Programmierkurs
Vorlesung 4
Objekt Orientiertes Programmieren

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

30. April 2023

Wiederholung/Korrektur

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

Wiederholung/Korrektur

- container: `array`,`vector`,`list`,`map`
  - kontinuierlicher Speicher, Indexzugriff: `array`, `vector`
  - unstrukturierter Speicher: `map`, `list`

Wiederholung/Korrektur

- container: `array`,`vector`,`list`,`map`
  - kontinuierlicher Speicher, Indexzugriff: `array`, `vector`
  - unstrukturierter Speicher: `map`, `list`
- Initialisierung:

```
std::vector<int> v1 = { 4,5,6 }; // direkt initialisiert
std::vector<int> v2 = {{ 4,5,6 }}; // indirekt per list -> copy
```

  - v1: direkt initialisiert über `initializer_list`
  - v2: wählt `initializer_list` im zweiten Schritt mit {`4`,`5`,`6`}
  - Ausführliche Erklärung
- Datenstrukturen `pair`,`tuple`
- Zugriff für pair/tuple
  - allgmein: `get<ind>(v)`
  - pair: `p.first`, `p.second`

Objektorientierte Programmierung

Bisher:

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

Objektorientierte Programmierung

Bisher:

- Programme aus Funktionen, die mit primitiven Datentypen arbeiten (Zahlen)
- 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
  - ...

```
struct Point {
  double x;
  double y;

  double getAngle() const;
  double getLength() const;
  void shift(double dx, double dy);
  void rotate(const Point& c, double phi);
};
```

Klassen I

- C++ erlaubt die Definition von Klassen
- Eine Klasse beschreibt den Inhalt eines Objekts:
  - Variablen
  - Funktionen
  - Instanzvariablen
  - Sichtbarkeit
- Alle Objekte einer Klasse sind einheitlich (Speicherbedarf etc.) werden

Klassen I

- C++ erlaubt die Definition von Klassen
- Eine Klasse beschreibt den Inhalt eines Objekts:
- Alle Objekte einer Klasse sind einheitlich (Speicherbedarf etc.)
- Klassen dürfen in jedem Scope:
  - Namespace
  - Klassen
  - Funktionen

  deklariert werden

```
namespace Geometrie {
  struct Point {
    double x;
    double y;
    ....
  };
}
```

```
namespace Geometrie {
  struct Mesh {
    struct Point {
      double x;
      double y;
      ....
    };
  };
}
```

```
std::shared<PointBase>
  getPoint(double x0, double y0) {
    struct Point : PointBase {
      double x;
      double y;
      ....
    };
  ...
}
```

Klassen I

- C++ erlaubt die Definition von Klassen
- Eine Klasse beschreibt den Inhalt eines Objekts:
- Alle Objekte einer Klasse sind einheitlich (Speicherbedarf etc.)
- Klassen dürfen in jedem Scope deklariert werden
- Variablen und Funktionen besitzen eine Sichtbarkeit

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

```
struct Point {
  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` (oder `struct`), 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 getLength() const {
    return std::sqrt(x*x + y*y);
  }

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

Member Functions

- muss man auf einer Instanz aufrufen:

```
Point p; double r = p.getLength();
```

Member Functions

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

```
struct Point {
  double x;
  double y;

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

  void scale(double factor) {
    this->x *= factor; // this-> ist optional
    y *= factor; // und wird nur wenn notwendig verwendet
  }
};
```

const und Klassen

Zugriff und Methodenaufruf bei einer `const` Instanz

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

const Point c{1.0,2.0}; // initialisiert x und y zu 1.0 und 2.0
std::cout << c.x << std::endl;
c.x = 3.0; // darf nicht funktionieren
std::cout << c.getLength() << std::endl; // funktioniert da getLength const ist
```

Kapselung

Klassen können die Sichtbarkeit von enthaltenen Variablen und Funktionen kontrollieren:

```
class Point {
// not visible outside Point
  double x,y;
// only visible to Point and classes that inherit from it
protected:
  double mass;
  double physProperty(int i) const;
// visible to everyone
public:
  double getLength() const;
  void rotate(const Point& c, double angle);
};
```

- Die Standard-Sichtbarkeit in `class` ist `private`.

