C++

LUG Frankfurt
Programmierworkshop
4.12.2010

Thomas Baumgart

Background

- 1984 – erste Kontakte zu Unix (IS/1) auf Z8000
- 1987 – THEOS O/S auf IBM/PS2 portiert
- 1988 – THEOS Basic Interpreter und Compiler von 16 auf 32 bit portiert
- 1998 – versucht GNU C/C++ zu portieren um THEOS binaries zu erstellen. Cross-Compiler zum Laufen gebracht
- Seit 2001 aktiv im KMyMoney Projekt aktiv
- Seit dem auch Qt / C++ Kenntnisse erlangt

First things first

C makes it easy for you to shoot yourself in the foot.

C++ makes that harder, but when you do, it blows away your whole leg.

-- Bjarne Stroustrup

Wir sortieren uns

Objekt

Public

Polymorphismus

Struktur

Methode

Instanz

Membervariable

Klasse

Protected

Funktion

Private

Vererbung

Attribut

Friend

Wir sortieren uns

Klasse
Struktur — Bauplan / Beschreibung einer Sache

Objekt
Instanz — Vorkommen der Sache

Membervariable
Attribut — Detail der Sache

Funktion
Methode — Veränderung oder Aktivität der Sache

Vererbung
Polymorphismus
Dazu kommen wir gleich!

Private
Protected
Public
Friend
Sichtbarkeit von Attributen und Methoden der Sache

From struct to class I

```c
/* Standard C */

typedef struct Car {
  int farbe;
  int achsen;
  int raeder;
} Car;


void f(void)
{
  Car MyCar;
  MyCar.farbe = 0;
  MyCar.achsen = 2;
  MyCar.raeder = 4;
}
```

```c
// C++

class Car {
public:
  int farbe;
  int achsen;
  int raeder;
};

void f(void)
{
  Car MyCar;
  MyCar.farbe = 0;
  MyCar.achsen = 2;
  MyCar.raeder = 4;
}
```

From struct to class II

```c
/* Standard C */

typedef struct Car {
  int farbe;
  int achsen;
  int raeder;
} Car;



void constructCar(Car *this)
{
  this->farbe = 0;
  this->achsen = 2;
  this->raeder = 4;
}

void main(void)
{
  Car MyCar;
  // MyCar.farbe == ???
  constructCar(&MyCar);
  // MyCar.farbe == 0
}
```

```c
// C++

class Car {
public:
  Car();
  int farbe;
  int achsen;
  int raeder;
};

Car::Car()
{
  farbe = 0;
  achsen = 2;
  raeder = 4;
}

void main(void)
{
  Car MyCar;
  // MyCar.farbe == 0
}
```

Zugriffschutz

```c
/* Standard C */

typedef struct Car {
  int farbe;
  int achsen;
  int raeder;
} Car;



void constructCar(Car *this)
{
  this->farbe = 0;
  this->achsen = 2;
  this->raeder = 4;
}

void main(void)
{
  Car MyCar;
  constructCar(&MyCar);
  MyCar.farbe = 5;
}
```

```c
// C++

class Car {
public:
  Car();
private:
  int farbe;
  int achsen;
  int raeder;
};

Car::Car()
{
  farbe = 0;
  achsen = 2;
  raeder = 4;
}

void main(void)
{
  Car MyCar;
  MyCar.farbe = 5;
}
```

Compile Error

Zugriffschutz: Lösung

```
// car.h

class Car {
public:
  Car();
  setFarbe(int);
  int farbe(void);
private:
  int farbe;
  int achsen;
  int raeder;
};
```

```
// car.cpp

#include “car.h”

Car::Car()
{
  farbe = 0;
  achsen = 2;
  raeder = 4;
}

Car::setFarbe(int f)
{
  farbe = f;
}

int Car::farbe(void)
{
  return farbe;
}
```

```
// main.cpp

#include “car.h”


void main(void)
{
  Car MyCar;
  // MyCar.farbe() == 0

  MyCar.setFarbe(5);
  // MyCar.farbe() == 5

  MyCar.setFarbe(7);
  // MyCar.farbe() == 7

  MyCar.setFarbe(-3);
  // MyCar.farbe() == -3
}
```

