onlyReading(ptr);           //Compiler?
    ...
}
```

## Beispiel 4

```
const int myFunc(int x)
{return x;}

int main(void)
{int y = myFunc(5);           //Compiler?
}
```

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

const int* myFunc(void)
{static int buf[2]={1,2};
 return (buf);
}

int main(void)
{cout << *(myFunc() + 1);    //Compiler?
 *myFunc() = 20;           //Compiler?
}
```

## Teste dein Wissen: Aufgabenstellung

```
//Notieren Sie sich die jeweils illegale Anweisung
//mit Begründung (Aufgabe aus /2/)

void t(int* a) {}

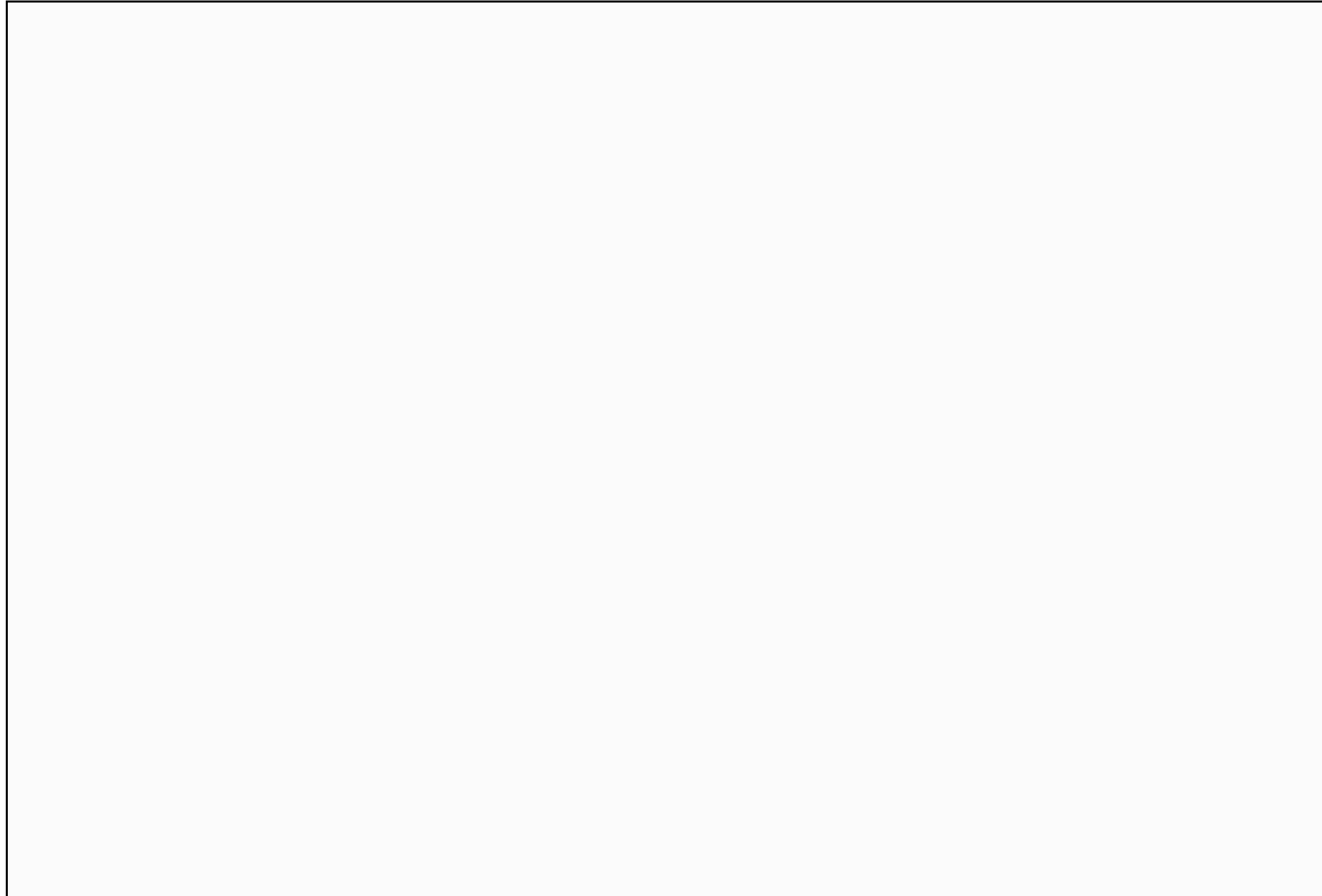
void u(const int* cip)
{
    *cip = 2;
    int i = *cip;
    int* ip2 = cip;
}

const char* v()
{
    return "result of function v()";
}
```

## Teste dein Wissen: Aufgabenstellung (2)

```
const int* const w()  
{static int i;  
  return &i;  
}  
  
int main(void)  
{int x = 0;  
  int* ip = &x;  
  const int* cip = &x;  
  t(ip);  
  t(cip);  
  u(ip);  
  u(cip);  
  char* cp = v();  
  const char* ccp = v();  
  int* ip2 = w();  
  const int* const ccip = w();  
  const int* cip2 = w();  
  *w() = 1;  
}
```

## Teste dein Wissen: Lösung



## Überladen von Funktionen

In C++ ist es möglich, an verschiedene Funktionen den gleichen Funktionsnamen zu vergeben. Dies ist sinnvoll, wenn diese verschiedenen Funktionen verwandte Aufgaben durchführen, wie z.B. zwei Funktionen, die eine Fläche berechnen, einmal für einen Kreis und einmal für ein Rechteck. In beiden Fällen können wir die Funktion z.B. **flaeche** nennen.

Im ersten Fall wird als Parameter der Radius übergeben: **double flaeche(double radius)**, im zweiten Fall als Parameter die Breite und Höhe: **double flaeche(double breite, double hoehe)**.

Der Compiler zieht für die Funktionsunterscheidung neben dem Namen auch die Anzahl und die Typen der Übergabeparameter heran und kann daher den richtigen Funktionsaufruf absetzen.

Ein Überladen durch den Typ des Rückgabewerts ist nicht möglich, da beim Funktionsaufruf der Compiler nicht immer eine exakte Identifikation durchführen kann, d.h. der Compiler weiß nicht, welche der unterschiedlichen Funktionen gemeint ist.

## Beispiel 1

```
//Funktion zum Drucken von Variablen, überladen
//durch den Typ der Parameter:

    void drucke(int);      //A
    void drucke(double); //B
    void drucke(char);    //C
//Aufruf:
    drucke (20.);          //Aufruf von B
    drucke ('A');          //Aufruf von C
    drucke (150);         //Aufruf von A
```

## Beispiel 2

```
//Überladen durch die Anzahl der Parameter
#include<iostream>
using namespace std;

//Kreisflaeche
double flaeche(double rad)
{const double pi = 3.14;
  return (pi*rad*rad);}

//Rechteckflaeche
double flaeche(double hoehe, double breite)
{return (hoehe*breite);}

int main(void)
{double radius=10.;
  cout <<"K-Flaeche = " << flaeche(radius) << endl;

  double dim1=5.,dim2=15.;
  cout << "R-Flaeche = " << flaeche(dim1,dim2);
}
```

## Beispiel 3

```
//Ueberladen durch die Anzahl der Parameter
#include<string.h>

void strcpy(char* dst,const char* src)
{strcpy(dst,src);
}

void strcpy(char* dst,const char* src,int len)
{strncpy(dst,src,len);
}

char stringA[20],stringB[20];

int main(void)
{strcpy(stringA,"Das");
  strcpy(stringB,"Dies ist ein String",4);
}
```

## Beispiel 4

```
//Überladen durch den Typ der Parameter

#include<iostream>
#include<time.h>
using namespace std;

void display_time(const struct tm* x)
{cout << "1. (struct tm) Zeit: " << asctime(x);}

void display_time(const time_t* y)
{cout << "2. (time_t ) Zeit: " << ctime(y);}

int main(void)
{time_t tim = time(NULL);
  struct tm* ltim = localtime(&tim);
  display_time(ltim);
  display_time(&tim);
}
```

## Teste dein Wissen: Aufgabenstellung (1)

```
//Was meint der Compiler zu folgenden Beispielen

//Beispiel 1
double flaeche(double rad)
{const double pi = 3.14;
  return (pi*rad*rad);
}

double flaeche(double hoehe,double breite = 2.5)
{return (hoehe*breite);
}

int main(void)
{flaeche(5.5);
}
```

## Teste dein Wissen: Aufgabenstellung (2)

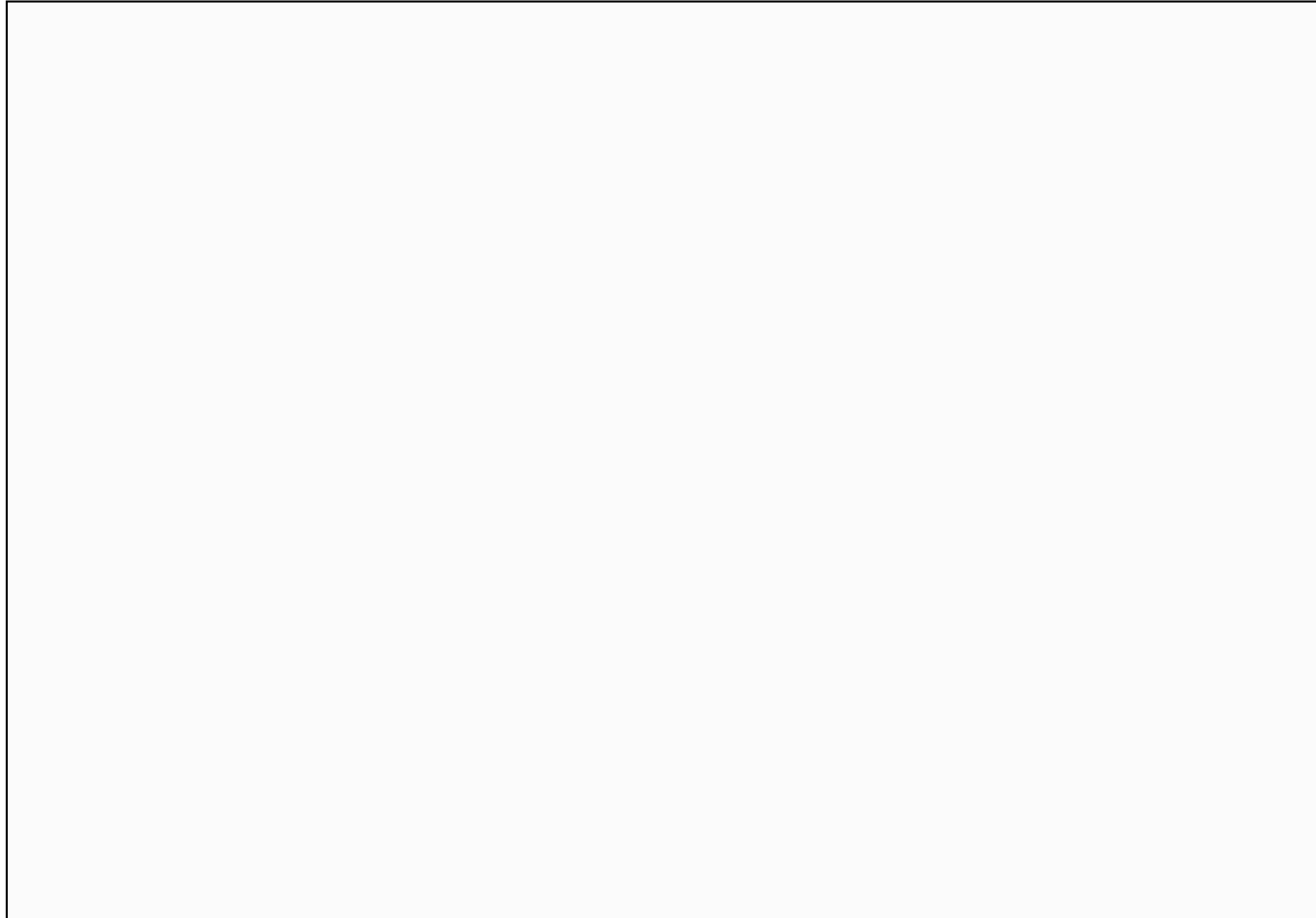
```
//Beispiel 2
char* myFunc(int x);
int* myFunc(int x);

int main(void)
{int a = 5;
  myFunc(a);
}

//Beispiel 3
#include <iostream>
using namespace std;
float yourFunc(float i);
double yourFunc(double i);

int main(void)
{cout << yourFunc(10.1) << endl;
  cout << yourFunc(10);
}
```

## Teste dein Wissen: Lösung



## Referenzen (1)

In C gibt es zwei Möglichkeiten Variable einer Funktion zu übergeben:

1.) Wird die Variable direkt als Parameter notiert, steht sie in der Funktion als Kopie zur Verfügung (**call-by-value**). Die Originalvariable im aufrufenden Programm kann von der Funktion aus nicht verändert werden; durch das Kopieren werden die Ressourcen Laufzeit und Speicherplatz strapaziert (Stichwort: Übergabe großer Strukturen).

2.) Es wird ein Zeiger übergeben, der auf die Variable zeigt. Die Kopie des Zeigers steht in der Funktion zur Verfügung und zeigt natürlich auch auf die Originalvariable im aufrufenden Programm. Der kopierte Zeiger benötigt unabhängig von der Variable einen (immer gleich) geringen Speicherplatz und das Kopieren benötigt folgedessen wenig Zeit. Mit dem kopierten Zeiger kann die Originalvariable von der Funktion aus verändert werden (Stichwort: Dereferenzierung) .

## Referenzen (2)

In C++ wird durch die Referenz eine dritte Möglichkeiten eingeführt (call-by-reference), die an Stelle der Zeigermethode verwendet werden kann. Durch die Referenzenmethode wird die Einführung eines Zeigers überflüssig und es wird die etwas umständliche Zeigernotation vermieden. Beispiel:

```
void myFunc(int& x){x++;}  
int a = 10;  
myFunc(a);      //sieht aus wie call-by-value oder?
```

Beim Aufruf **myFunc(a)** übergibt der Compiler automatisch die Adresse von **a** in die Funktion hinein und die Variable **x** stellt in der Funktion die Adresse von **a** dar.

**Man sagt, x ist eine Referenz (ein Verweis) auf a.**

Dies wird durch den **&**-Operator (Referenz-Operator) in der Funktionsdefinition festgelegt. Durch **x++** wird das Objekt, auf welches **x** verweist (referenziert), um Eins erhöht.

## Beispiel 1

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

void myFunc(int& iref);

int main(void)
{int val = 1;
  cout << "Alter Wert fuer val: " << val << '\n';
  myFunc(val);
  cout << "Neuer Wert für val: " << val << '\n';
}

void myFunc(int& iref)
{iref = 10;}
```

## Beispiel 2

```
//Verwendung einer konstanten Referenz
#include<iostream>
using namespace std;
...
int sqr_it(const int& x);

int main(void)
{int t = 10;
  cout << sqr_it(t);
}

int sqr_it(const int& x)
{return x*x;}
```

## Beispiel 3

```
int myNum = 0;

int& Num(void)
{return myNum;
}

//In diesem Beispiel ist der Rueckgabewert von Num
//eine Referenz auf eine Variable, daher kann der
//Funktionsaufruf auf der linken Seite einer
//Zuweisung stehen.

int main(void)
{int i;
 i = Num();      //Von wem erhält i den Wert?
 Num() = 5;     //Wer erhält den Wert 5?
 return i;
}
```

## Teste dein Wissen: Aufgabenstellung

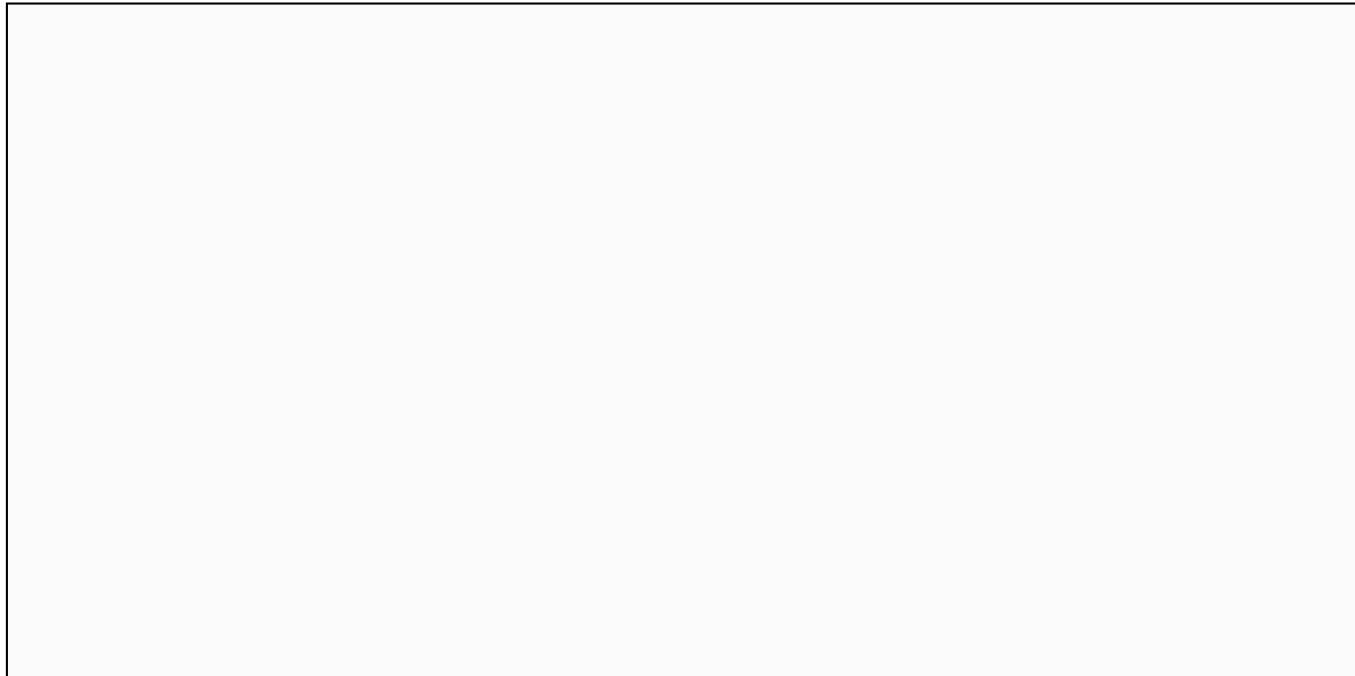
Entwickeln Sie ein Hauptprogramm (Funktion **main**) und die Funktion **getElement**. Die Funktion **getElement** erhält als Parameter den Zeiger **double\* vecptr**, der auf das Feld **arr** zeigt, ausserdem den Feldindex **k** (mit **k = 1,2,3,4**).

Das Feld **arr** mit 4 Elementen ist innerhalb von **main** definiert.

Der Funktionsrumpf von **getElement** besteht einzig aus der Anweisung **return vecptr[k+?]**. Innerhalb von **main** sollen durch eine Zuweisung die Feldelemente auf den Wert **1.0/Feldindex** gesetzt werden.

(Ersetzen Sie auch das Fragezeichen durch einen passenden Wert.)

## Teste dein Wissen: Lösung



## Freispeicherverwaltung

In C stehen für die dynamische Speicherverwaltung u.a. die Funktionen **malloc()** und **free()** zur Verfügung. In C++ werden dafür zwei neue Operatoren bereitgestellt: **new** und **delete**. Vorteile sind, dass **new** einen typisierten Zeiger auf den reservierten Speicherbereich zurückliefert und dass **new** die Größe des Speicherbereichs selbst bestimmt.

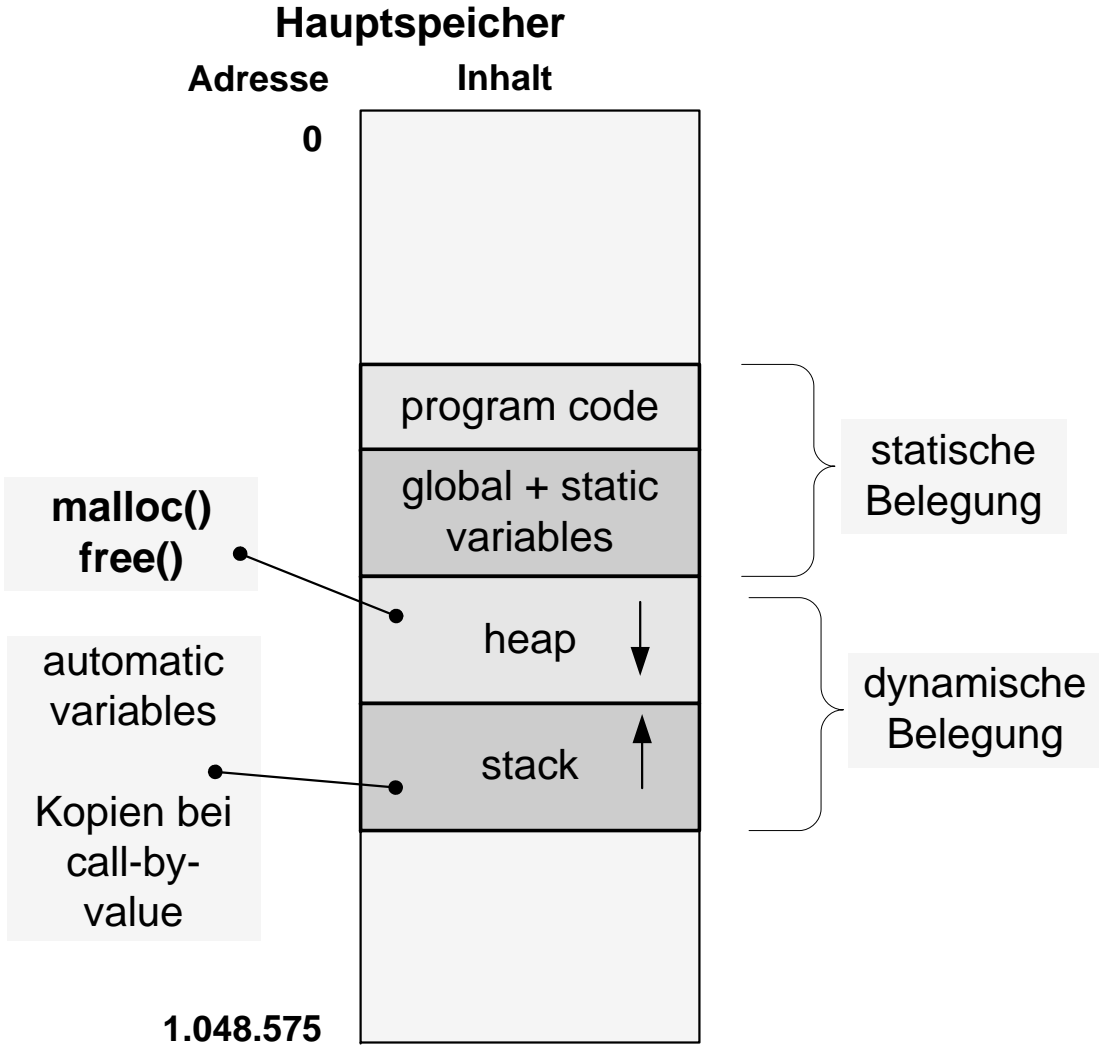
So erhält der Zeiger **ptm** die Adresse einer Variablen vom Typ **struct tm** durch **struct tm\* ptm = new struct tm;**

Mit der Anweisung **delete ptm;** wird der Speicherplatz im Freispeicher wieder freigegeben.

Für Felder wird durch folgende Syntax Speicherplatz reserviert:

**int\* pint = new int [20].** Damit wird Speicherplatz für 20 **int**-Variablen reserviert, deren Anfangsadresse dem Zeiger **pint** zugewiesen wird. Die Freigabe erfolgt durch **delete [] pint;**

# Speicherbelegung eines Programms zur Laufzeit



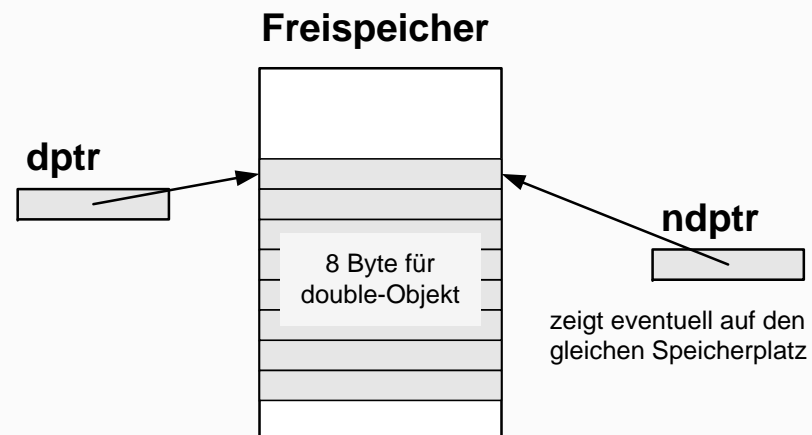
## Beispiel 1