Kapselung — Richtlinien

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

```
class Point {
  double _x, _y;
public:
  double x() const { // or getX()
    return _x;
  }

  void setX(double v) {
    _x = v;
  }
...};
```

- In Member-Funktionen direkt auf private Variablen / Funktionen zugreifen!
- Accessor-Methoden können das Einhalten von *Invarianten* sicherstellen

Kapselung — Richtlinien

- Alle Member-Variablen `private`
- Wenn externer Zugriff auf private Variablen erforderlich ist: Accessor-Methoden
- In Member-Funktionen direkt auf private Variablen / Funktionen zugreifen!
- Accessor-Methoden können das Einhalten von *Invarianten* sicherstellen
- Einheitliche Layouts, z.B.:

Variable	getter	setter
`double _x;`	`double x()`	`void x(double )`
`double x;`	`double getX()`	`void setX(double )`
`double x;`	`double get_x()`	`void set_x(double )`

- bei Referenzen: zwei Getter:

```
struct Point {
  double& getX();
  const double& getX() const;
};
```

```
Point c;
double& x = c.getX();
r = 0.5;
const Point c2;
const double& r2 = c2.getX();
// r2 = 0.5;
```

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 (siehe 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
  - die Umgebung endet, in dem die Variable angelegt wurde (für normale Variablen)
  - explizit `delete` aufgerufen wird (für Pointer)
- Strengere Garantien als viele andere Sprachen.

Konstruktor

```
  struct Shape {};
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 deren 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 Tilde "~".
- Destruktoren haben nie Argumente ⇒ es gibt nur einen pro Klasse.
- Cleanup-Aufgaben: Speicher freigeben (bei Pointern), Dateien schließen, Netzwerkverbindungen schließen, ...
- Muss man nur definieren, wenn tatsächlich eine dieser Aufgaben erfüllt werden muss

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

```
Point(const Point& other)
  : x(other.x), _y(other._y)
{}
```

- Neuen Wert in existierendes Objekt kopieren:
  Copy Assignment Operator

```
Point& operator=(const Point& 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 erzeugen

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

Resource 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 Standardbibliothek:
  - Speicher: `std::vector`, `std::map`, `std::unique_ptr`, ...
  - Dateien: `std::fstream`
  - Locks: `std::lock_guard`, ...

Eigentlich wäre DIRR (Destruction is Resource Release) ein besserer Name gewesen...

RAII: Beispiel

```
#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::getX() 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.h`

```
#ifndef POINT_H
#define POINT_H

class Point {
  double x;
  double y;

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

  double getX() const;
  double getY() const;

  void scale(double factor);
};

#endif // POINT_HH
```

Klassen in Headerdateien: Beispiel II

Implementierung `point.cc`

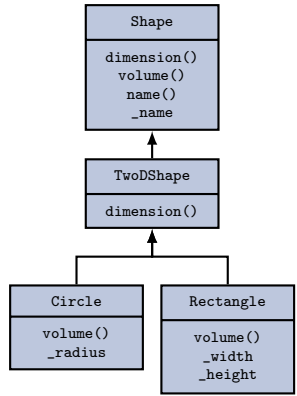
```
#include <point.hh>

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

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

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

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

Objekt Orientiertes Programmieren - Zusammenfassung

- Klassen und Objekte bündeln Daten und zugehörige Methoden
  - Bessere Abstraktion von abgebildetem Verhalten
  - Objekte verwalten ihren Zustand selbst $\rightarrow$ geringere Fehleranfälligkeit
  - Zugriff auf Daten kann durch Kapselung eingeschränkt werden um z.B. invariante Variablen zu implementieren
  - Ressourcenverwaltung nach dem RAII-Idiom
  - Eigenschaften und Verhalten von Klassen können Modular mittels Komposition und Vererbung erweitert und spezialisiert werden
- Deklaration von Klassen gehören bevorzugt in Header-Files und Definition in Source-Files