Vorteil: z.B. zentrale Bereichsprüfung

```
// car.h

class Car {
public:
  Car();
  setFarbe(int);
  int farbe(void);
private:
  int farbe;
  int achsen;
  int raeder;
};
```

```
// car.cpp

#include “car.h”

Car::Car()
{
  farbe = 0;
  achsen = 2;
  raeder = 4;
}

Car::setFarbe(int f)
{
  if(farbe >= 0 || farbe < 6)
    farbe = f;
}

int Car::farbe(void)
{
  return farbe;
}
```

```
// main.cpp

#include “car.h”


void main(void)
{
  Car MyCar;
  // MyCar.farbe() == 0

  MyCar.setFarbe(5);
  // MyCar.farbe() == 5

  MyCar.setFarbe(7);
  // MyCar.farbe() == 5

  MyCar.setFarbe(-3);
  // MyCar.farbe() == 5
}
```

Vererbung / Inheritance

```
// car.h

class Car {
public:
  Car();
  setFarbe(int);
  int farbe(void);
  int achsen(void) { return achsen; }
protected:
  int farbe;
  int achsen;
  int raeder;
};

// car.cpp

#include “car.h”

Car::Car()
{
  farbe = 0;
  achsen = 2;
  raeder = 4;
}

Car::setFarbe(int f)
{
  if(farbe >= 0 || farbe < 6)
    farbe = f;
}

int Car::farbe(void)
{
  return farbe;
}
```

```
// truck.h
#include “car.h”

class Truck : public Car {
public:
  Truck();
};



// truck.cpp

#include “truck.h”

Truck::Truck()
{
  achsen = 3;
}
```

```
// main.cpp

#include “car.h”
#include “truck.h”


void main(void)
{
  Car MyCar;
  Truck MyTruck;
  // MyCar.achsen() == 2
  // MyTruck.achsen() == 3

  MyTruck.setFarbe(7);
  // MyTruck.farbe() == 0
}
```

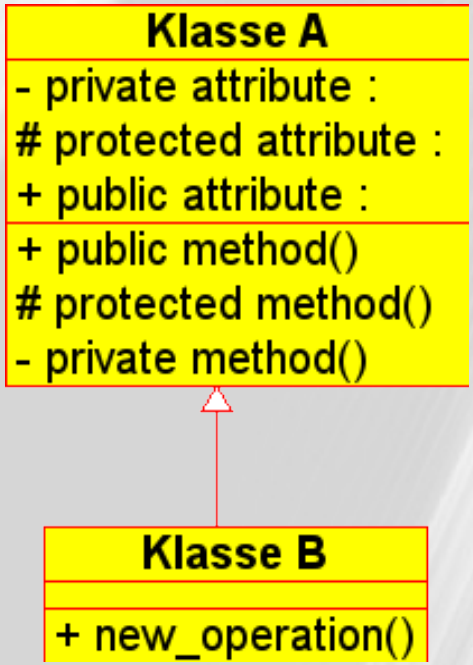
Include - Stopper

```
// car.h

#ifndef CAR_H
#define CAR_H

class Car {
public:
  Car();
  setFarbe(int);
  int farbe(void);
  int achsen(void) { return achsen; }
private:
  int farbe;
  int achsen;
  int raeder;
};

#endif // CAR_H
```

```
// truck.h

#ifndef TRUCK_H
#define TRUCK_H

#include “car.h”

Class Truck : public Car {
public:
  Truck();
};

#endif // TRUCK_H
```