```
//Dynamische Allokierung eines Feldes
#include<iostream>
#include<string>
using namespace std;

int main(void)
{char* sptr = "C++ from the GROUND UP";
  int len = strlen(sptr);
  char* cptr;
  cptr = new char[len + 1]
  strcpy(cptr,sptr);
  cout << sptr << endl;
  cout << cptr;
  delete cptr;
  cptr = NULL;
}
```

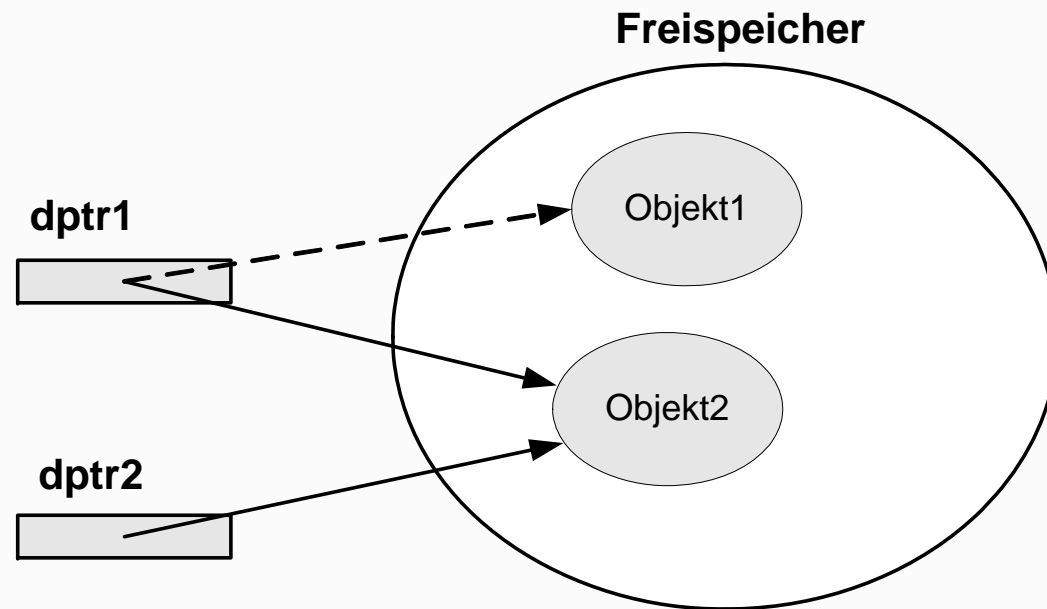
## Beispiel 2

```
double* dptr = new double(10.);  
...  
delete dptr;  
...  
double* ndptr = new double(20.);  
...  
*dptr = 30.;//Achtung: dptr darf nach delete nicht  
             //mehr verwendet werden,  
             //daher dptr = NULL;
```



### Beispiel 3

```
double* dptr1 = new double(10.);  
double* dptr2 = new double(20.);  
dptr1 = dptr2;  
...  
...  
delete dptr1;  
dptr1 = NULL;
```



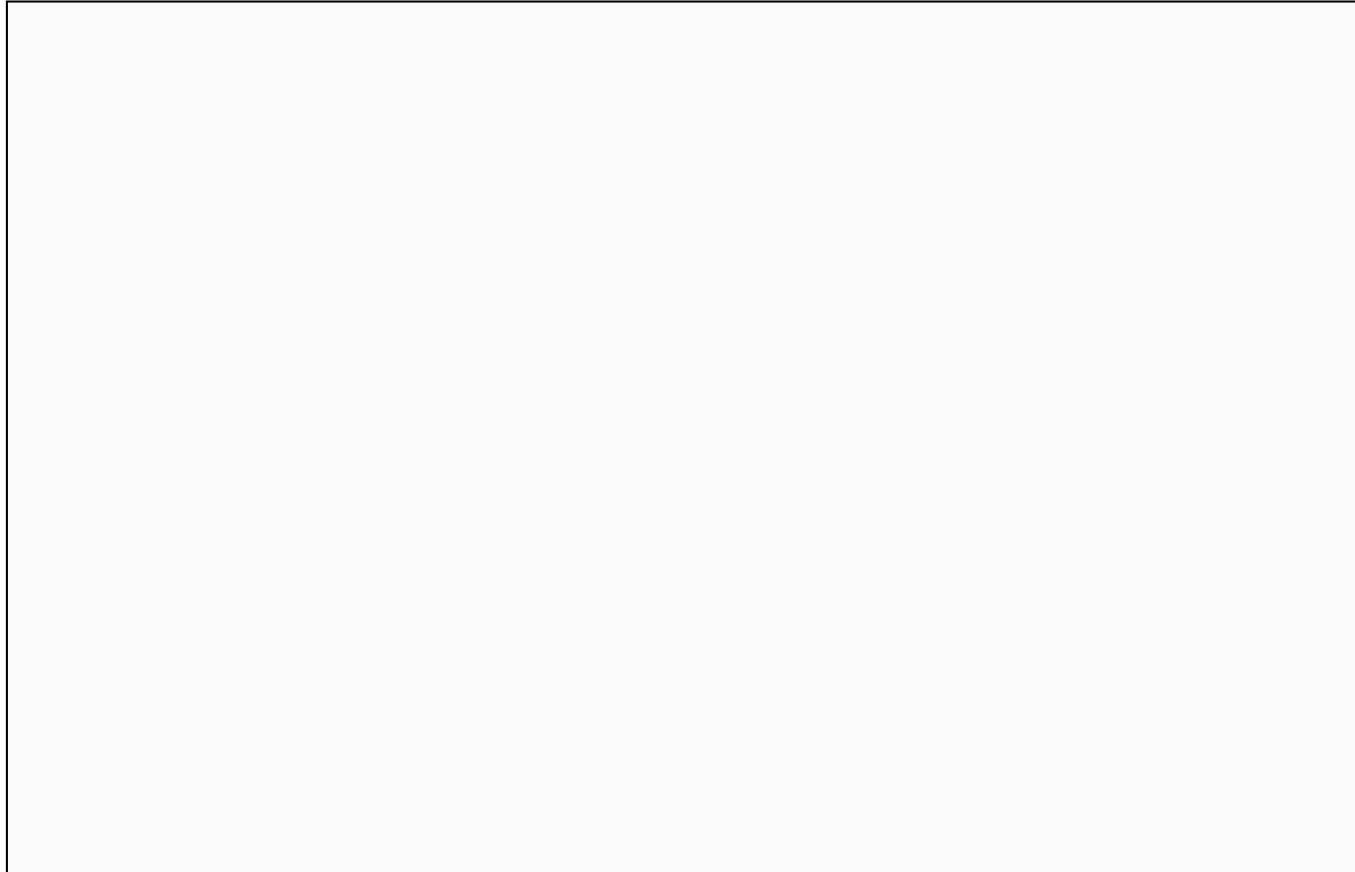
## Teste dein Wissen: Aufgabenstellung

Entwickeln Sie einen Programmausschnitt mit dem ein zwei-dimensionales Feld im Freispeicher erzeugt wird.

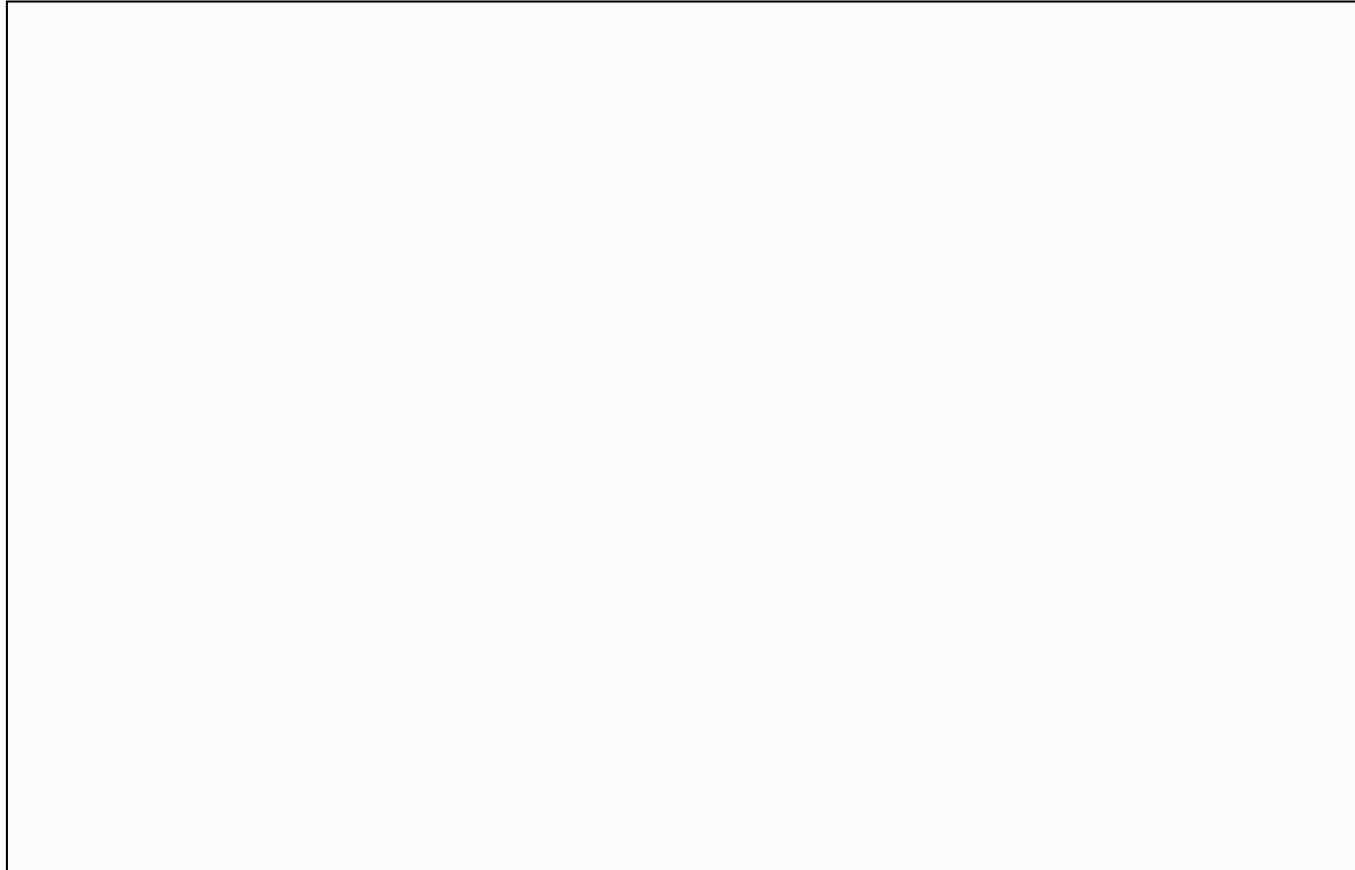
Geben Sie anschließend den Speicherplatz wieder frei.

Zeichnen Sie zu Ihrem Programm eine Skizze, die den Aufbau des zweidimensionalen Feldes sowie eventuell beteiligte Zeiger zeigt.

## Teste dein Wissen: Lösung (1)



## Teste dein Wissen: Lösung (2)



## Klassen (1)

In C bestehen Programme aus Daten und Funktionen. Die Funktionen selbst stellen abgeschlossene Teillösungen dar, die ihren Datenaustausch über definierte Schnittstellen realisieren. Auf die lokalen Variablen einer Funktionen können dabei keine fremden Funktionen zugreifen.

In C++ wurde dieser Kapselungsgedanke weiterentwickelt. Die aus C bekannten Strukturen wurden ergänzt durch Funktionen. Die Elemente der Struktur und die Funktionen bilden eine Einheit. Diese Einheit nennt man **Klasse**. Die Elemente nennt man jetzt **Attribute**, die Funktionen **Elementfunktionen** oder **Methoden**.

## Klassen (2)

Die Attribute gibt es in zwei Ausführungen:

1. Attribute, die von klassenfremden Funktionen gelesen und verändert werden können. Diese "freien" Attribute nennt man **public**-Attribute. Sie sind in der Klassendeklaration durch das Kennzeichen **public** markiert.
2. Attribute, die nur von klasseneigenen Funktionen gelesen und verändert werden können. Diese gekapselten Attribute nennt man **private**-Attribute. Sie sind in der Klassendeklaration durch das Kennzeichen **private** markiert. Diese **private**-Attribute sind von "ausen" nur durch Methoden nutzbar. Dies bedeutet, dass der Klassenprogrammierer bestimmen kann, was mit seinen **private**-Attributen geschehen soll. Gibt es z.B. nur eine Methode, die ein Attribut als Rückgabewert zurückliefert, dann ist dieses Attribut von ausen eben nur lesbar, aber nicht veränderbar. Diese Methoden bilden ein **Klasseninterface** zwischen einem Objekt und der "Aussenwelt".

## Klassen (3)

Neben der Datenkapselung steckt in dem Klassenbegriff ein weiterer Gedanke. Durch eine Klasse können reale Objekte aus der Umwelt in der Software nachgebildet werden. Man macht sich ein datentechnisches Modell von Teilen der realen Welt. Dabei sind die Attribute die Eigenschaften und die Methoden die Fähigkeiten.

Bildet man in einer Klasse ein Auto nach, so gibt es Attribute wie Farbe, Länge, Breite, Getriebausführung. Die Fähigkeiten eines Autos wie Fahren, Tanken, Bremsen werden in Methoden nachgebildet.

Klassen sind wie Strukturen Datentypen. Bildet man aus Klassen Variable, so nennt man diese Variable **Objekte** oder **Instanzen**. Beim Einsatz von Klassen spricht man deshalb von der **objektorientierten Programmierung (OOP)**. Dabei stellt die Klassenbildung wie bisher beschrieben nur einen ersten Schritt in die OOP dar. Weiteres folgt.

## Beispiel 1

```
class Date
{private:
  int month;
  int day;
  int year;

  public:
  int hour;
  int minute;
  int second;
};
```

```
int main(void)
{Date heute;
  int monat;
  int stunde;

  heute.month = 10;      //o.k.?
  monat = heute.month;  //o.k.?
  heute.hour = 12;      //o.k.?
  stunde = heute.hour;  //o.k.?
}
```

## Beispiel 2

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

class Date
{private:
    int month,day,year;
public:
    void display();};

void Date::display()
{static char* name[] = {"zero","Januar","Februar",
    "März","April","Mai","Juni","Juli","August",
    "September","Oktober","November","Dezember"};
    cout << name[month] << ' ' << day << ' ' << year;
}

int main(void)
{Date mydate;
    mydate.display(); //Was wird ausgegeben?
}
```

## Konstruktoren

Variable, denen keine Anfangswerte zugewiesen wurden, stellen eine große Fehlerquelle dar. (Berücksichtigt sind nicht-initialisierte Zeiger.) C++ stellt im Zusammenhang mit Klassen einen Automatismus zur Verfügung, mit dem neue Objekte direkt bei ihrer Erzeugung initialisiert werden:

Der Klassenentwickler schreibt eine Methode, die bei der Objekterzeugung automatisch aufgerufen wird. In dieser Methode werden alle Aktionen zur Initialisierung durchgeführt.

Diese Methode nennt man **Konstruktor** mit folgenden Eigenschaften:

- gleicher Namen wie die Klasse
- besitzt keinen Rückgabewert, daher kein Datentyp erforderlich
- Konstruktor ohne formale Parameter = **Standardkonstruktor**
- Überladung möglich, daher kann eine Klasse mehrere Konstruktoren besitzen
- aktuelle Parameter werden hinter dem Objektamen in Klammern übergeben

## Beispiel 1

```
class Date
{private:
    int month,day,year;
public:
    void display();
    Date(int a,int b,int c)
        {month = a; day = b; year = c;}
};

void Date::display()
{static char* name[] ={"zero","Januar","Februar",
    "März","April","Mai","Juni","Juli","August",
    "September","Oktober","November","Dezember"};
    cout << name[month] << ' ' << day << ' ' << year;
}

int main(void)
{Date mydate(10,22,1999);
    Date yourdate(2,15,2000);
    mydate.display();
    yourdate.display();
}
```

## Beispiel 2

```
class Date
{public:
    Date();
    Date(int mn,int dy,int yr);
    ...
};

Date::Date()
{month = day = year = 1;}

Date::Date(int mn,int dy,int yr)
{setMonth(mn); setDay(dy); setYear(yr);
}

int main(void)
{Date yourDate(12,25,1990);
  Date myDate;
}
```

## Beispiel 3 (1)

```
//Klasseninterface
class Date
{private:
    int month,day,year;
public:
    Date(int mn,int dy,int yr);
    int getMonth();
    int getDay();
    int getYear();
    void setMonth(int mn);
    void setDay(int dy);
    void setYear(int yr);
    void display();
};
```

Date
-month: int -day: int -year: int
+getmonth():int +getday():int +getyear():int +setmonth(mn:int) +setday(dy:int) +setyear(yr:int)

## Beispiel 3 (2)

```
inline int Date::getMonth(){return month;}
inline int Date::getDay(){return day;}
inline int Date::getYear(){return year;}

void Date::setMonth(int mn)
{month = max(1,mn); month = min(month,12);}

void Date::setDay(int dy)
{static int
leng[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
 day = max(1,dy); day = min(day,leng[month]);}

void Date::setYear(int yr){year = max(1,year);}

Date::Date(int mn,int dy,int yr);
{static int
leng[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
 month = max(1,mn); month = min(month,12);
 day = max(1,dy); day= min(dy,leng[month]);
 year= max(1,year);}
```

### Beispiel 3 (3)

```
//Wie gross sind die Attribute von deadline?  
int main(void)  
{int i;  
  Date deadline(3,10,1980);  
  i=deadline.getMonth();  
  deadline.setMonth(4);  
  deadline.setMonth(deadline.getMonth()+1);  
}
```

## Beispiel 4

```
//Referenzen als Funktionsergebnisse
class Date
{private:
    int monthMember;int dayMember;int yearMember;
public:
    Date(int mn,int dy,int yr);
    int& month(void);
};

int& Date::month(void)
{monthMember=max(1,monthMember);
 monthMember=min(monthMember,12);
 return monthMember;}

int main(void)
{int i;
 Date deadline(3,10,1980);
 i= deadline.month();
 deadline.month()=34;
 deadline.month()++ ;}
```

