2. Object Oriented Programming und Templates

AlDaBi Praktikum

René Rahn
WS 2014/15

Fundamental Programming Styles

PROGRAMMING PARADIGMS

Imperative Paradigmen

- Programm besteht aus Anweisungen, die den Status des Programms ändern
- Programmierer beschreibt, wie diese Anweisungen auszuführen sind, ähnlich einem Kochrezept
- Wiederverwendbare Anweisungsblöcke werden meist in Funktionen ausgelagert
- Ergebnisse von Funktionen sind oftmals vom Programmstatus abhängig

Beispiel Naives Pattern Matching

```
#include <iostream>
#include <string>
#include <vector>

// Iteration über das Pattern (inner loop)
int computeLocalScore(std::string const & text, std::string const & pattern,
                      int const &  textPos) {...}
// Iteration über den Text (outer loop)
std::vector<int> computeScore(std::string const & text, std::string const &
                              pattern){...}
int main() {
    // Initialisierung
    std::string text = "This is an awesome tutorial to get to know some C++!";
    std::string pattern = "some";
     // Computation of the similarities
    std::vector<int> score = computeScore(text, pattern);
    // Ergebnisausgabe
    for (unsigned i = 0; i < score.size(); ++i)
        std::cout << score[i];
    std::cout << std::endl;
    return 0;
}
```

OOP Paradigmen

Ausführung von Anweisungen mit Hilfe von interagierenden Objekten

- Objekte sind reale oder abstrakte Einheiten mit einer best. Rolle
- Objekte haben klar definierte Verhaltensweisen
- Objekte enthalten private Informationen
- Objekte interagieren miteinander über ihre Funktionalitäten

Programmierer entwerfen ein Set von Objekten für das aktuelle Problem

- Menschen sehen die Welt als eine Zusammenwirkung von Objekten

Wichtig für Softwarekonzeption, Wartung und Erweiterbarkeit

- Objekt ist kleinste Moduleinheit, erweiterbar und wiederverwendbar

Moderne Programmiersprachen unterstützen OOP

- Java und C++ wurden direkt für OOP konzipiert

OOP Prinzipien

Abstraktion

- Funktionalität gegeben über Schnittstellen (Memberfunktionen) wie bei abstrakten Datentypen
- Komplexität der Implementierung beschränkt sich auf das Objekt, d.h. für den Benutzer nicht notwendig und ersichtlich

Datenkapselung

- Interner Status des Objektes nicht nach außen ersichtlich
- Objekt (Member) kann nur über seine Methoden verändert und eingesehen werden
- Methoden sichern ab, dass Member nur gültige Zustände erreichen

Vererbung

- Erlaubt Wiederverwendung und Erweiterung von Objekten

Polymorphismus

- Erlaubt Spezialisierung von Subtypen von Objekten (z.B. nach Vererbung)