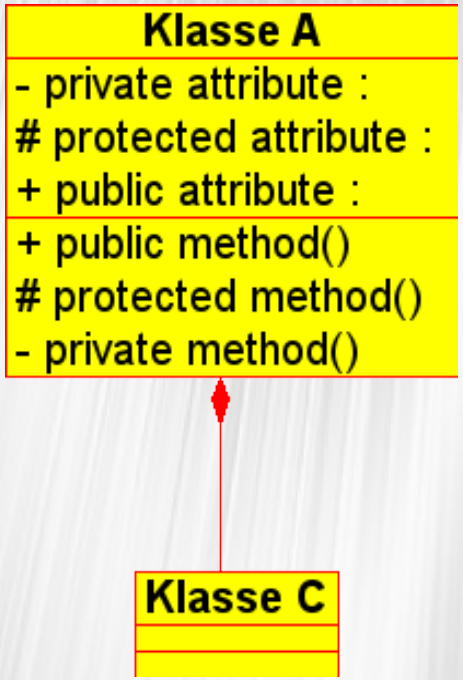
```
// main.cpp

#include “car.h”
#include “truck.h”


void main(void)
{
  Car MyCar;
  Truck MyTruck;
  // MyCar.achsen() == 2
  // MyTruck.achsen() == 3

  MyTruck.setFarbe(7);
  // MyTruck.farbe() == 0
}
```

UML



Car

Truck



Car

Motor

“Klasse B” ist von “Klasse A” abgeleitet

oder

“Klasse B” implementiert “Klasse A”

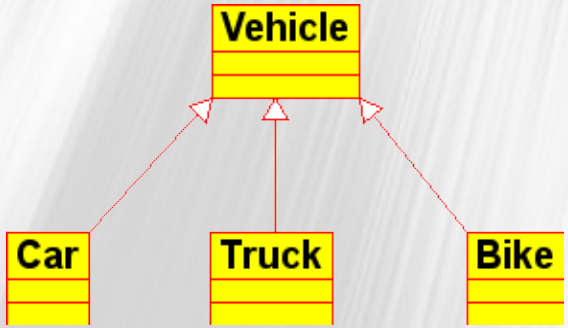
“Klasse C” ist in “Klasse A” enthalten

oder

“Klasse C” ist Member von “Klasse A”

Fahrzeuge: Design Issues

Bad ??



Good ??

```
class Vehicle {
public:
  int farbe(void);
  ...
protected:
  int farbe;
  int achsen;
  int raeder;
};

class Car : public Vehicle {
};

class Truck : public Vehicle {
};

class Bike : public Vehicle {
};
```

Konstruktor mit Parameter

```
class A
{
public:
  A(int x);
private:
  int m_a;
};

A::A(int x)
{
  m_a = x;
}
```

```
class B : public A
{
public:
  B(int x);
};

B::B(int x)
{
  m_a = x;
}
```

```
class B : public A
{
public:
  B(int x);
};

B::B(int x) : A(x)
{
}
```

Der Destruktor

```
class A
{
public:
  A();     // constructor
  ~A();    // destructor
};

A::A()
{
  std::cout << “Ich bin der CTOR” << std::endl;
}



A::~A()
{
  std::cout << “Ich bin der DTOR” << std::endl;
}

int main (void)
{
  A myObject;
  std::cout << “Ich bin main” << std::endl;
}
```

Ausgabe:

```
Ich bin der CTOR
Ich bin main
Ich bin der DTOR
```

```
A* f(void)
{
  retrun new A;
}

int main (void)
{
  A myObject;
  A *p;
  std::cout << “Ich bin main 1” << std::endl;
  P = f();
  std::cout << “Ich bin main 2” << std::endl;
  delete p;
}
```

Ausgabe:

```
Ich bin der CTOR   // myObject
Ich bin main 1
Ich bin der CTOR   // new
Ich bin main 2
Ich bin der DTOR   // delete
Ich bin der DTOR   // myObject
```

