C++

Klassen, Vererbung

Philipp Lucas
phlucas@cs.uni-sb.de

Sebastian Hack
hack@cs.uni-sb.de

Wintersemester 2008/09

Inhalt

C++

Übersicht

- Objektorientierte Erweiterung von C
- Viele andere Erweiterungen:
  - Schablonen (templates)
  - Operatorüberladung
  - Überladung von Funktionen
  - Referenzen
  - Umgebungen (namespaces)
  - ...
- Weitestgehend abwärtskompatibel zu C
- Interoperabel mit C:
  C kann C++ Funktionen rufen und umgekehrt

Klassen in C++

```
class Bruch {
    int _z, _n;

    void kuerze() {
        int g = ggT(_z, _n);
        _z /= g;
        _n /= g;
    }

    bool istGanz() { return _n == 1; }

    Bruch(int z, int n) : _z(z), _n(n) { }
};
```

- Deklaration ähnlich zu Java
- Nicht den ; am Ende der Vereinbarung vergessen!
- Konvention: `private`-Felder mit vorangestelltem _
- Konstruktor ist Methode ohne Rückgabetyp,
  die den Namen der Klasse trägt
- Spezialsyntax für die Initialisierung der Felder

Klassen in C++

Sichtbarkeit

```
class Bruch {
    int _z, _n;

    void kuerze() {
        int g = ggT(_z, _n);
        _z /= g;
        _n /= g;
    }

public:
    bool istGanz() { return _n == 1; }

    Bruch(int z, int n)
        : _z(z), _n(n) { }
};
```

- Wie in Java kann man die Sichtbarkeit von Methoden und Feldern einschränken:

`public`	überall sichtbar
`protected`	Nur in Klasse und Unterklassen sichtbar
`private`	Nur in Klasse selbst sichtbar

☞ Keine Paket-Sichtbarkeit wie in Java

Klassen in C++

Sichtbarkeit

```
class A {
public:
    ...
private:
    ...
public:
    ...
protected:
    ...
};
```

- Sichtbarkeitsbezeichner leitet Abschnitt ein
- Alle Vereinbarungen nach `public:` sind public bis zum nächsten Sichtbarkeitsbezeichner
- Standard-Sichtbarkeit bei Klassen: `private`
- Anstatt `class` kann man auch `struct` schreiben
  - Standard-Sichtbarkeit ist dann `public`
  - Ansonsten kein Unterschied

Klassen in C++

Erzeugen von Objekten

```
void test() {
    Bruch eins;
    Bruch einhalb(1, 2);
    ...
}
```

- Das „Name Tag“ kann direkt als Typ verwendet werden
  (im Unterschied zu C)
- Funktioniert auch mit `struct` und `union`
- Konstruktor-Argumente werden nach dem Bezeicher des Objekts in Klammern angegeben
- `Bruch eins;` ruft Default-Konstruktor `Bruch::Bruch()`
- Jede Klasse muss einen Default-Konstruktor! haben
  (im Unterschied zu Java)
- Gibt es keinen, wird er automatisch angelegt
  er tut aber nichts insb. initialisiert er keine Felder

Klassen in C++

Erzeugen von Objekten

```
void test() {
    Bruch eins;
    Bruch einhalb(1, 2);
    ...
}
```

- Die Lebenszeit von `eins` und `einhalb` ist auf die Ausführung von `test` beschränkt
- Kehrt `test` zurück, sterben beide Objekte
- `eins` und `einhalb` wurden in der Aufrufschachtel von `test` angelegt
- Man kann Objekte auch auf der Halde erzeugen

```
void test() {
    Bruch *eins = new Bruch();
    Bruch *einhalb = new Bruch(1, 2);
    ...
}
```

- Es gibt keine automatische Speicherbereinigung
- Mit `new` erstellte Objekte müssen weider freigegeben werden:

```
delete eins;
```

`new` und `delete`

- `new` reserviert Speicher auf der Halde und ruft den entspr. Konstruktor einer Klasse auf
- `delete` ruft den Destruktor auf und gibt den Speicher wieder frei
- Wozu Destruktoren?

