

Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

20. Dezember 2005

C++: Klassen mit Wertsemantik

- C++ unterstützt kleine, konkrete, benutzer-definierte Typen
 - Klassen ohne Speicher-Overhead
 - Effizienz: Inlining von Funktionen
 - Überladung von Operatoren für vertraute Syntax
 - Verwaltung des Lebenszyklus
- Wertsemantik: Funktionsaufruf und Zuweisung kopieren Werte
- Lebenszyklus von Werten / Variablen
 - Anlegen & Initialisieren
 - Zuweisung
 - Kopie
 - Freigabe

C++: Klassen mit Wertsemantik

- Compiler ruft spezielle Funktionen auf
 - Konstruktoren (für Initialisierung)
 - `operator=` (für Zuweisung)
 - Copy-Konstruktor (für Kopien)
 - Destruktor (zur Freigabe)
- ⇒ Programmierer kann für jeden Typ diese Operationen explizit definieren

The Rule of Three

A. Koenig, B. Moo: *Accelerated C++*, Addison-Wesley, 2000

§11.3.6 **Classes that allocate resources** in their constructors require that every copy deal correctly with those resources. Such classes almost surely need a **destructor** to free the resources. If the class needs a destructor, it almost surely needs a **copy constructor**, as well as an **assignment operator**. Copying or assigning objects of classes that allocate resources usually allocates those resources in the same way that creating an object from scratch does. [. . .] Because the copy constructor, destructor, and assignment operator are so tightly coupled, the relationship among them has become known as the **rule of three**: If your class needs a destructor, it probably needs a copy constructor and an assignment operator, too.

Nachtrag: friends

- Zugriffsschutz: `public:`, `private:` Abschnitte
 - Wollen: Globale Funktionen für neue Typen (Operatoren!)
 - Diese gehören zur Implementierung des Types
- ⇒ Sie sollen Zugriff auf `private:` Abschnitte bekommen
- Schlüsselwort `friend` führt Klassen und Funktionen auf
 - `friend` Klassen und Funktionen haben Zugriff auf `private` Daten

Beispiel: Die Klasse Num

```
class Num {
private:
    int n;
public:
    Num(int n) : n(n) { }           // Initialisierungslisten
    Num(const Num &b) : n(b.n) { }  // Copy-Konstruktor
    Num &operator=(const Num &b) {  // Zuweisung
        if (this != &b) {         // Schutz vor Selbstzuweisung
            n = b.n;
        }
        return *this;             // Rückgabe
    }
    friend Num operator+(const Num &a, const Num &b);
    friend ostream &operator<<(ostream &out, const Num &b);
};
Num operator+(const Num &a, const Num &b) {
    return Num(a.n + b.n);        // Zugriff auf private Daten
}
ostream &operator<<(ostream &out, const Num &b) {
    return out<<b.n;              // Zugriff auf private Daten
}
```

Vererbung

```
class A {  
public:  
    int a;  
};  
class B : public A {  
public:  
    int b;  
};
```

- B ist **abgeleitete Klasse (derived class)** von A
- A ist **Basisklasse (base class)**
- Lesart: Ein B-Wert enthält am Anfang einen A-Wert \approx

```
class B { public: A a; int b; };
```

- Einfacherer Zugriff auf A-Felder
- Ein B Wert gilt auch als ein A-Wert \rightarrow Konversion $B^* \rightarrow A^*$ möglich
- Prinzip: Die Basisklasse gilt als “besondere member variable”

Zugriffsschutz mit `protected`:

- `private`:-Abschnitt ist für Klasse selbst sichtbar
- Stroustrup: Programmierer soll nicht einfach dadurch Zugriff bekommen, dass er von einer Klasse erbt
- Kompromiss: `protected`: Abschnitt ist für abgeleitete Klassen sichtbar
- Hinweis: Bei Vererbung `public` angeben

```
class A {  
protected:  
    int a;  
};  
class B : public A {  
protected:  
    int b;  
};
```

Zugriffschutz auf Basisklasse

- Basisklasse ist “besondere member variable”
→ müssen Sichtbarkeit regeln
- ⇒ Deklariere bei Ableitung auch die Sichtbarkeit der Basisklasse (§15.3.2)
- Zugriff analog zu `public` / `protected` / `private` member variable