## Teste dein Wissen: Aufgabenstellung 1

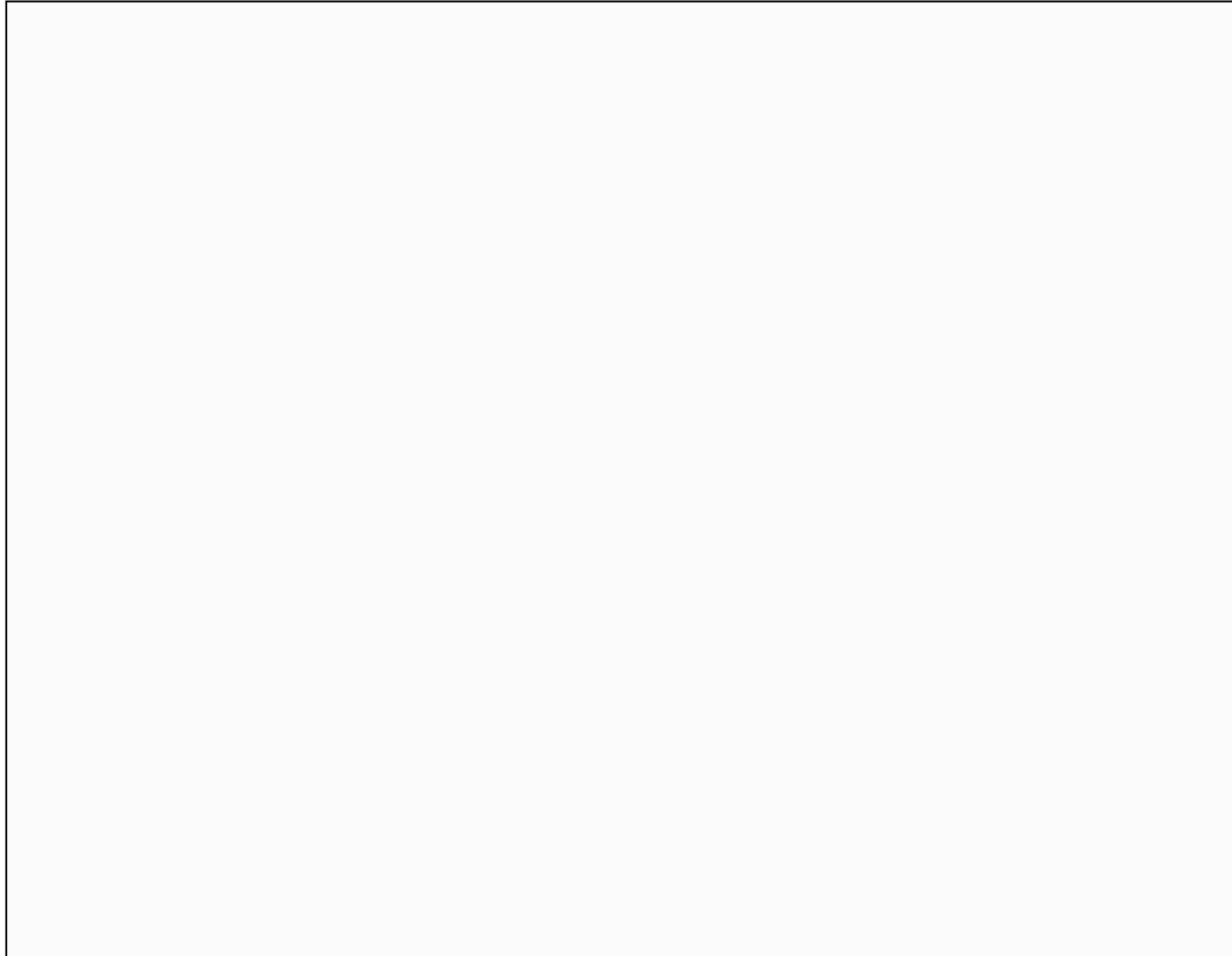
Entwickeln Sie eine Klasse **Gerade**, mit der geometrische Geraden dargestellt werden können. Die Gerade ist durch zwei Punkte definiert. Bei der Erzeugung von Objekten der Klasse Gerade sind drei Fälle möglich:

- Es wird ein Objekt durch  $0/0$  und  $1/1$  erzeugt.
- Es wird ein Objekt durch  $0/0$  und  $x_2/y_2$  erzeugt.
- Es wird ein Objekt durch  $x_1/y_1$  und  $x_2/y_2$  erzeugt.

Weiterhin gibt es eine Methode, die die einzelnen Punkte eines Objekts am Bildschirm ausgibt.

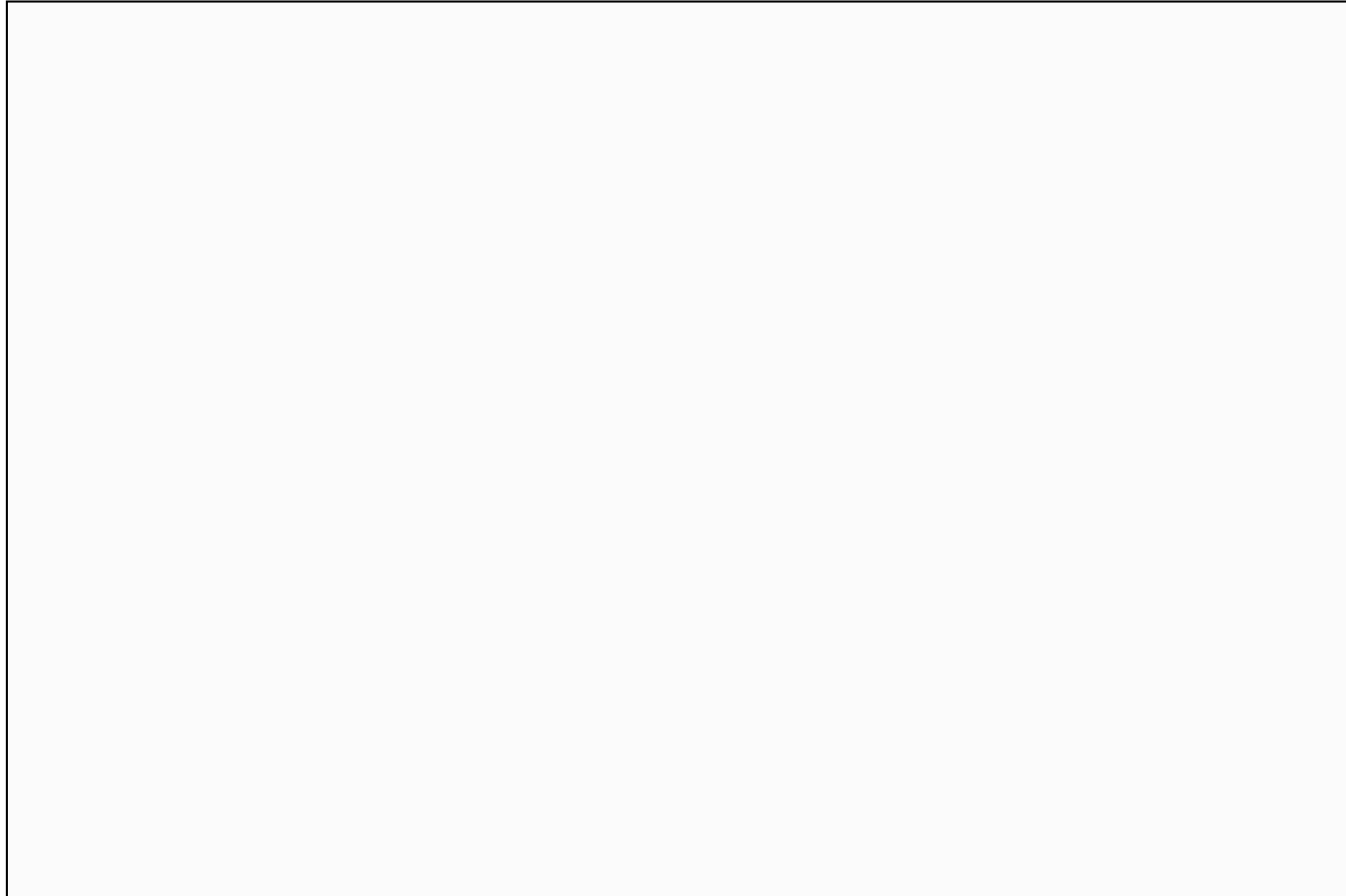
Entwickeln Sie eine Lösung mit Funktionsvoreinstellung und eine ohne.

## Teste dein Wissen: Lösung 1 (1)



---

## Teste dein Wissen: Lösung 1 (2)

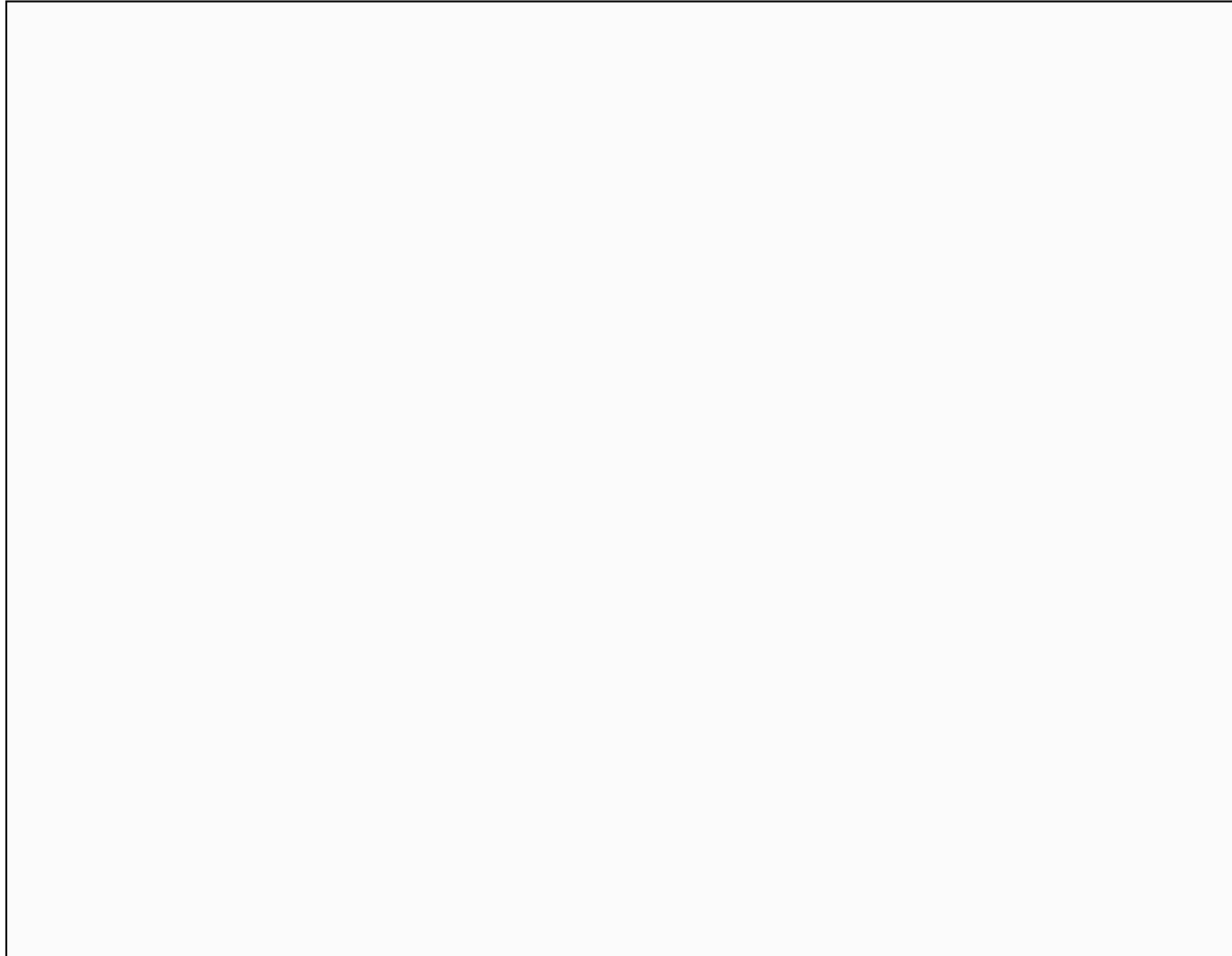


## Teste dein Wissen: Aufgabenstellung 2

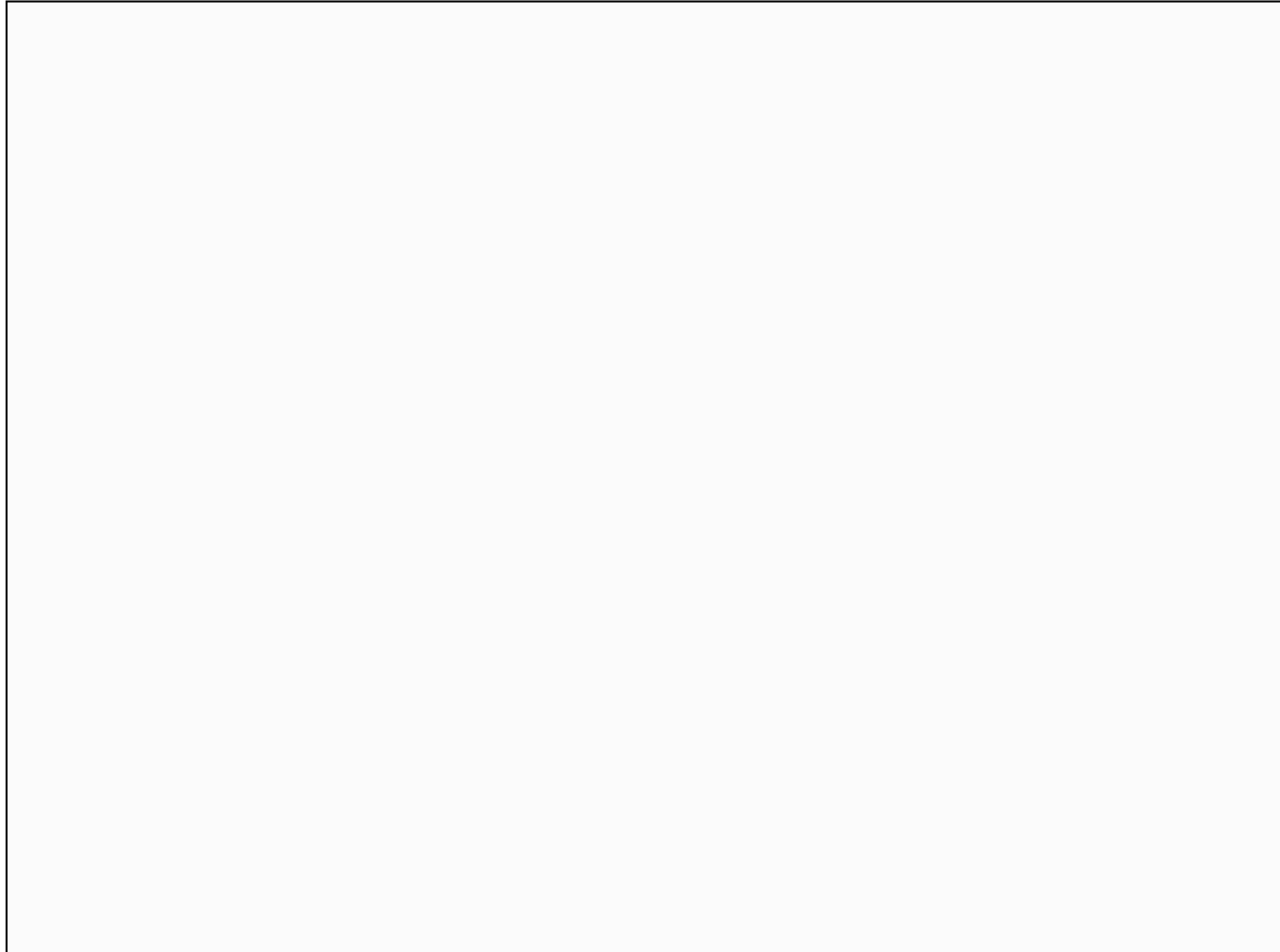
Entwickeln Sie eine Klasse Anrufbeantworter mit folgenden Eigenschaften:

1. speichert die Nachrichten und die Anzahl der Nachrichten
2. speichert einen Ansagetext
3. speichert maximal 10 Nachrichten mit maximal 80 Zeichen
4. bei der Erzeugung eines Anrufbeantworters wird die Nachrichtenanzahl auf 0 gesetzt und die Nachrichten als Nullstring initialisiert
5. alternativ wird zusätzlich ein Ansagetext eingestellt
6. wird ein Anrufbeantworter vernichtet, werden sämtliche vorhandene Nachrichtentexte auf dem Bildschirm ausgegeben
7. Interface zum Aufsprechen einer Nachricht
8. Interface zum Holen der Nachrichten

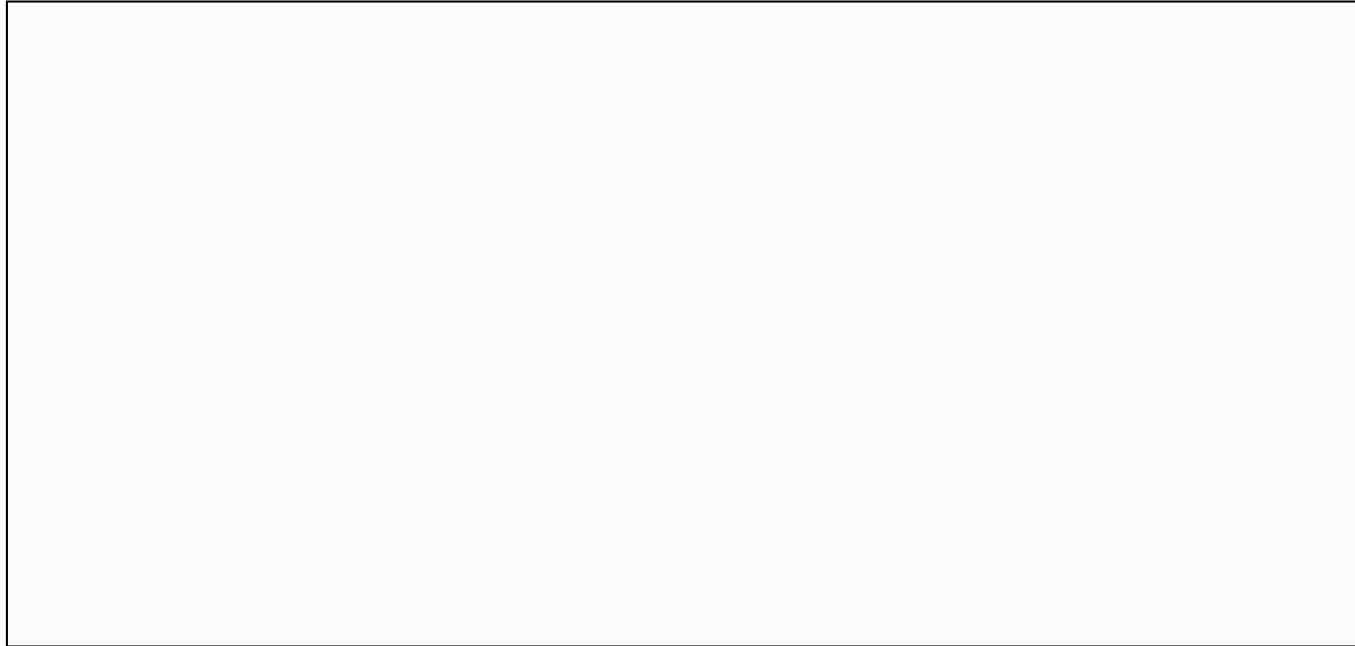
## Teste dein Wissen: Lösung 2 (1)



## Teste dein Wissen: Lösung 2 (2)



## Teste dein Wissen: Lösung 2 (3)



## const-Objekte

Ebenso wie bei einfachen Variablen, die als **const**-Variable notiert werden können, können auch **const**-Objekte formuliert werden:

```
const Date birthday(4,2,1965);
```

Dies bewirkt, dass die Attribute des Objekts **birthday** konstant sind. Der Compiler "traut" nun zunächst keiner Methode und verbietet den Methodenaufruf generell. Wenn nun die Nutzung der Methoden erlaubt sein soll, die keine Attribute ändern, müssen diese Methoden durch **const** gekennzeichnet werden (sogenannte Read-Only-Methoden).

Die Kennzeichnung mit **const** wird nach der Parameterliste der Funktion durchgeführt.

Mit **const** definierte Methoden dürfen also keine Attribute des Objekts verändern, aber auch keine nicht-**const**-Methoden aufrufen.

Vereinbaren Sie wenn möglich Methoden immer als **const**. Sie können dann bei **const**-Objekten die Read-Only-Methoden aufrufen.

Mit **const** werden Sie flexibler.

## Beispiel

```
class Date
{private: int month, day, year;
public:
    Date( int mn, int dy, int yr );
    int getMonth() const;
    int getDay() const;
    int getYear() const;
    void setMonth(int mn);
    void setDay(int dy);
    void setYear(int yr);
    void display() const;};

inline int Date::getMonth() const
{return month;}

int main(void)
{int i;
    const Date birthday(4,2,1965);
    i = birthday.getYear();
    birthday.setYear(1942);}
```

## Objekte als Klasselemente

Bisher bestanden die Klassenattribute aus Variablen mit einfachen Datentypen wie Typ **int**, **char**, **float**, etc. Diese Attribute wurden innerhalb der Konstruktoren durch Zuweisungen initialisiert.

Wenn als Attribute in einer Klasse **A** Objekte einer Klasse **B** verwendet werden, stellt sich die Frage, wie diese eingebetteten Objekte initialisiert werden. Die Lösung ergibt sich durch den Einsatz einer Initialisierungsliste. Dabei wird dem Konstruktor **A** der Klasse **A** über eine Liste die Initialisierungsparameter für das eingebettete Objekt der Klasse **B** mitgegeben. Bei der Erzeugung eines Objekts der Klasse **A** übergibt der Konstruktor **A** die Initialisierungsliste an den Konstruktor **B** weiter. Dieser initialisiert das eingebettete Objekt, anschließend initialisiert der Konstruktor **A** die restlichen Attribute.

