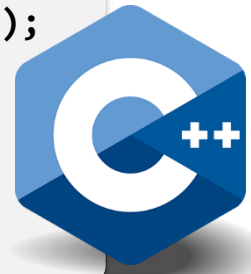
Einführung in die Softwareentwicklung

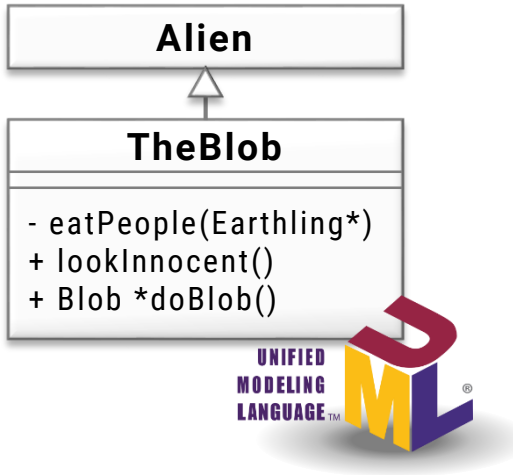
SOMMERSEMESTER 2020

Institut für Informatik

JG|U JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
// This C++ class models a famous B-movie monster
class TheBlob : public Alien {
 private:
   virtual void eatPeople(Earthling *e);
 public:
   virtual void lookInnocent() const;
   virtual Blob *doBlob();
}
```





Foliensatz #09

Objektorientierte Programmierung in C++ und Python

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

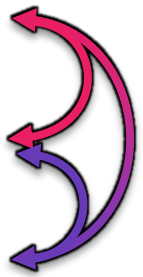
Übersicht

Objektorientierte Programmierung

- Prinzipien von objektorientierter Programmierung
- Unterstützung für OOP in C++
- Hilfsmittel für den Entwurf:
  UML & Klassendiagramme

Ausblick: (Objektorientierter) Softwareentwurf

- Design Patterns
- Architekturmuster
- Beispiele



Beispiel-/anwendungsorientierte Diskussion

„Objektorientierung“

Was ist OOP?

Objekt**o**rientierte **P**rogrammierung

- Kapselung in Objekte
  - Abstrakte Datentypen
  - Objekte haben Identitäten (Persistenz unabh. vom Aufrufstack)
- Polymorphie
  - Gleiche Schnittstelle für verschiedene Daten / Algorithmen