`new` und `delete`

- `new` reserviert Speicher auf der Halde und ruft den entspr. Konstruktor einer Klasse auf
- `delete` ruft den Destruktor auf und gibt den Speicher wieder frei
- Wozu Destruktoren?
  - Die freizugebenden Objekte müssen u.U. andere Objekte freigeben
- `malloc` und `free` sind rufbar, jedoch rufen beide nicht Kon-/Destruktoren

Regel

In C++ immer `new` und `delete` verwenden

Reihungen

- Reihungen von Objekten sind keine Reihungen von Referenzen
- Mit

```
A* arr = new A[10];
```

  werden 10 `A` Objekte angelegt, und 10 mal der entspr. Konstruktor aufgerufen
- Die Reihung enthält 10 `A` Objekte
- Das Freigeben muss mit `delete[]` erfolgen, und nicht mit `delete`

```
delete arr;      // Undefiniertes Verhalten
delete[] arr;    // Ruft Destruktoren aller Elemente auf
```

- Fallstrick: Einem Zeiger sieht man nicht an, ob er auf eines oder auf mehrere Objekte zeigt

Vererbung

Sichtbarkeit

```
class Bruch : public Paar {
    ...
};
```

- Sichbarkeitsbezeichner steuert Sichtbarkeit der ererbten Felder/Methoden

`public`	Sichtbarkeit wird nicht modifiziert
`protected public`	Felder werden `protected`
`private`	Alles wird `private`

Vererbung

Initialisierung

```
class Bruch : public Paar {
    Bruch() : Paar() { }
    Bruch(int z, int n) : Paar(z, n) { }
};
```

- Der Konstruktor der Oberklasse muss explizit gerufen werden
- Kein automatisches Rufen wie in Java

Methoden

Überladen

```
class Bruch {
public:
    void set(int z, int n)    { _z = z; _n = n; }
    void set(int z)           { _z = z; _n = 1; }
    void set(const Bruch& b) { _z = b._z; _n = b._n; }
};
```

- Methoden können überladen werden
- Methoden mit gleichem Namen müssen sich in den Typen der Parameter unterscheiden
- Wie in Java

Operatorüberladung

Ausblick

- Operatoren sind auch Methoden
- Sie werden nur anders hingeschrieben:

$$a \, \tau \, b \quad \text{anstatt} \quad \tau(a, b)$$

und

$$\tau \, a \quad \text{anstatt} \quad \tau(a)$$

- Man kann in C++ für eigene Typen eine eigene Implementierung der meisten Operatoren angeben (aka Operatorüberladung):

```
class Bruch {
    ...
public:
    Bruch operator*(const Bruch& b) {
        return Bruch(_z * b._z, _n * b._n);
    }
};
```

Default-Parameter

```
class Bruch {
    ...
public:
    void set(int z, int n = 1) { _z = z; _n = n; }
    void set(const Bruch& b)   { _z = b._z; _n = b._n; }
};
```

- Methodenparameter können mit voreingestellten Werten gekennzeichnet werden
- Wird kein anderer Wert vom Aufrufer gewünscht, kann das Argument entfallen
- Default-Parameter sind immer die letzten in der Parameterliste
- Es darf zu keinen Widersprüchen kommen,
  z.B. wäre im obigen Beispiel die Methode

  ```
  void set(int z) { ... }
  ```

  nicht erlaubt, da bei einer Aufrufstelle

  ```
  set(5);
  ```

  nicht klar wäre, welche Methode gerufen werden soll

  ```
  void set(int z, int n = 1) { ... }
  ```

  gerufen werden soll.

Überschreiben

- Im Gegensatz zu Java gibt es in C++ zwei Arten der Überschreibung
- Standardmässig wird der statische Typ zur Ermittlung der zu rufenden Methode herangezogen
- Durch Kennzeichnen der Methodendeklaration mit `virtual` kommt der dynamische Typ zum Einsatz
- Abstrakte Methode: Anhängen von `= 0;` an die Deklaration

```
struct A {
    char f()          { return 'A'; }
    virtual char g() { return 'A'; }
};
struct B : public A {
    char f()          { return 'B'; }
    virtual char g() { return 'B'; }
};
void test() {
    A* a = new B(); // statischer Typ: A
    B *b = new B(); // statischer Typ: B
    a->f();         // Ergebnis: 'A'
    a->g();         // Ergebnis: 'B'
    b->f();         // Ergebnis: 'B'
    b->g();         // Ergebnis: 'B'
}
```