Lifetime of an object

- Stack
  - Definition bis Ende Basic Block
- Heap
  - Von new bis delete
- Global
  - Von 'vor main()' bis 'nach main()'

Defaultvalue

```
class A
{
public:
  A(int x);
private:
  int m_a;
};

A::A(int x)
{
  m_a = x;
}


void f(void)
{
  A MyA(5);
  A MyOtherA;
}
```

```
class A
{
public:
  A(int x = 0);
private:
  int m_a;
};

A::A(int x)
{
  m_a = x;
}


void f(void)
{
  A MyA(5);
  A MyOtherA;
}
```

Defaultvalue

```
class A
{
public:
  A(int x = 0);
};
```

```
class A
{
public:
  A(int x, int y = 0);
};
```

```
class A
{
public:
  A(int x = 0, int y);
};
```

Methodenüberlagerung

```
class Car
{
public:
 void bremsen(void);
 void bremsen(int force);
 void bremsen(double force);
};

void f(void)
{
  Car myCar;
  myCar.bremsen();
  myCar.bremsen(4);
  myCar.bremsen(4.5);
}
```

```
class Car
{
public:
 void bremsen(void);
 void bremsen(int force = 0);
 void bremsen(double force = 0);
};

void f(void)
{
  Car myCar;
  myCar.bremsen();
  myCar.bremsen(4);
  myCar.bremsen(4.5);
}
```

```
class Car
{
public:
 void bremsen(void);
 void bremsen(double force);
};

void f(void)
{
  Car myCar;
  myCar.bremsen();
  myCar.bremsen(4);
  myCar.bremsen(4.5);
}
```

Operatorenüberladung

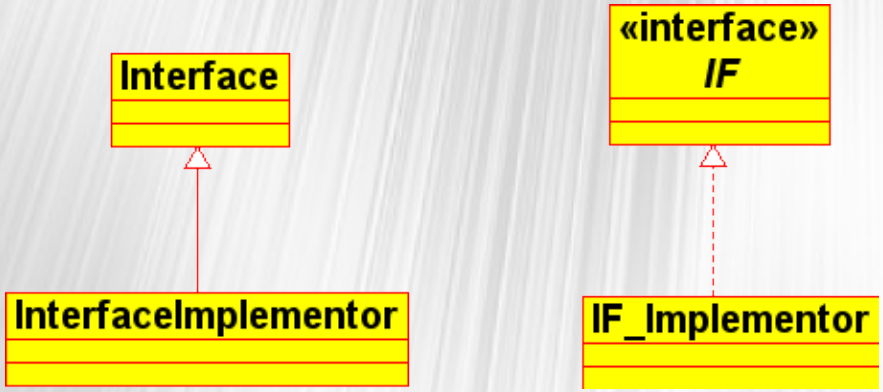
```
class Anhaenger;
class Truck
{
public:
 void anhaengen(const Anhaenger& haenger);
};

void Truck::anhaengen(const Anhaenger& haenger)
{
  // do something
}

void f(void)
{
  Truck myTruck;
  Anhaenger myHaenger;

  myTruck.anhaengen(myHaenger);
}
```

```
class Anhaenger;
class Truck
{
public:
 Truck& operator+=(const Anhaenger& haenger);
};

Truck& Truck::operator += (const Anhaenger& haenger)
{
  // do something
  return *this;
}

void f(void)
{
  Truck myTruck;
  Anhaenger myHaenger;

  myTruck += myHaenger;
}
```

Interface definition

```
// interface.h

class Interface {
public:
  virtual void methodA(void) = 0;
  virtual void methodB(void) = 0;
};
```

```
// interface-usage.h

#include “interface.h”

class Usage : public Interface {
public:
  Usage();
  void methodA(void);
  void methodB(void);
};
```