Die Initialisierungsliste wird ebenfalls bei **const**- und Referenz-Attributen eingesetzt.

## Beispiel 1

```
//Wie werden const-Attributen initialisiert?  
//durch Zuweisungen?  
  
class Student  
{private:  
    const int alter; const int matnum;  
public:  
    Student(int age,int num)  
        {alter=age; matnum=num;}  
};
```

```
//Nicht durch Zuweisungen, sondern durch eine  
//Initialisierungsliste !!  
  
class Student  
{private:  
    const int alter; const int matnum;  
public:  
    Student(int age,int num):alter(age),matnum(num)  
        {}  
};
```

## Beispiel 2

```
//Initialisierung von Referenz-Attributen durch  
//Zuweisung?
```

```
class Student  
{private:  
  int& alter; int& matnum;  
public:  
  Student(int& age,int& num)  
    {alter=age; matnum=num;}  
};
```

```
//Initialisierung durch eine Initialisierungsliste
```

```
class Student  
{private:  
  int& alter; int& matnum;  
public:  
  Student(int age,int num):alter(age),matnum(num)  
    {}  
};
```

## Beispiel 3

```
//Initialisierungsliste für eingebettete Objekte
class Student
{private:
    char name[30];
    char address[60];
    Date birthday;    // Attribut vom Typ Date

public:
    Student(char*,char*,int,int,int);
};

Student::Student(char *nm,char *adr,
                 int mn,int dy,int yr):birthday(mn,dy,yr)
{strncpy(name,nm,sizeof(name));
  strncpy(address,adr,sizeof(address));
}
```

## Beispiel 4

```
//Möglich, aber ineffizient. Warum ineffizient?  
class Student  
{private:  
    char name[30];  
    char address[60];  
    Date birthday;    // Attribut vom Typ Date  
public:  
    Student(char*,char*,int,int,int);  
};  
  
Student::Student(char* nm,char* adr,  
                 int mn,int dy,int yr)  
{birthday.setMonth(mn);  
    birthday.setDay(dy);  
    birthday.setYear(yr);  
    strncpy(name,nm,sizeof(name));  
    strncpy(address,adr,sizeof(address));  
}
```

## Teste dein Wissen: Aufgabenstellung 1

Gegeben sind folgende Klassendeklarationen:

```
class Arm
{ int z;
  ??? hd;
public:
  Arm (int c){z = c;}
};
```

```
class Hand
{ int y;
public:
  Hand (int b){y = b;}
};
```

Wie an den Namen der Klassen leicht zu erkennen ist, kann zwischen den Klassen eine Beziehung hergestellt werden.

Stellen Sie den unbekannt Typ des Attributs **hd** fest.

Welche Änderungen müssen Sie jetzt an den Konstruktoren vornehmen?

---

## Teste dein Wissen: Lösung 1

```
class Hand
```

```
class Arm
```

## Teste dein Wissen: Aufgabenstellung 2

Deklarieren Sie eine Klasse **Point** mit zwei **int**-Variable **x** und **y** als **private**-Elemente. Der dazugehörige Konstruktor soll bei einer Objekterzeugung die beiden Variablen auf von außen festgelegte Werte setzen.

Deklarieren Sie eine Klasse **Line**, die im **private**-Bereich neben einer **int**-Variable **v** auch zwei Variable **p1** und **p2** vom Typ **Point** enthält.

Formulieren Sie den Konstruktor **Line** der Klasse **Line**, wenn er den Wert von **v** von außen übernehmen soll.

## Teste dein Wissen: Lösung 2

```
class Point
```

```
class Line
```

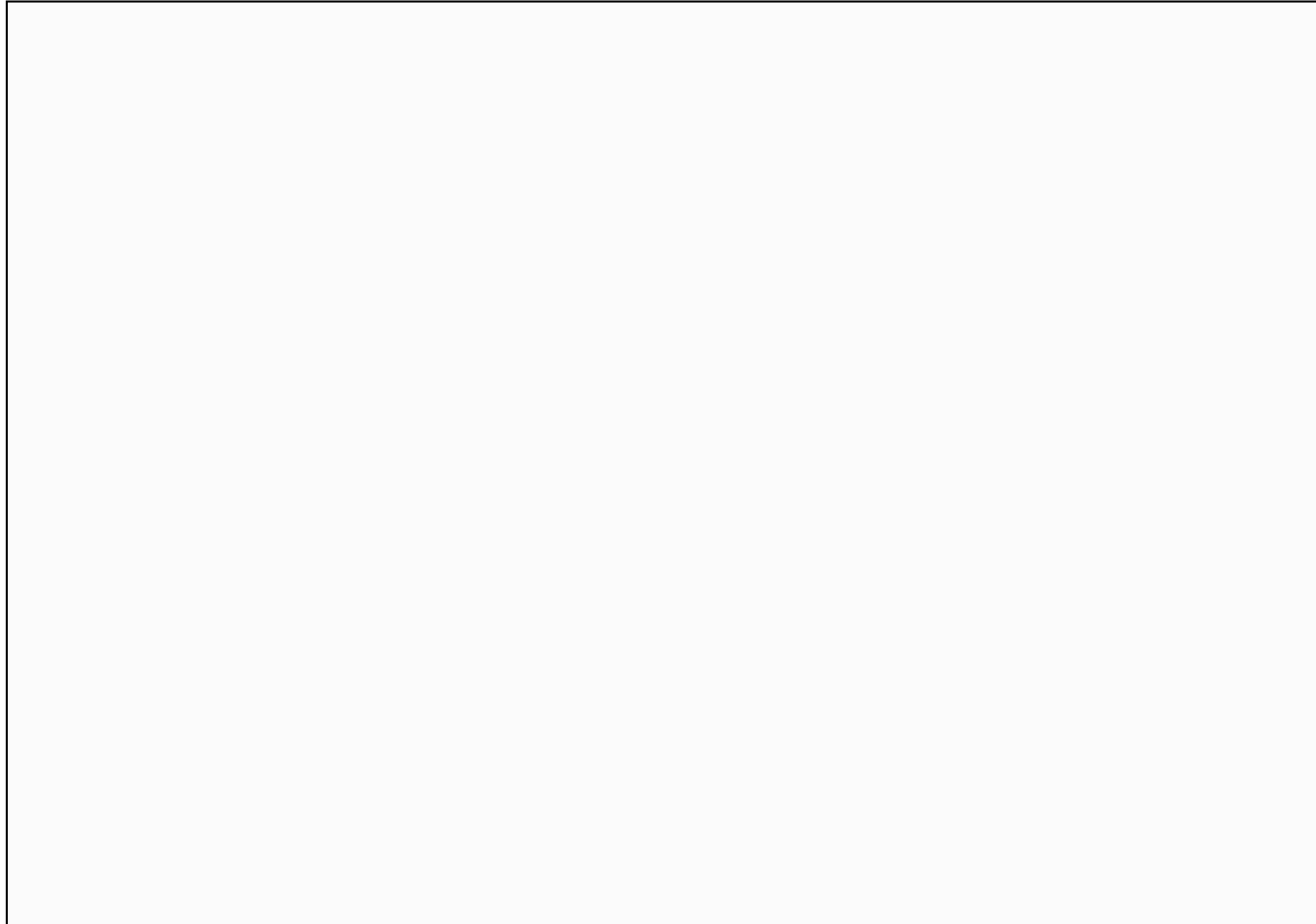
## Teste dein Wissen: Aufgabenstellung 3

Die Reihenfolge der Attribute in der Klassendeklaration und die Reihenfolge der Attribute in der Initialisierungsliste muß nicht übereinstimmen.

Frage: In welcher Reihenfolge werden nun die Attribute initialisiert?  
Gilt die Reihenfolge in der Deklaration oder die Reihenfolge in der Initialisierungsliste?

Schreiben Sie ein Testprogramm zur Klärung dieser Frage.

## Teste dein Wissen: Lösung 3



## Friends

Ein wichtiges Prinzip der OOP ist die Datenkapselung, realisiert durch **private**-Attribute zusammen mit Zugriffsmethoden. Eine gewisse Lockerung stellt die **friend**-Beziehung dar. Dadurch wird es möglich, dass

- globale Funktionen (**friend**-Funktionen),
- Methoden anderer Klassen (**friend**-Methoden) oder
- sämtliche Methoden einer anderen Klasse (**friend**-Klasse)

auf die **private**-Attribute unter Umgehung des Klasseninertices zugreifen können.

**friend**-Beziehungen werden z.B. benötigt, wenn globale Funktionen sehr schnell auf **private**-Attribute von Instanzen zugreifen müssen oder im Zusammenhang mit dem Überladen von Operatoren.

**friend**-Beziehungen sollten sorgsam benutzt werden, da sie dem Prinzip der Datenkapselung widersprechen.

## Beispiel 1

```
//friend-Funktion
#include <iostream>
using namespace std;

class MyClass
{int a, b;
 public:
   MyClass(int i,int j){a = i;b = j;}
   friend int sum(MyClass x);
};

int sum(MyClass x)
{return x.a + x.b;
}

int main(void)
{MyClass n(3,4);
 cout << sum(n);
}
```

## Beispiel 2

```
//friend-Methoden
#include<iostream>
using namespace std;

class Schwein;           //foreward-Deklaration
class Metzger
{public:
    void print_Gewicht(const Schwein& Hugo)
    {cout << Hugo.Gewicht;}};

class Schwein
{friend void Metzger::print_Gewicht(const Schwein&);
 private:
    int Gewicht;
 public:
    Schwein(int x){Gewicht = x;}};

int main(void)
{Schwein Herbert(250);
 Metzger Klaus;
 Klaus.print_Gewicht(Herbert);}
```

## Beispiel 3

```
//friend-Klasse
class YourClass
{ friend class YourOtherClass;
  private:
    int topSecret;};

class YourOtherClass
{public:
  void changeIt(YourClass &yc);};

void YourOtherClass::changeIt(YourClass &yc)
{yc.topSecret++;}
```

```
class HisClass
{ friend class YourClass;
  public:
    void changeIt(YourClass yc);};

void HisClass::changeIt(YourClass yc)
{yc.topSecret++;}
```

## Teste dein Wissen: Aufgabenstellung

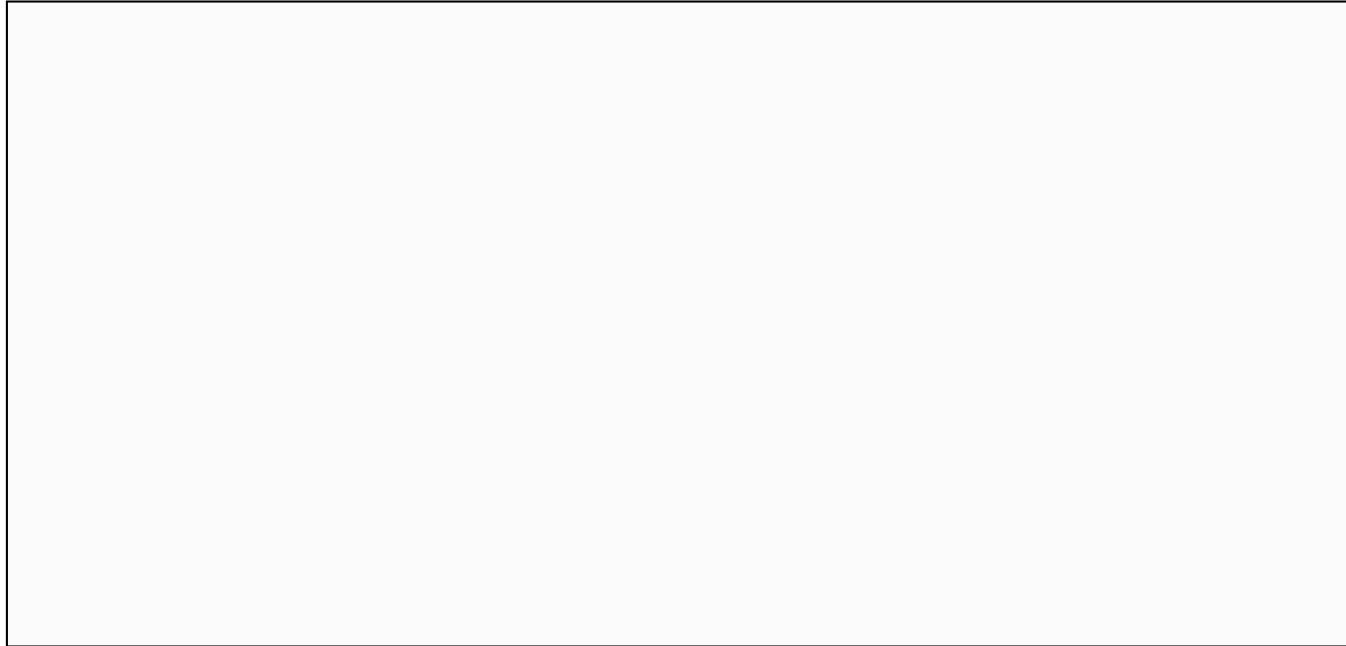
Programmieren Sie eine Klasse **Pixel**, welche die Pixelkoordinaten (Typ **int**) und die Pixelfarbe (Typ **color**) speichern kann. Weiterhin gibt es Methoden zum Lesen und zum Setzen von Koordinaten und Farbe.

Programmieren Sie eine Klasse **Picture**, die aus maximal 1000 Pixels besteht. Weiterhin enthält die Klasse **Picture** eine Methode **mirror**, die das Bild an der 1. Winkelhalbierenden spiegelt.

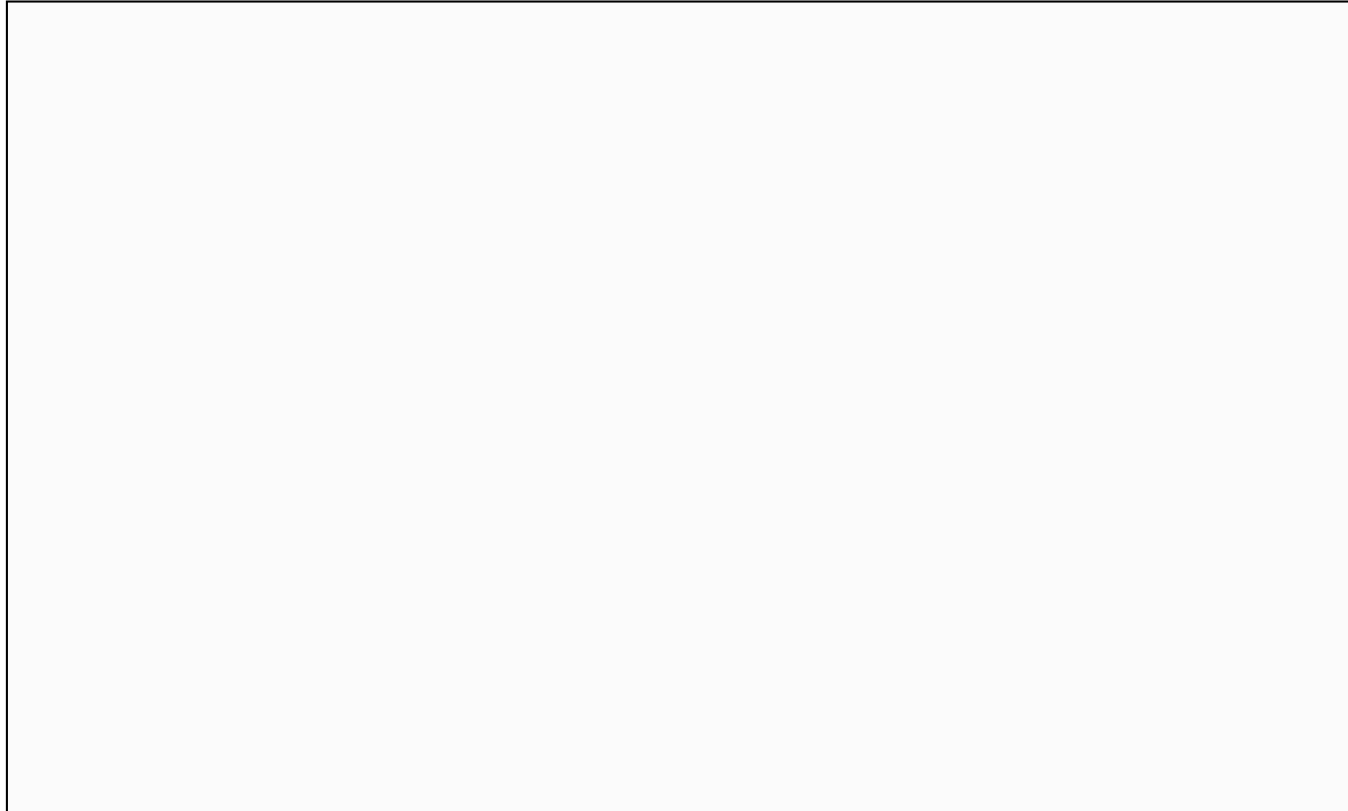
Entwickeln Sie zwei Lösungen für **mirror**:

1. Nutzung des Klasseninterface von **Pixel** (langsames Programm).
2. Ohne Klasseninterface (schnelles Programm).

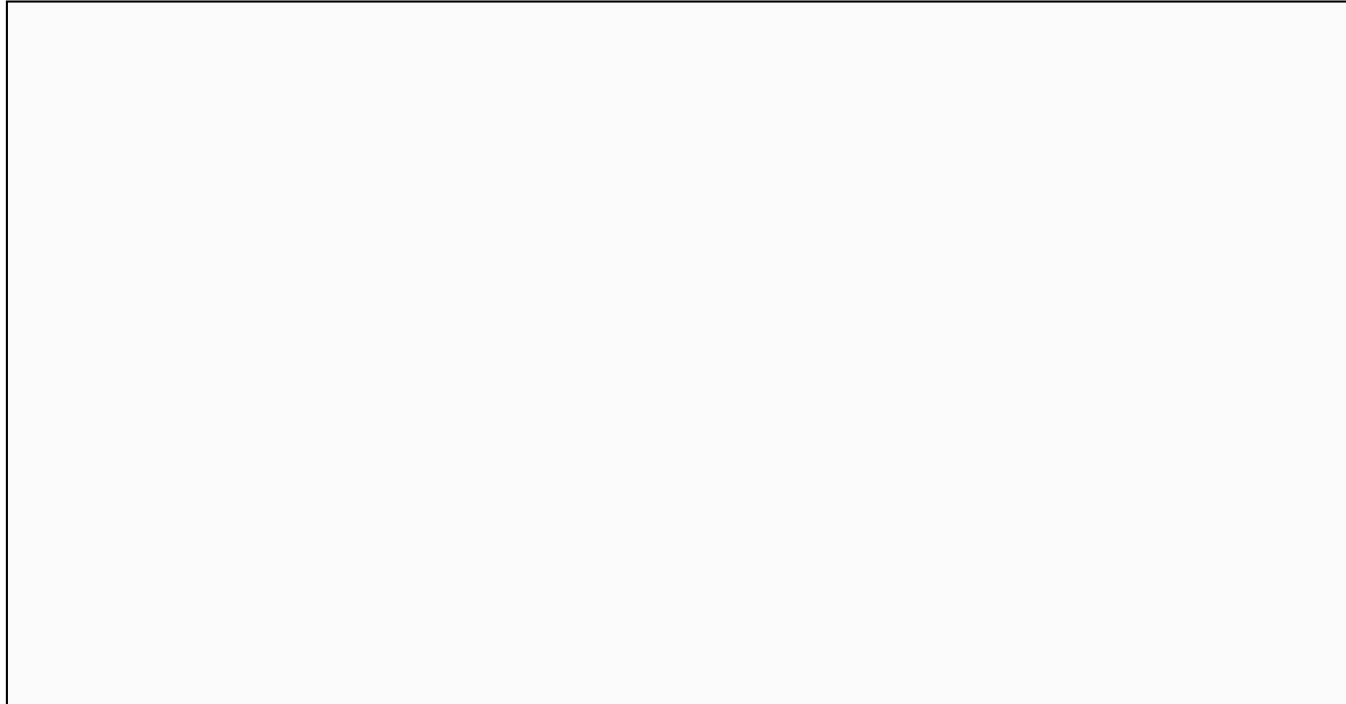
## Teste dein Wissen: Lösung (1)