- Vererbung
  - Wiederverwendung von Code + Datenstrukturen

Unterschiede

- Python: „Smalltalk“-OOP
- C++: „Simula“-OOP

Wiederholung

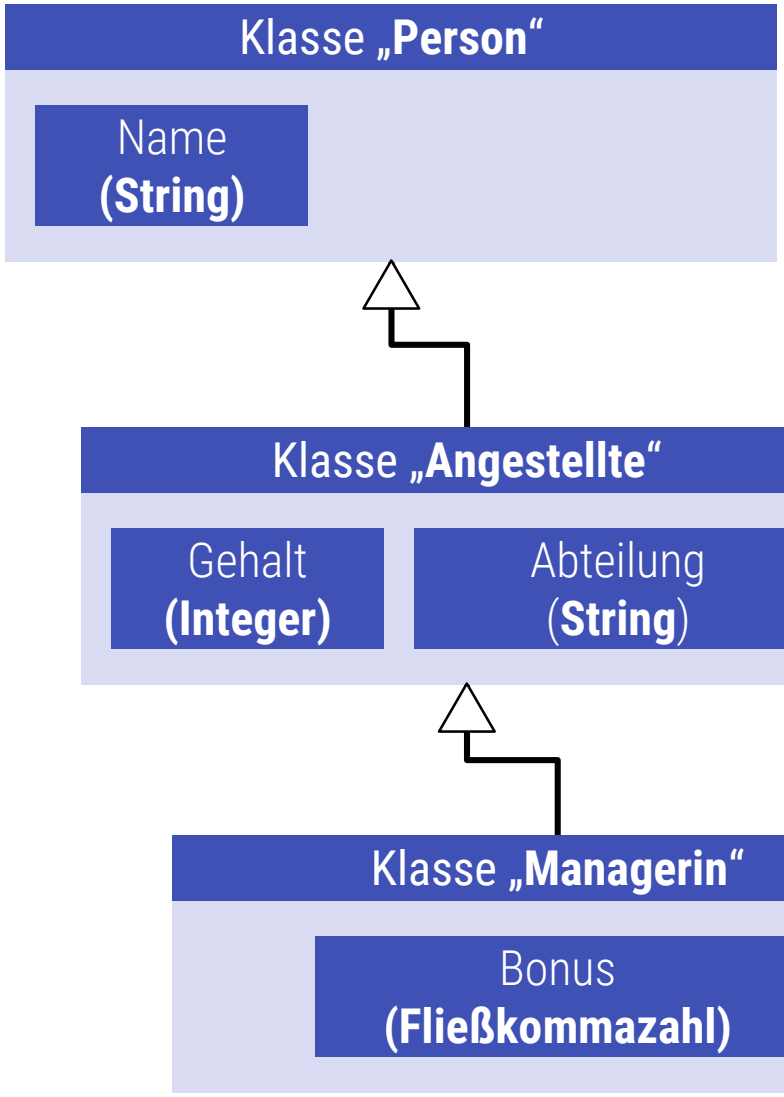
OOP in Python („Smalltalk“-Stil)

- Dynamische Typisierung
  - Polymorphie durch „Nachrichten“ (dyn. Methodenaufruf)
- Alle Daten sind „Identitäts“-Objekte (Pointer)
- Sehr flexibel (alles dynamisch)

OOP in C++ („Simula“-Stil)

- Statische Typisierung
  - Polymorphie durch „Subtyping“ via Vererbung
- Value- und Identity-Objects möglich
- Schneller & sicherer (Typprüfung)

Beispiel: Datenmodellierung



Person: Eintrag für Personen in unseren Datenbestand

Angestellte: Arbeit in einer Abteilung, bezieht festes Gehalt

Managerin: Leistungsabhängiges Gehalt; Bonus ist ein Faktor zwischen [0.5,...,2]

Griechische Philosphen

Definitionshierarchie

- „Genus proximum et differentia specifica“
- Geht auf Aristoteles zurück
- Definition als Hierarchie (Baum oder DAG)
- Schrittweise Spezialisierung von Oberklassen
  - Offensichtlich uneindeutig, wenn keine Baumstruktur vorliegt
  - Trotzdem oft nützlich
  - Motivation für Vererbung

Beispiel aus EiP: Vererbung in Python

