Praxisorientierte Einführung in C++
Lektion: "Klassen und Strukturen"

Christof Elbrechter

Neuroinformatics Group, CITEC

May 22, 2014

Table of Contents

Überblick

- Neben Funktionen und Kontrollstrukturen sind Strukturen wichtiger Baustein der strukturierten Programmierung
- Strukturen dienen der Aggregation von Daten

Beispiele

- Datensätze in einer Personalverwaltung (Name, Anschrift, ...)
- Spieler in einem Spiel (Name, Position, Orientierung, Frags, ...)
- Komplexe Zahlen (Real- und Imaginärteil)
- Vektoren, Matritzen (Komponenten)

Simple Strukturen (C)

- Strukturen werden durch Schlüsselwort struct definiert

struct-Definition

```
struct Bezeichner {
   // Elemente
};
```

Wichtig

In C++ enden Klassen und Struktur-Definitionen **immer** mit einem Semikolon

- Die Elemente sind Variablendeklarationen beliebiger anderer Typen
- Bezeichner ist nach Definition ein neuer Typ

Beispiel

Point2D.h

```
struct Point2D{
  float x;
  float y;
};
```

- Point2D kann nun wie ein eingebauter Typ verwendet werden:

Verwendung des neuen Typs Point2D

```
Point2D myPoint;
Point2D a,b,c;
Point2D *points = new Point2D[10];
delete [] points;
```

Zugriff auf Elemente

- Zugriff auf Elementvariablen erfolgt mit dem .-Operator

```
Point2D p;

p.x = 4;
p.y = 7;

std::cout << "Position: "
          << " x:" << p.x
          << " y:" << p.y
          << std::endl;
```

Strukturen Zeiger

- Zugriff über Zeiger mit dem ->-Operator

```
Point2D *p = new Point2D; // alternativ: new Point2D()

p->x = 4;
p->y = 7;

std::cout << "Position: "
          << " x:" << p->x
          << " y:" << p->y
          << std::endl;

// oder natuerlich auch so (gilt aber als schlechter Stil)
(*p).x = 5;

// oder
p[0].x = 4;
```

Initialisierung von Strukturen

Strukturen können ähnlich wie Arrays per Liste initialisiert werden (C-Relikt):

```
Point2D p = {1.0, -1.0};
```

- Kurzschreibweise, um den gesamten Speicher der Struktur (binär) mit 0 zu initialisieren:

```
Point2D meinPunkt = {0};
```

- Das geht allerdings **nur** für 0

Zuweisung

- Bei Zuweisung wird eine Stuktur standardmäßig *binär* kopiert.

Beispiel

```
struct Matrix{
   float *data;
   int dim[2];
   bool transposed;
};
int main(){
  Matrix a = {new float [20], {2,10}, false};
  Matrix b = a;
}
```

- Einzelne Datenelemente werden *tief* kopiert
- Arrays konstanter Größe werden *tief* kopiert
- Zeiger werden allerdings *flach* kopiert. D.h. der Zeiger wird kopiert, und nicht das worauf der Zeiger zeigt

Genau genommen ...

- Tatsächlich wird für jedes Element der sog. Copy-Constructor aufgerufen
- Das war der C-Teil:
- C++:
  - Methoden, Konstruktor(en), Destruktor
  - Zugriffsstufen (Sichtbarkeit)
  - Operatoren
  - Templates
  - Vererbung

Methoden

- Methoden erlauben es, Funktionen an Daten zu binden

Point2D.h

```
struct Point2D {
   // Elemente
   float x;
   float y;

   // Methodendeklarationen
   void translate(float dx, float dy);
   void rotate(float angle);
};
```

Aufruf von Methoden

- Aufruf von Methoden auch über . und ->-Operator

```
#include <Point2D.h>
#include <cmath>
int main() {
   Point2D p={0};
   p.translate(4,3);

   p.rotate(M_PI/2);

   Point2D *pp = new Point;
   pp->x = pp->y = 0;
   pp->translate(4,3);
   pp->rotate(M_PI/2);
   delete pp;
}
```