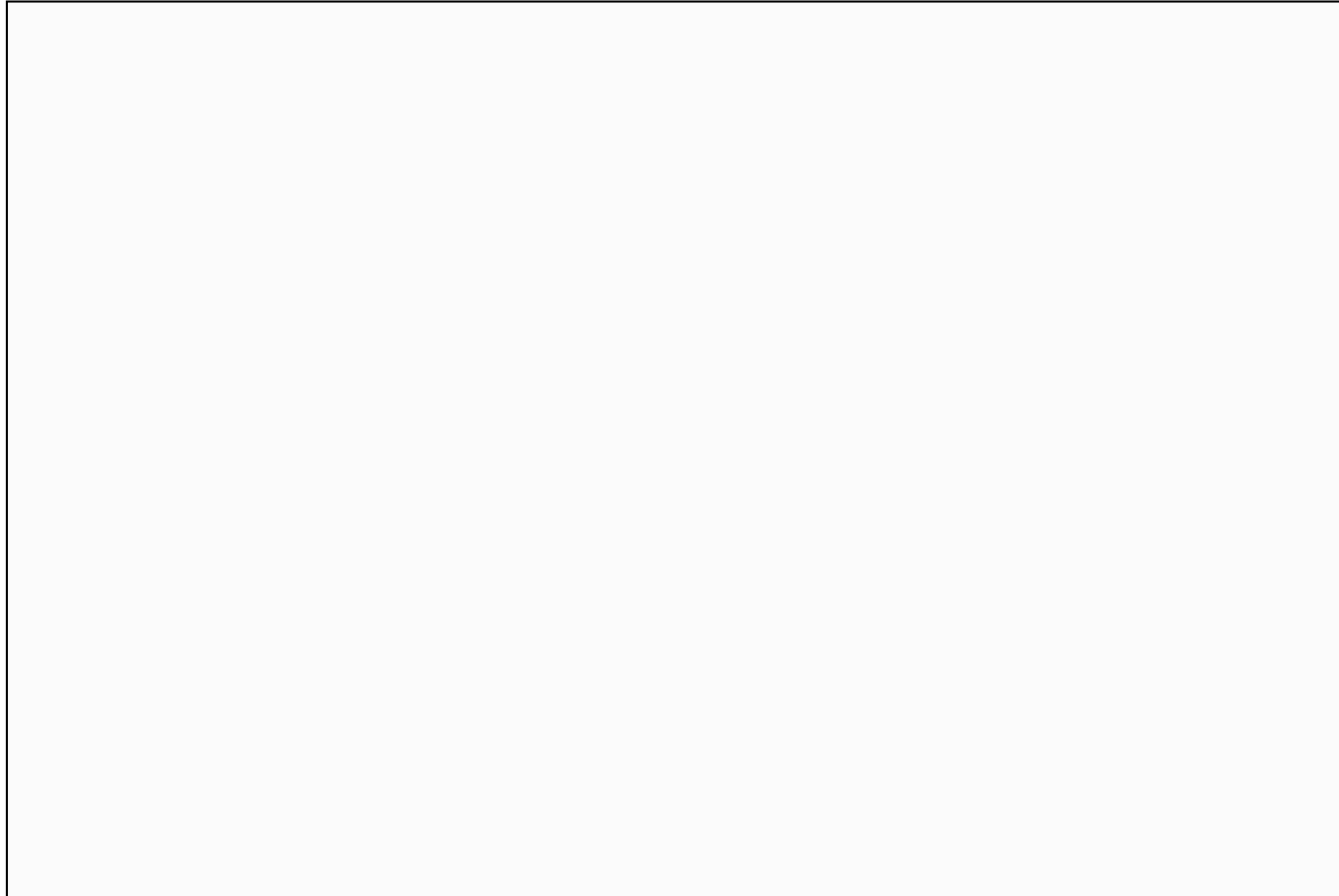
## Teste dein Wissen: Lösung (2)



## Teste dein Wissen: Lösung (3)



## Teste dein Wissen: Lösung (4)

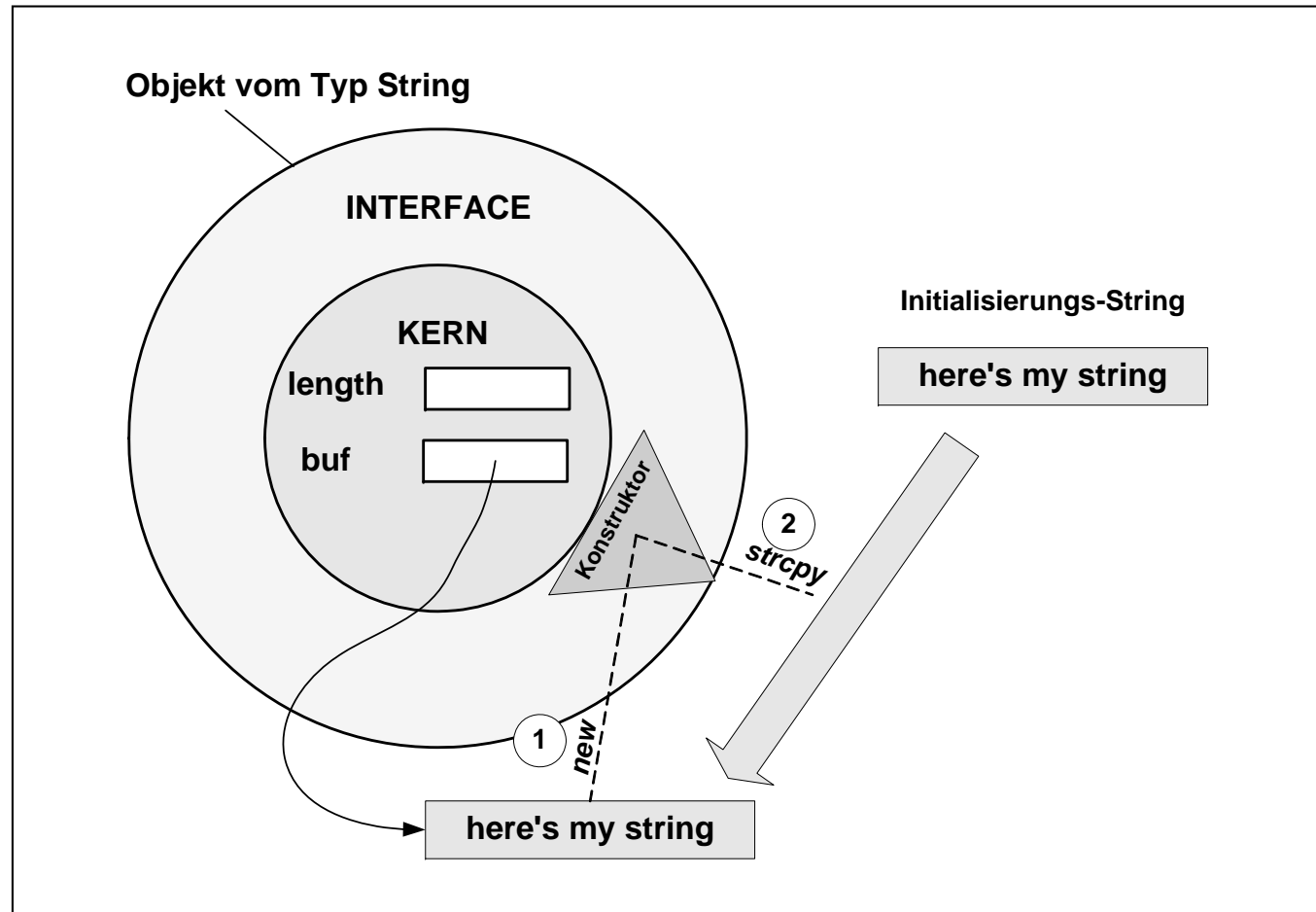


## Klassen mit Zeigerelementen

Ausgangssituation ist eine Klasse mit Zeigerelementen, über die im Freispeicher Speicherplatz allokiert wird. Jede Instanz einer Klasse bekommt bei ihrer Erzeugung ihren eigenen Speicherplatz, d.h. der Speicherplatz wird durch den Konstruktor allokiert. Der Konstruktor wird mit dem Operator **new** Speicherplatz anfordern.

Ist die Instanz eine lokale Variable in einer Funktion, beendet die Instanz ihre Existenz, wenn die Funktion beendet wird. Wer gibt in diesem Zusammenhang den allokierten Speicherplatz frei? Die Lösung ist die Einführung einer Funktion, die automatisch aufgerufen wird, wenn die Instanz ihre Existenz verliert. Diese Funktion nennt sich **Destruktor**. Der Destruktor besitzt keinen Rückgabewert, erhält keine Übergabeparameter und wird mit dem Klassennamen bezeichnet mit vorgestellter Tilde. Innerhalb des Destruktors wird der Speicherplatz im Freispeicher durch den Operator **delete** freigegeben.

## Beispiel (1)



## Beispiel (2)

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

class String
{private:
    int length;
    char* buf;

public:
    String();
    String(const char* s);
    String(char c,int n);

    void set(int index,char newchar);
    char get(int index) const;
    int getLength() const {return length;};
    void display() const {cout<<buf;};
    ~String();
};
```

### Beispiel (3)

```
String::String()
{length = 0;buf = 0;}

String::String(const char *s)
{length = strlen(s);
  buf = new char[length + 1];
  strcpy(buf,s);
}

String::String(char c,int n)
{length = n;
  buf = new char[length + 1];
  memset(buf,c,n );
  buf[length]='\0';
}

String::~String()
{delete buf;
}
```

## Beispiel (4)

```
void String::set(int index, char newchar)
{if((index >= 0) && (index < length))
  buf[index] = newchar;
}

char String::get(int index) const
{char res;
  if((index >= 0) && (index < length))
    res=buf[index];
  return res;
}

int main(void)
{String myString("here`s my string");
  myString.set(0, 'H');
  myString.display();
}
```

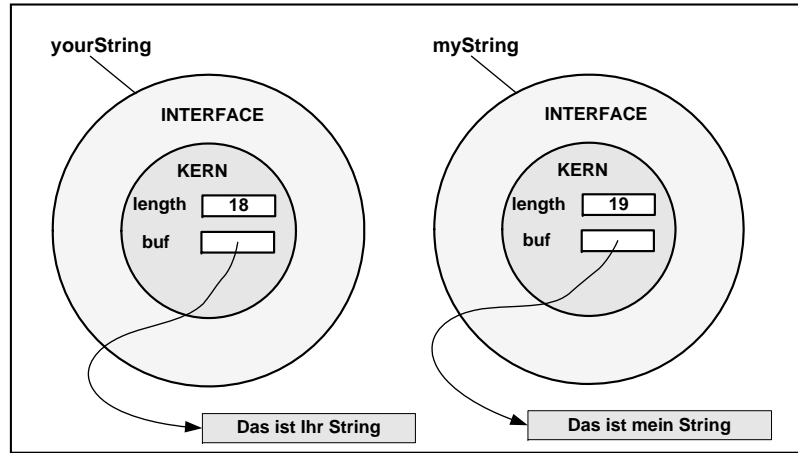
## Überladen des Zuweisungsoperators (1)

Findet eine Zuweisung zwischen Instanzen mit Zeigerelementen statt, so werden wie bisher die Attribute einander zugewiesen. Dies bedeutet bei unserer Stringklasse, dass nach der Zuweisung die Zeiger auf den selben Bereich im Freispeicher zeigen. Die Instanzen sind dadurch nicht mehr unabhängig voneinander. Verliert z.B. eine Instanz ihre Existenz, wird durch den Destruktor der allokierte Speicher freigegeben. Die andere Instanz zeigt nun auf einen Speicherbereich, der gar nicht mehr existiert. Diese Instanz ist nicht mehr brauchbar.

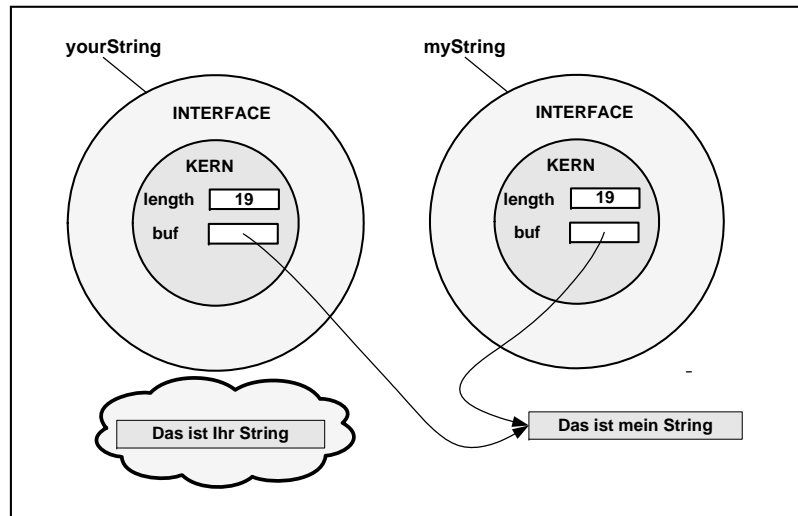
Bei einer Funktionsweise der Zuweisung wie hier beschrieben, spricht man von einer **flachen Kopie**:

```
String myString("Das ist mein String");  
String yourString("Das ist Ihr String");  
yourString = myString;  
//yourString ist eine flache Kopie von myString:  
//yourString.length = myString.length  
//yourString.buf = myString.buf
```

## Überladen des Zuweisungsoperators (2)



vorher



nachher

## Überladen des Zuweisungsoperators (3)

Die "richtige" Zuweisung müßte wie folgt durchgeführt werden:

- Freispeicher von **yourString** wird freigegeben
- neuer Speicher für **yourString** wird allokiert
- String von **myString** wird in den neuen Speicher kopiert

Dies kann dadurch erreicht werden, dass der Zuweisungsoperator eine neue klassenspezifische Wirkung bekommt. Der Zuweisungsoperator wird "überladen", indem eine Methode definiert wird, die den Namen **operator=** bekommt und die die obengenannten Aufgaben durchführt. Der Software-Entwickler kann demnach dem Zuweisungsoperator eine neue, eigendefinierte Bedeutung geben.

Die Zuweisung **yourString = myString;** entspricht daher **yourString.operator=(myString);**

Wird zwischen Instanzen der Zuweisungsoperator verwendet, informiert sich der Compiler, ob eine Funktion **operator=** in der Klasse des links stehenden Operanden vorhanden ist. Wenn nicht, wird eine flache Kopie erzeugt, wenn ja, wird die Methode **operator=** aufgerufen.

## Beispiel

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

class String
{private:
    int length; char* buf;
public:
    String();
    String(const char *s);
    String(char c, int n);
    void operator=(const String &src);
};

void String::operator=(const String &src)
{delete buf;
    length = src.length;
    buf = new char[length+1];
    strcpy(buf,src.buf);
}
```

## Teste dein Wissen: Aufgabenstellung 1

### Überladen des unären Operators ~

```
//Deklaration der Klasse Point
class Point
{ int x,y;
  public:
    Point(int a,int b){x=a;y=b;}
};
```

Gegeben ist die Klasse **Point** mit den Koordinatenattribute **x** und **y**. Mit dem Operator ~ soll ein Objekt der Klasse **Point** an der 1. Winkelhalbierenden gespiegelt werden.

## Teste dein Wissen: Lösung 1

```
//Klasse Point mit dem überladenen Operator ~  
class Point  
{ int x,y;  
  public:  
    Point(int a,int b){x=a;y=b;}  
  
};
```

## Teste dein Wissen: Aufgabenstellung 2

Entwickeln Sie die Klasse **Arr**, die ein Feld **f[2]** mit zwei **int**-Werten enthält.

Überladen Sie den **+**Operator, so dass die Addition zweier **Arr**-Instanzen sowie die Addition einer **Arr**-Instanz mit einer Konstanten möglich ist:

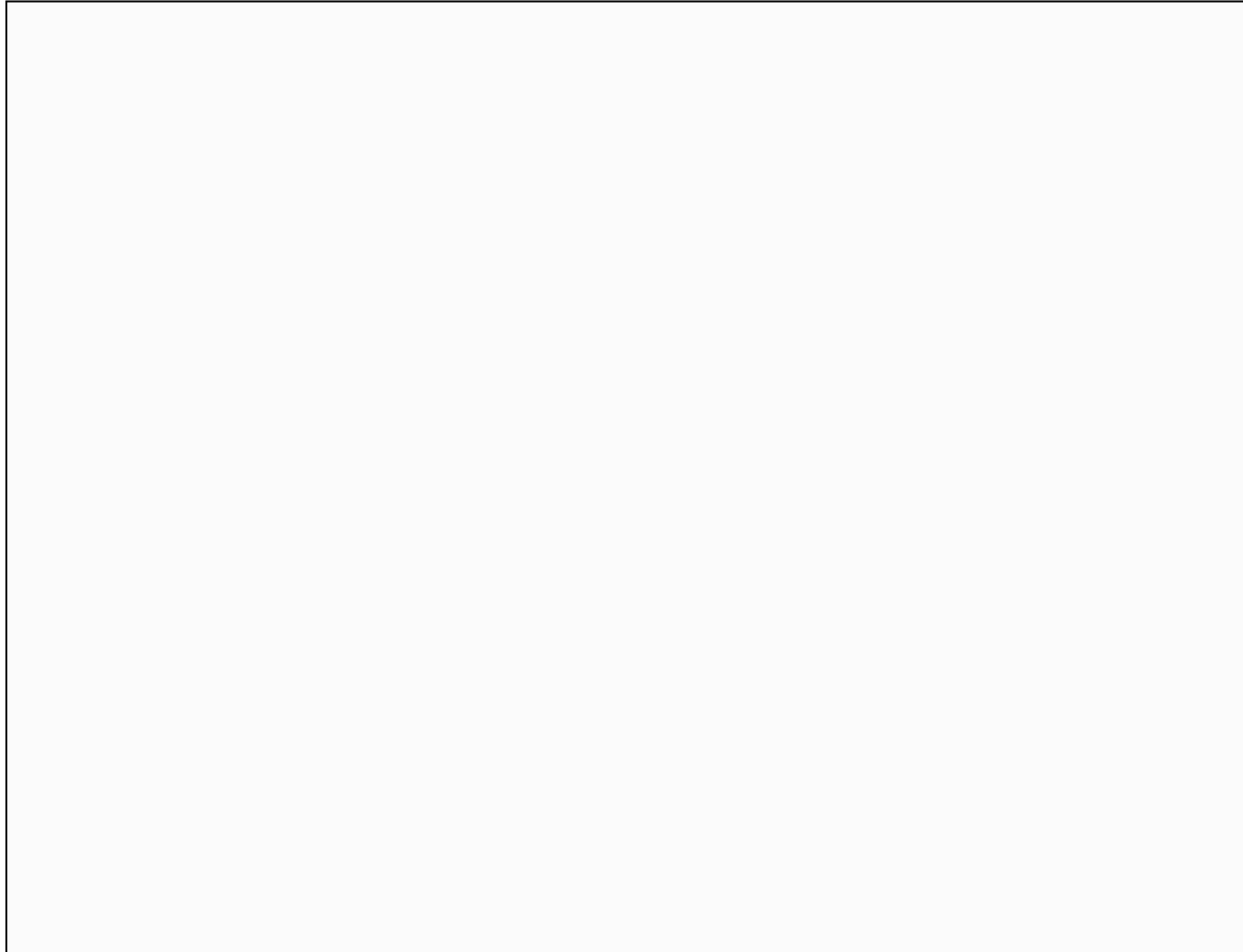
```
arr1 = arr2 + arr3    (Realisierung durch Methode)  
arr1 = arr2 + Konst  (Realisierung durch Methode)  
arr1 = konst + arr3  (Realisierung durch friend-Funktion)
```

## Teste dein Wissen: Lösung 2 (1)

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

class Arr
{
```

## Teste dein Wissen: Lösung 2 (2)

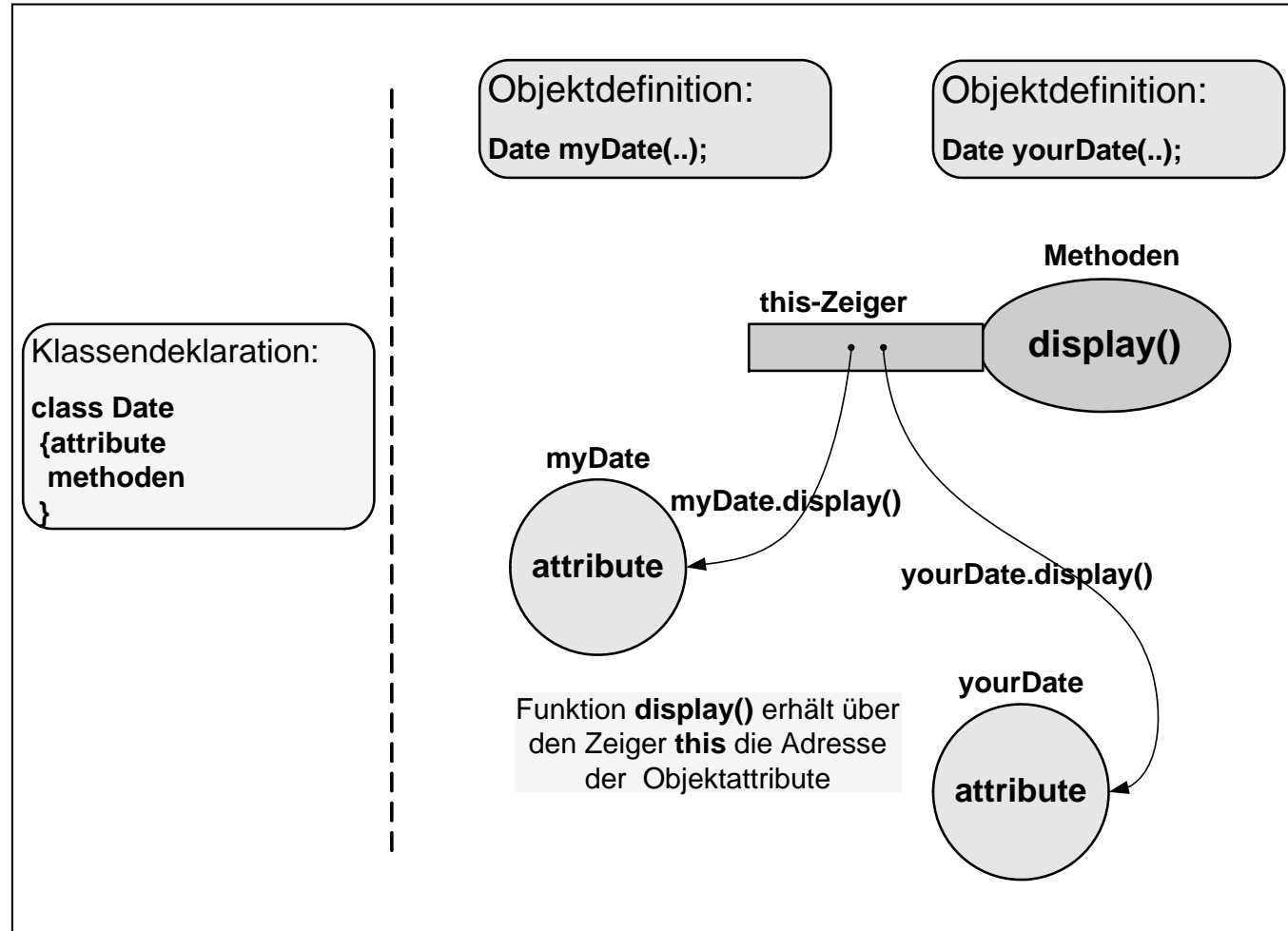


## Zeiger this (1)

Ist der Zuweisungsoperator überladen, ergibt sich bei der Selbstzuweisung **myString = myString** das Problem, dass zuerst der allokierte Speicher mit dem String "Das ist mein String" freigegeben wird und damit verloren geht. Anschliessend gibt es nichts mehr zum Kopieren. Die richtige Reaktion der Methode **operator=()** wäre, bei der Selbstzuweisung nichts zu tun. Dazu müsste die Methode feststellen, dass als aktueller Parameter die "eigene Instanz" übergeben wurde. Dies führt zu dem Zeiger **this**, den der Compiler für jede Klasse anlegt.

Die Methoden einer Klasse werden nur einmal im Speicher abgelegt. Die einzelnen Instanzen sind ebenfalls gespeichert, bestehen aber nur aus den speziellen Attributen. Wird eine Methode zu einer speziellen Instanz aufgerufen, dann braucht diese Methode die Information, wo die Instanzattribute sich befinden. Diese Information stellt der Compiler im Zeiger **this** zur Verfügung, der innerhalb eines Programms auch explizit verwendet werden kann.

## Zeiger this (2)



## Zeiger this (3)

Zwei Anwendungen des **this**-Zeigers:

1. Innerhalb der Methode **operator=** kann nun durch Vergleich des Zeigers **this** mit der übergebenen Referenz (siehe Beispiel) die Selbstzuweisung festgestellt werden.
2. Als Syntax sind Ausdrücke wie **a = b = c** zulässig. Dies entspricht dem Ausdruck **a.operator=(b.operator=(c))**. Soll dies funktionieren, muss die Methode **operator=** die Instanz auf der linken Seite des Zuweisungsoperators zurückliefern. Dies erfolgt durch die Anweisung **\*this** (siehe Beispiel).

## Beispiel 1

