Forschungszentrum Telekommunikation Wien
[Telekommunications Research Center Vienna]

CAMPARI-Trainings-Workshop C++ und Objektorientierung

Martina Umlauft
umlauft@ftw.at

Zielgruppe und Vorkenntnisse

- Praktikanten CAMPARI und Interessierte
- C-Kenntnisse
- Kenntnis einer OO-Sprache (JAVA)
- grundlegende Kenntnisse OO/UML

Overview

- Zielgruppe & Vorkenntnisse
- Klassenkonzept
- C++ Syntax
- Wiederholung UML
- STL - Standard Template Library
- Design Patterns

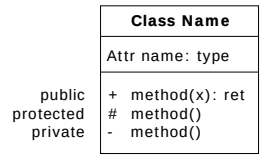
Klassenkonzept

Klasse

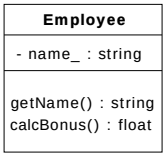
- **Typ** zur Konstruktion von Objekten
- Class Members - `static`

Objekt

- Members - eine Kopie per Objekt
- Konstruiert mit new



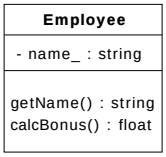
Klassen: Syntax / 1



Unterschied zu Java:

- **Definition** im .h File
- **Implementierung** im .cc oder .cpp File
- .cc/.cpp File: **#include**s das .h File
- mehrere Klassen pro File möglich

Klassen: Syntax / 2



Deklaration - .h File

```
class Employee {
  private:
    string name_;

  public:
    string getName();
    float calcBonus();
};
```

Klassen: Syntax / 3

Implementierung - .cc File

Employee
- name_ : string
getName() : string calcBonus() : float

```
string Employee::getName() {
  return name_;
}

float Employee::calcBonus() {
  ...
}
```


Klassen: Syntax / 4

Inline-Deklaration - .h File

Employee
- name_ : string
getName() : string calcBonus() : float

```
class Employee {
  private:
    string name_;

  public:
    string getName() {
      return name_;
    }
    float calcBonus();
};
```


Klassen: Syntax / 5

Verwendung

Employee
- name_ : string
setName() getName() : string calcBonus() : float

```
int main(int argc, char *argv[]){
  Employee martina;

  martina.setName("Umlauft");
  ...
}
```

Unterschied zu JAVA:

- **kein** new nötig!
- Java: NUR Referenzen auf Objekte
- C++: AUCH normale Variablen mögl.
- Pointer auf Objekt => new nötig

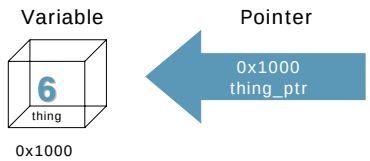
Klassen: Syntax / 6

Verwendung - Pointer auf Objekt

Employee
- name_ : string
setName() getName() : string calcBonus() : float

```
int main(int argc, char *argv[]){
  Employee *e_ptr;

  e_ptr = new Employee;

 (*e_ptr).setName("Umlauft");
  e_ptr->setName("Umlauft");

  delete e_ptr;
}
```


Wiederholung: Pointer




```
int thing;
int * thing_ptr;

thing = 6;
thing_ptr = &thing;
cout << "Inhalt: " << *thing_ptr;
```


Klassen: Syntax / 7

Constructor

Employee
- name_ : string
getName() : string calcBonus() : float

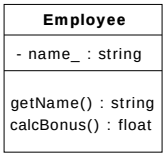
.h

```
class Employee {
public:
  Employee(string name);
};
```

.cc

```
Employee::Employee(const
  string name) {
    name_ = name;
}
```

Klassen: Syntax / 8

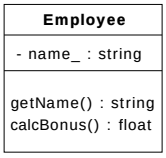


Destructor

.h
```
class Employee {
public:
  ~Employee();
};
```

.cc
```
Employee::~Employee() {
    ...
}
```


Klassen: Syntax / 9



Copy-Constructor

.cc
```
Employee::Employee(const
  Employee& e) {
    name_ = e.name;
    ...
}
```

```
void funct(Employee e){
  ...
}

Employee a("Umlauft");
funct(a);
```

- Wird bei **call-by-value** Parameter-Übergabe autom. aufgerufen
- Wichtig bei komplexen Objekten, die Members mit **new** anlegen!