	B : public A	B : protected A	B : private A
Zugriff auf			
A private	nur A	nur A	nur A
A protected	A, B, B ↓	A, B, B ↓	A, B
A public	Alle	A, B, B ↓	A, B
Cast B* → A*	Alle	B, B ↓	B

B ↓ sind die von B abgeleiteten Klassen

Mit einer Klasse haben auch alle ihre friends Zugriff

Verdecken von Member Functions

- Member functions können in abgeleiteten Klassen neu definiert werden
- Sie verdecken **alle** überladenen Funktionen mit gleichem Namen
- Achtung: Differenz zu Java
 - Dieser Vorgang heißt noch nicht “Überschreiben”
 - Member functions werden nach **statischem Typ** ausgewählt (kein dynamic dispatch)
 - Gruppierung in Klassen ist nur Aufteilung des Namensraumes
- Prinzip: In C++ zahlt man nur für das, was man auch bestellt.

Verdeckung rückgängig machen

```
struct A {
    void f(int i) {
        cout<<"A::f " <<i<<endl;
    }
};
struct B : A {
    using A::f;
    void f() {
        cout<<"B::f"<<endl;
    }
};
int main() {
    B b;
    b.f();
    b.f(4);
}
```

- Klassen sind auch Namensräume (→ namespace)
- Überladungsauflösung nur innerhalb eines Namensraums
- `using` macht Deklarationen aus anderen Namensräumen sichtbar

super-Zugriffe in C++

- Es gibt kein Schlüsselwort `super` in C++
 - Ziel von `super`-Aufrufen kann statisch ermittelt werden (Blatt 9)
- ⇒ Der Scope-Operator `::` erledigt bereits das Gewünschte

```
struct A {  
    void f() { ... }  
};  
struct B : A {  
    void f() { A::f(); ... }  
};
```

- Allgemeine Syntax für Zugriff auf Member m von Klasse C in Objekt x

$$x.C :: m$$

- Im Beispiel: `this` wird implizit übergeben

Abgeleitete Klassen als Werte

- Prinzip: Die Basisklasse gilt als “besondere member variable”

⇒ Verständnis von

- Konstruktion
- Zuweisung
- Kopie
- Destruktion

anders als in Java

Konstruktion

- Erinnerung: Initialisierungslisten
 - Rufen Konstruktoren von member variables auf
 - Abarbeitung in Reihenfolge der Deklaration
 - Erweiterung auf Basisklassen
 - Name der Basisklasse in Initialisierungsliste ruft deren Konstruktor auf
 - Basisklassen werden vor member variables initialisiert
 - Einschränkung: Abgeleitete Klassen dürfen keine
 - Members von Basisklassen initialisieren
 - Basisklassen von Basisklassen initialisieren
- ⇒ Prinzip: Für die Initialisierung der Basisklassen sind diese selbst zuständig

Beispiel: Konstruktoren

```
class A {
protected:
    int a;
    A(int i) : a(i) { cout<<"Construct A " <<*this<<endl; }
    A(const A&y) : a(y.a) { cout<<"Copy A " <<*this<<endl; }
    ...
};
class B : public A {
protected:
    int b;
public:
    B(int i, int j) : A(i), b(j) {
        cout<<"Construct B " <<*this<<endl;
    }
    B(const B&y) : A(y), b(y.b) {
        cout<<"Copy B " <<*this<<endl;
    }
    ...
}
```

Destruktoren

- Compiler ruft Destruktoren für member variables auf
 - Die Reihenfolge ist genau umgekehrt wie bei Konstruktoren
 - Basisklasse ist nur besondere member variable
- ⇒ Destruktoren von Basisklassen werden automatisch aufgerufen, nachdem
- Der Rumpf des Destruktors abgelaufen ist
 - Alle anderen member variables zerstört sind

Beispiel: Destruktoren

```
class A {  
public:  
    ~A() { cout<<"Destructor A " <<*this<<endl; }  
    ...  
};  
class B : public A {  
public:  
    ~B() { cout<<"Destructor B " <<*this<<endl; }  
    ...  
};
```

- Bei Freigabe eines B Wertes
 - Destruktor-Rumpf von B
 - (Destruktoren für member variables von B)
 - Destruktor-Rumpf von A
 - (Destruktoren für member variables von A)