```
//Zugriffe auf die Attribute von myDate erfolgen
//implizit über den this-Zeiger
void Date::setMonth(int mn)
{month=mn;}

//Syntax ist möglich, aber redundant
void Date::setMonth(int mn)
{this->month=mn;}

//analog in C
void Date_setMonth(Date* const this,int mn)
{this->month= mn;}

Date_setMonth(&myDate,3);
```

## Beispiel 2

```
//Selbstprüfung und Instanzrückgabe
String String::operator=(const String& src)
{if(&src!=this)
  {delete buf;
   buf = new char[length=src.length];
   strcpy(buf,src.buf);
  }
return *this;
}
```

## Teste dein Wissen: Aufgabenstellung

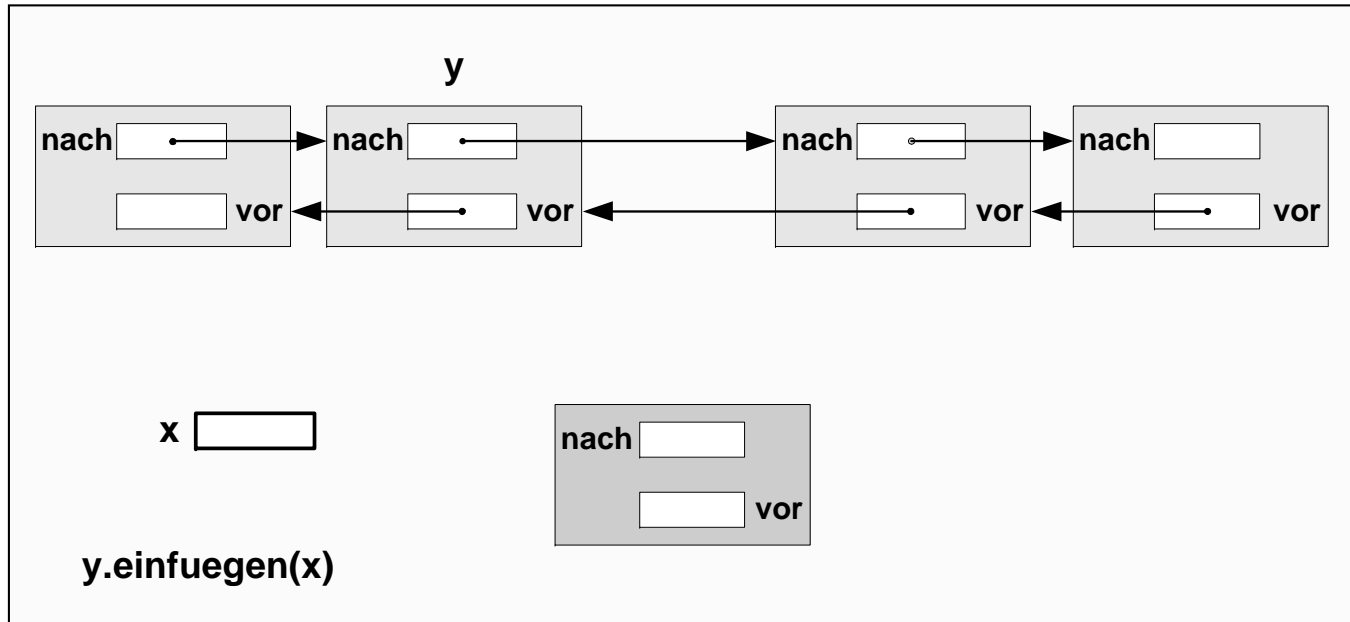
**this**-Zeiger und eine doppelt verkettete Liste

```
//Klassendeklaration
class Dkette
{Dkette* vor;
  Dkette* nach;
public:
  void einfuegen(Dkette* x);
};
```

Programmiere die Methode **einfuegen**, die ein **Dkette**-Objekt nach dem Objekt **y** einfügt. Erstelle für den Vorgang eine Skizze, in der die beteiligten Objekte und die Zeiger-Attribute ersichtlich sind.

## Teste dein Wissen: Lösung

```
void Dkette::einfuegen(           )
{
}
}
```



## Kopierkonstruktor (1)

In C++ wird zwischen einer Zuweisung und einer Initialisierung strikt unterschieden:

Zuweisung:     `yourString = myString;`

Initialisierung: `String yourString = myString;`

Eine Zuweisung erfolgt immer an ein bereits existierendes Objekt, verändert es und kann an beliebiger Stelle mehrmals auf dieses Objekt angewendet werden. Eine Initialisierung erfolgt immer mit der Definition, also zu Beginn der Lebenszeit eines Objekts; mit der Initialisierung erhält ein Objekt einen quasi voreingestellten inneren Zustand. Eine Initialisierung kann auf ein Objekt nur ein einziges Mal angewendet werden. Bei einer Initialisierung wird ein Konstruktor aufgerufen. Da im oberen Fall eine Kopie von `myString` erzeugt wird, nennt man den Konstruktor, der als Übergabeparameter eine Instanz der eigenen Klasse erhält, den **Kopierkonstruktor**.

Eine Initialisierung wie oben wird daher wie folgt präziser ausgedrückt:

`String yourString(myString);`

## Kopierkonstruktor (2)

Zwischen dem Kopierkonstruktor und dem Zuweisungsoperator gibt es folgende Unterschiede:

Der Zuweisungsoperator wird auf bereits existierende Objekte angewendet, der Kopierkonstruktor legt ein neues Objekt erst selbst an. Deshalb muß der Zuweisungsoperator im Gegensatz zum Kopierkonstruktor ggf. verschiedene Datenelemente vorschriftsmäßig löschen, das Objekt also teilweise abbauen, bevor er die Daten übernehmen kann.

Bei der Initialisierung kann es keine Selbstzuweisung geben. Der Zuweisungsoperator muß die Selbstzuweisung berücksichtigen, der Kopierkonstruktor hingegen nicht.

Damit Mehrfachzuweisungen bzw. verkettete Zuweisungen möglich werden, muß der Zuweisungsoperator als Ergebnis **\*this** zurückliefern. Konstruktoren, also auch ein Kopierkonstruktor, haben gar kein Funktionsergebnis.

## Beispiel

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

class String
{ int length;
  char* buf;
public:
  String(); //Standardkonstruktor
  String(const char* s);
  String(char c,int n);
  String(const String& src); //Kopierkonstruktor
  void operator = (const String& src);
};

String::String( const String &src)
{buf = new char[length = src.length];
  strcpy(buf,src.buf);
}
```

## Teste dein Wissen: Aufgabenstellung

Erzeugen Sie eine Klasse **Auto** mit folgenden Attributen:

- **int**-Variable zum Führen des Kilometerstands
- Zeiger auf einen String für die Automobilmарke; die Länge des Strings ist vor der Instanzerzeugung nicht bekannt, daher wird der Speicher dynamisch angefordert

und folgenden Konstruktoren:

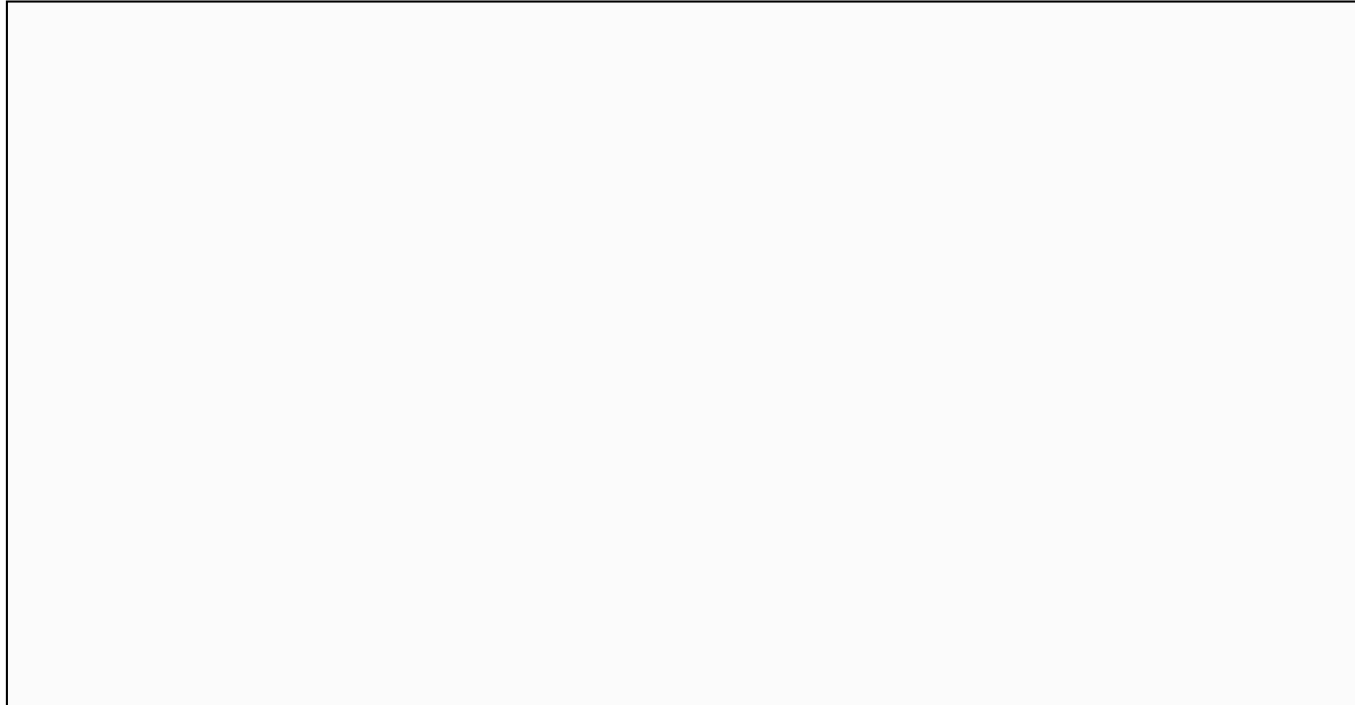
- Standardkonstruktor
- Konstruktor zur Übernahme des Markenstrings
- Kopierkonstruktor mit der Fähigkeit, den Kilometerstand des neuen Autos bei Bedarf auf 0 zu setzen

Ergänzen Sie die Methoden, damit die Klasse erprobt werden kann.

## Teste dein Wissen: Lösung (1)

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

## Teste dein Wissen: Lösung (2)



## Funktionen und Instanzen (1)

Instanzen können wie gewöhnliche Variable als Funktionsübergabeparameter und als Funktionsrückgabeparameter verwendet werden. Als Technik ist erlaubt call-by-value und call-by-reference.

Beispiel für call-by-value:

```
void consume(String par){...}

int main(void)
{String myString("Das ist mein String");
  consume(myString);
}
```

**myString** wird per call-by-value an die Funktion **consume** übergeben, d.h. beim Aufruf der Funktion wird eine Kopie von **myString** erzeugt und steht als Kopie in der Funktion zur Verfügung. Wird nur eine flache Kopie erzeugt und endet die Lebenszeit der Kopie am Funktionsende, gibt der Destruktor den allokierten Bereich im Freispeicher frei.

## Funktionen und Instanzen (2)

Dadurch geht auch der Instanz **myString** im aufrufenden Programm der allokierte Speicher verloren.

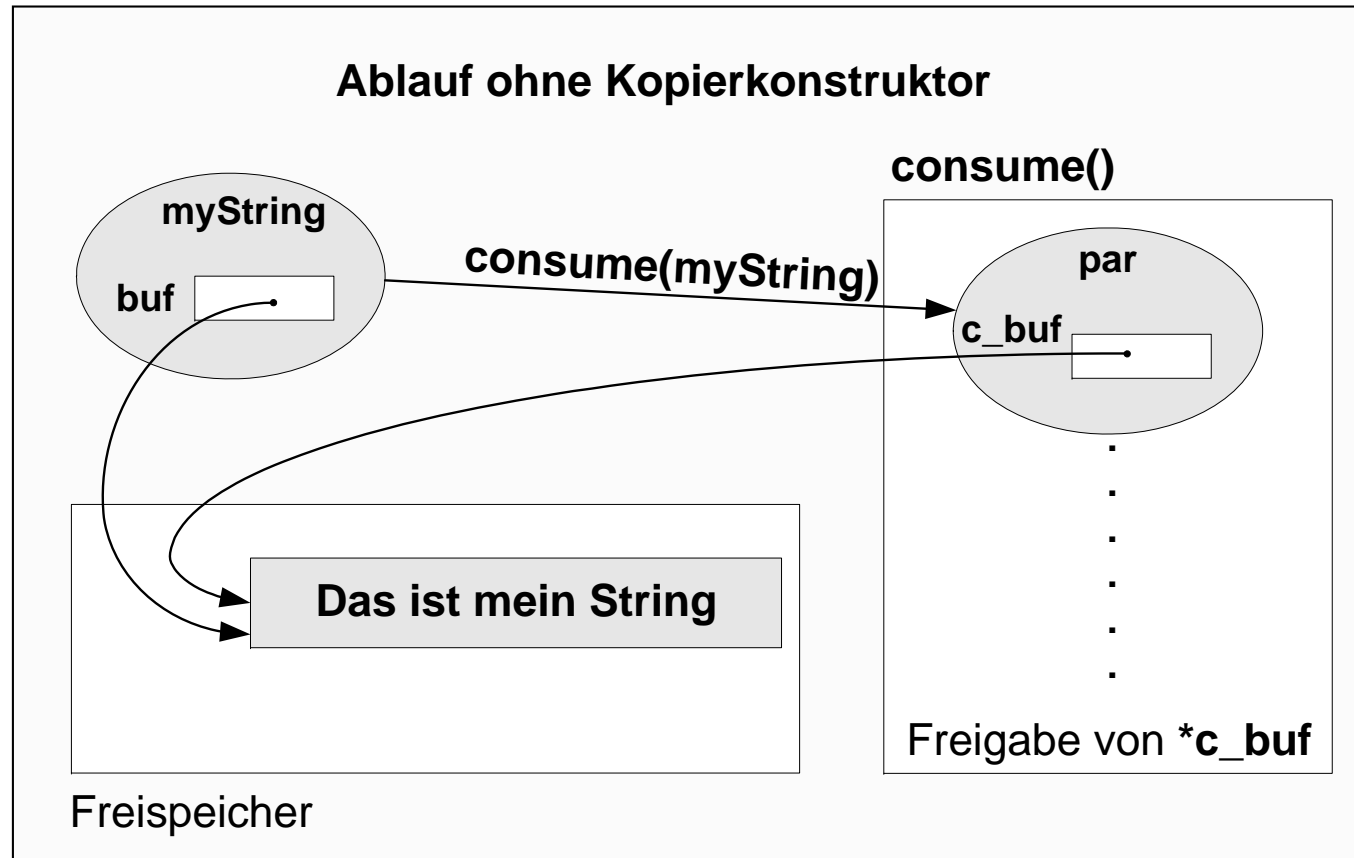
Die Instanz **myString** kann nicht mehr vernünftig benutzt werden.

Der richtige Ablauf (so wird es auch gemacht) sieht wie folgt aus:

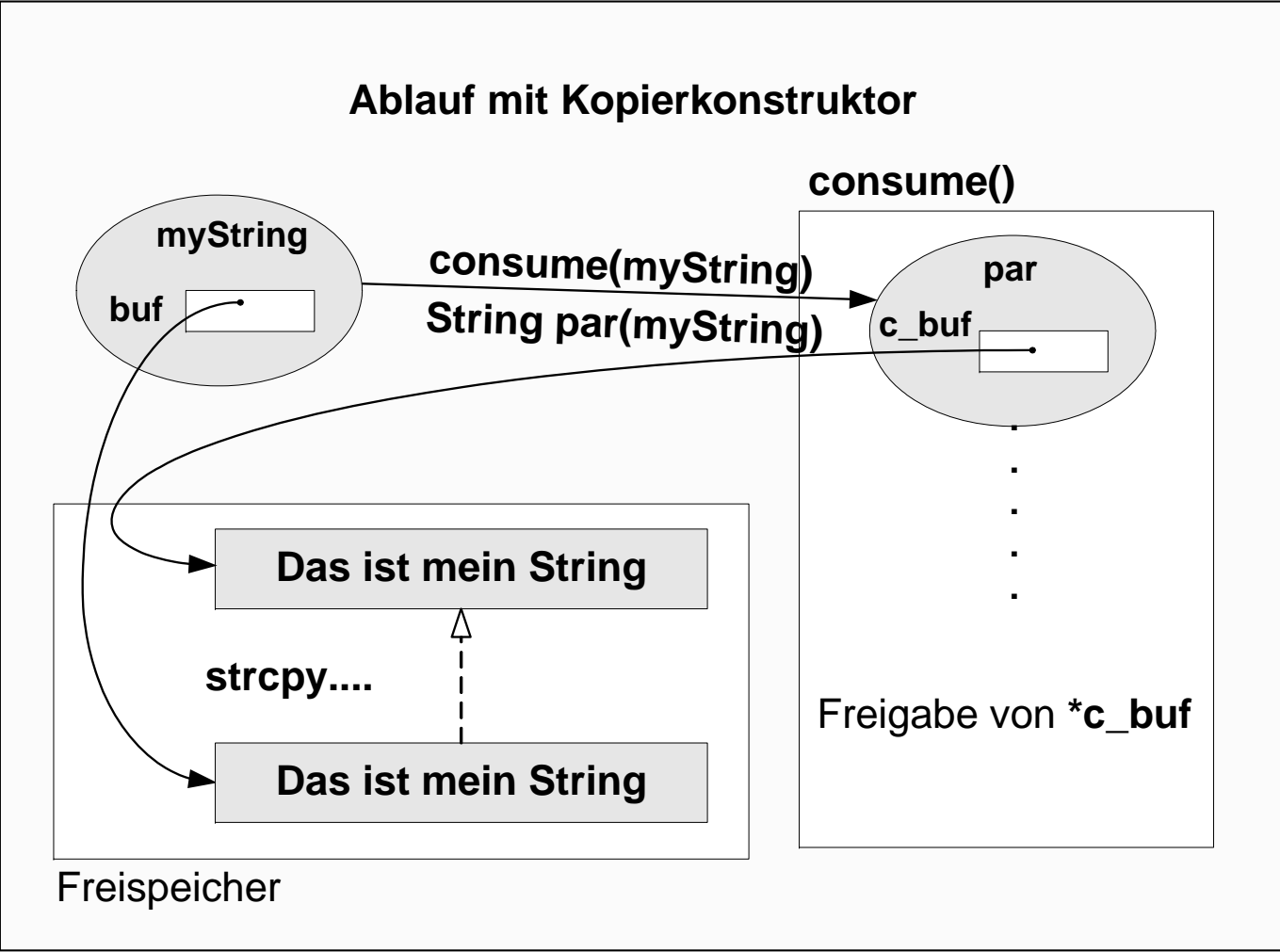
1. Compiler fügt implizit den Aufruf des Kopierkonstruktors ein; damit verwendet die Kopie ihren eigenen Speicher:  
**String par(myString).**
2. Freigabe dieses Speichers bei Funktionsende.
3. **myString** (im aufrufenden Programm) verwendet ihren eigenen Speicher, der mit dem Speicher der Kopie nichts zu tun hat.

Jetzt wird auch klar, warum der Kopierkonstruktor seinen Übergabeparameter per Referenz bekommt. Wird dem Kopierkonstruktor der Wert per call-by-value übergeben, wird wiederum der Kopierkonstruktor aufgerufen, usw., usw., usw. Bei call-by-reference wird keine Kopie gebildet und daher der Kopierkonstruktor auch nicht aufgerufen.