Klassen: Syntax / 10

Constant Members

.h
```
class foo {
  public:
    const int size;
    foo():size(500) {
      ...
    }
};
```

Normal Constants

```
const int size = 500;
```


Klassen: Syntax / 11

Static Member Variables

.h
```
class foo {
  private:
    static int general_counter;
    int per_object_var;
};
```

.cc
```
int foo::general_counter = 0;
```

- Existieren nur 1x pro **Klasse**, NICHT per Objekt
- Alle Objekte dieser Klasse greifen auf dieselbe Variable zu

Klassen: Syntax / 12

Static Constant Member Variables

.h
```
class foo {
  private:
    static const int size = 500;
};
```

- Initialisierung wie bei normalen Konstanten
- Ebenfalls nur 1x pro Klasse, sowie
- alle Objekte dieser Klasse greifen auf dieselbe Variable zu

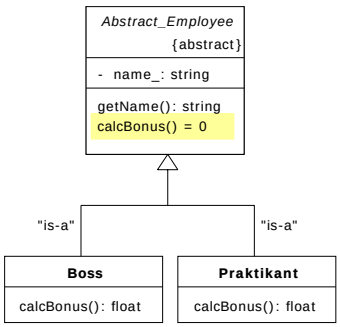
Klassen: Syntax / 13

Static Member Functions

.h
```
class foo {
  private:
    static int general_counter;
  public:
    static int getCount() {
      return general_counter;
    }
};
```

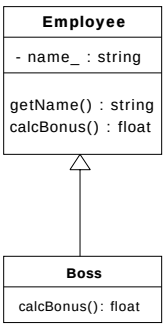
- Can only access `static` vars and methods

Wiederholung: Vererbung



- Gemeinsame Eigenschaften in Parent class
- Spezialisierung in Kindern
- **"is-a"** Relation
- NICHT "has-a"!

Klassen: Syntax / 14

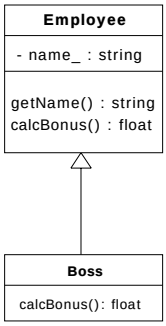


Vererbung

```
class Employee {
  public:
    virtual float calcBonus();
    ...
};

class Boss : public Employee {
  public:
    virtual float calcBonus();
    ...
};
```

- Tip: Destruktoren **immer** `virtual` machen!

Klassen: Syntax / 15

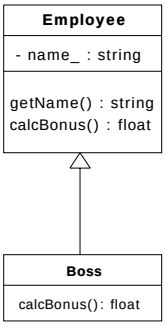


Vererbung - Verwendung

```
Employee * e;
float bonus;

e = new Employee();
e = new Boss();
e = new Praktikant();

bonus = e->calcBonus();
```

ruft dank `virtual` immer die richtige Methode auf

Klassen: Syntax / 16

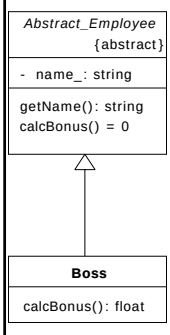


Vgl. "super" in JAVA:

```
virtual float Boss::calcBonus();
  return (5 * Employee::calcBonus());
}
```

"super"

Klassen: Syntax / 17



Abstrakte Klasse

.h

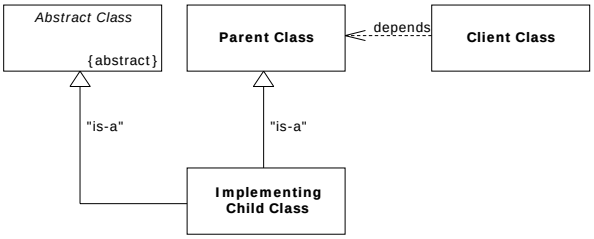
```
class Abstract_Employee {
  public:
    virtual float calcBonus() = 0;
    ...
};

class Boss : pub  "pure virtual"  oyee {
  public:
    virtual float calcBonus();
    ...
};
```

.cc

```
virtual float Boss::calcBonus() {
  ...
}
```

Implementierung notwendig!