Zuweisung

- Der Zuweisungsoperator muß überschrieben werden
 - `A::operator=(const A&)` kopiert einen A-Wert
 - `B::operator=(const B&)` kopiert B-Wert = A-Wert + b-Variable
- Der neue Operator soll den alten verwenden
- Hinweis: Wenn die members von A `protected:` sind, könnte `B::operator=` auch direkt auf `A::a` zugreifen, das ist aber schlechter Stil, da die Basisklasse für ihre eigene Zuweisung zuständig ist

Beispiel: Zuweisungsoperator

```
A &operator=(const A&y) { // Deklaration in class A
    cout<<"Assignment A " <<*this<<endl;
    if (this != &y)
        a = y.a;
    return *this;
};
```

```
B &operator=(const B&y) { // Deklaration in class B
    cout<<"Assignment B " <<*this<<endl;
    if (this != &y) {
        A::operator=(y);
        b = y.b;
    }
    return *this;
};
```

- operator= ist member function
- ⇒ this wird implizit an operator= übergeben → Methodenaufruf
- Scope operator in A::operator= ersetzt das super-Schlüsselwort

Slicing

- A gilt als Supertyp von B
 - B-Zeiger (Referenzen) werden in A-Zeiger (Referenzen) konvertiert
 - B-Werte können als A-Werte verwendet werden
- “Ay, there’s the rub”

```
void f(A a) {  
    cout<<"f has: "<<a<<endl;  
};  
int main() {  
    B b(4,5);  
    f(b);  
}
```

- Aufruf Copy-Konstruktor `A(const A&)` bei Aufruf von `f()`
 - `b` ist Typ `B&` in Aufruf `f(b)`
 - Compiler konvertiert `B&` in Basisklasse `A&`
- ⇒ Kopiert nur der A Anteil von B → [slicing](#)

Beispiel: Slicing

```
void f(A a) {  
    cout<<"f has: " <<a<<endl;  
};  
void g(B b) {  
    cout<<"g has: " <<b<<endl;  
};  
int main() {  
    B b(4,5);  
    g(b);  
    f(b);  
}
```

```
> ./a.out  
Construct A a=4  
Construct B a=4, b=5  
Copy A a=4  
Copy B a=4, b=5  
g has: a=4, b=5  
Destructor B a=4, b=5  
Destructor A a=4  
Copy A a=4  
f has: a=4  
Destructor A a=4  
Destructor B a=4, b=5  
Destructor A a=4
```

- Die Funktion `f` arbeitet auf einer Kopie eines Teils von `b`
- ⇒ Informationen gehen verloren
- ⇒ A Funktionen bearbeiten B-Anteil → Überschriebene Funktionen vergessen
- ⇒ Invarianten von B gelten, aber auch alle von A?

Virtuelle Funktionen

- Member functions können `virtual` deklariert werden
- Virtuelle Funktionen (und nur diese!) heißen auch **Methoden**
- Klassen mit virtuellen Funktion heißen auch **polymorphic types**
- Dynamischer Typ bestimmt ausgeführten Code
- Implementierungsmodell: Virtuelle Tabellen
- Objekte von polymorphen Typen belegen 1 Wort mehr Speicher (→ Zeiger auf virtuelle Tabelle)
- (Virtuelle Tabellen sind sehenswert: Schon in *A Tour of C++* (§2.5.5))

Überschreiben von Virtuellen Funktionen

§12.2.6 A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to **override** the base class version of the virtual function.

- Nur virtuelle Funktionen, also Methoden, können überschrieben werden
- Grundsatz: “Einmal virtuell, immer virtuell”
- ⇒ Überschreibende Funktion ist implizit `virtual`
- Aufruf von virtuellen Funktionen mit dynamic dispatch
- Ausnahme: Angabe der gewünschten Klasse über Scope-Operator
- ⇒ `super`-Schlüsselwort kann wieder durch `Scope` ersetzt werden

Dynamic Dispatch und Slicing

- Dynamic dispatch wird nur verwendet, wenn
 - eine virtuelle Funktion aufgerufen wird
 - der Zugriff auf das Objekt über einen Zeiger (eine Referenz) erfolgt
- In allen anderen Fällen wird die Funktion des statischen Typs aufgerufen
 - Bei Zuweisung werden abgeleitete Klassen konvertierte (→ Slicing)
 - ⇒ Für normale Variablen ist der statische Typ auch der dynamische Typ.
- ⇒ Für Objekt-orientierte Programmierung müssen Objekte per Referenz (per Zeiger) verwaltet werden
- (Durch Adress-Operator & bzw. Referenztypen müssen Objekte aber nicht auf dem Heap angelegt werden.)

