Praxis der Programmierung

Kopieren von Objekten, abstrakte Klassen, Ausnahmefehler, Namensräume

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

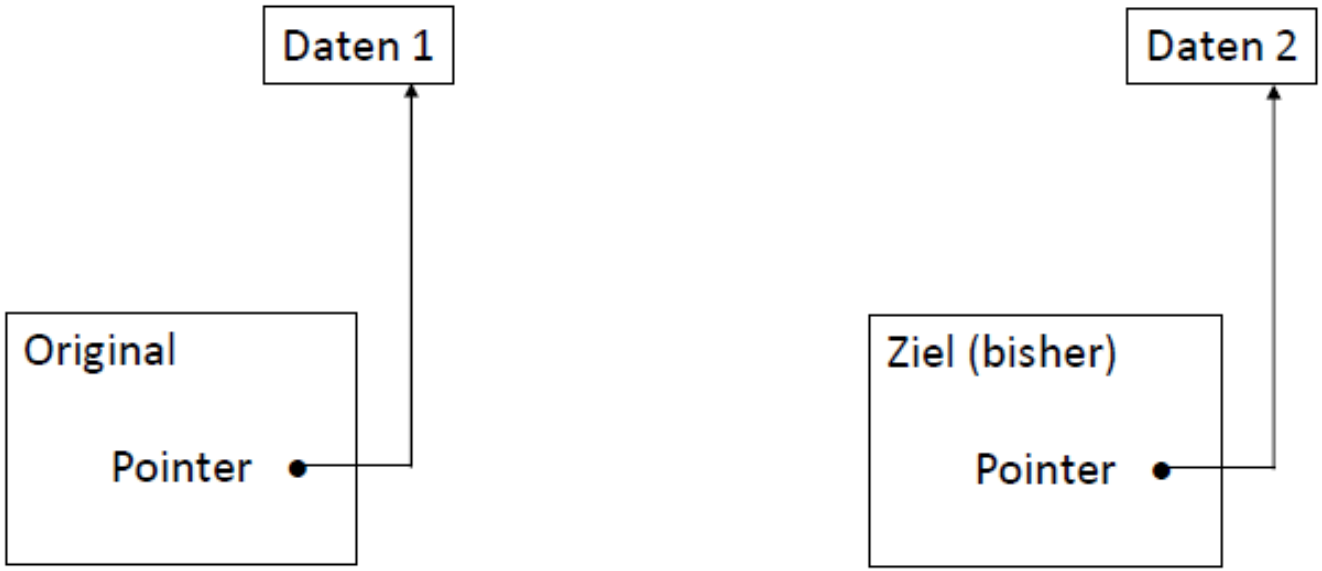
Kopieren von Objeken

Wann werden Objekte kopiert?

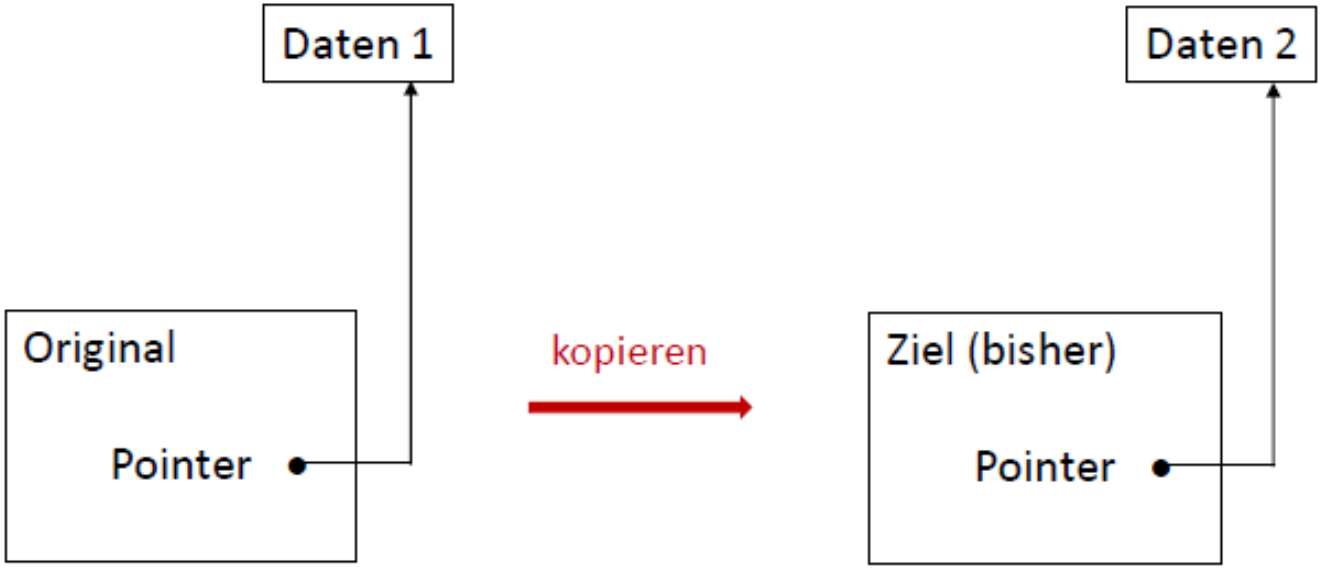
- Zuweisung `objekt1 = objekt2;`
- Objekte als Parameter von Methoden/Funktionen
  (call by value!!!)
- Objekte als Rückgabewerte von Funktionen

