

18.911: Kompaktkurs C++ für Java-Programmierer

3. Tag

Reinhard Zierke

Universität Hamburg, MIN-Fakultät, Informatik-Rechenzentrum

8.-12. Oktober 2007

Übersicht 3. Tag

- 1 Klassen in C++
 - Klassen
 - Felder und Methoden
 - Codeaufteilung
 - Konstruktoren und Destruktoren
 - Überladen von Operatoren
 - Vererbung

Klassendeklaration

```
class Staff : public Person {  
public:                                // öffentliche Methoden  
    Staff();                            // Konstruktor  
    ~Staff();                            // Destruktor  
    void setId(const int id);  
    int getId() const;  
private:                               // klasseninterne Felder und Methoden  
    int id_;  
};
```

- Klassendeklaration ist Typdeklaration: Semikolon am Ende
- `class` ist Erweiterung von `struct` in C
- Zugriffsprivilegien (`public`, `protected`, `private`) nur einmal angeben; gelten für alle folgenden Deklarationen

Zugriff auf Felder und Methoden (2)

- Felder und Methoden des aktuellen Objektes können in der Methode direkt angesprochen werden:

```
void Person::setName(const string &name) {  
    name_ = name;  
}
```

- alternativ Zugriff auf Datenelemente des aktuellen Objektes mit dem Zeiger `this`:

```
void Person::setName(const string &name_) {  
    this->name_ = name_;  
}
```

- Zeiger `this` ist impliziter erster Parameter aller Methoden einer Klasse:

```
void Person::setName(Person *this, const string &name);
```


Definition von Methoden (3): Accessor und Mutator

- Mutator: Methode, die Felder verändern kann
- Accessor: Methode, die keine Felder verändert. Muss als `const` deklariert werden (Hinweis an Compiler zur Codeoptimierung):

```
string Person::getName() const {  
    return name_;  
}
```

- wirkt wie `const` bei implizitem ersten Parameter `this`:

```
string getName(const Person *this) {  
    return this->name_;  
}
```

- Accessor / Mutator ist Bestandteil der Signatur. Es kann beide gleichzeitig mit sonst gleicher Parameterliste geben.

Codeaufteilung: Interface

- Klassendeklaration (Interface) in Header-Datei Person.h;
wird von Klassendefinition und vom Anwender eingebunden:

```
#ifndef PERSON_H_
#define PERSON_H_
class Person {
public:
    Person(string name="", int year=0);
    void setName(const string &name);
private:
    string name_;
};
#endif
```

- `#ifndef`, `#define`: Header wird nur einmal eingebunden
- Faustregel: keinen Speicherplatz reservieren → `extern`

Codeaufteilung: Implementation und Anwendungsprogramm

- Klassendefinition (Implementation) in Datei `Person.cc`, bindet Klassendeklaration ein:

```
#include "Person.h"
...
void Person::setName(const string &name) {
    name_ = name;
}
```

- Anwendungsprogramm benötigt Klassendeklaration und muss Header-Datei `Person.h` einbinden:

```
#include "Person.h"
...
Person p("Sandy_Denny", 1947);
```

Konstruktor

- Konstruktor für Speicherzuweisung und Feld-Initialisierung.
- Hat gleichen Namen wie Klasse und keinen Rückgabetyt.
- Mehrere Konstruktoren mit verschiedenen Signaturen möglich.
- Verschiedene Signaturen auch mit Defaultparametern:

```
class Person {  
    Person(string name="", int year=0);  
    ...  
}
```

- Feld-Initialisierung über Initialisierungsliste:

```
Person::Person(string name, int year)  
    : name_(name), year_(year) {  
}
```

Default-Konstruktor

- Default-Konstruktor ist der Konstruktor, der ohne Argumente aufgerufen wird.
- Ist nur dann automatisch verfügbar, wenn kein (anderer) Konstruktor definiert wird.
- Hat entweder keine Argumente:

```
Person::Person()  
    : name_(""), year_(0) {  
}
```

- oder ist Konstruktor mit Defaultwerten für alle Argumente:

```
Person(string name="", int year=0);  
...  
Person::Person(string name, int year)  
    : name_(name), year_(year) {  
}
```

Kopier-Konstruktor

- Ein Konstruktor mit einem Parameter gleichen Typs heißt Kopier-Konstruktor:

```
Pic3D::Pic3D(const Pic3D &pic)
    : sx_(pic.sx), sy_(pic.sy), sz_(pic.sz) {
    copy(*bitmap_, pic->bitmap_);
}
```

- Kopier-Konstruktor muss nicht explizit programmiert werden, aber:
- Default-Kopier-Konstruktor kopiert „flach“: Referenzen werden direkt kopiert, aber nicht die Daten, auf die diese zeigen.
- Für „tiefe“ Kopien muss der Kopier-Konstruktor selbst programmiert werden.