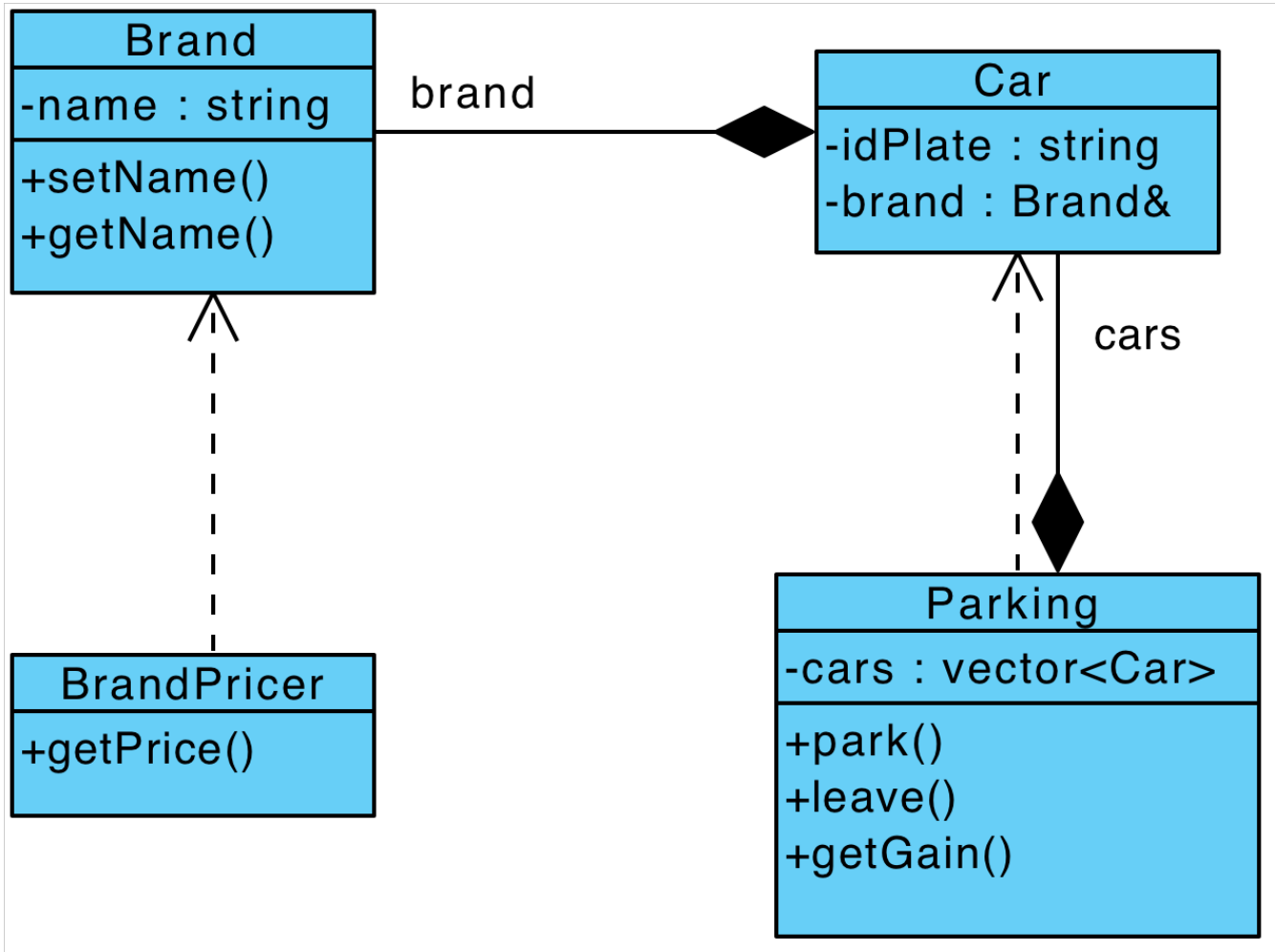
Beispiel Parkplatz für Autos

- Anweisungen
  - Programmiere Anwendung zum Parkplatzmanagement
  - Berechne zu jeder Zeit den aktuellen Gewinn durch die parkenden Autos
  - Parkgebühr abhängig von der Automarke

- Objekte
  - Parking: enthält Autos
  - Car: Auto
  - Brand: Automarke
  - BrandPricer: Bestimmt Gebühr anhand der Automarke

Beispiel entnommen aus der Vorlesung „La programmation objet “ of Rémi Forax, WS07, Univ. Paris-Est MLV

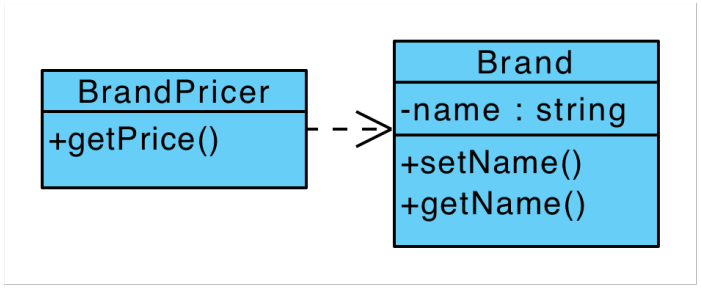
Beispiel Parkplatz für Autos (II)