Dabei werden Objekte bitweise kopiert.
$\rightsquigarrow$**flache Kopie**

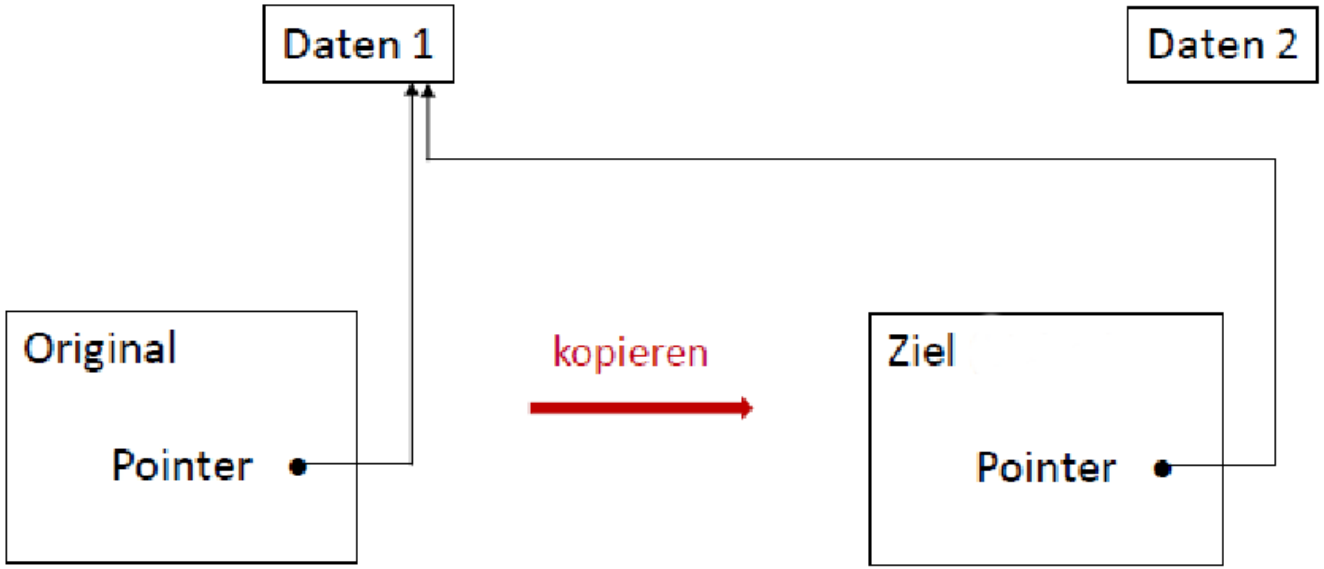
Flache Kopien



Flache Kopien



Flache Kopien



Tiefe Kopie mittels Kopierkonstruktor

neuen Pointer anlegen und Wert an neue Adresse kopieren

```
class Cls {
    int * ptr;   // referenzierte Daten
    int n;       // sonstige Daten
public:
    // Kopierkonstruktor
    Cls(Cls * orig) {
        ptr = new int;
        *ptr = *(orig->ptr);
        n = orig->n;
    }
    void setData(int value) {
        *ptr = value;
    }
};
```

Tiefe Kopie mittels Kopierkonstruktor

neuen Pointer anlegen und Wert an neue Adresse kopieren

```
class Cls {
    int * ptr;   // referenzierte Daten
    int n;       // sonstige Daten
public:
    // Kopierkonstruktor
    Cls(Cls * orig) {
        ptr = new int;
        *ptr = *(orig->ptr);
        n = orig->n;
    }
    void setData(int value) {
        *ptr = value;
    }
};
```

```
int main() {
    Cls obj;
    obj.setData(16);
    Cls flat = obj;
    Cls deep(&obj);
    obj.setData(66);

    // *(obj.ptr) = 66
    // *(flat.ptr) = 66
    // *(deep.ptr) = 16
```