```
class Person:
    def __init__(self, name):
        self.name = name
    def get_name(self):
        return self.name


class Angestellte(Person):
    def __init__(self, name, gehalt):
        super.__init__(name)
        self.gehalt = gehalt
    def berechne_gehalt(self):
        return salary

class Managerin(Angestelle):
    def __init__(self, name, gehalt, bonus = 1.1):
        super.__init__(name, address)
        self.bonus = bonus
    def berechne_gehalt(self):
        return int(super.calc_salary()*bonus)
```

Beispiel aus EiP: Vererbung in Python

```
class Person:
    ...
    def __str__(self):
      return name
    ...

class Angestellte(Person):
    ...
    def __str__(self):
      return name \
           + super.__str__() + " verdient " \
           + str(berechne_gehalt()) + "€"
    ...

class Managerin(Angestellte):
    ...
    def __str__(self):
      return super.__str__() + ", davon " + str((bonus-1.0)*100.0)\
             + "% leistungsabhängiger Bonus."
    ...
```

Polymorphie

Welche Methode wird aufgerufen?

Möglichkeiten:

- (Statischer Typ: Nach Programtext)
- Dynamischer Typ: Objekt zur Laufzeit

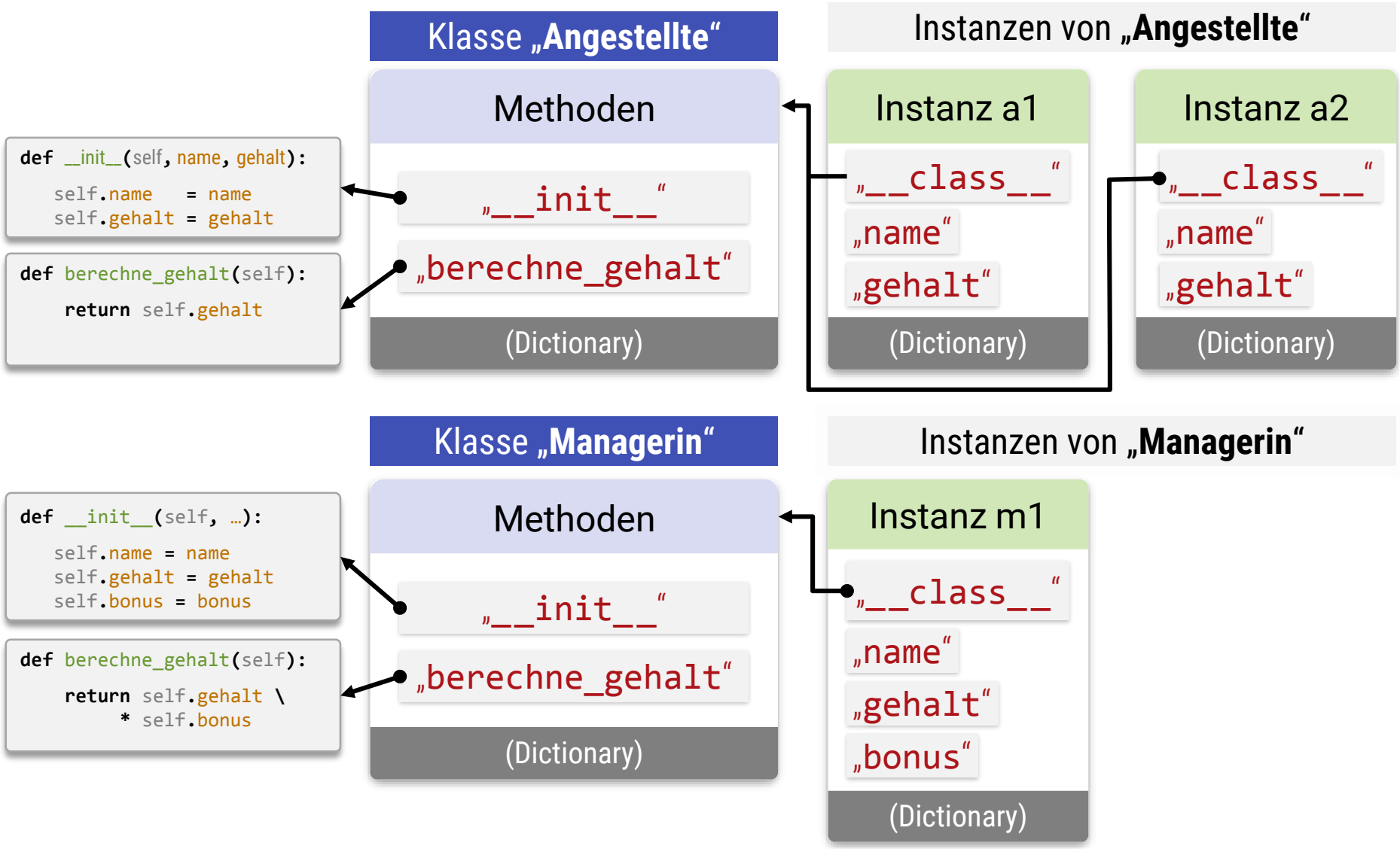
„Dynamic Dispatch“

- In Python immer dynamisch!

Beispiel aus EiP: Vererbung in Python