Beispiel

```
struct A {
    virtual void f(int i) {
        cout<<"A::f " <<i<<endl;
    } };
struct B : A {
    void f(int i) {
        cout<<"B::f " <<i<<endl;
        cout<<"super: ";
        A::f(i);
    } };
void call_dynamic1(A *a) { a->f(42); }
void call_dynamic2(A &a) { a.f(42); }
void call_static(A a) { a.f(42); }
int main() {
    B b;
    call_dynamic1(&b);
    call_dynamic2(b);
    call_static(b);
}
```

```
> ./a.out
B::f 42
super: A::f 42
B::f 42
super: A::f 42
A::f 42
```

Pure Virtual Funktions

- Methoden ohne Implementierung: `virtual ... =0`
- `pure virtual functions`
- Webster's zu "pure": 2b) Syn. für "abstract"
- `abstract classes` sind Klassen mit `pure virtual functions`
- Abstrakte Klassen können nicht instanziiert werden

Virtueller Destruktor

```
class A {
public:
    A() { cout<<"Constructor A"<<endl; }
    ~A() { cout<<"Destructor A"<<endl; }
};
class B : public A {
public:
    B() { cout<<"Constructor B"<<endl; }
    ~B() { cout<<"Destructor B"<<endl; }
};
int main() {
    A *a = new B();
    delete a;
}
```

> ./a.out
Constructor A
Constructor B
Destructor A

- Problem: delete ruft Destruktor nach statischem Typ auf
- Lösung: Destruktor muß auch virtuell sein!

Virtueller Destruktor

```
class A {
public:
    A() { cout<<"Constructor A"<<endl; }
    virtual ~A() {
        cout<<"Destructor A"<<endl;
    }
};
class B : public A {
public:
    B() { cout<<"Constructor B"<<endl; }
    virtual ~B() {
        cout<<"Destructor B"<<endl;
    }
};
```

> ./a.out
Constructor A
Constructor B
Destructor B
Destructor A

⇒ Prinzip: Für OOP in C++ sind Destruktoren virtuell

- Achtung: Brauchen evtl. leeren virtuellen Destruktor schon in Basisklassen, die noch gar keine Ressourcen allozieren (Man kann auch pure virtual destructors deklarieren)

Laufzeit-Typinformation §15.4

- Operator: `dynamic_cast<T*>(p)`
 - p ist Zeiger auf Klassentyp
 - Ergebnis:
 - Wenn dynamischer Typ von p Subklasse von T ist:
→ Zeiger auf Instanz von T innerhalb von Objekt $*p$
 - Ansonsten: 0 (*keine Exception!*)
 - (Ergebnis ist nicht notwendig p selbst → Mehrfachvererbung)
 - Implementierung (§15.4.1)
 - Laufzeit-Typinformation
 - Tabellen von Super-Typen in virtueller Tabelle
 - Suche nach gewünschter Typinformation
- `instanceof` : Teste Ergebnis auf 0

Mehrfachvererbung

- C++ erlaubt mehrere Basisklassen ([multiple inheritance](#))
- Abgeleitete Klasse
 - hat alle Basisklassen als “besondere member variables”
 - erbt alle Members aus allen Klassen
 - kann alle virtuellen Funktionen aus allen Klassen überschreiben
 - hat alle Basisklassen als Supertypen
- Virtuelle Funktionen wie bei einfacher Vererbung
- Zugriffsschutz für jede Basisklasse einzeln
- Häufige Anwendung:
 - `public`-Basisklassen erweitern die Schnittstelle
 - `protected/private`-Basisklassen erben Implementierung