Implementierung von Methoden

- Implementierung kann direkt im Header in der Klassendefinition erfolgen
  - Ermöglicht *inline* Übersetzung
  - Compiler muss die Methode dann aber bei jeder Verwendung übersetzen
  - Verlangsamt Entwicklung in großen Projekten sehr stark
  - Sollte nur in Ausnamefällen verwendet werden
- ... oder separat im zugehörigen .cpp-File
  - Methode muss nur einmal übersetzt werden
  - Methode wird nicht mehr `inline` übersetzt[1]
  - Ändert sich die Methoden-Implementierung muss **nur** das zugehörige .cpp-File übersetzt werden, und nicht alle Files, in denen die Methode aufgerufen wird

[1] tatsächlich kann der Compiler die Methode innerhalb des .cpp-Files immer noch `inline` übersetzen

Inline-Implementierung von Methoden

- Bei der Inline Implementierung kann die Methode entweder direkt deklariert und implementiert werden

Point2D.h

```
class Point2D{
   float x,y;

   inline void translate(float dx, float dy){ // (inline)
      x += dx;
      y += dy;
   }
   // weitere Methoden ...
};
```

Inline-Implementierung von Methoden

- Einige Programmierer bevorzugen allerdings die eigentliche Klassendefinition übersichtlicher zu halten:

```
// Reine Klassendefinition
class Point2D{
   float x,y;
   void translate(float dx, float dy);
   // Weitere Methoden ...
};

// Implementierung von inline-Methoden anschliessend
inline void Point2D::translate(float dx, float dy){
  x += dx;
  y += dy;
}
// Implementierung weiterer inline-Methoden
```

Nicht-inline-Implementierung von Methoden

- Von der Separierung von Klassendefinition und Methoden-Implementierung ist es nun nur noch ein kleiner Schritt zur *normalen* Methoden Implementierung

Point2D.h

```
class Point2D{
   float x,y;
   void translate(float dx, float dy);
};
```

Point2D.cpp

```
#include <Point2D.h>
void Point2D::translate(float dx, float dy){
  x += dx;
  y += dy;
}
```

Überladen von Methoden

- Für das Überladen von Methoden gelten die gleichen Regeln, die auch für das Überladen von globalen Funktionen gelten

Beispiel

```
struct Point2D {
   float x;
   float y;

   inline void rotate(float angle){
      Point2D p0 = {0};
      rotate(p0,angle);
   }

   void rotate(const Point2D &origin, float angle){
      // ...
   }
};
```

Der this-Zeiger

- Falls Methoden-Argumente den gleichen Bezeichner wie Member-Variablen haben, muss der this-Zeiger verwendet werden

Beispiel

```
struct Point2D {
   float x;
   float y;

   void setX(float x){
      // x = x; ?? -> welches x ist gemeint?
      this->x = x; // x: local scope, this->x: member
   }
};
```

Sichtbarkeits-Modifikatoren

- Schlüsselwörter `public`, `protected` und `private` dienen dem *Information-Hiding*
- C++: Sichtbarkeit-Scopes, d.h., nach dem `private`:-Label ist solange alles versteckt, bis `protected`: oder `public`: folgt

Point2D.h

```
struct Point2D {
  private: // versteckte Daten und Funktionen
    float m_x; // m_ wird gerne fuer 'Member' verwendet
    float m_y;
  public: // sichtbare Daten und Funktionen
    void translate(float dx, float dy){
       m_x += dx;
       m_y += dy;
    }
    float getX() { return m_x; }
    float getY() { return m_y; }
};
```

Sichtbarkeits-Modifikatoren

- `public`: Von außen sichtbar
- `protected`: In der aktuellen und in Unterklassen sichtbar – nicht aber von außen
- `private`: Nur in der aktuellen Klasse sichtbar

Klassen vs Strukturen

- Klassen werden mit dem Schlüsselwort `class` deklariert,
  Strukturen (wie bereits gesehen) mit `struct`
- Es existieren einige Mythen bzgl. dem Unterschied von Klassen und Strukturen
  - Es gibt z.B. das Gerücht, Strukturen wären C-Relikte und dürften daher keine Methoden haben ($\Rightarrow$ falsch!)
- Die einzigen Unterschiede sind marginal:

class vs struct

1. Bei Klassen ist die Standardsichtbarkeit `private`,
   bei Strukturen ist sie `public`
2. Später werden wir sehen, dass Klassen auch standardmäßig *private beerbt* werden und Strukturen *public*

