Praxisorientierte Einführung in C++
Lektion: "Static Members"

Christof Elbrechter

Neuroinformatics Group, CITEC

May 28, 2014

Table of Contents

Motivation

- Bis jetzt: Alle Elemente und Methoden einer Klasse/Struktur an Instanz gebunden:

```
struct Foo { int x; int bar() { return x; } };

int main() {
  Foo f;
  f.x = 0; // Okay!!
  f.bar(); // Okay!!

  Foo::x = 0; // Fehler!!
  Foo::bar(); // Fehler
}
```

static

- Manchmal werden aber klassenspezifische Daten/Methoden benötigt
- Beispiele:
  - Performancemessungen (Anzahl der Methodenaufrufe mitzählen, usw.)
  - Methoden, die Instanzen erstellen (Singleton Pattern, Factory Pattern, usw.)
- In Klassen/Strukturen kann Schlüsselword `static` benutzt werden

Beispiel

static-Beispiel

```
class Foo {
  static int s_instanceCount;

  public:
    Foo() { ++s_instanceCount; }
    ~Foo() { --s_instanceCount; }

    static int getInstanceCount() {return s_instanceCount; }
};
```

- Frage: Wo wird `s_instanceCount` initialisiert?
  - Wenn `const`: u.U. direkt im Header
  - Wenn nicht: In *einer* Quellcode-Datei
    - Sonst wird die One-Definition-Rule verletzt

Beispiel - Initialisierung

foo.h

```
class Foo {
  /// direkt im header (nur in wenigen Faellen: s.u.)
  const static int classID = 100;
  static int s_instanceCount;   // s_ prefix wegen _s_tatic

  public:
    Foo();
    ~Foo();

    static int getInstanceCount(){...}
};
```

foo.cpp

```
/// so wird das richtige Symbol erzeugt und initialisiert
int Foo::s_instanceCount = 0;
```

Wann darf ein `static` Member direkt initialisiert werden?

```
class Foo{
      static const int BAR = 5;
};
```

- Direkte Initialisierung nur in *speziellen Spezialfällen*
  - ...für konstante Ganzzahltypen oder Aufzählungstypen
  - ...wenn Initialisierungsausdruck auch *konstanter Ganzzahlausdruck*
- Dann darf der Typ aber immer noch nur in konstanten Ganzzahlausdrücken vorkommen
  → Das geht dann sehr stark ins Detail

Wann darf ein `static` Member direkt initialisiert werden?

```
class Foo{
      static const int BAR = 5;
};
```

- Direkte Initialisierung nur in *speziellen Spezialfällen*
  - ...für konstante Ganzzahltypen oder Aufzählungstypen
  - ...wenn Initialisierungsausdruck auch *konstanter Ganzzahlausdruck*
- Dann darf der Typ aber immer noch nur in konstanten Ganzzahlausdrücken vorkommen
  → Das geht dann sehr stark ins Detail
- Ausweg: einfach **immer** außerhalb der Klassendefinition in einem .cpp File deklarieren

Singletons (auch *singelton pattern*)

- Manchmal braucht man eine Klasse, von der es maximal eine Instanz gibt
- Instanz sollte global erreichbar sein
- Beispiele:
  - Klasse, die Konfigurationsfile repräsentiert
  - Hauptfenster einer Applikation
  - ....
- Lösung: Singleton

Singleton - Beispiel

foo.h

```
class Foo {
  static Foo* s_instance;
  Foo(); // privater Konstruktor -> Instanzierung verboten

  public:
  static Foo* getInstance();
  ~Foo();
};
```

foo.cpp

```
#include "foo.h"

Foo* Foo::s_instance = 0;

Foo::Foo() { }
Foo::~Foo() { s_instance = 0; }

Foo* Foo::getInstance() {
  if (s_instance == 0) s_instance = new Foo;
  return s_instance;
}
```

- Implementation jedoch fragwürdig: wer gibt `s_instance` frei?

Alternative Version

```
// ---- foo.h -------
class Foo {
  Foo(){}; // privater Konstruktor -> Instanzierung verboten

  public:
  static Foo* getInstance();
};

// ---- foo.cpp -------
#include "foo.h"
Foo* Foo::getInstance() {
  static Foo instance;      // instance wird automatisch bei Programm-
  return &instance;         // ende zerstoert
}
```

- Achtung: auch diese Implementation kann zu Problem führen
  - Es ist nicht direkt klar, in welcher Reihenfolge lokale `static` Variablen freigegeben werden
- Ausweg: sog. *Smart-Pointer* verwenden
- Genaugenommen geht das schon mit den herkömmlichen C++ `auto_ptr<T>` aus dem header `<memory>`

static Methoden - Anmerkung

- Es ist auch möglich static Methoden über Objektnamen aufzurufen

```
#include <iostream>
struct X {
  static void f() { std::cout << "X" << std::endl; }
};

int main() {
  X x;
  x.f();
}
```