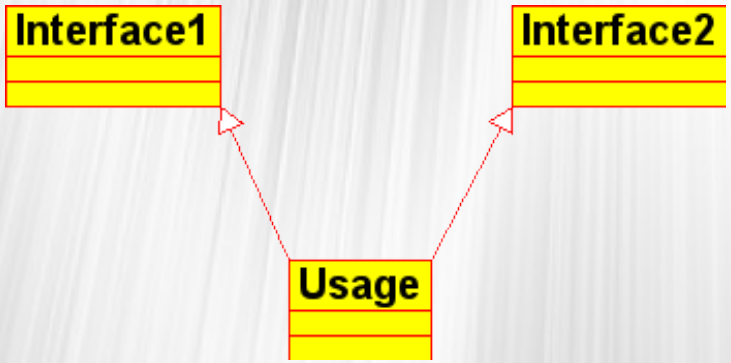
Falls eine Methode (hier methodA) in dem Objekt nicht implementiert ist, dann:

```
test.cpp:12: error: cannot declare variable ‘Test’ to be of abstract type ‘Usage’
interface-usage.h:6: note:   because the following virtual functions are pure within ‘Usage’:
interface.h:4: note:   virtual void Interface::methodA()
```

Multiple Inheritance I

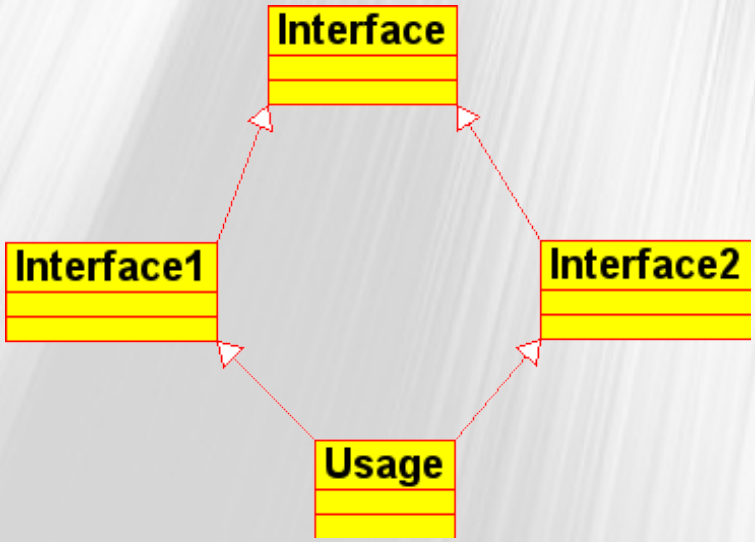
```
// interface.h

class Interface1 {
public:
  virtual void methodA(void) = 0;
  virtual void methodB(void) = 0;
};

class Interface2 {
public:
  virtual void methodC(void) = 0;
  virtual void methodD(void) = 0;
};
```

```
// interface-usage.h

#include “interface.h”

class Usage : public Interface1, Interface2 {
public:
  Usage();
  void methodA(void);
  void methodB(void);
  void methodC(void);
  void methodD(void);
};
```



Multiple inheritance II

Aber Vorsicht!!!



Polymorphismus

```
class Vehicle {
public:
  virtual int raeder(void) = 0;
  ...
};

class Car : public Vehicle {
  int raeder() { return 4; }
};

class Truck : public Vehicle {
  int raeder() { return 6; }
};

class Bike : public Vehicle {
  int raeder() { return 2; }
};
```

```
void f(void)
{
  Truck myTruck;
  Car myCar;
  Bike myBike;

  MyTruck.raeder();  // 6
  MyCar.raeder();    // 4
  MyBike.raeder();   // 2
}
```

```
void f(void)
{
  Vehile *arr[3];
  arr[0] = new Truck;
  arr[1] = new Car;
  arr[2] = new Bike;

  for(int i = 0; i < 3; ++i)
    arr[i].raeder();

  // i == 0 → 6
  // i == 1 → 4
  // i == 2 → 2
}
```