Wiederholung UML: Vererbung



"is-a": Vererbung
"has-a": Member variable

C++: Referenzen / 1

- Call-by-value

```
void increase(int counter) {
  counter++;
}
```

Tut nicht, was es soll!

- Referenz-Parameter

```
void increase(int & counter) {
  counter++;
}
```


C++: Referenzen / 2

- Referenz als Return-value

```
int & search(int array[]) {
  ... //zb. Suche größtes Element
  return (array[biggest]);
}
```

- Nicht veränderbare Return-Referenz

```
const int & search(int array[]) {
  ...
  return (array[biggest]);
}
```


Java-Pitfall: Dangling References!

```
const int & min(const int & i1, const int & i2) {
  if (i1 < i2)
    return (i1);
  return (i2);
}

int main() {
  const int & i = min(1+2, 3+4);
  return (0);
}
```

temporäre Variablen für diese angelegt

diese sind am Ende der Funktion `min()` nicht mehr gültig!

=> `i` zeigt ins Nirvana!!

- In C++ sind Referenzen nicht wirklich besser als Pointer und **machen böse Dinge!**

Overview

- Zielgruppe & Vorkenntnisse
- Klassenkonzept
- C++ Syntax
- Wiederholung UML
- STL - Standard Template Library
- Design Patterns

STL - Standard Template Library

- **Container:**
  nützliche Behältnisse für Daten;
  sequential und associative (key, value)
- **Iteratoren:**
  um durch Container durchzugehen
  Zugriff auf Elemente
- **Algorithmen:**
  häufig gebrauchte Algorithmen für Datenstrukturen in effizienter Implementierung

STL - Container & Iteratoren

Container

- **`vector`**
  "intelligentes Array", erweiterbar, Elemente aus der Mitte löschbar
- **`list`**
  doppelt verkettete Liste
- **`set`**
  ungeordnete Menge von Elementen
- **`map`**
  key, value-Paare von Elementen
- **...**

Iteratoren: "intelligente Pointer"

STL - Algorithmen

- **find**
  findet Element in Container
- **sort**
  ordnet Element in Container
- **for_each**
  geht alle Elemente 1x durch
- **reverse**
  dreht Element-Reihenfolge um
- **copy**
  kopiert Container
- **...**

STL - Beispiel / 1

```
#include <vector>

vector<int> vec;
vector<int>::iterator vecIt;

vec.push_back(3);
vec.push_back(5);

for(vecIt = vec.begin();
    vecIt != vec.end();
    vecIt++) {
    cout << *vecIt << endl;
}
```


STL - Beispiel / 2

```
vector<int> vec (10, 0); //init 10 elements =0
vector<int>::iterator vecIt;

vec[3] = 1;
vec[4] = 2;

int i = vec.size();

vec[15] = 3;

vec.resize(n+1);
```

Kein Range-check!

Tut was böses

Wie bei Array: n+1 => 0...n

STL - Beispiel / 3

```
vector<int> vec;
vector<int>::iterator vecIt;

sort(vec.begin(), vec.end());

vecIt = find(vec.begin(), vec.end(), 2);

vec.erase(vecIt);
```


Overview

- Zielgruppe & Vorkenntnisse
- Klassenkonzept
- C++ Syntax
- Wiederholung UML
- STL - Standard Template Library
- Design Patterns

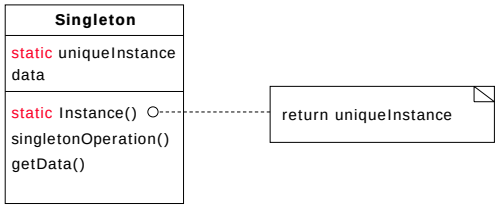
Design Patterns

- "GoF"-Book: *Design Patterns - Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, Vlissides
- "Pattern" = Muster
- Katalog mit "Design Experience"

Bekannter Erfahrungsschatz an Lösungen für immerwiederkehrende Probleme

Singleton Pattern / 1

- **Zweck:**
  - 1 globaler Zugriffspunkt auf Objekt
  - genau 1 Instanz des Objekts (oder genau *n*)