## Funktionen und Instanzen (3)



### Funktionen und Instanzen (4)



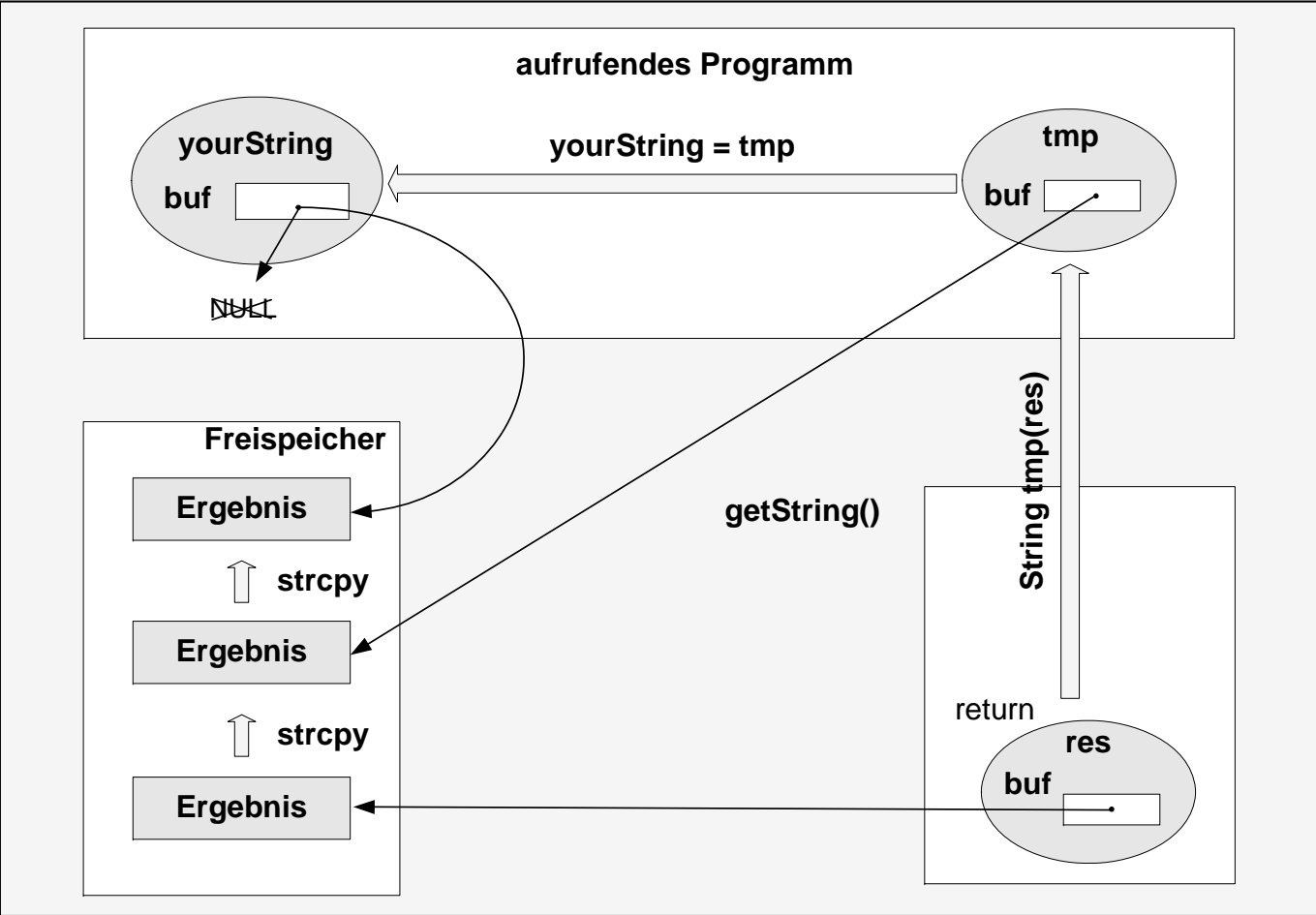
## Funktionen und Instanzen (5)

Die Rückgabe eines Werts aus einer Funktion erfolgt durch call-by-value, d.h. im aufrufenden Programm steht eine Kopie des Wertes aus der Funktion zur Verfügung.

```
String getString(void)
{String res("Ergebnis");
  return res;
}
```

```
int main(void)
{String yourString;
  yourString = getString();
}
```

# Funktionen und Instanzen (6)



## static-Attribute und static-Methoden (1)

Folgende Vorstellung: Eine Klasse **SavingsAccount** (Sparkonto) enthält den Kontostand und den aktuellen Zinssatz (**dailyInt**). Jede Instanz entspricht einem Sparkonto. Ändert sich der Zinssatz, muss die Änderung mühsam in jeder Instanz durchgeführt werden. Der Einsatz einer globalen Variablen kommt wegen dem Prinzip der Datenkapselung nicht in Frage. Die Lösung ist eine Variable, die pro Klasse nur einmal vorhanden ist. Diese Variable ist in jeder Instanz sichtbar, kann aber nach außen gekapselt werden (private!). Diese Art von Variable nennt sich **static**-Attribut oder auch Klassenattribut:

```
class SavingsAccount
{public:
    SavingsAccount();
    void earnInterest(){total=total(1 + dailyInt);}

private:
    char name[50];
    float total;
    static float dailyInt;    //Klassenattribut
};
```

## static-Attribute und static-Methoden (2)

Die Initialisierung von **static**-Attributen kann **nicht** durch Konstruktoren durchgeführt werden, da:

- **static**-Attribute schon vor dem ersten Klassenobjekt existieren können
- pro erzeugter Instanz wird jeweils der Konstruktor aufgerufen

Die Initialisierung wird wie bei einer globalen Variablen durchgeführt und ist unabhängig davon, ob die Variable **public** oder **private** erklärt wurde.

```
float SavingsAccount::dailyInt = 0.000154;

..
int main(void)
{SavingsAccount myAccount,yourAccount;
  cout << myAccount.dailyInt;
}
```

## static-Attribute und static-Methoden (3)

**static**-Attribute können **public** und **private** sein.

**public-static**-Attribute können von außen, **private-static**-Attribute können über Methoden verändert werden.

```
int main(void)
{SavingsAccount myAccount,yourAccount;
  myAccount.dailyInt = 0.000154; //private/public?
  cout << yourAccount.dailyInt;
}
```

Da **static**-Attribute vor jeder Instanz existieren, können sie auch ohne Instanz verändert werden.

```
int main(void)
{SavingsAccount::dailyInt = 0.000154;
  SavingsAccount myAccount,yourAccount;
  cout << yourAccount.dailyInt;
}
```

## static-Attribute und static-Methoden (4)

Sind **static**-Attribute **private**, kann man über Methoden auf sie zugreifen. Dies setzt aber das Vorhandensein von Instanzen voraus. Will man auf **static-private**-Attribute ohne eine spezielle Instanz zugreifen, verwendet man **static**-Methoden.

```
class SavingsAccount
{ static float dailyInt;
  public:
    static void setInterest( float newValue );
};
```

```
int main(void)
{SavingsAccount::setInterest(0.000154);}
```

**static**-Methoden haben keinen Zugriff auf nicht-**static**-Attribute und können auch keine sonstigen Methoden der Klasse verwenden.

## Beispiel

```
class Airplaine
{public:
    Airplaine(){count++;}
    static int howMany(){return count;}
    ~Airplaine(){count--;}
private:
    static int count;
};

int Airplaine::count=0;
Airplaine myAirplaine;

int main(void)
{Airplaine yours;
  cout << Airplaine::howMany() << '\n';
}
```

## Verwandte Datentypen in C (1)

Als nächstes Thema der objektorientierten Programmierung steht die **Vererbung** an. Das Thema soll anhand eines Beispiels eingeführt werden. Zunächst wird dieses Beispiel mit den Möglichkeiten von C betrachtet und gelöst, um damit den Fortschritt durch die Vererbungstechnik zu zeigen.

Die angesprochene Aufgabe besteht darin, eine Datenverwaltung sämtlicher Beschäftigten einer Firma zu erstellen. Drei Typen von Firmenmitarbeitern werden betrachtet, für die jeweils eine spezielle Verdienstabrechnung durchgeführt wird:

- Arbeiter (Anzahl Stunden mal Stundenlohn)
- Verkäufer (Anzahl Stunden mal Stundenlohn plus Provision)
- Manager (Festgehalt)

## Verwandte Datentypen in C (2)

Die Aufgabe besteht nun zunächst darin, eine Datenstruktur zu finden, in die sämtliche Mitarbeiter aufgenommen werden können:

1. eine Struktur für alle Beschäftigungstypen (Nachteil:Speicheraufwand)
2. pro Beschäftigungstyp eine eigene Struktur  
(großer Programmieraufwand)
3. Verwendung einer union  
(Lösung ist vertretbar)

## Beispiel (1)

```
//Deklaration der drei Strukturen für die union
struct mgrPay          //Bezahlung Manager
{float weeklySalary;  //Wochengehalt
};

struct wagePay         //Bezahlung Lohnempfänger
{float wage;          //Lohn
 float hrs;           //Stundenzahl
};

struct salesPay        //Bezahlung Verkaufspersonal
{float wage;          //Lohn
 float hours;         //Stundenzahl
 float commission;    //Provision
 float salesMade;     //Umsatz
};
```

## Beispiel (2)

```
//Datentyp der Beschäftigungsarten
enum EMPLOYEE_TYPE
    {WAGE_EMPLOYEE, SALES_PERSON, MANAGER};
```

```
//Struktur für einen Beschäftigten
struct employee
{char name[30];
  EMPLOYEE_TYPE type;

  union
  {struct wagePay worker;
    struct salesPay seller;
    struct mgrPay mgr;
  };
};
```

## Beispiel (3)

```
//Verdienstberechnung
float computePay(struct employee* emp)
{float res = 0.0;

switch(emp->type)
{case WAGE_EMPLOYEE:
    res = emp->worker.hrs*emp->worker.wage;
    break;

case SALESPERSON:
    res = emp->seller.hrs*emp->seller.wage;
    res+=emp->seller.commission*emp->sellerMade;
    break;

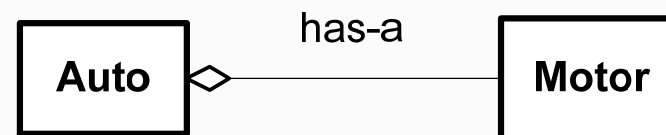
case MANAGER:
    res = emp->mgr.weeklySalary;
    break;
}
return res;
}
```

## Vererbung (1)

Die Bildung von Klassen (Objekten) und die Beziehungen zwischen Klassen (Objekten) dient zur Modellierung von Teilen der realen Welt. Wir wollen zwei Beziehungen betrachten.

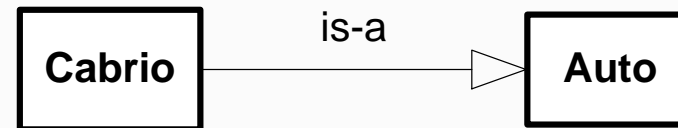
Durch die Zerlegung eines realen Objekts in einzelne Komponenten erhalten wir zwischen dem Gesamtobjekt und seinen Komponenten sogenannte **has-a-Beziehungen**. Zu der Beziehung zwischen einem Auto und seinem Motor kann man sagen:

Ein Auto "has-a" Motor. In C++ wird diese Beziehung durch eine Zusammensetzung realisiert, d.h. die Klasse Auto enthält ein Attribut vom Typ Motor, wobei Motor ebenfalls eine Klasse darstellt. Grafisch sieht diese Beziehung wie folgt aus:



## Vererbung (2)

Eine zweite Beziehung folgt aus dem Zusammenhang zwischen Auto und Cabrio. Hier kann man nicht sagen, ein Auto "has-a" Cabrio, sondern ein Cabrio "is-a" Auto. Es liegt eine **is-a-Beziehung** vor. Man kann auch sagen, ein Cabrio ist eine Spezialisierung eines Autos. D.h. ein Cabrio besitzt alle Attribute und Methoden eines Autos, aber eben noch weiter, die charakteristisch für ein Cabrio sind. Grafisch sieht diese Beziehung wie folgt aus:

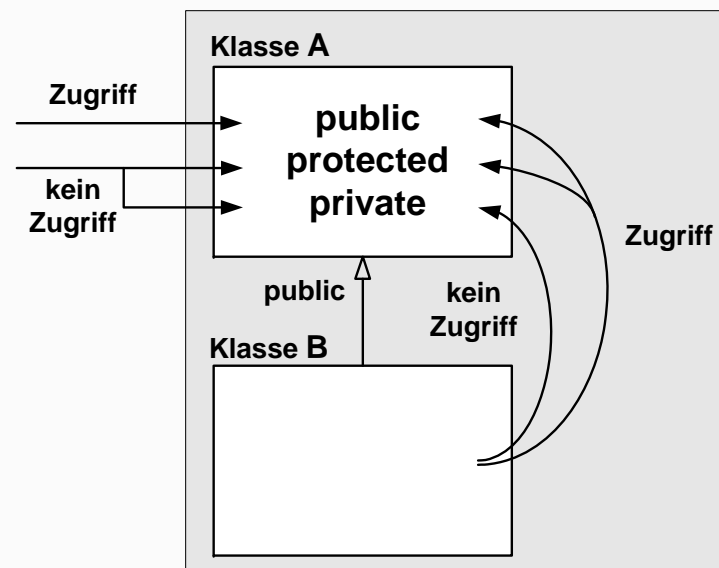


In C++ wird diese Beziehung als **Vererbung** realisiert. Wir wollen nun die Datenverwaltung der Mitarbeiter mit den Ausprägungen Arbeiter, Verkäufer und Manager durch Vererbungsbeziehungen realisieren.

## Vererbung (3)

Üblicherweise werden Klassen **public** vererbt (abgeleitet). Die **public**-Attribute der Basisklasse sind in der abgeleiteten Klasse auch **public** zugänglich. **Private**-Attribute der Basisklasse bleiben auch in der abgeleiteten Klasse **private**. Die Datenkapselung bleibt erhalten.

Um die Verwandtschaft zwischen Basisklasse und abgeleiteter Klasse zu verstärken, wird das Zugriffsrecht **protected** eingeführt. **protected**-Attribute der Basisklasse sind in der abgeleiteten Klasse nutzbar wie **public**-Attribute. Von außen sind **protected**-Attribute genauso geschützt wie **private**-Attribute.



## Vererbung (4)

```
//Employee ist die Klasse, in der sämtliche Attribute und
//Methoden enthalten sind, die für alle Mitarbeiter gelten,
//hier exemplarisch der Name eines Mitarbeiters.
class Employee
{ char name[30];
  public:
    Employee();
    Employee(const char* name);
    char* getName() const;};

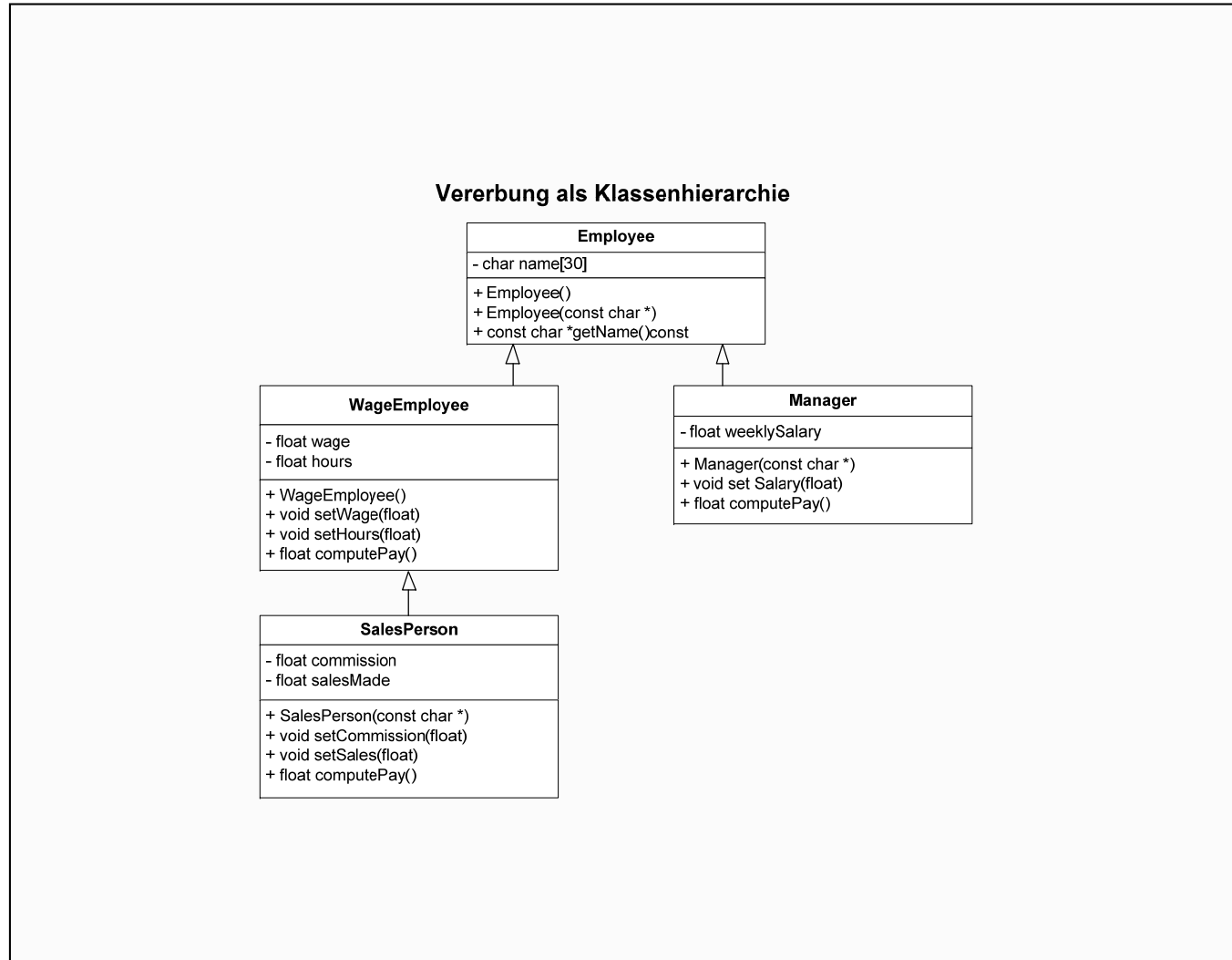
//WageEmployee ist die Klasse, die Attribute und Methoden
//enthält, die spezifisch für einen Arbeiter sind.
class WageEmployee : public Employee
{ float wage;      //Stundenlohn
  float hours;    //Arbeitszeit in Stunden
  public:
    WageEmployee();
    void setWage(); //Uebergabe des Stundenlohns
    void setHours();//Uebergabe der geleisteten Arbeitszeit
};
```

## Vererbung (5)

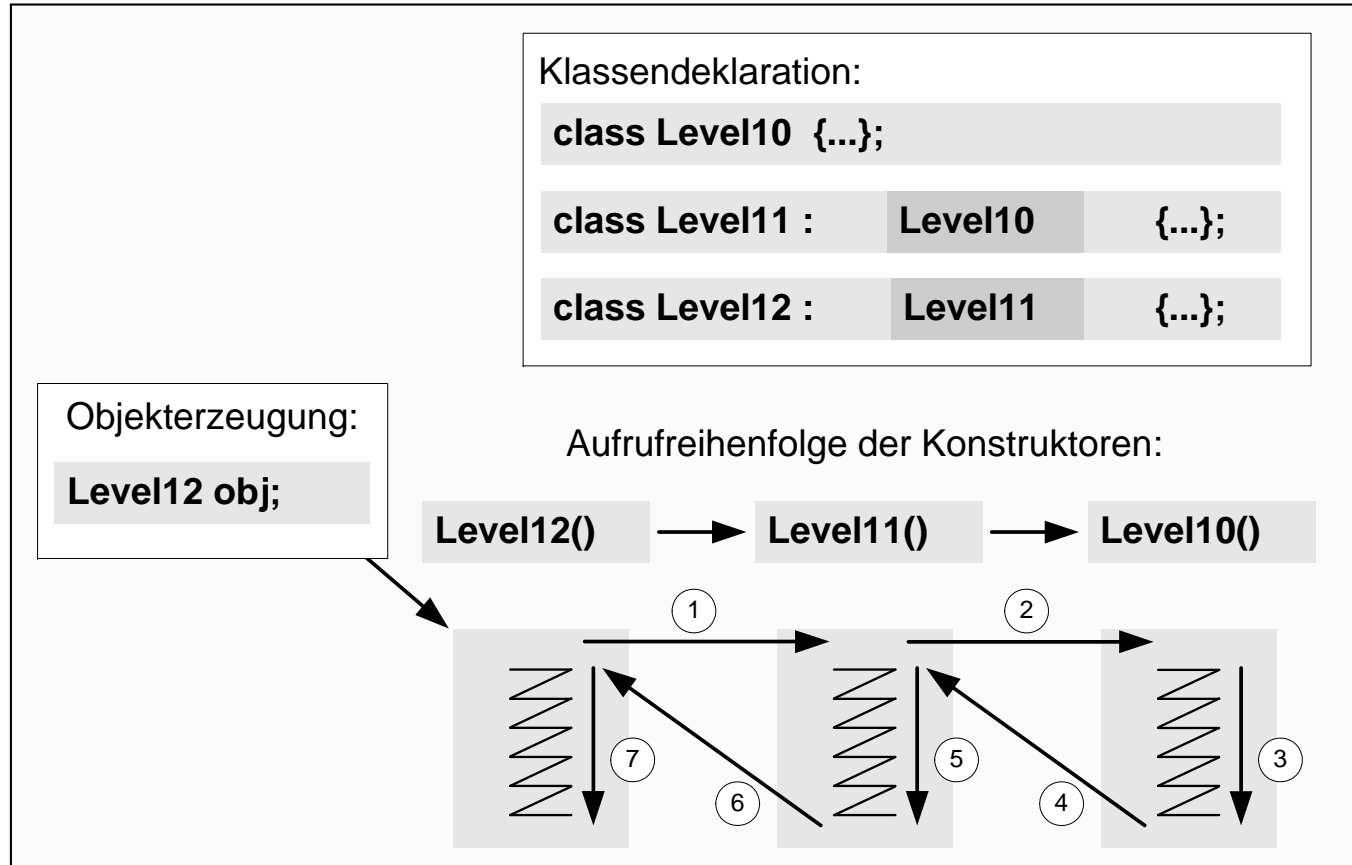
```
//SalesPerson ist die Klasse, die Attribute und Methoden
//enthält, die spezifisch für einen Verkäufer sind. Da ein
//Verkäufer eine Spezialisierung eines Arbeiters ist,
//leiten wir SalesPerson von WageEmployee ab.
class SalesPerson : public WageEmployee
{ float commission;           //Umsatzbeteiligung
  float salesMade;           //Umsatz
public:
  SalesPerson(const char *name);
  void setCommission(float comm);
  void setSales(float sales);
};

//Manager ist die Klasse, die Attribute und Methoden
//enthält, die spezifisch für einen Manager sind.
class Manager : public Employee
{ float weeklySalary;
public:
  Manager(const char* nm);
  void setSalary(float salary);
};
```