Objektbeziehungen

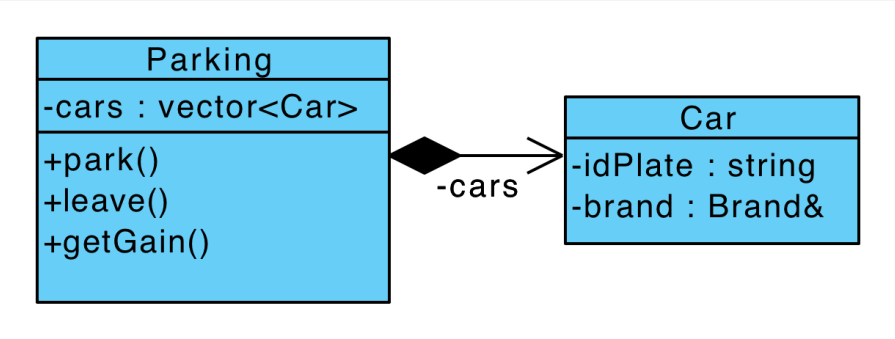
Assoziation

- Generische Beziehung zwischen zwei Objekten
- Objekte benutzen andere Objekte oder stellen sie zur Verfügung



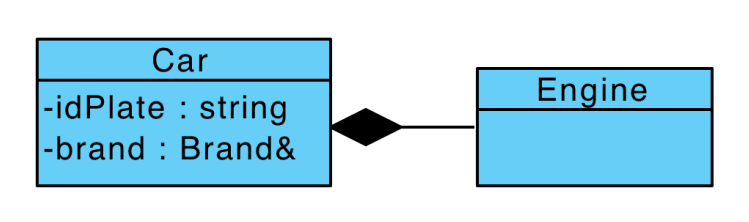
Aggregation

- „Hat ein(e)“ Beziehung
- Tritt z.B. in Containern (z.B. Vektoren) auf



Komposition

- „Teil eine(s)“ Beziehung
- Objekt existiert nur als Teil eines anderen Objektes



A Very Brief Overview

OOP IN C++

Klassen und Objekte

Klasse definiert *Implementierung* von Objekten

```
class Car
{   // private:
};
```

```
struct Car
{   // public:
};
```

Objekt ist Instanz einer Klasse

– a, b, c sind alles Instanzen derselben Klasse Car

```
void main() {
   Car a;
   Car b;
   Car c;
}
```

Member

- Eigenschaften eines Objektes sind die Member (Typen, Variablen) der Klasse
- Sichtbarkeit der Member kann eingeschränkt sein
  - Schlüsselwort *private* nur sichtbar innerhalb der Klasse
  - Schlüsselwort *protected* erlaubt Sichtbarkeit für Subklassen
  - Schlüsselwort *public* erlaubt Sichtbarkeit für alle Klassen

```
class Car {
private:
  string idPlate;
protected:
  unsigned seats;
public:
  string brandName;
};
```

```
void main() {
  Car c;
  // OK
  c.BrandName = "BMW";
  // Compile Error
  c.seats = 5;
  // Compile Error
  cout << c.idPlate;
}
```

Methoden

- Methoden sind Funktionen mit einem impliziten Argument *this*
- *this* ist ein Pointer auf das spezielle, instanziierte Objekt

```
class Car {
private:
  string idPlate;

public:
  string getIdPlate() {
    return this->idPlate;
  }

  bool setIdPlate(string &idPlate) {
    if (!idPlate.empty())
      return false;
    this->idPlate = idPlate;
    return true;
  }
};
```

```
int main() {
  Car c;
  // Returns false
  c.setIdPlate("");
  // Returns true
  c.setIdPlate("B ER 2025");
  // Prints B ER 2025
  cout << c.getIdPlate();
}
```

Methoden extern definieren

- Methoden müssen in der Klasse **deklariert** werden
- Der Übersichtlichkeit wegen werden sie aber oft außerhalb **definiert**

```
// Deklaration in car.cpp
class Car {
private:
  string idPlate;
public:
    bool setIdPlate(string &);     // Deklaration
};

// Definition in car.h
bool Car::setIdPlate(string &idPlate) {
  if (!idPlate.empty())
    return false;
  this->idPlate = idPlate;
  return true;
}
```

Method Overloading

Methoden (und Funktionen) können überladen werden

- Zwei Funktionen können denselben Namen, aber andere Signaturen haben
- Der Compiler wählt den mit der passendsten Signatur aus
- Überladung bezieht sich dabei nicht auf den Rückgabewert!!

```
struct Class {
  static void m(int a)  { cout << "1" }
  static bool m(char a) { cout << "2" }
  static void m(double a, double b)
                        { cout << "3" }
};
```

```
void main() {
  // Prints 1
  Class::m(5);
  // Prints 2
  Class::m((char)5);
  // Prints 3
  Class::m(3.1, 2);
}
```

Methoden nur überladen, wenn sie *inhaltlich* (mehr oder weniger) dasselbe machen!