Auflösung von Namenskonflikten

- Situation: Mehrere Basisklassen definieren dieselbe member function
- Regel in C++: Namenskonflikte müssen explizit aufgelöst werden
- Namenskonflikte entstehen aus zwei Gründen
 - Zugriff auf Member
 - Zwei Definitionen einer virtuellen Funktion
 - Füllen der virtuellen Tabelle
- Lösung
 - Verwendung des Scope-Operators ::
 - Erneutes Überschreiben der strittigen Funktion

Beispiel: Namenskonflikt

```
class A1 {
public:
    void f() { cout<<"A1::f " <<endl; }
};

class A2 {
public:
    void f() { cout<<"A2::f " <<endl; }
};

class B : public A1, public A2 { };

int main() {
    B b;
    // b.f(); ==> error: request for member 'f' is ambiguous
    b.A1::f(); // disambiguated
}
```

Beispiel: Namenskonflikt

```
class C : public A1, public A2 {
public:
    void f() {
        cout<<"C::f -> ";
        A1::f();
        cout<<"C::f -> ";
        A2::f();
    }
};
int main() {
    C c;
    c.f();
}
```

- Explizite Auflösung für Namen ist umständlich

⇒ Konflikten vorbeugen durch Verdecken / Überschreiben

Interfaces in C++

- Java-Interfaces enthalten Methoden ohne Code
 - In C++: abstrakte Klassen mit pure virtual functions
 - Mehrfachvererbung → mehrere Interfaces
 - Gleichzeitig Vererbung für Implementierung
 - Stroustrup: Mehrfachvererbung bleibt verständlich, wenn nur eine Basisklasse wirklich member functions implementiert und die anderen nur pure virtual functions deklarieren.
- ⇒ Semantik ist klar: Implementierung ergibt sich wie bei Einfachvererbung

Beispiel: Observer-Pattern

```
class subject; // forward declaration
struct change_listener { // interface
    virtual void notify_change(subject *obj) = 0;
};
class subject {
    typedef vector<change_listener*> listeners_type;
    listeners_type listeners;
    int id;
    int data;
public:
    subject(int id) : id(id), data(0) { }
    void set_data(int d);
    int get_id() { return id; }
    int get_data() { return data; }
    void add_change_listener(change_listener *l);
    void fire_change();
};
void subject::fire_change() {
    for (listeners_type::iterator i=listeners.begin();
         i!=listeners.end();++i)
        (*i)->notify_change(this); }
```

Beispiel: Observer-Pattern

```
class text_field { // this should be a real GUI class
public:
    void set_text(string s) { cout<<"TEXT: "<<s<<endl; }
};
class display_change : protected text_field,
                       public change_listener {
public:
    virtual void notify_change(subject *obj);
};
void display_change::notify_change(subject *s) {
    ostringstream buf;
    buf<<"Have been informed about "<<s->get_id()
        <<": Data is "<<s->get_data();
    set_text(buf.str());
}
```

- display_change erbt seine Implementierung von text_field
- Es implementiert das Interface change_listener
- Es muß alle pure virtual functions überschreiben

Für Liebhaber: Pointers to Members

```
struct A {
    int j;
    A(int j) : j(j) { }
    void f(int i) { cout<<"A::f "<<i<<"(j = "<<j<<)"<<endl; }
};
typedef void (A::* A_f_ptr)(int);
void call_it_A(A_f_ptr f) {
    A a(3);
    (a.*f)(5);
}
int main() {
    call_it_A( &A::f);
}
```

- Pointers to members erlauben indirekten Zugriff auf Members
- Zugriffscode hängt ab vom Klasse $\rightarrow C::*$ ersetzt einfachen $*$
- Verwendung $x.*p$ beziehungsweise $x->*p$
- $*$ ist wieder Dereferenz, diesmal des Member-Zeigers

Zusammenfassung

- Vererbung in C++ orientiert sich an Wertsemantik für Objekte
 - Konstruktion
 - Copy
 - Destruktion
- Prinzip: Die Basisklasse ist eine besondere member variable
- Objekt-orientierte Programmierung in C++
 - Virtuelle Funktionen \approx Methoden
 - Virtuelle Funktionen können überschrieben werden
 - Dynamic dispatch nur bei virtueller Funktion auf Referenz / Zeiger
- Bemerkung: Ansatz der Vorlesung
 - Alle C++-Konzepte auf Java Konzepte zurückgeführt
 - Funktionierte, weil wir Java so detailliert besprochen hatten