Programmierkurs C/C++

"Freitag"

Sandro Andreotti
andreott@inf.fu-berlin.de
WS 2010/11
(Slides: Andreas Döring)

Vererbung

Klassen „erben“ die Member von Basisklassen:

```
struct Tier
{
   int gewicht;
   void fuettern() { ++gewicht; }
};

struct Goldfisch: Tier { int wert; };

Goldfisch glupschi;
glupschi.gewicht = 100;
glupschi.fuettern();
```

Im Beispiel wird Tier eine „Basisklasse“ von Goldfisch genannt; Goldfisch ist von Tier „abgeleitet“.

Vorsicht: Bei Vererbung von Klassen, die mit „class“ definiert worden sind, wird default-mäßig **private** vererbt. In diesem Fall müsste man z.B. so schreiben:

class Goldfisch: public Tier ...

Member überladen

Die Member-Definitionen in der abgeleiteten Klasse überladen die der Basisklassen:


```
struct Auto
{
   void tanken() { std::cout << "Benzin"; }
};

struct Solarmobil: Auto
{
   void tanken() { std::cout << "Sonne"; }
};
```

Wenn nun **mob.tanken()** von einem Object **Solarmobil mob** aufgerufen wird, wird „Sonne“ ausgegeben.

Auswahl bei Überladung

...

```
Solarmobil mob;
mob.tanken();          //Ausgabe "Sonne"
```

Zugriff auf die Basisklassen-Funktion:

```
mob.Auto::tanken();    //Ausgabe "Benzin"
```

Alternative:

```
Auto * aut = & mob;
aut->tanken();         //Ausgabe "Benzin"
```

Base kennt Derived nicht! (1)

Das Problem:


```
struct Auto
{
   void tanken() { printf("Benzin"); }
   void starten()
   {
       if (Tankfuellung == 0) tanken();
       ...
   }
};
struct Solarmobil: Auto
{
   void tanken() { printf("Sonne"); }
};
```

Base kennt Derived nicht! (2)


```
...
Solarmobil mob;
mob.starten();
```

Ausgabe: **`Benzin !!!`**

Grund: Die Basisklasse **`Auto`** kennt die abgeleitete Klasse **`Solarmobil`** nicht.

```
void starten()
{
    tanken();
}
```

⇐ this-Pointer hat Typ **`Auto *`**

Lösung: virtuelle Funktionen

Lösungsmöglichkeit: **`virtual`**


```
struct Auto
{
   virtual void tanken() { ...}
};

...

Solarmobil mob;
mob.starten();    //Ausgabe: "Sonne"
```

virtual hat vielfältige Nachteile: Das Programm wird dadurch langsamer und die Objekte verbrauchen mehr Speicher (mindestens +4 Bytes pro Objekt).

Aus der Sicht von C++ sind in Java alle Funktionen virtual (außer sie sind final); aus der Sicht von Java sind alle Funktionen in C++ final (außer sie sind virtual).

Private Basisklassen

Die Member von privaten Basisklassen sind nur den Member-Funktionen der abgeleiteten Klasse zugänglich:

```
struct Geldboerse { int Betrag; };

struct Panzerschrank: private Geldboerse
{
   void ausrauben() { Betrag = 0; }
};

Panzerschrank p;
p.Betrag = 10;          //Fehler!
```

Man beachte, dass die Member-Variable **Betrag** in **Geldboerse** noch **public** deklariert ist. Erst durch die private Vererbung ändert sich das: Bei einem Objekt vom Typ **Panzerschrank** ist **Betrag** nicht mehr **public**, sondern **private**.

Auch **p.Geldboerse::Betrag = 10;** ist verboten.

Bei Klassen, die mit dem Schlüsselwort **class** statt **struct** erzeugt werden, sind die Basisklassen ohne Angaben des Schlüsselwortes **public** automatisch **private**.