Referenzen und Referenzparameter

- Referenz: Variable, die wie ein *Aliasname* auf ein Speicherobjekt verweist
- keine Adresse (Pointer), sondern ein (zweiter) Name, mit dem auf die Speicherstelle zugegriffen wird
- Referenzparameter: Übergabe der Referenz
  $\rightsquigarrow$ Manipulation am Speicherobjekt möglich
- Definition: *Datentyp* & *Bezeichner*
- Beispiel: `int & reference`

```
void inc(int & n) {
    n++;
}
```

```
int main() {
    int number = 22;
    inc(number);     // number = 23
}
```

Referenz- *versus* Pointerparameter

- Pointer können verändert werden, also später auf andere Speicherobjekte zeigen
- Referenzen können nach der Parameterübergabe nicht mehr auf andere Speicherstellen "umgebogen" werden
- aktueller Parameter bei Referenzen: (dereferenzierte) Variable (kein Pointer!)

$\rightsquigarrow$ Aufrufer sieht nicht, ob die Variable als Wert oder Referenz übergeben wird
$\rightsquigarrow$ *auf mögliche Seiteneffekte achten!!!*

Kopierkonstruktor mit Referenz auf das Original

```
class Cls {
    int * ptr;   // referenzierte Daten
    int n;       // sonstige Daten
public:
    // Kopierkonstruktor
    Cls(Cls & original) {
        ptr = new int;
        *ptr = *orig.ptr;
        n = orig.n;
    }
    void setData(int value) {
        *ptr = value;
    }
};
```

```
int main() {
    Cls obj;
    obj.setData(16);
    Cls flat = obj;
    Cls deep(obj);
    obj.setData(66);

    // *(obj.ptr) = 66
    // *(flat.ptr) = 66
    // *(deep.ptr) = 16
```

Kopierkonstruktor in der Klasse Highscore

```
class HighScore {
   int score;
   Date * date;
public:
   HighScore(HighScore & orig) {
      score = orig.score;
      date = new Date();
      *date = *orig.date;
   }
   ...
}
```

```
int main() {
   HighScore hsc;
   // Daten setzen
   Highscore cpy(hsc);
   /* Daten von hsc und cpy gleich */
   // hsc veraendern
   /* Daten von hsc und cpy
            sind jetzt verschieden */
```

Abstrakte Klassen

Abstrakte Klassen als abstrakte Oberbegriffe

- Quadrate, Kreise, Rechtecke, Dreiecke, ...
  ... sind *ebene Figuren*

- Idee: gemeinsame Oberklasse `Figure`

- aber: keine Exemplare von `Figure` (*abstrakt!*)

Design von `Figure`

- gemeinsame Datenelemente (`Point`)
- gemeinsame Methoden (Getter und ggf. Setter für gemeinsame Datenelemente, `moveTo()`, `moveRel()`)
- gemeinsame Methoden**signaturen** (area(), perimeter(), ...)
  $\rightsquigarrow$ Implementierung??? ... keine!!!
- `virtual double area() = 0;`
  - abstrakte Methoden: virtuelle Methoden, denen 0 zugewiesen wird
  - Implementierung erfolgt in Unterklassen
  - abstrakte Klasse: Klasse mit mindestens einer abstrakten Methode
    $\rightsquigarrow$ keine Exemplare erzeugbar

Ausnahmefehler

Begriff Ausnahmefehler

- Ausnahmefehler sind Laufzeitfehler oder logische Fehler,
  die ein unerwartetes Verhalten oder einen Absturz zur Laufzeit verursachen

- Beispiele:
  - Zugriff auf Array- oder Listenelemente ausserhalb des alloziierten Bereichs
  - Zugriff auf Datenelemente von Instanzvariabeln, die auf kein Objekt zeigen
  - Aufruf von Methoden mit Instanzvariabeln, die auf kein Objekt zeigen
  - Division durch 0
  - Öffnen einer Datei, die nicht voranden ist
  - Lesen aus einer Datei, die während des Zugriffs gelöscht wird

- Ausnahmefehler = Exception

Behandlung von Ausnahmefehlern

- Ausnahmefehler können vom Programmierer *abgefangen* werden
  - ⇝ kein Programmabbruch
  - ⇝ kontrolliertes, vom Programmierer festgelegtes Verhalten im Ausnahmefall
  - ⇝ Trennung von normalem Verhalten und Fehlerbehandlung

```
try {
   // Versuch, das Programm fehlerfrei auszufuehren
   //
   // Anweisungen, die einen Fehler ausloesen koennten
}
catch(...) {  // ... muss dort stehen!
   // Anweisungen, die ausgefuehrt werden,
   // falls ein Ausnahmefehler aufgetreten ist
}
```