```
class Person:
    ...
    def berechne_gehalt(): …


class Angestellte(Person):
    ...
    def berechne_gehalt(): …


class Managerin(Angestellte):
    ...
    def berechne_gehalt(): …


# somewhere else:
p1 = Person()
p2 = Managerin()

g1 = p1.berechne_gehalt() # Person
g2 = p2.berechne_gehalt() # Managerin!
```

Implementation: Dictionaries

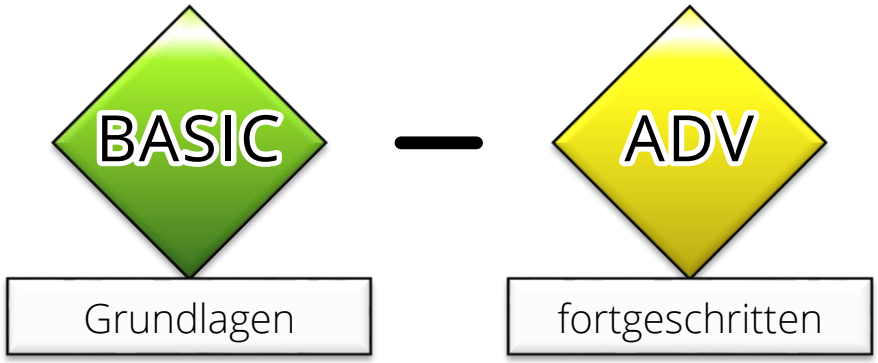


OOP in Python

Polymorphie

- Durch dynamischen Methodenaufruf!
  - Klasse des Objektes bestimmen
  - Suche nach Methodenname in Klasse
  - Aufruf falls gefunden
    - Sonst Fehler
- Jede Variable kann beliebige Objekte enthalten
  - Keine Typeinschränkung – alles zur Laufzeit
  - „Dynamische Typisierung“
  - Duck Typing: Wenn Methode gefunden, wird sie aufgerufen
- Klassenhierarchien weniger wichtig
  - Nur zum (direkten) wiederverwenden von Code

Werkzeuge für OOP in C++



Werkzeuge

Werkzeuge

- Klassen (**`struct`**/**`class`**)
- Memberfunktionen / -datenfelder
- Kontrolle der Sichtbarkeit (neu!)
- Vererbung (wie in Python) / Subtyping (neu!)
- Explizite Zeiger (für Identitätsobjekte; optional)
- Virtueller Methodenaufruf
  für dynamic Dispatch (neu, optional!)
- **`dynamic_cast<…>(…)`** (neu!)
- Mehrfachvererbung (auch in Python, dort einfacher)

Klassen, Members

Containerklasse: Listen

```
struct IntList {
    int *memory;
    int size;
    // Konstruktur
    IntList(unsigned initialSize = 0);
    // Copy-Konstruktur
    IntList(const IntList &other);
    // Destruktor
    ~IntList(unsigned initialSize = 0);
    // Zugriff auf Elemente
    int &operator[](unsigned index);
    // Zuweisungsoperator
    void operator=(const IntList &other);
};
```

Kontrolle der Sichtbarkeit

Containerklasse: Listen