Destruktoren

- Destruktoren werden (automatisch) aufgerufen, wenn ein Objekt stirbt
- Sei es auf der Halde oder auf dem Keller
- Destruktoren sollte man angeben, wenn das Objekt Ressourcen belegt, die freigegeben werden sollten
- Ansonsten ist der Destruktor leer
- Destruktoren sollten immer `virtual` sein
- Destruktion muss nach dem dynamischen Typ erfolgen
- An der Stelle der Destruktion kann der statische Typ unpräziser sein

```
class Vector {
    int _len;
    double* _arr;
public:
    Vector(int len) : _len(len), _arr(new double[_len]) { }
    virtual ~Vector() { delete[] _arr; }
};
```

Inhalt

Kompatibilität zwischen C und C++

Übersicht

- C und C++ sind weitestgehend kompatibel
- C++ Übersetzer können die meisten C Programme übersetzen
- Man kann C++ Funktionen für C Programme rufbar machen
- Die C-Kompatibilität von C++ basiert auf C90
  ☞ Das meiste aus C99 ist in C++ nicht verfügbar

Binden von C und C++ Code

- Zur Auflösung der Überladung verwendet C++ name mangling
- Beispiel:

```
// foo.c
int foo(double x, char s) {
    ...
}
```

- Übersetzt man die Datei mit `c++`, dann heißt das Symbol `__Z3foodc`
  ☞ Die Parametertypen werden in den Symbolnamen kodiert
- Übersetzt man die Datei mit `cc`, dann heißt das Symbol `foo`
  ☞ C kennt keine Überladung; Symbolname = Funktionsname

Binden von C und C++ Code

- Möchte man C-Funktionen von C++ aus rufen, so muss man das name mangling für diese deaktivieren

```
// myprg.cpp

extern "C" int foo(double x, char s);

int myfunc() {
    return foo(1.234, 'A');
}
```

- C++ Übersetzer weiß nun, dass `foo` in einer C-Bibliothek liegt
- Das name mangling wird für `foo` deaktiviert
- Der Binder (Linker) sucht nach dem Symbolnamen `foo`, nicht `__Z3foodc`
- Zum Verwenden der C-Standardbibliothek in C++ muss man andere Header einbinden:
  `#include <cstdio>` anstelle von `#include<stdio.h>`, usw.

Binden von C und C++ Code

C Header für C++ nutzbar machen

- Die Headerdateien vieler C-Bibliotheken sehen so aus:

```
#ifndef MYHEADER_H
#define MYHEADER_H

#ifdef __cplusplus
extern "C" {
#endif

/* Inhalt */

#ifdef __cplusplus
}
#endif

#endif  /* MYHEADER_H */
```

- Dies macht die C-Funktionen für C++ rufbar

Inkompatibilitäten zwischen C und C++

Namensraum von `struct`/`union` Tags

- In C bilden die Typtags von `struct`s und `union`s einen eigenen Namensraum

```
int foo;
struct foo {
    int x, y;
};
struct foo foo_struct;
```

- In C++ sind sie im Namensraum der Variablen
- Vorsicht: Der Übersetzer kann nicht immer warnen

```
extern int foo;
int test() {
    struct foo { };
    return sizeof(foo);
}
```

- In C und C++ gültig
- Rückgabewert in C: `sizeof(int)`, in C++: `sizeof(struct foo)`

Inkompatibilitäten zwischen C und C++

Verschiedene

- C++ hat mehrere neue Schlüsselwörter

```
int class;
double template;
```

  ist ok in C, aber nicht in C++

- `'a'` hat Typ `int` in C und `char` in C++
  `sizeof('a')` kann unterschiedliche Werte in C und C++ haben
- `enum`-Konstanten sind `int` in C, haben aber eigene Typen in C++
- `void *` nicht mehr implizit in andere Zeiger umwandelbar

```
int *x = malloc(20 * sizeof(*x));
```

  geht in C, aber nicht in C++

- Filescope `const` Variablen sind `static` in C++ und `extern` in C
- usw.

Interoperabilität zwischen C und C++

Fazit

- C Programme sind nicht ohne weiteres mit C++ Übersetzern verarbeitbar
- Viele kleine Unterscheide
- Das Schlimmste: Manches ist in C und C++ erlaubt, wird aber anders interpretiert
- Daher: C-Dateien mit `cc` übersetzen, C++-Dateien mit `c++`
- Gemeinsam genutzte Funktionen mittels `extern "C"` deklarieren
- Den Binder beide Welten zusammenfügen lassen