Konstruktoren und Destruktoren

- Bei komplexen Typen ist manuelle Initialisierung bei Erstellung umständlich
- Konstruktor ist spezielle Methode
- Konstruktor wird bei der Erstellung des Objekts aufgerufen nachdem Speicher für das Objekt reserviert wurde
- Gegenstück: Destruktor
- Destruktor wird bei der Freigabe des Objekts aufgerufen bevor der Speicher freigegeben wird

Konstruktoren und Destruktoren (Stack vs Heap)

- Objekte, die auf dem Stack alloziert werden, werden automatisch freigegeben, wenn der aktuelle Stack-Frame verlassen wird

```
void foo(){
  Point2D p1; // Konstruktor wird aufgerufen
  Point2D p2; // Konstruktor wird aufgerufen
  //...
}// Destruktoren werden in umgekehrter Reihenfolge aufgerufen
```

- Bei Objekten die auf dem Heap (mit `new` oder `new[]`) alloziert wurden, erfolgt der Destruktor-Aufruf erst bei der expliziten Speicherfreigabe (mit `delete` oder `delete[]`)

```
void foo(){
  Point2D *p1 = new Point2D; // Konstruktor wird aufgerufen
  Point2D *p2 = new Point2D; // Konstruktor wird aufgerufen
  //...
}// Destruktoren werden nicht automatisch aufgerufen
```

Konstruktoren und Destruktoren (Stack vs Heap)

```
void foo(){
  Point2D *p1 = new Point2D; // Konstruktor wird aufgerufen
  Point2D *p2 = new Point2D; // Konstruktor wird aufgerufen
  //...
  delete p1; // Destruktor wird aufgerufen
  delete p2; // Destruktor wird aufgerufen
}
```

- Für dynamische und fixed-size Arrays wird für jedes Element der Konstruktor/Destruktor aufgerufen

```
void foo(){
  Point2D *p1 = new Point2D[5]; // Konstruktor wird 5x aufgerufen
  Point2D p2[5]; // Konstruktor wird 5x aufgerufen
  //...
  delete [] p1; // Destruktor wird 5x aufgerufen
}// Destruktor wird 5x fuer p2 aufgerufen
```

Der Konstruktor

- Wenn `Typ` Name einer Klasse ist, dann heißen auch die Konstruktoren `Typ`
- Es kann verschiedene Konstruktoren mit unterschiedlichen Argumenten geben (Überladung)
- Der Konstruktor hat keinen Rückgabetyp (nicht mal `void`)

Point2D.h

```
struct Point2D {
   float m_x;
   float m_y;
   // Sog. Default-Konstruktor (oder leerer Konstruktor)
   Point2D();
   // Ueberladener bestimmter Konstruktor
   Point2D(float x, float y);
   // ... andere Methoden
};
```

Der Konstruktor

- Separate Implementierung: wie bei anderen Methoden
- Jedoch auch hier: Ohne Angabe eines Rückgabetyps

Point2D.cpp

```
#include <Point2D.h>
Point2D::Pointd2D(){
    m_x = 0;
    m_y = 0;
}
Point2D::Point2D(float x, float y){
    m_x = x;
    m_y = y;
}
```

Hinweis

Von nun an werden wir das meiste direkt `inline` in der Klassendefinition implementieren. Dieses ist zwar nicht so gebräuchlich, erhöht aber die Übersicht!

Wann wird welcher Konstruktor aufgerufen?

Stack

```
Point2D p; // Default-Konstruktor (x,y)=(0,0)
// Nun: Java-Programmierer aufgepasst!
Point2D p(2,3); // Bestimmter Konstruktor
                // Java-Programmierer muessen sich davon verabschieden,
                // dass man staendig new verwenden muss

// Beliebte Fehler:
Point2D p = new Point2D(2,3);  // Fehler new gibt Pointer zurueck
Point2D p = *new Point2D(2,3); // wird uebersetzt, aber SEPICHERLECK: erzeugt zunaechst einen
                               // Point2D auf dem Heap, und kopiert diesen auf den Stack -> der
                               // Point2D auf den Heap wird niemals freigegeben

Point2D ps[3]; // Fuer jedes Element: Standardkonstruktor
Point2D ps[3] = { Point2D(1,2),
                  Point2D(3,4),
                  Point2D(5,6) }; // 3x bestimmter Konstr.
```