```
struct IntList {
 private: // Zugriff nur innerhalb IntList (aus Memberfunktionen)
   int *memory;
   int size;
 public: // Zugriff von überall her
   // Konstruktur
   IntList(unsigned initialSize = 0);
   // Copy-Konstruktur
   IntList(const IntList &other);
   // Destruktor
   ~IntList(unsigned initialSize = 0);
   // Zugriff auf Elemente
   int &operator[](unsigned index);
   // Zuweisungsoperator
   void operator=(const IntList &other);
};
```

Zugriffskontrolle

Drei Optionen

- „`private:`“
  - Zugriff für nachfolgende Members nur aus Memberfunktionen der exakt gleichen Klasse
- „`protected:`“
  - Relevant nur bei Vererbung
  - Zugriff für nachfolgende Members nur aus Memberfunktionen der Klasse und aller Nachfahren
- „`public:`“
  - Zugriff ohne Einschränkungen

Wozu ist das gut?

Drei Optionen

- „`public:`“
  - Öffentliche Schnittstelle der Komponente
  - Hinweis an Benutzer: Änderungen brechen Kompatibilität
- „`private:`“
  - Interne Implementationsdetails
  - Hinweis an Benutzer: „Änderungen jederzeit möglich“
    - Keine stabile Schnittstelle
- „`protected:`“
  - Schnittstelle für Nachfahren
  - Nur nützlich, um Nachfahren zu bauen
  - Benutzer von außen kann diese Details ignorieren

class vs. struct

„Klassen“ in C++

- In C++ sind die Schlüsselworte „**`struct`**“ und „**`class`**“ *fast* äquivalent
  - „**`struct`**“ stammt noch aus C und wurde übernommen
- Einziger Unterschied:
  - **`class`** hat „**`private:`**“ als Standard-Sichtbarkeit
  - **`struct`** hat „**`public:`**“ als Standard-Sichtbarkeit
  - Ich habe bisher „struct“ verwendet, um die public/protected/private Diskussion zu verschieben

class vs. struct

„Klassen“ in C++

- Äquivalent:

```
class … {
 public:
   …
};
```

```
struct … {
    …
};
```

- Äquivalent:

```
struct … {
 private:
    …
};
```

```
class … {
    …
};
```

Wann was?

Wann soll ich was benutzen?

- Völlig egal – *Geschmackssache!*

Empfehlung?

- Meine Empfehlung (Konvention, übern. aus C#)
  - **`class`** für „Identitätsobjekte“
    - „Echte“ Objekte
    - Mit virtuellen Methoden, nur via Pointer
  - **`struct`** für „Wertobjekte“
    - Wertsemantik
    - Überladene Operatoren
    - Keine Vererbung, auch ohne Pointer

Statische Members

IntList.h

```
struct IntList {
 private: // Zugriff nur innerhalb IntList (aus Memberfunktionen)
   int *memory;
   int size;
 public: // Zugriff von überall her
   // Statische Memberfunktion – braucht keine Instanz
   static IntList createExampleList();
   // Statische (globale) Variable
   static int meaningOfLife;
};
```

IntList.cpp

```
#include "IntList.h"
IntList IntList::createExampleList() { // „static“ hier weglassen!
   return IntList(3);
}
int IntList::meaningOfLife=42; // muß nochmal definiert werden! „static“ auch weglassen!
```

„static“

Statische Members

- Klassenmethoden
  - Keine Instanz
  - Kein „this“ pointer
  - Globale Methode für ganze Klasse
  - Gleich wie eine normale Methode (ohne Klasse), lediglich im Namensraum `Klasse::Methode` statt global `Methode`
  - public/protected/private möglich
- Klassenvariablen
  - Im Namensraum der Klasse, public/protected/private
  - Globale Variable (immer noch schlechter Stil)
    - Akzeptierte Ausnahmen: Konstanten (static const…)

Werkzeuge

Werkzeuge

- Klassen (`struct`/`class`)
- Memberfunktionen / -datenfelder
- Kontrolle der Sichtbarkeit (neu!)
- Vererbung (wie in Python) / Subtyping (neu!)
- Explizite Zeiger (für Identitätsobjekte; optional)
- Virtueller Methodenaufruf
  für dynamic Dispatch (neu, optional!)
- **`dynamic_cast<…>(…)`** (neu!)
- Mehrfachvererbung (auch in Python, dort einfacher)

Vererbung in C++

Containerklasse: Listen