Initialisierung von Objekten

- Aufruf des Default-Konstruktors:

```
Person p1;  
Person func(); // Kein Konstruktor, Funktionsdeklaration !
```

- Aufruf eines Konstruktors mit mehreren Argumenten:

```
Person p2("Sandy", "Denny", 1947);
```

- Aufruf des Kopier-Konstruktors:

```
Person p3(p2);
```

- Für Zeigervariablen muss ausdrücklich Speicherplatz bereitgestellt (**new**) und freigegeben (**delete**) werden:

```
Person *pp = new Person;  
(*pp).name = 'Otto'; pp->year = 1999;  
delete pp; pp = 0;
```

Destruktor

- wird aufgerufen, wenn ein Objekt gelöscht wird:

```
Pic3D::~~Pic3D() {  
    delete [] bitmap_;  
}
```

- Destruktor muss nicht explizit programmiert werden, aber:
- Default-Destruktor löscht „flach“: Referenzen werden direkt gelöscht, aber nicht die Daten, auf die diese zeigen.
⇒ Speicherlecks!
- Für „tiefes“ Feld-Löschen muss der Destruktor selbst programmiert werden.

Überladen von Operatoren

- In C++ sind Operatoren Methodenaufrufe und können „überladen“ werden: neue Funktionalität
- Beispiel: Klassen mit dynamischen Feldern brauchen einen eigenen Zuweisungs-Operator `operator=`
- Beispiel: Klassen mit arithmetischen Funktionen brauchen überladene Rechenoperationen zur intuitiven Nutzung, z.B. `operator+`, `operator+=`, ...
- aber: nur bestehende Operatoren überladbar,
- aber: Priorität und Assoziativität der Operatoren nicht veränderbar.

Zuweisungs-Operator

- Bei Zuweisung von Objekten

```
Pic3D p1(50, 50, 50), p2;  
p2 = p1;
```

wird der Zuweisungs-Operator aufgerufen:

```
Pic3D& Pic3D::operator=(const Pic3D &pic) {  
    sx_ = pic.sx; sy_ = pic.sy; sz_ = pic.sz;  
    copy(*bitmap_, pic->bitmap_);  
    return *this;  
}
```

- Default-Zuweisungs-Operator kopiert „flach“: Zeiger werden direkt kopiert, aber nicht die Daten, auf die diese zeigen.
- Für „tiefe“ Kopien muss der Zuweisungs-Operator selbst programmiert werden.

Ausgabe von Objekten

- Ausgabe von Objekten durch Überladen von << :

```
ostream& operator<<(ostream &os, const Person &rhs) {  
    return rhs.print(os);  
}
```

```
ostream& Person::print(ostream &os) const {  
    os << name_ << ", " << year_;  
    return os;  
}
```

Beispiel komplexe Zahlen

- Klassendefinition:

```
class complex {
public:
    complex(double=0., double=0.);
    void setReal(const double);
    double getReal() const; ...
    complex operator+=(const complex&);
    complex operator+(const complex&) const;
    ostream& print(ostream&) const;
private:
    double real_, im_;
};
// complex operator+(const complex&, const complex&);
ostream& operator<<(ostream&, const complex&);
```

Komplexe Addition

- Operator +=
- Definition:

```
complex complex::operator+=(const complex &rhs) {  
    real_ += rhs.real_;  
    im_   += rhs.im_;  
    return *this;  
}
```

- Beispiel:

```
complex a(1.,1.), b(0.,1.);  
a += b += complex(1.,1.);
```

Komplexe Addition (2)

- Operator +
- Definition: Zurückführung auf Operator += :

```
complex complex::operator+(const complex &rhs) {  
    complex result(*this);  
    result += rhs;  
    return result;  
}
```

- Beispiel:

```
complex a, b(0.,1.);  
a = b + 1.; // a = (1.,1.)  
a = 1. + b; // error: no match for 'operator+' in '1. + b'
```

Komplexe Addition (3)

- Operator +
- bessere Definition, symmetrisch:

```
complex operator+(const complex &lhs,  
                  const complex &rhs) {  
    complex result(lhs);  
    result += rhs;  
    return result;  
}
```

- Beispiel:

```
complex a, b(0.,1.);  
a = b + 1.; // a = (1.,1.)  
a = 1. + b; // a = (1.,1.)
```