Heap

```
Point2D *pp = new Point2D; // Standard-Konstruktor
Point2D *pp = new Point2D(3,4); // Bestimmter Konstruktor
Point2D *pp = new Point2D[4]; // 4x Standard-Konstruktor
```

Konstruktoren von aggregierten Objekten

- Wenn eine Struktur andere Strukturen oder Klassen als Elemente enthält, werden Elemente mit Default-Konstruktor initialisiert

Line2D.h

```
#include <Point2D.h>
struct Line {
   Point2D start;
   Point2D end;
   Line2D(){
     // hier sind start und end schon initialisiert
     // d.h. fuer start und end wurde bereits der
     // Standard-Konstruktor aufgerufen
   }
   Line2D(const Point2D &start, const Point2D &end){
     // hier auch -> sub-optimal
     this->start = start;
     this->end = end;
   }
};
```

Initialisierungsliste

- Im Falle des `Line2D(const Point2D&,const Point2D&)` Konstruktors werden die Elemente zunächst mit (0,0) und dann erst mit den richtigen Werten initialisiert
- In diesem Fall vernachlässigbar
- **Aber:** In vielen Fällen bedeutet dies erheblichen Mehraufwand
- **Ausweg:** Verwendung der sog. Initialisierungsliste

Line2D.h

```
struct Line {
   Point2D start,end;
   Line2D():start(0,0),end(0,0){} // nicht unbed. notwendig!
   Line2D(const Point2D &start, const Point2D &end):
      start(start),end(end){} // Hier wird der sog. Copy-Konstr.
                              // aufgerufen -> kommt gleich
};
```

Initialisierungsliste

- Die Initialisierungsliste beginnt mit einem Doppelpunkt
- Sie enthält Konstruktoraufrufe für die Member-Variablen
- Für alle Members, die nicht in der Initialisierungsliste auftauchen wird automatisch der Default-Konstruktor aufgerufen
- Members werden **immer** automatisch in ihrer Auftrittsreihenfolge in der **Klassendefinition** initialisiert
- Initialisierunsliste sollte die gleiche Reihenfolge verwenden (meist warnt der Compiler)

```
stuct Point2D{
   int x,y;                          // Warnung:
   Point2D(int x, int y):y(y),x(x){} // falsch-herum!
};
```

Konstruktoren und Default-Argumente

- Wie für globale Funktionen und Methoden können auch Konstruktoren Default-Argumente haben
- In vielen Fällen kann damit der Default-Konstruktor sinnvoll mit-abgefangen werden

Point2D.h

```
struct Point2D{
  int x, y;
  Point2D(int x=0, int y=0):x(x),y(y){}
};
```

- Die obige Implementierung implementiert beide Konstruktoren zusammen ohne deren Semantik zu ändern
- Wird der leere Konstruktor verwendet, so werden x und y automatisch mit 0 initialisiert

Der Copy-Konstruktor

- Spezieller Konstruktor
- Erlaubt zu bestimmen, was bei Initialisierung mit anderem Objekt des selben Typs passiert

Wann wird der Copy-Construktor aufgerufen?

```
Point2D a;

Point2D b(a);   // Copy-Construktor
Point2D c = a;  // Copy-Construktor (Sonderfall!)

Point2D d;
d = a;          // Zuweisungsoperator (->spaeter)
```

- Viele andere Programmiersprachen: implizit flache Kopie
- In C++ kann das Verhalten individuell bestimmt werden

Wann wird der Copy-Konstruktor Aufgerufen?

- Darüber muss man sich im Klaren sein ($\Rightarrow$ Performance)

```
void set_pos(Point2D p){ ... }
Point2D get_pos(){ ... }

int main(){
  // Call-by-Value
  Point2D p;
  set_pos(p);            // Copy-Konstruktor

  // Dies gilt allerdings nicht fuer Temporaries
  set_pos(Point2D());    // nichts!

  // Aehnliches gilt fuer den Zuweisungsoperator
  Point2D p;
  p = get_pos();         // Zuweisungs-Operator

  // Bei direkter Initialisierung allerdings nicht!
  Point2D p = get_pos(); // nichts!
}
```

Wie Implementiert man den Copy-Konstruktor?