Vererbung

- Klassen können von anderen Klassen erben.
- Wichtigste Regel: is-a
  Is a circle a shape?
- Abgeleitete Klasse enthält alle Variablen und Methoden der Basisklasse.
- Methoden können überschrieben werden.
- Variablen vom Typ der Basisklasse können Objekte von abgeleiteten Klassen zugewiesen werden.
- Erweitern der Basisklasse um zusätzliche Funktionalität.



Verwendung von Klassenhierarchien

- Referenzen und Pointer auf Basisklassen funktionieren auch mit abgeleiteten Klassen:

```
Circle c(...);
Shape& s_ref = c;
```

- Beim Kopieren von Objekten werden nur die enthaltenen Daten der Basisklasse kopiert, Informationen aus abgeleiteten Klassen gehen verloren:

```
// only copies member variable _name
Shape s_copy = c;
```

- Eine Referenz auf die Basisklasse hat nur Zugang zu den Methoden und Variablen der Basis:

```
s_ref._name;  // ok
s_ref._radius // compile error
```

- Aufgerufene Funktionen sind immer aus der Basisklasse:

```
c.volume()     // calls Circle::volume()
s_ref.volume() // calls Shape::volume()
```

Dynamische Polymorphie

- Idee: Beim Aufruf einer Methode die Implementierung aus der abgeleiteten Klasse verwenden:

```
Circle c(...);
Shape& s_ref = c;
s_ref.volume(); // calls Circle::volume()
```

- Funktioniert mit `virtual` Funktionen:

```
class Shape {
  virtual double volume() const;
  // always make destructor virtual as well!
  virtual ~Shape();
};
```

- Methode ist dadurch auch in allen abgeleiteten Klassen `virtual`.
- Funktioniert nur mit Pointern / Referenzen:

```
s_ref.volume(); // calls Circle::volume()
Shape s_copy = c;
s_copy.volume(); // calls Shape::volume()
```

Dynamische Polymorphie: Pitfalls

- Keyword `virtual` ist in abgeleiteten Klassen implizit, aber die Redeklaration ist erlaubt.
- Methoden-Signatur in abgeleiteten Klassen muss exakt identisch sein, inklusive `const`-Deklarationen:

```
class Circle {
  // does NOT override the volume() method in Shape!
  // (we forgot the const)
  virtual double volume();
}
```

- Besser: `override`, um Tippfehler zu vermeiden:

```
class Circle {
  double volume() const override; // ok
  // compile error: no virtual function defined in base class
  double volume() override;
```

- Immer auch den Destruktor `virtual` machen, ansonsten oft Speicherlücken und ähnliche Probleme!

Abstrakte Klassen

- Es gibt auch die Möglichkeit abstrakte Klassen, welche nicht instantiiert werden können zu implementieren.

```
class Abstract {
  virtual void doSomething() = 0; // pure virtual function
};
```

- Abstrakte Klassen werden typischerweise zur Definition von Interfaces genutzt.
- Eine Klasse wird abstrakt durch Definition einer rein virtuellen Funktion.
- Die Ableitende Klasse muss die rein virtuelle Funktion implementieren!

```
class Derived {
  void doSomething() override { ... }
};
```

Ableitung von mehreren Klassen

- Eine Klasse kann auch von mehreren Basisklassen ableiten
- Hierbei kann das virtual Keyword verhindern dass die Basisklasse mehrfach in der abgeleiteten Klasse vorkommt.

```
class Base {
public:
        int n;
        Base(int x) : n(x) {}
};
class D1 : virtual Base { public: D1() : Base(1) {} };
// virtual prevents multiple instaces of Base in inheritance tree
class D2 : virtual Base { public: D2() : Base(2) {} };
class DMulti : D1, D2 { public: DMulti() : Base(3), D1(), D2() };
```

- In der abgeleiteten Klasse müssen alle Konstruktoren der Basisklassen aufgerufen werden ansonsten wird der Default-Konstruktor wendet (wenn möglich).
- Die Klassen von denen abgeleitet wird müssen keine gemeinsame Basisklasse haben.

