Praxisorientierte Einführung in C++
Lektion: "Virtuelle Methoden"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

Motivation

- Bis jetzt haben wir Vererbung *ohne* Polymorphismus kennengelernt
- In C++ gilt die Faustregel: Man kriegt nur die Features, die man explizit will
- Für Polymorphismus benötigt man ein weiteres Schlüsselwort: `virtual`
- Es folgt ein Beispiel zur Motivation

Beispiel

```
#include <iostream>

struct Animal{
   void show() const { std::cout << "Animal" << std::endl;  }
};

struct Dog : public Animal{
   void show() const { std::cout << "Dog" << std::endl; }
};

void show_animal(Animal &a) { a.show(); }

int main() {
  Dog d;
  show_animal(d);
}
```

- Programm gibt (wider Erwarten) "Animal" aus und nicht "Dog".

Statische Typanalyse vs. dynamische Typanalyse

- Im vorigen Beispiel wurde die Methode zur Compile-Zeit ermittelt
  - Da `show_animal` eine `Animal`-Referenz übergeben wurde, und `Animal` kein polymorpher Typ ist, ermittelt der Compiler die Methode `Animal::show`
- Alternative: Methode wird zur Laufzeit ermittelt (dynamisch)
  - Teuer, da Typ ermittelt werden muss und deshalb optional

Angepasstes Beispiel

```
#include <iostream>

struct Animal{
   virtual void show() const { std::cout << "Animal" << std::endl;  }
};

struct Dog : public Animal{
   void show() const { std::cout << "Dog" << std::endl; }
};

void show_animal(Animal &a) { a.show(); }

int main() {
  Dog d;
  show_animal(d);
}
```

- Programm gibt "Dog" aus

Once Virtual - Always Virtual

- `virtual` wurde nur in der Basisklassenmethode verwendet
- Alle Methoden, die `virtual`-Methoden überschreiben sind automatisch auch `virtual`
- Zu Dokumentationsszwecken sollte man sich aber vielleicht angewöhnen auch überschriebene Methoden als `virtual` zu markieren
- Im neuen Standard wird man überschriebene Methoden mit "override" kennzeichnen können

```
struct Dog : public Animal{
   // 'const' wurde extra vergessen: Kompiler beschwert sich:
   // show () ohne 'const' gibt's nicht in der Basisklasse
   void show() override { std::cout << "Dog" << std::endl; }
};
```

- Dadurch sollen Fehler vermieden werden, wenn man die Funktion oder deren Signatur falsch schreibt

Anmerkungen: Pointer, Referenzen, Slicing

- Achtung: auch bei polymorphen Objekten gibt es Slicing

```
void show(Animal a) {
  a.show();
}
```

- Hier keine Referenz, sondern Kopie
- Wenn *Kopie* gemacht wird: *Slicing*
- Ausgabe ist wieder `"Animal"`
- Ausweg: Pointer oder Referenzen verwenden

Faustregel für echten Polymorphismus

- `virtual` verwenden
- mit Zeigern oder Referenzen arbeiten

Beispiel mit Pointer

```
#include <iostream>

struct Animal{
   virtual void show() const { std::cout << "Animal" << std::endl;  }
};

struct Dog : public Animal{
   void show() const { std::cout << "Dog" << std::endl; }
};

void show_animal(Animal *a) { a->show(); }

int main() {
  Dog d;
  show_animal(&d);
}
```

- Geht: Programm gibt "Dog" aus

vtable

- Für nicht-polymorphe Typen kann zur Übersetzungszeit ermittelt werden, welche Methoden aufgerufen werden sollen
  - Anhand des Typs der Variable, über die die Methode aufgerufen wird
- Für polymorphe Typen muss zur Laufzeit Information über Objektinstanz vorhanden sein
- Normalerweise über "virtual table" (`vtable`) geregelt
  - Aber implementierungsabhängig

vtable

- Für Jede Typhierarchie gibt es eine `vtable`
- Jedes Objekt eines polymorphen Typs trägt eine `vtable`-Pointer mit sich herum, die automatisch erstellt wird
- Objekte werden um die Größe eines Pointers größer
  - Das kann bei vielen kleinen Objekten schon mal ein Faktor 2 sein
- Methodenaufruf bei virtuellen Methoden langsamer, da erst ein Zeiger dereferenziert werden muss
  - Kein inlining möglich (das macht der Kompiler i.d.R. auch dann nicht, wenn absolut klar ist, welche Methode gemeint ist)

Pure Virtual - Motivation und Syntax

- Oftmals gibt es Anforderungen an ein Interface, ohne dass Funktionalität vorgegeben wird
  - Java: Interfaces
  - C++: Pure `virtual` methods
- Suggestive Schreibweise

```
struct X {
  virtual void f() = 0;
};
```

Beispiel: Threading Library

- Klasse `Thread` stellt Funktionalität zur Verfügung, um Thread zu starten, weiß aber nicht, wie der Code aussehen soll

```
struct Thread {
  virtual void run() = 0;
};
```

- Es ist **nicht** möglich, Instanzen von `Thread` zu erstellen

```
int main() {
  Thread t; // geht nicht!
}
```

- Erzeugt Compiler-Fehler
- `run()` ist "rein virtuelle Methode"
- Solange eine Klasse *mindestens* eine rein virtuelle Methode hat, nennt man sie auch "abstrakt"

Beispiel cntd.