Syntax: Copy-Construktor (am Bsp. Point2D)

```
struct Point2D{
   Point2D(const Point2D &other); // Copy-Construktor
};
```

- Für eine Klasse `X` hat der Copy-Construktor immer die Signatur `X(const X&)`
- Andere Sachen wie `X(X)` oder `X(X&)` sind nicht erlaubt
- Pointer-Versionen (z.B. `X(X*)`) sind möglich, sind aber nur normale Konstruktoren

Automatisch generierte Klassenbestandteile

C++ generiert automatisch für jede Klasse die folgenden Bestandteile

Copy-Konstruktor, Default-Konstruktur, Zuweisungs-Operator und Destruktor

- **Es gilt allerdings:**
  - Alle können auch selbst implementiert werden
  - Der Default Konstruktor wird nur generiert, falls kein anderer Konstruktor deklariert wurde (auch wenn es nur der Copy-Konstruktor ist)
- Die automatisch generierten Bestandteile verhalten sich nach einfachen Regeln:
  - Der automatisch generierte Default-Konstruktor ruft die Default-Konstruktoren aller Member auf
  - Automatisch generierte Kopien rufen für alle Elemente den Copy-Construktor auf
  - Default-Destruktor ruft die Destruktoren der Members auf

Noch was zum Thema Konstruktor ...

Gemein:

Sobald irgendein Konstruktor definiert wurde, funktioniert die C-Zuweisung mit den Array-Klammern nicht mehr

z.B.

```
Point2D p = {1,3};
```

Default-Konstrukturen von POD-Typen

- Die Default-Konstruktoren von builtin-Typen machen **nichts**

Wichtig

- Nach dem Ausdruck
  `int i;`
  wurde nur der Speicher für `i` reserviert
- Der Inhalt von `i` ist dann zufällig (je nachdem was vorher dort im Speicher stand
- Sehr oft steht dort eine 0, was die Sache allerdings eher schlimmer macht!

- Somit machen auch die Default-Konstruktoren von Typen, die nur POD-Typen enthalten, **nichts**
- Z.B. `struct Point2D{ int x,y; };`

Wann benötigt man einen speziellen Copy-Konstruktor?

- Lässt sich grundsätzlich nicht sagen

I.d.R. gilt:

Verwendet einen Klasse explizit dynamischen Speicher, welcher im Konstruktor mit `new` oder `new[]` alloziert wird, so wird auch ein **Destruktor**, **Copy-Konstruktor** und **Zuweisungs-Operator** benötigt!

- In Ausnamefällen gilt das allerdings nicht
- Der Grund hierfür ist vor allem der **Destruktor**, daher wird dieser zunächt erläutert, bevor ein komplexeres Beispiel vorgestellt wird

Implementierung des Destruktors

- Wie bereits zuvor gesehen: Destruktor wird **automatisch** aufgerufen bevor der Speicher eines Objektes freigegeben wird
- Jede Klasse hat immer nur einen Destruktor
- Der Destruktor hat weder Rückgabe-Typ noch Argumente
- Der Destruktor heißt wie die Klasse mit vorangestellter Tilde

Syntax: Destruktor (am Bsp. Point2D)

```
struct Point2D{
   ~Point2D(); // Destruktor
};
```

- I.d.R. werden im Destruktor mögliche Resourcen freigegeben:
  - Speicher
  - Netzwerk-Sockets
  - File-Handles
  - Geräte-Locks

Simples Beispiel: Player

```
#include <cstring>
#include <Point2D.h>
class Player{
  char *name;  // name des Spielers
  Point2D pos; // aktuelle Position
  // Hilfsfunktion
  void save_assign(const char *name, const Point2D &pos);
public:
  // Default Konstruktor
  Player();
  // Konstruktor
  Player(const char *name, const Point2D &pos);
  // Copy-Konstruktor
  Player(const Player &p);
  // Zuweisungsoperator
  Player &operator=(const Player &p);
  // Destruktor
  ~Player();
};
```

Simples Beispiel: Player

```
Player::Player():name(0){}
Player::Player(const char *name, const Point2D &pos):name(0){
   save_assign(name,pos);
}
Player::Player(const Player &p):name(0){
   save_assign(p.name,p.pos);
}
Player &Player::operator=(const Player &p){
   save_assign(p.name,p.pos);
   return *this;
}
Player::~Player(){  if(name) delete [] name; }
```