Speichern von polymorphen Objekten

- Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt: `sizeof(Circle) != sizeof(Rectangle)`
- Container brauchen Objekte fixer Grösse
- Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Speichern von polymorphen Objekten

- Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt: `sizeof(Circle) != sizeof(Rectangle)`
- Container brauchen Objekte fixer Grösse
- Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Lösung: Dynamische Speicherverwaltung

- Wir speichern eine Liste von Pointern auf Objekte und lassen uns dynamischen Speicher für das eigentliche Objekt geben
- Man spricht davon, dass das Objekt auf dem Heap angelegt wird, normale Variablen liegen auf dem Stack
- Objekte auf dem Heap werden NICHT automatisch aufgeräumt, wenn das aktuelle Scope endet
- Bei manueller Verwaltung: Speicher geht eventuell verloren
- Daher: smart pointer verwenden!

Smart Pointers

- Ein Smart Pointer reserviert Speicher für ein Objekt auf dem Heap und räumt das Objekt auf, wenn es nicht mehr verwendet wird.
- `unique_ptr` erzeugt das neue Objekt beim Anlegen und gibt es frei, sobald die Pointer-Variable aufhört zu existieren:

```cpp
#include <memory>

std::unique_ptr<int> foo(int i) {
  return std::make_unique<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  return *p + b;
} // memory gets freed here
```

- `unique_ptr` kann nur verschoben werden, nicht kopiert

Anwendungsbeispiel mit unique pointer

- Bei Containern unbedingt mit inplace mit `emplace_back` anlegen.
- `push_back` macht eine Kopie, was bei unique Pointern nicht erlaubt ist.

```
vector<unique_ptr<Base>> v;
v.emplace_back(make_unique<Base>(Base()));
v.emplace_back(make_unique<Derived1>(Derived1()));
v.emplace_back(make_unique<Derived2>(Derived2()));

for (auto& br : v)
  br->foo();
```

Smart Pointers für geteilte Objekte

- Oft ist es nicht möglich, einen eindeutigen Eigentümer für ein Objekt festzulegen.
- Hierfür gibt es `shared_ptr`.
- Mehrere `shared_ptr` können auf das gleiche Objekt zeigen.
- Das Objekt wird genau dann freigegeben, wenn der letzte `shared_ptr` auf das Objekt zerstört wird.
- Wichtig: `shared_ptr` immer nur mit `make_shared` anlegen oder aus anderen `shared_ptrn` kopieren!

```cpp
#include <memory>

std::shared_ptr<int> foo(int i) {
  return std::make_shared<int>(i);
}

int add(int a, int b) {
  auto p = foo(a);
  auto p2 = p;
  return *p + *p2 + b;
} // memory gets freed here
```

Zusammenfassung

- Programm entscheidet zur Laufzeit, welche Methode ausgeführt wird.
  - Vorteil: Hohe Flexibilität (die gleiche Funktion kann zur Laufzeit für zwei Objekte unterschiedlichen Typs jeweils die richtige Methode aufrufen).
  - Nachteil: Laufzeit-Overhead (die richtige Methode muß zur Laufzeit identifiziert werden).
- Erfordert Planung und Disziplin beim Programm-Design:
  - Gemeinsame Hierarchie für alle Klassen.
  - Gemeinsame Funktionalität muß in Basisklasse vorgesehen sein (`virtual`-Deklarationen).
  - Vorhandene Klassen (z.B. aus Standard Library) nicht integrierbar.