- Subklassen, die `Thread` beerben, können – müssen aber nicht – `run` überschreiben

```
struct MyThread : public Thread {

  virtual void run() {
    ...
  }
};
```

- Es ist nun möglich Instanzen von `MyThread` zu erstellen

```
int main() {
  MyThread thread1;
  MyThread thread2;
}
```

Beispiel: Multiplattform-Threading-Library

- Klasse kann mehr als eine rein virtuelle Methode enthalten

```
struct Thread {
  // User implements run()
  virtual void run() = 0;

  // OS-dependent Thread subclasses implement start()
  virtual void start() = 0;
};

struct PosixThread : public Thread {
  virtual void start() { ... }
}

struct Win32Thread : public Thread {
  virtual void start() { ... }
}
```

- `PosixThread` und `Win32Thread` sind immer noch abstrakt, da `run` immer noch rein virtuell ist

Posix Threads

- Wie startet man einen Thread auf einem Posix-System?
  - Posix Threading library `pthread`
  - Header `pthread.h`
  - Compiler Flag `-pthread`
- Uns interessieren nur zwei Funktionen

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void*), void *restrict arg);

int pthread_join(pthread_t thread, void **value_ptr);
```

- Wir behandeln hier kein Locking, Condition Variables, etc..

Posix Threads - Beispiel

- Ein kleines Beispiel

```
#include <pthread.h>
#include <iostream>

void *run(void *) { std::cout << "." << std::endl; return 0; }

int main() {
  pthread_t thread1;
  pthread_t thread2;

  pthread_create(&thread1, 0, run, 0);
  pthread_create(&thread2, 0, run, 0);

  pthread_join(thread1, 0);
  pthread_join(thread2, 0);
}
```

- Ausgabe kann je nach Ausführungsreihenfolge der Instruktionen in den beiden Threads varieren

PosixThread-Klasse, erster Versuch

- Erster Versuch für PosixThread-Klasse

```
#include <pthread.h>
struct PosixThread : public Thread {
  pthread_t thread; bool running;

  PosixThread() : running(false) { }

  virtual void start() {
    pthread_create(&thread, 0, do_run, 0);
    running = true; // Error Checking omitted
  }

  ~PosixThread() { if (running) pthread_join(thread, 0); }

  protected: virtual void do_run() { run(); }
};
```

- Problem: Typ von `do_run` ist nicht globale Funktion, sondern Methode (Compiler meckert)

`PosixThread`-Klasse, zweiter Versuch

- Dieses Problem tritt häufiger auf: Ein C-Interface erwartet Function-Pointer, aber wir haben Methode
- IMHO ein C++-Pattern, dessen Name mir leider nicht geläufig ist
- Lösung immer sehr ähnlich: Erstelle global Hilfsfunktion, die ein `void *`-Argument erwartet
- Diese globale Funktion castet den Pointer in Klassenpointertyp und ruft die Methode auf..
- ...

PosixThread-Klasse, zweiter Versuch

```
#include <pthread.h>

void *PosixThreadRun(void *p);

struct PosixThread : public Thread {
  pthread_t thread; bool running;
  PosixThread() : running(false) { }
  virtual void start() {
    pthread_create(&thread, 0, PosixThreadRun, (void*)this);
    running = true; // Error Checking omitted
  }
  ~PosixThread() { if (running) pthread_join(thread, 0); }
};
void *PosixThreadRun(void *p) { ((PosixThread *)p)->run(); }
```

- Implementierung hat einen subtilen Bug - Wer sieht ihn (Fix: Condition Variables)?
- Analog könnte man `Win32Thread` implementieren

Virtuelle Destruktoren

- Konstruktoren sind niemals `virtual`
- Destruktoren aber können, und müssen (manchmal) `virtual` sein
- Wann immer man `delete` auf einem Basisklassenpointer aufruft, hinter dem sich ein Objekt eines abgeleiteten Typs verbergen könnte, muss virtueller Destruktor deklariert und definiert werden

Virtuelle Destruktoren - Beispiel

```
#include <iostream>

struct X { ~X() { std::cout << "~X()" << std::endl; };

struct Y : public X { ~Y() { std::cout << "~Y()" << std::endl; };

int main() {
  X *x = new Y;
  delete x;
}
```

- Welche Destruktoren werden aufgerufen? Nur `~X()`
- Wenn `Y` Resourcen benutzt, die ordentlich freigegeben werden müssen, kann das fatal sein
  - Speicherlecks
  - Netzwerkverbindungen
- Ausweg: Virtueller Destruktor

Virtuelle Destruktoren

```
#include <iostream>

struct X {
  virtual ~X() { std::cout << "~X()" << std::endl;
};

struct Y : public X {
  ~Y() { std::cout << "~Y()" << std::endl;
};

int main() {
  X *x = new Y;

  delete x;
}
```

- Jetzt wird auch ~X() aufgerufen

Anmerkung: Rein virtuelle Methoden mit Definition

- Rein virtuelle Funktionen können Definition haben
- Aufruf nur über explizite Typ-Nennung möglich

```
struct X {
  virtual void f() = 0;
};

void X::f() {

}

struct Y : public X {
  virtual void f() {
    X::f();
  }
};

int main() { Y y; y.X::f(); }
```

Anmerkung: `static` Methoden

- `static` Methoden sind niemals `virtual`
- Der Compiler ermittelt zur Compilezeit die Methode, die aufgerufen wird entweder
  - anhand des Klassenscopes
  - anhand des Referenz-Typen

```
struct X { static void f() { } };

struct Y : public X { static void f() { } };

int main() {
  X x;
  Y y;
  X &rx = y;

  x::f();
  y::f();
  rx::f();
}
```