Übung

Wir bauen libalkimia

Wir bauen was??

```
svn co svn://anonsvn.kde.org/home/kde/trunk/playground/office/alkimia/libalkimia

cd libalkimia
mkdir build
cd build
cmake ..
make
```

Exceptions

Wie der Name sagt: wird zur Behandlung von Ausnahmen verwendet.

```
// exception.cpp

#include <cstdio>
#include <iostream>

using namespace std;

void f(void)
{
  throw 1;
}

int main(void)
{
  try {
    f();
    cout << "None" << endl;
  } catch(int i) {
    cout << "Integer" << endl;
  } catch(double d) {
    cout << "Double" << endl;
  } catch(...) {
    cout << "Anything else" << endl;
  }
}
```

```
// exception.cpp
// Alternative flows

  // throw 1;
   → None

  throw 1.0;
   → Double

  throw “1.0”;
   → Anything else
```

Singleton

Für manche Klassen ist es wichtig, dass nur eine einzelne Instanz existiert. Obwohl es mehrere Drucker in einem System gibt, gibt es nur einen Druckerspooler. Es gibt nur ein Dateisystem und einen Window-Manager. Ein Digitaler Filter hat nur einen A/D Umsetzer und ein Buchhaltungssystem wird einer Firma gewidmet sein.

Wie stellen wir sicher, dass eine Klasse nur eine Instanz erzeugen kann und dass diese Instanz einfach anzusprechen ist? Eine globale Variable macht ein Objekt einfach ansprechbar aber es wird nicht verhindert, dass mehrere Instanzen erzeugt werden können.

Eine bessere Lösung besteht darin, die Klasse selbst dafür sorgen zu lassen, dass nur eine Instanz existiert. Die Klasse kann sicherstellen, dass keine weitere Instanz erzeugt werden kann (durch Abfangen von Anfragen zum Erzeugen neuer Objekte) und sie kann einen einfachen Zugriffsmechanismus zur Verfügung stellen. Dies ist das Singleton-Pattern.

Singleton (implementation)

```
// singleton.h

class Singleton
{
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};
```

```
// singleton.cpp

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance ()
{
  if (_instance == 0) {
    _instance = new Singleton;
  }
  return _instance;
}
```

```
// main.cpp

int main(void)
{
  Singleton *s1 = Singleton::Instance();
  Singleton *s2 = Singleton::Instance();
  // s1 == s2
}
```

Diese Version ist **nicht** thread-safe!!

Singleton (thread-safe)

```
// singleton.h

class Singleton
{
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};
```

```
// singleton.cpp

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance ()
{
  Mutex.lock()
  if (_instance == 0) {
    _instance = new Singleton;
  }
  Mutex.unlock();
  return _instance;
}
```

Diese Version ist thread-safe!!

Singleton (optimized)

```
// singleton.h

class Singleton
{
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};
```

```
// singleton.cpp

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance ()
{
  if (_instance == 0) {
    Mutex.lock()
    if (_instance == 0) {
      _instance = new Singleton;
    }
    Mutex.unlock();
  }
  return _instance;
}
```

Diese Version ist thread-safe und schnell !!

Tools für diesen Vortrag

- **Open Office** (Präsentation)
- **Umbrello** (UML Diagramme)
- **Gimp** (entfernen des Backgrounds in UML Diagrammen)
- **Kate** (schreiben von Source-Code)

Fun stuff

```
auto accident;
register voters;
static electricity;
struct by_lightning;
void *where_prohibited;
char broiled;
short circuit;
short changed;
long johns;
unsigned long letter;
double entendre;
double trouble;
union organizer;
float valve;
short pants;
union station;
void check;
unsigned check;
struct dumb by[sizeof member];
```

Links

- http://bruce-eckel.developpez.com/livres/cpp/ticpp/v1/
- http://bruce-eckel.developpez.com/livres/cpp/ticpp/v2/
-

Finale

Vielen Dank für Eure Aufmerksamkeit und Geduld