Multiple Vererbung

Eine Klasse kann mehrere Basisklassen haben:


```
struct Fahrzeug { int Geschwindigkeit; };
struct Wertgegenstand { int Wert; };

struct Ferrari:
   Fahrzeug,
   Wertgegenstand
{ };

Ferrari meinAuto;
meinAuto.Geschwindigkeit = 280;
meinAuto.Wert = 280000;
```

Mehrdeutigkeiten

Vorsicht vor Mehrdeutigkeiten bei multipler Vererbung! Bsp.


```
struct Hamster
{
    void fuettern();
};
struct Polster
{
    void fuettern();
};

struct Hamstersofa: Hamster, Polster
{ };

Hamstersofa s;
s.fuettern(); //Error!
```

Objekte kopieren

Es gibt zwei Fälle, in denen Objekte kopiert werden:

1. Initialisierung:

```
Hamster billy;
Hamster sunny = billy;
```

2. Zuweisung:

```
Hamster billy, sunny;
sunny = billy;
```

1. Kopieren bei Initialisierung

Objekte werden mit dem Copy Konstruktor bei der Initialisierung kopiert:


```
struct Hamster
{
   Hamster() { Alter = 0; }
   Hamster(Hamster const & ham)
   {
       Alter = ham.Alter;
   }
   int Alter;
};
Hamster billy;
Hamster sunny = billy;
```

Der Copy Konstruktor ist derjenige Konstruktor, der eine Referenz auf ein konstantes Objekt der Klasse nimmt (im Beispiel also **Hamster const &**). Weder auf das **const** noch auf das **&** kann man verzichten:

Das **const** gibt an, dass das Argument **ham** nicht vom Konstruktor verändert wird.

Das **&**-Zeichen Kennzeichnet **ham** als eine Referenzvariable.

Aufruf **Hamster sunny** = **billy;** ist äquivalent zu **Hamster sunny(billy);**

Copy-Ctor im Detail

C++

```
Hamster (Hamster const & ham)
{
    Alter = ham.Alter;
}
```

Anmerkungen:

Das Argument **ham** ist **const**: Es wird durch den CTor nicht verändert.

Alternative Schreibweise für Initialisierung:

```
Hamster billy;
Hamster sunny(billy);
```

CTor sei Kurzform für „Kostruktor“, DTor für „Destruktor“.

2. Kopieren bei Zuweisung

Bei Zuweisungen wird statt dessen der *assignment*-Operator ausgeführt:

```
struct Hamster
{ ...
   Hamster & operator= (Hamster const & ham)
   {
       Alter = ham.Alter;
       return *this;
   }
   int Alter;
};
Hamster billy, sunny;
sunny = billy;
```

operator= im Detail


```
Hamster & operator= (Hamster const & ham)
{
    Alter = ham.Alter;
    return *this;
}
```

Anmerkungen:

Argument **`ham`** ist **`const`**: Es wird durch **`operator=`** nicht verändert.

Es wird das Objekt selbst zurückgegeben: **`*this`**

Der Typ der Rückgabe ist **`Hamster &`**.

"Die großen Vier"

Die folgenden vier Memberfunktionen baut der Compiler, wenn man sie nicht selbst definiert:

- Der Default Constructor: **`C()`**
- Der Copy Constructor: **`C(C const &)`**
- Der Copy Assignment Operator:
  `C const & operator = (C const &)`
- Der Destructor: **`~C()`**

Regel: "Definiere sie immer selbst!"

Operatoren für Objekte?

Manchmal möchte man weitere Operatoren für Objekte definieren, z.B.:

```
//Klasse fuer komplexe Zahlen
struct Complex
{
    double m_R; double m_I;
    Complex(double r, double i = 0)
    {
          m_R = r; m_I = i;
    }
};

Complex x(1);
Complex y(2, 1);
Complex z = x + y;
```

Operatoren überladen

Man kann fast alle Operatoren überladen.