Vererbung

Vererbung besteht aus drei Konzepten

- Strukturelle Vererbung der Methoden und Member
- Subtyping
- Method overloading

```
struct Car {
  void refill() {}
  void drive() {}
};
struct ElectricCar : Car {}
```

```
void main() {
  ElectricCar e;
  e.drive();
}
```

Die abgeleitete Klasse ElectricCar

- Erbt Methoden refill und drive von Car
- Ist ein Subtyp der Klasse Car
- Hat die Möglichkeit die Klasse Car zu erweitern und spezialisieren

Subtyping

- Problem: Können wir ein ElectricCar Auto auf einem Parkplatz für Car Autos parken?
  - Yes we can! ☺

```
struct Parking {
  vector<Car> cars;

  Parking(unsigned places) {
    cars.resize(places);
  }
  void park(unsigned place, Car &car) {
    cars[place] = car;
  }
  Car & leave(unsigned place) {
    return cars[place];
  }
};
```

```
void main() {
  Parking p(2);

  Car c;
  p.park(0, c);

  ElectricCar e;
  p.park(1, e);

  Car & b = p.leave(1);
  b.drive();
}
```

Note: due to space constraints, class Parking is not implemented as it should be!

Quiz

Woher weiß die Memberfunktion Date::month(), dass sie beim ersten Aufruf `d1.m` und beim zweiten `d2.m` zurückgeben muss?

```
Class Date {
  // ...
  int month() {return m;}
  // ...
 private:
  int y, m, d; // Jahr, Monat, Tag
};

void f(Date d1, Date d2) {
  cout << d1.month() << d2.month() << endl;
};
```

Quiz

Woher weiß die Memberfunktion Date::month(), dass sie beim ersten Aufruf `d1.m` und beim zweiten `d2.m` zurückgeben muss?

```
Class Date {
  // ...
  int month() {return m;}
  // ...
 private:
  int y, m, d; // Jahr, Monat, Tag
};

void f(Date d1, Date d2) {
  cout << d1.month() << d2.month() << endl;
};
```

```
int month(Date *this) {
  return *this.m;     }
```

```
month(&d1)
```

```
month(&d2)
```

TEMPLATES

Templates: Motivation

Aufgabe: Schreibe eine Funktion max(a, b), die das Maximum zweier Zahlen a und b ausgibt:

```
int max (int a, int b)
{
    if (a > b) return a;
    return b;
}
```

Problem: Funktion wird für alle Typen benötigt

```
float x = max(1.4, y);
```

Lösung: Templates

Templates sind Schablonen, nach denen der Compiler Code herstellt

```
template <typename T>
T max (T a, T b)
{
    if (a > b) return a;
    return b;
}

...

float x = 1, y = 2;
float z = max (x, y);
```

Template-Argumente

Beim Kompilieren müssen die Template-Argumente bestimmt werden.

Es gibt zwei Möglichkeiten:

1. Explizit:

```
template <typename T1, typename T2>
void f (T1 a, T2 b) { ... }

f<int,float>(0, 3.5);
```

2. Implizit:

```
int   x = 0;
float y = 3.5;

f(x, y);
```

Template-Argumente, gemischt

Mischung von expliziter und impliziter Bestimmung von Template-Argumenten:

```
template <typename T1, typename T2, typename T3>
void f (T3 x, T2 y, T1 z)
{ ... }

f<int>( (char) 0, 3.5F, 7 );
```

1. Argument: `char x`
2. Argument: `float y`
3. Argument: `int z`

Implizit: Problem 1

Folgender Code verursacht ein Problem:

```
template <typename T>
T max (T a, T b)
{
    //...
}

double x = 1.0;
double y = max(x, 0);
```

```
error: template parameter 'T' is ambiguous
```

Grund:

- x (double) und 0 (int) haben unterschiedliche Typen
- In Funktionssignatur müssen beide Argumente denselben Typ haben

Implizit: Problem 2

Bei folgendem Beispiel funktioniert eine implizite Bestimmung des Template-Arguments gar nicht:

```
template <typename T>
T zero ()
{
    return 0;
}

int x = zero();
```

`error: could not deduce template argument`

Grund:

