Programmierkurs

Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

November 30, 2018

Objektorientiertes Programmieren

- Kapselung
- const und Klassen
- Initialisierung und Cleanup
- Default-Konstruktor
- Objekte kopieren
- Ressourcenverwaltung
- Klassen in Headerdateien

Objektorientierte Programmierung

Bisher:

- Programme aus Funktionen, die mit primitiven Datentypen arbeiten (Zahlen, Strings)
- Verwendung von komplexeren Datentypen aus der STL

Objektorientierte Programmierung

Bisher:

- Programme aus Funktionen, die mit primitiven Datentypen arbeiten (Zahlen, Strings)
- Verwendung von komplexeren Datentypen aus der STL

Jetzt:

- Programme aus Objekten, die
  - einen internen Zustand haben (member variables)
  - Operationen ausführen können (member functions / Methoden)
- Jedes Objekt ist eine *Instanz* einer *Klasse*.
- Eine Klasse definiert das Verhalten all ihrer Instanzen.
- Wichtige Konzepte:
  - Kapselung
  - `const`ness
  - Komposition vs. Vererbung
  - Initialisierung und Cleanup

Reale Objekte in C++ abbilden

- Einfaches Beispiel: $x \in \mathbb{R}^2$
- Eigenschaften:
  - x-Koordinate, y-Koordinate
  - Betrag, Winkel
- gespeicherte Daten vs. Eigenschaften
  - Eine Repräsentation zum Speichern aussuchen
  - Andere Eigenschaften bei Bedarf ausrechnen
- Operationen
  - Verschieben
  - Rotieren
  - Spiegeln
  - ...

```
class Point {
public:
  double x;
  double y;
};
```

Klassen I

- C++ erlaubt die Definition von Klassen
- Eine Klasse beschreibt eine bestimmte Art von Objekt
- Alle Objekte einer Klasse sind einheitlich (Speicherbedarf etc.)
- Klassen dürfen nur in globalem Scope, in Namespaces oder in anderen Klassen definiert werden, nicht in Funktionen

```
class Point {
public:
  double x;
  double y;
};

int main() {
  Point p;
  p.x = 1.;
  p.y = 2.;
  std::cout << p.x << std::endl;
}
```

Klassen II

- Klassen beginnen mit dem keyword `class`, gefolgt von einem Scope, gefolgt von einem Semikolon
- Eine Klasse kann member variables enthalten
- Eine Klasse kann member functions enthalten

```
class Point {
public:
  double x;
  double y;

  double norm() {
    return std::sqrt(x*x + y*y);
  }

  void scale(double factor) {
    x *= factor;
    y *= factor;
  }
};
```

Member Functions

- muss man auf einer Instanz aufrufen
- erhalten einen impliziten ersten Parameter `this`, der die Instanz repräsentiert, für die die Methode aufgerufen wurde
- können auf die Member-Variablen der Instanz zugreifen

```
class Point {
public:
  double x;
  double y;

  double norm() {
    return std::sqrt(x*x + y*y);
  }

  void scale(double factor) {
    x *= factor;
    y *= factor;
  }
};
```

Kapselung

Klassen können die Sichtbarkeit von enthaltenen Variablen und Funktionen kontrollieren:

```
class Polygon {
// not visible outside Polygon
private:
  std::vector<Point> _corners;
// only visible to Polygon and classes that inherit from it
protected:
  const Point& corner(int i) const;
// visible to everyone
public:
  double area() const;
  void rotate(double angle);
};
```

- Die Standard-Sichtbarkeit in `class` ist `private`.
- Es ist sinnvoll, für private Member ein Namensschema einzuführen (z.B. beginnen mit Unterstrich)

Kapselung

Klassen können die Sichtbarkeit von enthaltenen Variablen und Funktionen kontrollieren:

```
class Polygon {
// not visible outside Polygon
private:
  std::vector<Point> _corners;
// only visible to Polygon and classes that inherit from it
protected:
  const Point& corner(int i) const;
// visible to everyone
public:
  double area() const;
  void rotate(double angle);
};
```

- Die Standard-Sichtbarkeit in `class` ist `private`.
- Es ist sinnvoll, für private Member ein Namensschema einzuführen.