Ausnahmen (u.a.):

- . (Member selection)
- :: (Scope resolution)

Zwei Möglichkeiten:

1. Durch Member-Funktion
2. Durch globale Funktion

1. Operatoren als Member-Funktion

unäre Operatoren: geht auf aktuelles Objekt

```
Complex Complex::operator~ () {...}
```

binäre Operatoren: Aktuelles Objekt steht links.

```
 Complex Complex::operator+
   ( int const right )
{
   ...
}
```

2. Operatoren als globale Funktion

unäre Operatoren:

```
Complex operator~ (Complex const & obj)
{...}
```

binäre Operatoren: Aktuelles Objekt steht links.

```
Complex operator+
   (Complex const & left, int const right)
{
   ...
}
```

Aufgabe: Complexe Zahlen

Schreiben Sie die Operation **+** für **Complex**!


```
Complex Complex::operator+ (Complex const & right)
{
    Complex z(m_R + right.m_R, m_I + right.m_I);
    return z;
}
```

besser:

```
Complex Complex::operator+ (Complex const & right)
{
    return Complex(m_R + right.m_R, m_I + right.m_I);
}
```

Zum Abschluss: Überblick C++

Einige wichtige C++-Erweiterungen:

- Klassen für OOP (siehe Script 4 und 5)
- Referenzen (siehe Script 2)

- Exceptions
- Templates
- Namespaces
- ...

Exceptions dienen einem kontrollierten Fehlerhandling. Beispielsweise liefert der **new**-Operator (anders als **malloc**) niemals 0 zurück: Steht nicht genug Speicher zur Verfügung, wird statt dessen eine Exception geworfen, die dann von einem geeigneten Handler abgefangen werden kann. Man kann sich bei der Benutzung von **new** also den Test auf 0 sparen, auf den man bei **malloc** niemals verzichten sollte.

Templates sind so etwas ähnliches wie Bauanleitungen für Klassen oder Funktionen. Der Compiler erzeugt anhand dieser Anleitungen nach Bedarf Klassen bzw. Funktionen.

Mit *Namespaces* können Bezeichner gekapselt werden, so dass sie sich nicht gegenseitig in die Quere kommen. Dies ist besonders nützlich bei größeren Projekten. Die Standardbibliotheken (siehe unten) sind beispielsweise (zum großen Teil) im Namespace **std** gekapselt.

Ende!

Konfuzius sagt:

» Programmieren lernt man nur durch‘s Programmieren! «

Aufgaben zum Freitag:

11. Aufgabe:

Schreiben Sie eine Klasse „Person“, die folgenden Daten enthält: **Name** (string bis maximal 100 Zeichen), **Vorname** (string bis maximal 100 Zeichen), **Alter** (unsigned int). Stellen Sie Member-Funktionen zur Verfügung, mit denen man Namen, Vornamen und Alter setzen kann.

Leiten Sie anschließend eine Klasse „Angestellter“ ab, die zusätzlich Daten für **Aufgabe** (string bis maximal 200 Zeichen) und **Gehalt** (unsigned int) hält.

12. Aufgabe:

Erzeugen Sie eine Hierarchie von verschiedenen Klassen, wobei manche Klassen von anderen abgeleitet werden oder andere als Member-Variablen enthalten. Implementieren Sie jeweils Ctor und Default Dtor. Testen Sie nun, in welcher Reihenfolge die Ctors und Dtors der Basisklassen und der Member-Variablen bei der Erzeugung eines Objektes aufgerufen werden. Leiten Sie daraus eine allgemeine Regel ab.

13. Aufgabe:

Schreiben Sie ein kleine String-Klasse **String**, die wichtige Funktionalität im Umgang mit Strings kapselt: Strings bestimmter Länge erzeugen, Länge des Strings bestimmen, String verlängern, Strings hintereinander hängen (konkateniert).

Stellen Sie insbesondere einen Konstruktor bereit, der folgende Objektdefinition zulässt:

```
String str("hallo");
```

Definieren Sie auch einen operator+, mit dessen Hilfe Strings nach folgendem Muster hintereinander verkettet (konkateniert) werden können:

```
String s1("erster Teil ");
String s2("zweiter Teil");
String s3;
s3 = s1 + s2;    // jetzt steht in s3 "erster Teil zweiter Teil“
```