- Implizite Bestimmung benutzt Argumente, nicht den Rückgabewert

Parameter-Deklarationen

Statt eines Types kann auch eine Konstante als Template-Parameter spezifiziert werden:

```
template <int I>
void print ()
{
    std::cout << I;
}

print<5>();
```

Ausgabe: **5**

Template-Klassen

Wie Template-Funktionen lassen sich auch Template-Klassen definieren:

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
};

Pair <int> p;
```

Beispiel:

- vector, map, list (STL Template Klassen)

Bei Template-Klassen müssen die Argumente immer explizit angegeben werden

Defaults für Template-Parameter

Für Template-Klassen können Default-Argumente definiert werden:

```
template <typename T = int>
struct Pair
{
    T element1;
    T element2;
};

Pair < > p;
```

Nachfolgende Parameter müssen dann auch Defaults haben.

Default nur bei Template-Klassen, nicht bei Template-Funktionen

Template Beispiele

Frage: Wozu kann man die Argumente von Template-Klassen verwenden?

- Beispiel: Typen für Member

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
    T get_max();
    void set_both(T elm1, T elm2);
};
```

typename

Hinweis für den Compiler: „hier kommt ein Type!“

```
template <typename T>
struct A
{
    typename T::Alphabet x;    // abhängig, T ist Template-Arg
    typedef int MyInteger;
};

A<char>::MyInteger y;          // char ist kein Template-Arg
```

`typename` steht vor ...

1. Template-Argumenten, die Typen sind (keine Konstanten)
2. Typen, die von Template-Argumenten abhängen

Achtung: In der originalen Fassung von C++ wurde „class“ statt „typename“ für offene Typen verwendet.

"Patterns" von Template Typen

Typen von Klassentemplates können als eine Art "Pattern" angegeben werden:

```
template <typename T1, typename T2>
struct MyClass;

template <typename T>
void function1 (T & obj);

template <typename T1, typename T2>
void function2 (MyClass<T1, T2> & obj);

template <typename T>
void function3 (MyClass<T, int> & obj);
```

Template Spezialisierung

Templates können für bestimmte Template-Argumente spezialisiert werden.

```
template <typename T>
struct MyClass
{ int foo; };

template < >
struct MyClass <int>
{ int x; };

MyClass<int> obj;
obj.x = 17;

obj.foo = 18;          // ERROR! Keine Vererbung
```

Grund:

- `foo` ist in `MyClass<int>` nicht bekannt.

Quiz: Was wird ausgegeben?

```
template <typename T>
struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < >
struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T>
void call_f(T & t)
{
    C<T>::f();
}

typedef char * c_ptr;
c_ptr x;
call_f(x);
```

Quiz: Was wird ausgegeben?

```
template <typename T>
struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < >
struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T>
void call_f(T & t)
{
    C<T>::f();
}

typedef char * c_ptr;
c_ptr x;
call_f(x);
```

Quiz: Und jetzt?

```
template <typename T>
struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < >
struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T>
void call_f(T & t)
{
    C<T>::f();
}

call_f("hallo");
```

Quiz: Und jetzt?

```
template <typename T>
struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < >
struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T>
void call_f(T & t)
{
    C<T>::f();
}

call_f("hallo");
```

```
const char*
```

Metafunctions

Metafunctions sind eine Anwendung für Template Spezialisierung

- Beispiel: Eine Funktion auf Strings

```
template <typename TString>
char first_character(TString & str)
{
    return str[0];
}
```

- Problem:
  - Was machen wir, wenn der Alphabettyp von `TString` nicht `char` ist?

Metafunctions

Idee: Man benötigt eine Art "Funktion", die Typen zurückliefert

- Pseudo-Code:

```
template <typename TString>
Value(TString) first_character(TString & string)
{
    return string[0];
}
```

Der angegebene Code ist natürlich kein gültiges C++

Metafunctions

Lösung: Verwende Klassen-Templates

```
template <typename T>
struct Value
{
    typedef char Type;
};


template <typename TString>
typename Value<TString>::Type
first_character(TString & str)
{
    return str[0];
}
```

Metafunctions

Spezialisiere Template für verschiedene Typen:

```
template <
    typename TChar,
    typename TTraits,
    typename TAlloc >
struct Value < basic_string<TChar, TTraits, TAlloc> >
{
    typedef TChar Type;
};


template <>
struct Value <char *>
{
    typedef char Type;
};
```