Kapselung - Richtlinien

- Alle Member-Variablen **private**
- Wenn externer Zugriff auf private Variablen erforderlich ist: Accessor-Methoden

```
class Complex {
  double _real, _imaginary;
public:
  double real() const { // or getReal()
    return _real;
  }

  void setReal(double v) {
    _real = v;
  }
...};
```

- In Member-Funktionen direkt auf private Variablen / Funktionen zugreifen!

const und Klassen

```
const double x = 2.0;
std::cout << x << std::endl; // ok, x wird nur gelesen
x = x + 2; // Compile-Fehler

const Complex c(1.0,2.0);
std::cout << x.real() << std::endl; // ????
c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer `const` Instanz

const und Klassen

```
const double x = 2.0;
std::cout << x << std::endl; // ok, x wird nur gelesen
x = x + 2; // Compile-Fehler

const Complex c(1.0,2.0);
std::cout << x.real() << std::endl; // ????
c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer `const` Instanz

- Woher weiss der Compiler, dass die Methode die Instanz nicht verändert?
- Lösung: Funktionssignatur so verändern, dass die Instanz (und alle Member) in der Funktion `const` sind:

```
double real() const { // this makes instance const
  return _real;       // cannot modify _real here
}
```

Initialisierung und Cleanup

- Objekte müssen vor Verwendung initialisiert werden (Speicher allokieren, Dateien öffnen etc.) und danach Ressourcen wieder freigeben.
- C++ macht hier strikte Garantien:
  - Für jedes Objekt wird ein Konstruktor aufgerufen, bevor der Programmierer Zugriff bekommt.
  - Das gilt auch für Objekte, die Member von anderen Objekten sind, und Basisklassen (Vererbung).
  - Für jedes Objekt, dessen Konstruktor erfolgreich beendet wurde, wird ein Destruktor aufgerufen, bevor das Objekt aufhört zu existieren.
- Ein Objekt hört auf zu existieren, wenn
  - Das Scope (-Paar) endet, in dem die Variable angelegt wurde (normale Variablen).
  - `delete` aufgerufen wird (Pointer).
- Strengere Garantien als viele andere Sprachen.

Konstruktor

```
class Triangle : public Shape {
  Point _x1, _x2, _x3;
public:
  Triangle(const Point& x1, const Point& x2, const Point& x3)
    : Shape(), _x1(x1), _x2(x2), _x3(x3)
  {}
};
```

- Konstruktoren sind Methoden, die genauso heissen wie die Klasse und keinen Rückgabewert haben.
- Es kann mehrere Konstruktoren mit unterschiedlichen Argumenten geben.
- Vor dem Body des Konstruktors kommt die constructor initializer list:
  - Liste von Konstruktor-Aufrufen für Basisklassen und Member-Variablen.
  - Wenn Basisklassen oder Variablen hier nicht aufgeführt werden, wird der Default-Konstruktor (ohne Argumente) aufgerufen.
  - Variablen immer hier initialisieren, nicht im Body!

Destruktor

```
class Pointer {
  double* _p;
public:
  Pointer(double v)
    : _p(new double(v))
  {}

  ~Pointer()
  {
    delete _p;
  }
};
```

- Destruktoren heissen wie die Klasse mit vorgestellter "~".
- Destruktoren haben nie Argumente $\Rightarrow$ es gibt nur einen pro Klasse.
- Cleanup-Aufgaben
  - Speicher freigeben
  - Dateien schliessen
  - Netzwerkverbindungen schliessen
  - ...

Default-Konstruktor

Der Default-Konstruktor ist der Konstruktor ohne Argumente:

```
class Empty {
public:
  Empty()
  {}
};
```

- Wenn eine Klasse keinen Konstruktor definiert, erzeugt der Compiler einen Default-Konstruktor.
- Ansonsten muss man ihn von Hand schreiben, wenn man ihn braucht.
- Der Compiler erzeugt auch keinen Default-Konstruktor wenn eine der Member-Variablen oder eine Basisklasse keinen Default-Konstruktor hat (entweder von Hand geschrieben oder default).