Simples Beispiel: Player

```
void Player::save_assign(const char *name, const Point2D &pos){
   this->pos = pos;
   if(this->name) delete [] this->name;
   if(name){
      int namelen = strlen(name)+1;
      this->name = new char[namelen];
      for(int i=0;i<namelen;++i){
        this->name[i] = name[i];
      }
   }else{
      this->name = 0;
   }
}
```

Entwarnung

- Ok, das muss aber nicht so kompliziert sein (war nur zur Verdeutlichung)
- C++ bietet eine super String-Klasse `std::string`

Player.h (diesmal mit STL string)

```
#include <string>
#include <Point2D.h>
class Player{
   std::string name;
   Point2D pos;
   Player(const std::string &name="",
          const Point2D &pos=Point2D()):
     name(name),pos(pos){}
};
```

Merke ...

Falls möglich *managed* STL-Datentypen verwenden $\Rightarrow$ einfacher!

Vorausdeklaration von Klassen

- Falls Klassen sich gegenseitig (als Pointer) enthalten sollen, benötigt man analog zur Vorausdeklaration von Funktionen eine Vorausdeklaration von Klassen
- Gutes Beispiel: *Verkettete Liste*
  - `List` enthält `Note *first;`
  - `Node` enthält `List *parent;`

Beispiel Vorausdeklaration

```
class Node;
```

- Analog zur Vorausdeklaration von Funktionen kann die Klassen im Nachhinein normal definiert werden
- Bevor die Tatsächliche Klassendefinition nicht zur Verfügung steht
  - ... sind nur Pointer und Referenzen davon erlaubt
  - ... können keine Funktionen der Klasse Verwendet werden
  - ... kann nicht auf Membervariablen zugegriffen werden

Vorausdeklarationsbeispiel List und Node

list.h

```
// Node wird vorausdeklariert -> daher
// kein #include <node.h>
class Node;
struct List{
  Node *first;
  int size;
  // ...
};
```

node.h

```
#include <list.h>
// hier ist List bereits bekannt -> daher
// keine Vorausdeklaration notwending
struct Node{
  List *parent;
  Node *next;
  // ...
};
```

const Instanzen und Funktionen

- Wenn ein Objekt `const` ist, so sind automatisch alle Member-Variablen `const`

const Objekte

```
struct Point{
  int x,y;
};

int main(){
  Point p;
  p.x = 5;
  p.y = 7;

  const Point k;
  k.x = 5; // Fehler: x ist const
}
```

const Instanzen und Funktionen

- Die gilt auch manchmal obwohl die eigentlichen Objekt-Instanz gar nicht `const` ist

Beispiel

```
// const-Versprechen: Funktion verspricht,
// dass p nicht veraendert wird
void use_point(const Point &p){
   p.x = 5; // Fehler: const-Versprechen gebrochen
}

int main(){
  Point p; // hier nicht const
  foo(p);
}
```

const Objekte

- Wenn ein Objekt const ist, so dürfen auch nur Methoden darauf aufgerufen werden, welche die Member-Variablen des Objektes nicht verändern
- Wie kann der Übersetzer ermitteln, welche Methoden das sind?
- **Antwort: Gar nicht!** $\Rightarrow$ Muss manuell angegeben werden!

const ...

Funktionen, welche ein Objekt nicht verändern, müssen[a] explizit `const` deklariert werden!

[a] tatsächlich muss man es nicht machen, man bekommt dann aber i.d.R. Probleme

const Methoden

Beispiel: const Methoden

```
class Point{
   int x,y;
 public:
   // setter: nicht const
   inline void setX(int x){ this->x=x; }
   inline void setY(int y){ this->y=y; }
   // getter: veraendern das Objekt nicht
   inline int getX() const { return x; }
   int getY() const; // const gehoert mit zur Signatur!
};
// bei getrennter Impl. muss const mit angegeben werden
inline int Point::getY() const { return y; }
int main(){
   const Point p;
   std::cout << p.getX() << std::endl; // ok
   p.setX(7); // Fehler
}
```

const Methoden

- Damit nicht *aus Versehen* innerhalb einer `const`-Methode doch Member-Variablen verändert werden, sind diese innerhalb der Implementierung automatisch auch `const`
- Bei Member-Variablen die Pointer sind, sind allerdings nur die Pointer und nicht die dahinterliegenden Daten const
- Standardmäßig bezieht sich also die *const-ness* darauf, dass sich das Binärmuster der Objekt-Instanz nicht ändert
- *Constness* dient vor allem der Optimierung
- Z.B. können Speicherbereiche, die nur gelesen werden, an verschiedenen Stellen gleichzeitig verwendet werden (usw...)