STL Containers und Iteratoren enthalten value_type Member, z.B vector<char>::value_type ist char

TEMPLATE SUBCLASSING

Das Delegation-Problem

Problem: Es sei z.B. folgendes Programm gegeben:

```
struct Car {
  void refill() { cout << "Gas Please"; }

  void drive() {
    if (tank.empty())
      this->refill();
    ...
  }
};

struct ElectricCar: Car {
  // Override function from Car
  void refill() { cout << "Energy Please"; }
};
```

Das Delegation-Problem

Problem: Was passiert hier?

```
ElectricCar e;
e.drive();
```

```
struct Car {
  void refill() { cout << "Gas Please"; }

  void drive() {
    if (tank.empty())
      this->refill();
    ...
  }
};

struct ElectricCar: Car {
  // Override function from Car
  void refill() { cout << "Energy Please"; }
};
```

?

Das Delegation-Problem

Ausgabe: Gas Pleases

```
ElectricCar e;
e.drive();
```

```
struct Car {
  void refill() { cout << "Gas Please"; }

  void drive() {
    if (tank.empty())
      this->refill();
    ...
  }
};

struct ElectricCar: Car {
  // Override function from Car
  void refill() { cout << "Energy Please"; }
```

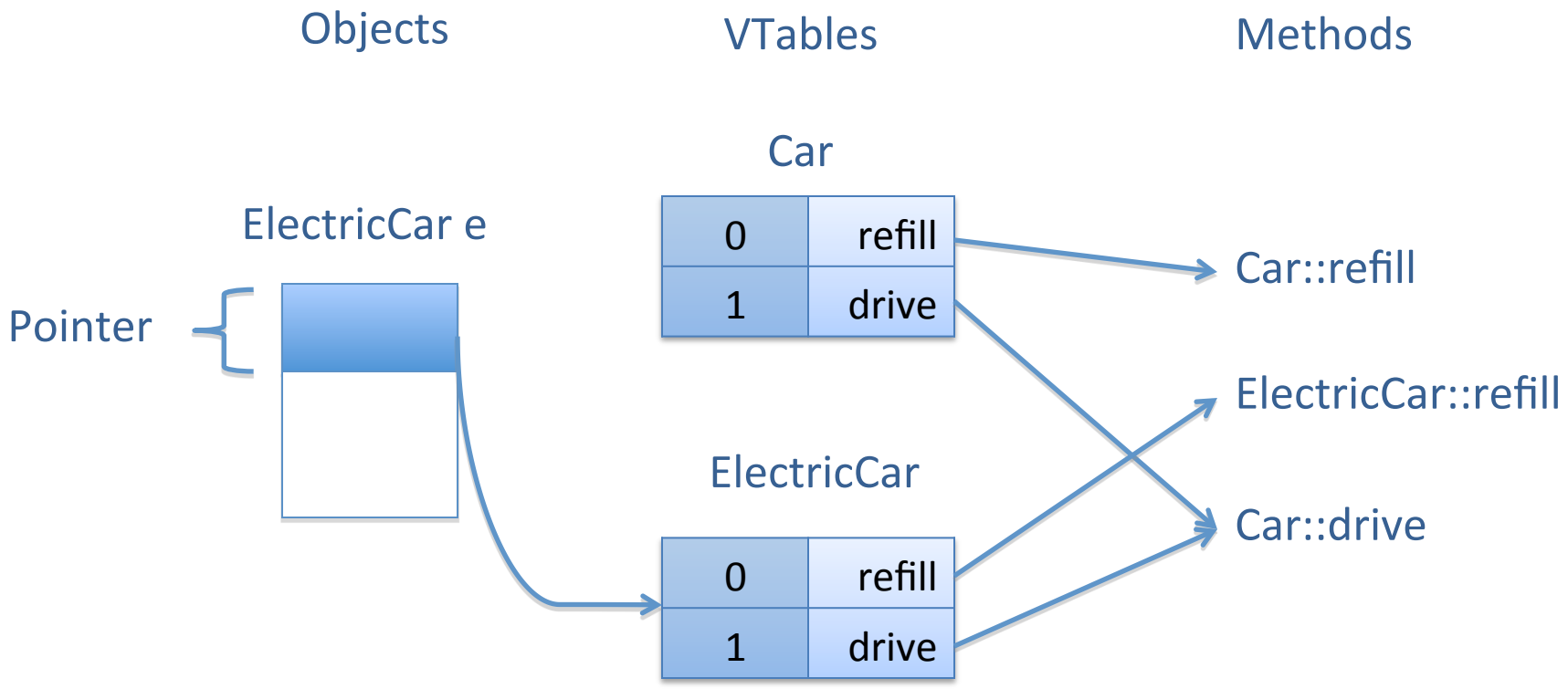
Lösung:

virtual functions

```
struct Car {
  virtual void refill()
    { ... }
};
```

Wie funktioniert `virtual`?