Objekte kopieren

- Um Objekte kopieren zu können, muss der Compiler wissen, wie er das machen soll
- Objekt beim Anlegen kopieren:
  Copy Constructor

```
Pointer(const Pointer& other)
  : _x(other._x)
  , _y(other._y)
{}
```

- Neuen Wert in existierendes Objekt kopieren:
  Copy Assignment Operator

```
Pointer& operator=(const Pointer& other) {
  _x = other._x;
  _y = other._y;
  return *this;
}
```

- Wenn alle Member-Variablen kopierbar sind und wir kein spezielles Verhalten benötigen, kann der Compiler die Funktionen automatisch erzugen

Ressourcenverwaltung

Programme müssen alle Ressourcen (Speicher etc.), die sie allokieren, auch wieder freigeben (sonst Bugs)!
Methoden:

- Manuell Irgendwo Speicher organisieren und von Hand überlegen, wann man ihn nicht mehr braucht
  - aufwendig
  - fehleranfällig
- Garbage Collection Speicher wird speziell markiert, in periodischen Abständen wird im Hintergrund unbenutzter Speicher gesucht und freigegeben
  - komfortabel
  - kann zu Programm-Rucklern führen
  - Funktioniert nicht für andere Ressourcen (Dateien etc.)
- RAII Die C++-Lösung

Ressource Acquisition is Initialization (RAII)

C++ verwaltet Ressourcen mit dem RAII-Idiom:

- Klasse, die genau eine Ressource kapselt.
- Ressource wird im Konstruktor allokiert.
- Ressource wird im Destruktor freigegeben.
- C++ garantiert, dass der Destruktor aufgerufen wird, falls der Konstruktor erfolgreich beendet wurde.
- Funktioniert für beliebige Arten von Ressourcen.
- Der Programmierer muss die RAII-Klasse bewusst verwenden.
- Diverse Implementierungen in der standard library:
  - Speicher: `std::vector`, `std::map`, `std::unique_ptr`, ...
  - Dateien: `std::fstream`
  - Locks: `std::lock_guard`, ...

RAII: Beispiel

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main(int argc, char** argv)
{
  {
    std::ofstream outfile("test.txt");
    outfile << "Hello, World" << std::endl;
  } // file gets flushed and closed here

  std::ifstream infile("test.txt");
  std::string line;
  std::getline(infile,line);
  std::cout << line << std::endl;
  return 0;
}
```

Klassen in Headerdateien

- Wenn man Klassen in Headerdateien deklariert, schreibt man die Klasse selbst in die Headerdatei.
- Funktionen werden wie üblich nur deklariert (jetzt innerhalb der Klasse).
- In der Implementierung wird die Klasse nicht erneut deklariert.
- Die Implementierung enthält nur noch Definitionen der Member-Funktionen.
- Um anzuzeigen, dass eine Funktion Member einer Klasse ist, wird dem Funktionsnamen der Klassenname gefolgt von `::` vorangestellt:

```
double Point::x() const {
  return _x;
}
```

- Die Signatur der Funktion muss exakt mit dem Header übereinstimmen, inklusive des möglicherweise angehängten `const`!

Klassen in Headerdateien: Beispiel I

Headerdatei `point.hh`

```
#ifndef POINT_HH
#define POINT_HH

class Point {
  double _x;
  double _y;

public:
  // Function Declarations
  Point(double x, double y);

  double x() const;
  double y() const;

  void scale(double factor);
};

#endif // POINT_HH
```

Klassen in Headerdateien: Beispiel II

Implementierung `point.cc`

```
#include <point.hh>

Point::Point(double x, double y)
  : _x(x), _y(y)
{}

double Point::x() const {
  return _x;
}

double Point::y() const {
  return _y;
}

void Point::scale(double factor) {
  _x *= factor;
  _y *= factor;
}
```