const Methoden

```
#include <string>
#include <set>

class Car{
 public:
   std::string getVendor(std::string prospect) const{
     if(!prospects.count(prospect)){
       prospects.insert(prospect); // Fehler: nicht erlaubt
     }
     return vendor;
   }
 private:
   // Liste von Interessenten
   std::set<std::string> prospects;
   // Hersteller
   std::string vendor;
   // ...
};
```

Das Schüsselwort: mutable

- Es existieren Ausnahmefälle, in denen auch in `const` Funktionen Member-Variablen verändert werden müssen
- In diesem Fall müssen die entsprechen-den Variablen `mutable` deklariert werden

mutable deklarierte Member-Variablen ...

- ... dürfen auch in `const` Funktionen verändert werden!
- ... dürfen auch von außen bei `const` Instanzen verändert werden
- ... $\Rightarrow$ sind niemals `const`

Beispiel: Car mit mutable-Schlüsselwort

```
#include <string>
#include <set>

class Car{
 public:
   std::string getVendor(std::string prospect) const{
     if(!prospects.count(prospect)){
       prospects.insert(prospect); // nun gehts!
     }
     return vendor;
   }
 private:
   // Liste von Interessenten
   mutable std::set<std::string> prospects;
   // Hersteller
   std::string vendor;
   // ...
};
```

Anmerkungen zum Schlüsselwort mutable

- Standard `const`-Konzept von C++ bezieht sich auf *binär-Kompatiblität*
- Das ist ein guter Standard, in vielen Fällen aber nicht angebracht
- Prominentes Beispiel: Klassen, deren Methoden aus unterschiedlichen Threads simultan aufgerufen werden können (Thread-safe Klassen)
- I.d.R. muss paralleler Zugriff auf die Member-Variablen von den Funktionen aus durch einen *Mutex* geschützt werden
- Die Mutex Variable ist dann i.d.R. `mutable`

mutable ...

Auch wenn `mutable` relativ selten verwendet wird, ist es doch ein wichtiges Werkzeug, um ein eigenes const-Konzept zu definieren

const und nicht const ??

- In einigen Fällen müssen Funktionen einmal `const` und einmal nicht-`const` implementiert werden
- Hierfür wollen wir noch mal ein etwas komplexeres Beispiel betrachten: eine Vector-Klasse

Vector.h

```
class Vector{
  float *data;
  int dim;
public:
  Vector(int dim=1) { data = new float[dim]; }
  ~Vector() { delete [] data; }

  float at(int idx) const { return data[idx]; }
  void set(int idx, float val){ data[idx] = val; }
};
```

- **Problem:** relativ unschöne Syntax `v.set(idx,val)`

const und nicht const ??

- Schöner: `v.at(idx)` *referenziert* einfach das entsprechende Vector-Element
- Damit könnte `v.at(idx)` zum Lesen und zu Schreiben verwendet werden
- Sehr gängiges Konzept in C++ (man benötigt aber Referenzen)

Vector.h

```
class Vector{
  // Daten, Konstruktur und Destruktor ...
  float& at(int idx) { return data[idx]; }
};
```

const und nicht const ??

- **Problem:** Soll `at(idx)` `const` oder nicht-`const` sein?
  - `const`-Fall: nur lesen
  - nicht-`const`-Fall: lesen und schreiben
- **Lösung:** beides anbieten (const-Überladung ist nämlich möglich)

Vector.h

```
class Vector{
  // Daten, Konstruktur und Destruktor ...
  float& at(int idx) { return data[idx]; }
  const float& at(int idx) const { return data[idx]; }
};
```

`const`-Überladung

Wird eine Funktion als `const` und als nicht-`const`-Version angeboten, so wählt der Compiler immer automatisch die richtige Version aus!