Prinzip: Objekt hält Zeiger auf eine Tabelle mit Zeigern auf die richtigen Funktionen



Eigenschaften virtueller Funktionen

dynamic binding

- Die tatsächlich aufgerufene Funktion wird erst zur Laufzeit bestimmt

Nachteile des dynamic binding

- zusätzlicher Speicherbedarf pro Objekt
- langsamer, indirekter Sprung
- kein Inlining

Beobachtung

- oft braucht man gar kein *dynamic binding*
- Typen stehen schon zur Compile-Zeit fest

Template Subclassing

Eine alternative Lösung des Delegation-Problems mit static binding

Bisher (objektorientiert):

```
struct Car { /*Car*/ };
struct ElectricCar : Car { /*ElectricCar*/ };
```

Template Subclassing

Schritt 1: “Template Spezialisierung statt Ableitung”

```
template <typename T>
struct Car { /*Car*/ };

struct ElectricCar;

template <>
struct Car <ElectricCar> { /*ElectricCar*/ };
```

ElectricCar ist ein Tag, das nur deklariert werden muss

Template Subclassing (II)

Schritt 2: “Globale Funktionen statt Member Funktionen”

```
template <typename T>
void refill (Car<T> & obj)
{
    std::cout << “Gas Please"; ...
}

void refill (Car<ElectricCar> & obj)
{
    std::cout << ”Energy Please"; ...
}

template <typename T>
void drive (Car<T> & obj)
{
    refill(obj); ...
}
```

Template Subclassing (III)

Template Subclassing löst das Delegation Problem:

```
Car<ElectricCar> car;
drive(car);

// Output: “Energy Please"
```

Anwendungsbeispiel

Gesucht: Funktion, die das größte Element eines Feldes bestimmt

- Gegeben ist eine Feld von Zeigern auf eigentliche Elemente
- Vergleichsfunktion soll frei wählbar sein

Lösung 1: Objektorientiert

- Basisklasse `Comparable` mit virtueller Vergleichsfunktion `less`
- Definiere Elementtyp als Kindklasse und überlade `less`

Lösung 2: Templates

- Definiere globale `less`-Funktion und templatisiere `maxArg`
- Spezialisiere `less` für Elementtyp

Objektorientiert

```
struct Comparable
{
    virtual bool less(Comparable &right) {
        return *this < right;
    }
};

struct Element: public Comparable
{
    int a;
    bool less(Comparable &right) {
        return a < static_cast<Element&>(right).a;
    }
};

Comparable* maxArg(Comparable* arr[], int size)
{
    Comparable *max = NULL;
    for (int i = 0; i < size; ++i)
        if (max == NULL || *max.less(*arr[i]))
            max = arr[i];
    return max;
}
```

Templates

```
template <typename T>
inline bool less(T &left, T &right)
{
    return left < right;        // allgemein
}

template < >
inline bool less(Element &left, Element &right)
{
    return left.a < right.a;    // speziell
}

template <typename T>
T * maxArg(T *arr[], int size)
{
    T *max = NULL;
    for (int i = 0; i < size; ++i)
        if (max == NULL || less(*max, *arr[i]))
            max = arr[i];
    return max;
}
```

Vergleich

Objektorientiert mit virtueller Funktion

- Es existiert genau eine `maxArg`-Funktion
- Indirektion beim Lesen der Funktionsadresse von `less`
- Sprung und Rücksprung, Stackframe auf- und abbauen

Templates und inline

- Für jeden benutzten Elementtyp wird eine eigene `maxArg`-Funktion erzeugt
- `less`-Funktion wird direkt in `maxArg` eingebaut (inline)