Singleton Pattern / 2

.h
```
class Singleton {
  public:
    static Singleton * Instance();
  protected:
    Singleton();
  private:
    static Singleton * instance_;
}
```

.cc
```
Singleton * Singleton::instance_ = NULL;

Singleton * Singleton::Instance() {
  if(instance_ == NULL)
    instance_ = new Singleton;
  return instance_
}
```


Singleton Pattern / 3

Verwendungsbeispiel

```
Singleton * printer_spooler;

printer_spooler = Singleton::Instance();

printer_spooler->doSomething();
```

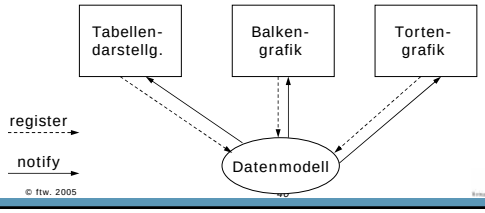
Kurzform

```
Singleton::Instance()->doSomething();
```

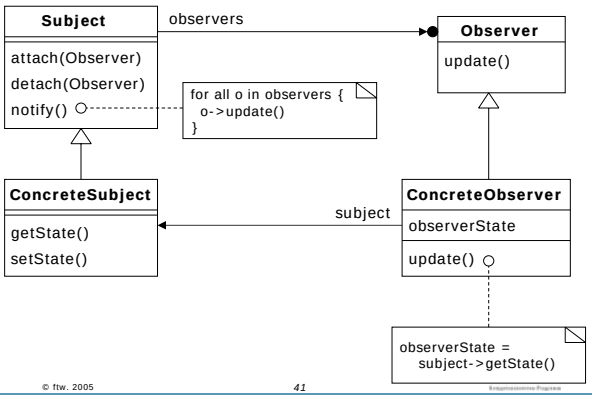
Kann von überall im Code aufgerufen werden!

Observer Pattern / 1

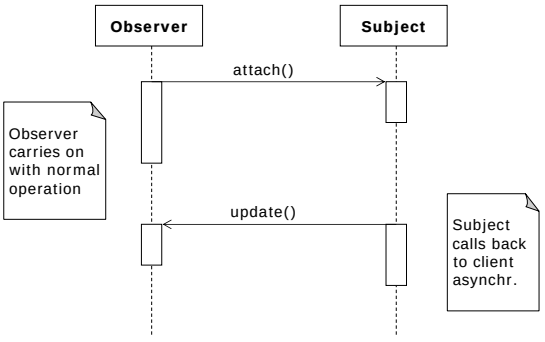
- Auch bekannt als Listener, Callback Idiom
- **Zweck:**
  - Konsistenten Status zwischen mehreren, lose gekoppelten Objekten erhalten
  - Beispiel: 2 unterschiedliche grafische Darstellungen "lauschen" an Datenobjekt und reagieren auf Veränderungen



Observer Pattern / 2



Observer Pattern / 3



References

- **Practical C++ Programming,** 2nd Edition, Steve Oualline, O'Reilly, ISBN 0-596-00419-2
- **Effectiv C++ programmieren** - *50 Wege zur Verbesserung Ihrer Programme und Entwürfe,* Meyers, Addison-Wesley, ISBN 3-8273-1305-8
- **Design Patterns** - *Elements of Reusable Object-Oriented Software,* Gamma, Helm, Johnson, Vlissides, Addison-Wesley, ISBN 0-201-63361-2
- **UML Distilled Second Edition** - *A Brief Guide to the Standard Object Modeling Language,* Fowler, Scott, Addison-Wesley, ISBN 0-201-65783-X

Folgetermine

- **Fr, 5. Aug, 9:00-12:00**
  Version control and collaborative working with Subversion (SVN)
- **Di, 9. Aug, 9:00-12:00**
  Tools: make, doxygen, Kdevelop, vi,...
- **Di, 23. Aug, 9:00-12:00**
  Extreme Programming light and good development practices
- **Fr, 26. Aug, 9:00-12:00**
  offen, wahrscheinlich Kurzvortrag Python

The End

Danke für die Aufmerksamkeit!