Vererbung

- Vererbung drückt eine „ist eine“-Beziehung aus
- Erweiterung oder Spezialisierung vorhandener Klassen, ohne diese verändern zu müssen
- „Interface“-Definition (`virtual` class) möglich
- Ableitung von bestehender Klasse durch Operator :
(entspricht `extends` in Java) :

```
class Staff : public Person {  
public:  
    Staff(string name="", int year=0, int id=0);  
    void setId(int id);  
    int getId() const;  
    void print() const;  
private:  
    int id_;  
};
```

Konstruktor

- Konstruktor der Basisklasse wird in der Initialisierungsliste angegeben:

```
Staff::Staff(string name="", int year=0, int id=0)
    : Person(name, year), id_(id) {
}
```

- Wenn kein Basisklassen-Konstruktor angegeben, wird automatisch deren Defaultkonstruktor aufgerufen

Slicing

- Wenn ein Objekt einer abgeleiteten Klasse in ein Objekt der Basisklasse kopiert wird, werden nur die Basisklassen-Felder kopiert ("Slicing"):

```
Staff s("Reinhard_Zierke", 1958, 4711);  
Person p = s;
```

- Das gleiche passiert bei call-by-value-Übergabe von Objekten:

```
ostream& operator<<(ostream& out, Person p) {  
    p.print(out);  
    return out;  
}  
  
...  
cout << s << endl;
```

Vermeiden von Slicing

Zum Vermeiden von Slicing muss man Objekte aus Klassenhierarchien wie in Java behandeln:

- Alle Methoden werden als `virtual` deklariert
- Nur Objekte mit `new` erzeugen
- Alle Objekte nur mit Zeigern ansprechen
- Collections von Zeigern auf Objekte benutzen

Mehrfachvererbung

- Java** Jede Klasse erbt von genau einer Klasse, ggfs. Object
- Klassenstruktur: Baum mit Object an der Wurzel
- C++** Mehrfachvererbung: eine Klasse erbt von beliebig vielen (auch keinen) Klassen
- Klassenstruktur: zyklensfreie gerichtete Graphen
 - Konflikt von gleichnamigen Feldern oder Methoden möglich

Virtuelle Methoden

- drücken aus, dass eine Methode in abgeleiteten Klassen
 - überschrieben werden kann ...
 - oder überhaupt erst zu implementieren ist („pure **virtual**“ Methode)
- ⇒ Polymorphie, late binding

C++ Damit eine Methode in einer abgeleiteten Klasse überladen werden darf, muss sie ausdrücklich als **virtual** deklariert werden (Laufzeit-Effizienz).

Java Alle Methoden sind virtuell; „pure **virtual**“ Methoden heißen „abstract“.

Polymorphie

- Objekt einer abgeleiteten Klasse kann statt eines Objekts der Basisklasse verwendet werden
- Im Programmcode ist u.U. der definierte Typ des Objekts gar nicht mehr sichtbar
- Laufzeit-Polymorphie: aufgerufene Methode abhängig vom definierten Typ des Objekts,
 - wenn die Methoden `virtual` sind ...
 - und sie in Basis- und abgeleiteter Klasse die gleiche Signatur haben ...
 - und sie über Pointer- oder Referenz-Variablen aufgerufen werden.
- Wichtig: Sobald eine Methode virtuell ist, muss auch der Destruktor virtuell sein.
- Bemerkung: weitere Polymorphie-Arten: statische Polymorphie (Überladen), parametrische Polymorphie (Templates)

Beispiel für virtuelle Methode

```
class Person {
    ...
    virtual void print() const;
}
class Staff : Person {
    Staff (string name="", int year=0, int id=0);
    ...
    virtual void print() const;
}
int main() {
    Person *p = new Staff("Reinhard_Zierke", 1958, 4321);
    p->print();
}
```

Pure virtual Methoden

- Eine virtuelle Methode ohne Implementierung wird durch Zuweisung von 0 als „pure **virtual**“ Methode deklariert:

```
class Person {  
    ...  
    virtual void print() const = 0;  
}  
class Staff : Person {  
    ...  
    virtual void print() const;  
}  
void Staff::print() const {  
    ...  
}
```

Nutzung abgeleiteter Klassen

„**Upcasting**“: Zuweisung eines Objektes an Objekt der Basisklasse.
Die zusätzlichen Felder und Methoden der abgeleiteten Klasse gehen verloren:

```
Staff st("Reinhard_Zierke", 1958, 4321);  
Person p = st;  
cout << p.getName() <<endl; // OK  
cout << p.getId() <<endl; // Fehler!
```

„**Downcasting**“: Laufzeit-Polymorphie geht nur mit virtuellen Methoden und Zeigervariablen oder Referenzen.

```
Staff* pst("Reinhard_Zierke", 1958, 4321);  
Person* pp = pst;  
pp->print(); // Staff . print ()  
cout << dynamic_cast<Staff*>(pp)->getId() <<endl;
```