Laufzeitmessung mit 1 Mrd. Elementen

- 3350 ms mit OOP
- 2150 ms mit Templates

BEMERKUNGEN ZUR P-AUFGABE

Hinweise

- Punkte erscheinen in Datei punkte.txt im Repository
- Musterlösung zum ShiftAnd unter https://svn.imp.fu-berlin.de/aldabi/WS14/material/
- Beachten Sie die Abgabehinweise!
  - Datei sollte aufgabe1.cpp heissen, nicht anders
  - Die nächste soll also aufgabe2.cpp heissen (!)
  - Aufruf muss über die Kommandozeile erfolgen, nicht über `cin`

Tipps zu Aufgabe 2

- Enums um Aufzählungstypen zu definieren.

```
enum TracebackDirection
{
    TRACEBACK_DIRECTION_DIAGONAL,
    TRACEBACK_DIRECTION_HORIZONTAL,
    TRACEBACK_DIRECTION_VERTICAL,
    TRACEBACK_DIRECTION_NONE
};


std::vector<TracebackDirection> traceback;
```

Tipps zu Aufgabe 2

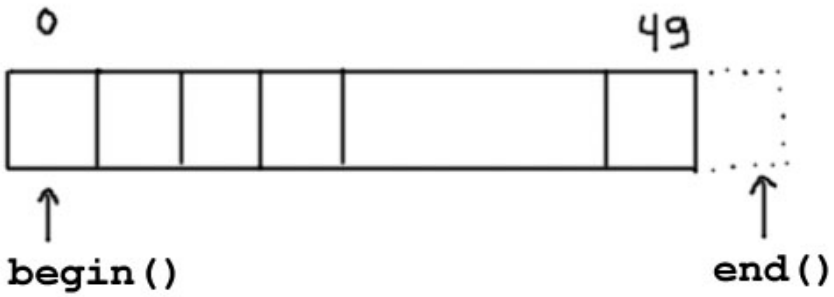
- Speichern einer Matrix im eindimensionalen Raum

```
// Matriximplementierung mit 2 Vektoren.
std::vector<int> rows(20, 0);
std::vector<std::vector<int> > matrix(15, rows)

int val = matrix[3][4];

// Matriximplementierung mit einem Vektor.
std::vector<int> matrix(20 * 15, 0);

int val = matrix[3 * 15 + 4];
```

Iteratoren

Iteratoren = verallgemeinerter Pointer

```
vector<char> vec(50);
vector<char>::iterator anfang = vec.begin();
vector<char>::iterator ende = vec.end();

cout << ende - anfang;   //Ausgabe: 50
```



Warum Iteratoren benutzen?

- summiere alle Elemente von Feld *f*

```
int sum = 0;
for (int i = 0; i < n; ++i)
   sum += f[i];
```

- wird vom Compiler genauso übersetzt wie:

```
int sum = 0;
for (int i = 0; i < n; ++i)
   sum += *(f + i);
```

- effizienter ist (ohne Optimierung):

```
int sum = 0;
int *f_end = f + n;
for (int *p = f; p < f_end; ++p)
   sum += *p;
```

Warum Iteratoren benutzen? (II)

- Warum sorgen, wenn der Compiler optimiert?
  - die Optimierung funktioniert nur bis zu einer gewissen Komplexität
- Was passiert, wenn *f* eine verkettete Liste ist?
  - `f[i]` hat dann eine Laufzeit von O(i)
  - die ganze Schleife hat dann also $O(n^2)$ Laufzeit

```
std::list<int> f(n);
for (int i = 0; i < n; ++i)
    sum += f[i];
```

- Iteratoren sind optimal für sequentielle Zugriffe
  - Schleife mit Iteratoren hat O(n) Laufzeit

```
std::list<int> f(n);
std::list<int>::iterator i = f.begin();
std::list<int>::iterator i_end = f.end();
for (; i != i_end; ++i)
    sum += *i;
```

Iteratoren (II)

Mit Iteratoren kann man durch Container traversieren.

```
template<typename TIn, typename TOut>
inline TOut copy(TIn first, TIn last, TOut dest)
{
 while ( first != last )
 {
    *dest = *first;
    ++dest; ++first;
 }
 return (dest);
}
```

Quiz: Was läuft hier schief?

Frage: Wieso produziert dieses Programm einen Laufzeitfehler?

```
vector<int> vec1;
vector<int> vec2;

for (int i = 0; i < 10; ++i) vec1.push_back(i);

copy(vec1.begin(), vec1.end(), vec2.begin());
```