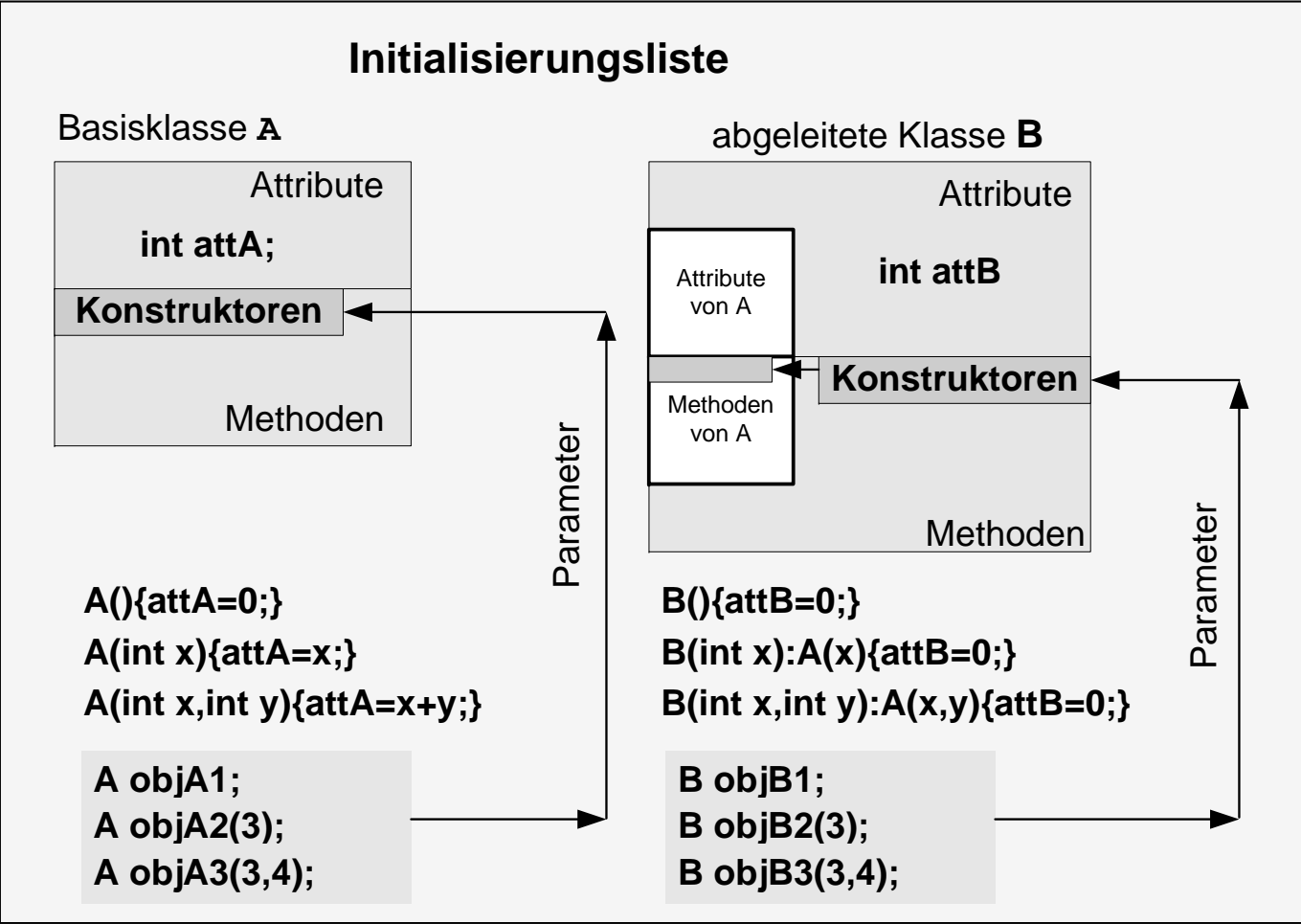
## Vererbung (6)



## Vererbung (7)



# Vererbung (8)



## Redefinition von Elementen

Elemente (Attribute und Funktionen) in abgeleiteten Klassen, die namensgleich sind mit Elementen der Basisklasse, überdecken diese bei ihrer Verwendung in den abgeleiteten Klassen. Sollen in den abgeleiteten Klassen die Elemente der Basisklasse Verwendung finden, muss der Scope-Operator eingesetzt werden. Beispiel Lohnberechnung:

```
float WageEmployee::computePay() // public vereinbart
{return wage*hours;}
```

```
//Lösung 1
```

```
float SalesPerson::computePay()
{return (wage*hours + commission*SalesMade);}
```

```
//Lösung 2
```

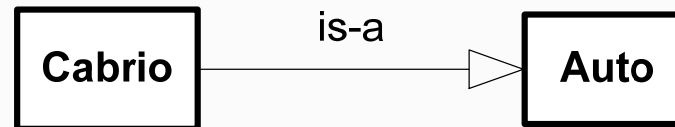
```
float SalesPerson::computePay()
{return computePay() + commission*SalesMade;}
```

```
//Lösung 3
```

```
float SalesPerson::computePay()
{return WageEmployee::computePay() + commission*SalesMade;}
```

## Konvertierung abgeleiteter Klassen

Gegeben sei die Klasse **Auto** und davon abgeleitet die Klasse **Cabrio**.



Daraus kann man folgern, dass eine Instanz der Klasse **Cabrio** sämtliche Elemente der Basisklasse **Auto** enthält, d.h. mindestens eine Instanz der Klasse **Auto** ist. Umgekehrt kann dies nicht behauptet werden.

Daraus kann man folgern, dass überall da, wo Instanzen vom Typ der Basis-klasse erwartet werden, auch Instanzen der abgeleiteten Klassen stehen können. (Der Umkehrschluß gilt allerdings nicht.) Dies wird erreicht durch **implizite Typenkonvertierung** durch den Compiler. Ihre Meinung zu den folgenden Fällen?

```
Auto aAuto1      = aCabrio1;    //Fall 1
Cabrio aCabrio2 = aAuto2;      //Fall 2
```

## Zeigerverwendung (1)

Zeiger, die typisiert auf die Basisklasse sind, zeigen üblicherweise auf Objekte vom Typ der Basisklasse. Da Objekte vom Typ der Basisklasse durch Objekte der abgeleiteten Klassen ersetzt werden können, kann ein Zeiger, typisiert auf die Basisklasse, auch auf Objekte der abgeleiteten Klassen zeigen. (Der Umkehrschluss gilt wiederum nicht.)

Dazu folgendes Beispiel:

```
Employee* empPtr;
```

```
WageEmployee aWageEmployee("A. Houston");
```

```
SalesPerson aSalesPerson("L. Sprewell");
```

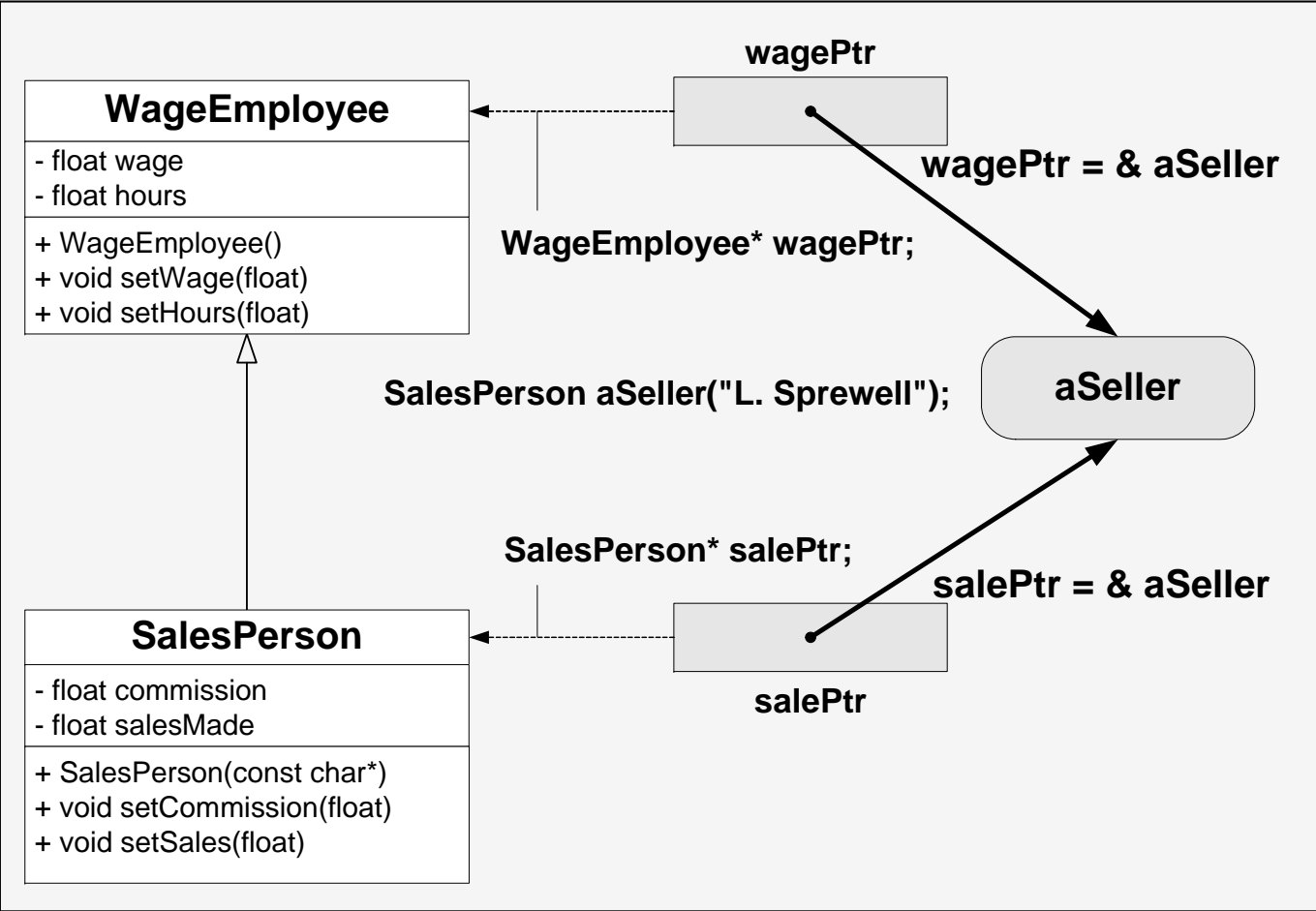
```
Manager aBoss("P. Ewing");
```

```
empPtr = &aWageEmployee; //WageEmployee* nach Employee*
```

```
empPtr = &aSalesPerson; //SalesPerson* nach Employee*
```

```
empPtr = &aBoss; //Manager* nach Employee*
```

# Zeigerverwendung (2)



## Zeigerverwendung (3)

Werden mit Zeigern, die typisiert auf die Basisklasse sind, Methoden aufgerufen, dann können (zunächst) nur Methoden der Basisklasse selbst aufgerufen werden.

Dazu folgendes Beispiel, welches Sie bitte kommentieren wollen:

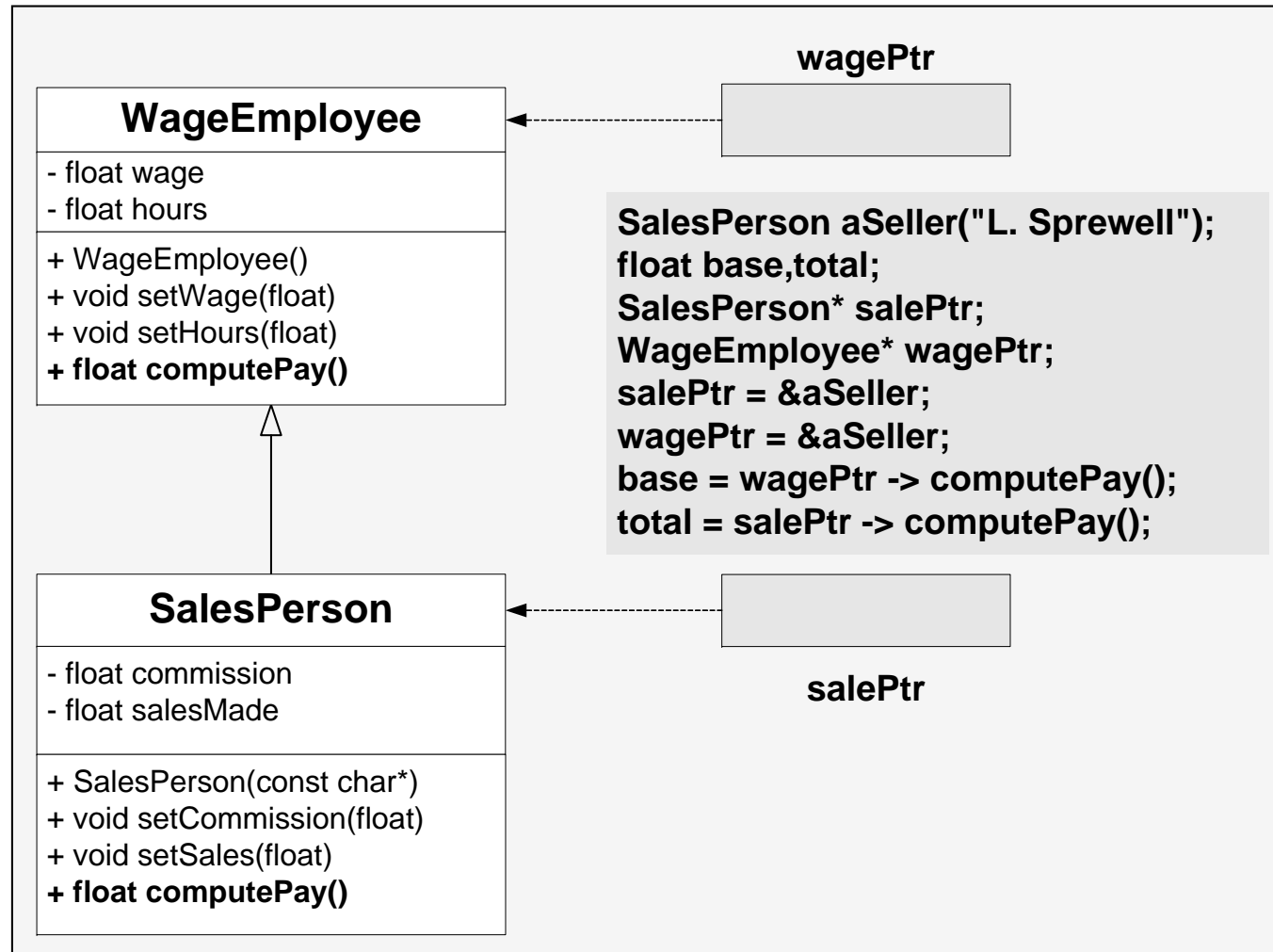
```
SalesPerson aSeller("L. Sprewell");

SalesPerson* salePtr;
WageEmployee* wagePtr;

salePtr = &aSeller;
wagePtr = &aSeller;

wagePtr->setHours(40.0);      //WageEmployee::setHours;
salePtr->setWage(6.0);        //WageEmployee::setWage;
wagePtr->setSales(1000);      //WageEmployee::setSale;
salePtr->setSales(1000);      //SalesPerson::setSale;
salePtr->setCommission(0.05); //SalesPerson::setCommission
```

## Zeigerverwendung (4)



## Beispiel 1(1)

```
//Funktionen, denen ein Zeiger auf Objekte vom Typ
//Employee übergeben wird, können auch über diesen
//Zeiger Objekte von abgeleiteten Klassen übergeben
//werden

class EmployeeList
{public:
    int x;
    Employee* Liste[3];
    EmployeeList();
    void add(Employee* newEmp)
        {Liste[x] = newEmp; x++;}
};
```

## Beispiel 1(2)

```
EmployeeList empList;      //Liste für alle Beschäftigten
WageEmployee* wagePtr;    //Zeiger auf Lohnempfänger
SalesPerson* salePtr;     //Zeiger auf Verkäufer
Manager* mgrPtr;          //Zeiger auf Manager

//Beschäftigte anlegen
wagePtr = new WageEmployee("A. Houston");
salePtr = new SalesPerson("L. Sprewell");
mgrPtr = new Manager("P. Ewing");

//Beschäftigte in die Liste eintragen
empList.add(wagePtr);      //Eintrag eines Lohnempfängers
empList.add(salePtr);     //Eintrag eines Verkäufers
empList.add(mgrPtr);      //Eintrag eines Managers
```

## Beispiel 1(3)

```
//Namen der Beschäftigten ausgeben
cout << empList.Liste[0]->getName();
    // A. Houston
cout << empList.Liste[1]->getName();
    // L. Sprewell
cout << empList.Liste[2]->getName();
    // P. Ewing

//Verdienst ausgeben
cout << empList.Liste[0]->computePay();
cout << empList.Liste[1]->computePay();
cout << empList.Liste[2]->computePay();
```

Welcher Verdienst wird ausgegeben?

## Beispiel 2

Gegeben ist folgender Programmausschnitt:

```
WageEmployee* wagePtr;  
SalesPerson* salePtr;  
  
WageEmployee aWorker;  
wagePtr = &aWorker;  
  
salePtr = (SalesPerson*) wagePtr;  
  
salePtr -> setCommission(0.05);
```

Welche Zeile ergibt einen Syntaxfehler?

Zeichnen Sie die Abhängigkeiten von Zeiger und Objekt!

Kommentieren Sie die letzte Zeile!

## Virtuelle Funktionen (1)

Für den Compiler sieht die bisherige Situation wie folgt aus: Die Elemente in der Liste **empList** sind auf **Employee** typisiert. In die Liste können Zeiger auf Objekte der Basisklasse, aber auch Zeiger auf Objekte der abgeleiteten Klassen gespeichert werden und dies während der Programmaufzeit in beliebiger Reihenfolge.

Muß der Compiler einen Funktionsaufruf wie

```
empList.Liste[i]->computePay( );
```

übersetzen, kann er auf Grund seines Wissens nur die Funktion der Basisklasse **Employee** verwenden. Man nennt dies **statische Bindung**.

## Virtuelle Funktionen (2)

In C++ gibt es eine Lösung des Problems:

Beim Übersetzen wird an Stelle des statischen Funktionsaufrufs ein Mechanismus eingebaut, der zur Laufzeit prüft, auf welchen Typ der Zeiger tatsächlich zeigt. Dies nennt man **dynamische Bindung**. Damit die dynamische Bindung aber vom Compiler verwendet wird, braucht die Funktion das Schlüsselwort **virtual**.

Das Schlüsselwort **virtual** braucht nur in der Basisklasse verwendet werden.

Die Funktion ist in den abgeleiteten Klassen dann ebenfalls **virtual**.

Ist in der Basisklasse keine Funktion deklariert, muß dort eine Funktion pro forma eingebaut werden.

## Beispiel 3(1)

```
class Employee
{public:
    Employee(const char* name);
    char* getName const;
    virtual float computePay() const;
private:
    char name[30];
};

class WageEmployee : public Employee
{public:
    WageEmployee(const char* name);
    void setWage(float wg);
    void setHours(float hrs);
    float computePay() const;    //implizit virtuell
private:
    float wage;
    float hours;
};
```

## Beispiel 3(2)

```
class SalesPerson(const char* nm):public WageEmployee
{public:
    SalesPerson(const char* name);
    void setCommission(float comm);
    void setSales(float sales);
    float computePay() const;        //implizit virtuell
private:
    float commission;
    float salesMade;
};

class Manager:public Employee
{public:
    Manager(const char* name);
    void setSalary(float salary);
    float computePay() const;        //implizit virtuell
private:
    float weeklySalary;
};
```

## Beispiel 3(3)

```
class EmployeeList
{public:
    int x;
    Employee* Liste[3];
    EmployeeList();
    void add(Employee* newEmp)
        {Liste[x] = newEmp; x++;}
};

int main(void)
{EmployeeList empList; //Liste aller Beschaeftigten
 WageEmployee* wagePtr; //Zeiger auf Lohnempfaenger
 SalesPerson* salePtr; //Zeiger auf Verkaeufer
 Manager* mgrPtr; //Zeiger auf Manager
```

## Beispiel 3(4)

```
//Beschäftigte anlegen
wagePtr = new WageEmployee("A. Houston");
salePtr = new SalesPerson("L. Sprewell");
mgrPtr  = new Manager("P. Ewing");

//Beschäftigte in die Liste eintragen
empList.add(wagePtr);      //Eintrag Lohnempfängers
empList.add(salePtr);      //Eintrag Verkäufers
empList.add(mgrPtr);       //Eintrag Managers

//Gehaltsberechnung
empList.liste[0]->computePay();
empList.liste[1]->computePay();
empList.liste[2]->computePay();
}
```