Vordefinierte Ausnahmefehler

- in Klassen der Satndardbibliothek
- Basisklasse: `exception`
- mehrere abgeleitete Klassen, *z.B.*
  - `ios_base::failure` – Fehlerklasse der Stream-Klassen
  - `length_error` – maximale Größe wird überschritten
  - `out_of_range` – Zugriff mit unzulässigem Index
  - `bad_alloc` – `new` kann keinen Speicher anfordern
  - ...
- Man kann eigene Unterklassen von `exception` definieren
  $\rightsquigarrow$ Überschreiben der Methode `what()`:
  $\rightsquigarrow$ liefert einen C-String (also `char`-Pointer) mit Fehlerinformationen

Unterscheidung von Ausnahemefehlern

- mehrere `catch`-Blöcke zu einem `try`-Block
- erst spezielle Fehler (Vererbungshierarchie beachten!), dann allgemeinere
- `catch(...)` zuletzt (Behandlung "aller weiteren" Ausnahmefehler)

```
try { // Anweisungen }
catch(MyException&) {
      // Behandlung von Fehlern eines selbst definierten Typs
}
catch(ios::failure&)
      // Behandlung von Fehlern bei Stream-Nutzung
}
catch(...) {
      // wenn sonst etwas schief geht
}
```

`throw`: Ausnahmefehler ohne Fehlerklasse erzeugen

```
void foo(int problem) {
   if (problem > 0)
      throw 0;      // erzeugt eine Exception
   // ...
}

int main() {
   try {
      foo(1);
   }
   catch(...) {
      cout << "Ein Fehler beim Aufruf von foo(int)."
   }
```

Übergabe der Fehlernummer?!

`throw`: Ausnahmefehler ohne Fehlerklasse erzeugen

```
void foo(int problem) {
   if (problem == 1)
      throw 1;      // erzeugt eine Exception
   if (problem == 2)
      throw 2;
   if (problem > 2)
      throw (char *) "message";
}

int main() {
   int n = 0;                    catch(int i)   { // Anweisungen fuer int }
   cin >> n;                     catch(char* s) { // Anweisungen fuer char* }
   try { foo(n); }               catch(...)     { // Anweisungen sonst }
}
```

Namensräume

Erinnerung: Namensräume

- definieren Bereiche, in denen Namen / Bezeichner eindeutig sein müssen
- in verschiedenen Namensräumen kann der gleiche Name verwendet werden
- *Beispiele*:
  - Telefonnummern in Vorwahlbereichen: `0331 123456`
    `030 123456`
  - Telefonnummern mit Vorwahl in Ländernetzen: `+49 30 123456`
    `+36 30 123456`
  - Dateinamen in Ordnern; diese in übergeordneten Ordnern, ...
  - in Netzwerken absolute Pfadnamen auf Hosts (Rechnernamen), ...

Erinnerung: Qualifizierte Namen

- unqualifizierte Namen: die Bezeichner selbst

  Beispiel: `meineDatei`

- qualifizierte Namen: mit Angabe des Namensraums

  Beispiel: `/home/rlehre/meineDatei`

- in **C++**: `namensraum::bezeichner`

  Beispiel: `std::cout`, `std::cin`, `std::endl`

- Namensraum `std` enthält Bezeichner der Standardbibliothek

- Definition eigener Namensräume möglich ...

Benutzerdefinierte Namensräume

- Zuordnung von Definitionen zu einem Namensraum:

```
namespace name {
   void function1();     // Definitionen, die dem Namensraum 'name'
   int function2();      // zugeordnet werden
}
```

- Zugriff mit `name::function1();`

  oder durch

```
using namespace name;
// ...
function1();
```

Anonyme Namensräume

- Namensräume ohne Namen: `namespace { // ... }`
- Zugriffe sind auf Funktionen beschränkt, die in derselben Quelltextdatei definiert sind.
- Alternative: alle Funktionen `static` definieren
  - `static` definierte Funktionen werden dem Linker nicht bekannt gegeben
    $\rightsquigarrow$ stehen nur innerhalb der Quelltetdatei mit ihrer Definition zur Verfügung
  - *aber:* statische Methoden von Klassen sind Klassenmethoden

Beispiel

util.h:

```
namespace util {
    void foo();
}
```

bsp.cpp:

```
#include "util.h"
#include <iostream>
using namespace std;
namespace {
    void myFunction(int a) {
        cout << a << endl;
    }
}
void util::foo() {
    myFunction(4);
}
```