```
// Very simple example, we use “class”es this time, for a change…
class Person {
 public:
   std::string name;
};

class Angestellte : public Person {
 public:
   int gehalt;
   std::string abteilung;
};

class Managerin : public Angestellte {
 public:
   double bonus;
};
```

Vererbung

Vererbung in C++

- Schlüsselwort „`: public`“ für „öffentliches“ Erben

Hinweis (ignore if you like)

- Man kann auch „**`privat`**“ und „**`protected`**“ erben
  - Die Sichtbarkeit der geerbten Bestandteile ändert sich entsprechend
  - Subtypbeziehungen entsprechend unsichtbar
- Meine Empfehlung: nur „**`public`**“-Vererbung
  - Ich habe die anderen Optionen noch nie benutzt

Vererbung in C++

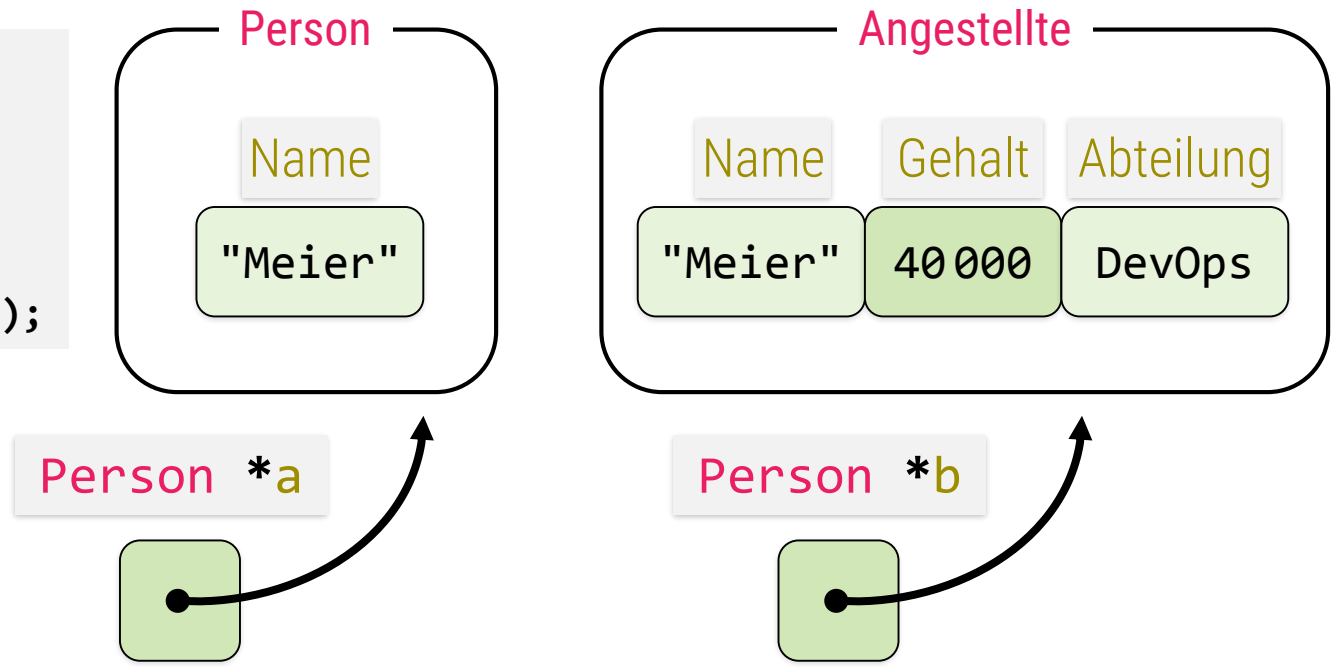
Containerklasse: Listen

```
// Very simple example, we use “class”es this time, for a change…
class Person {…};                                  // name
class Angestellte : public Person {…};             // name, gehalt, abteilung
class Managerin : public Angestellte {…};          // name, gehalt, abteilung, bonus

// Polymorphie und Subtyping („richtig“, via Zeiger)
{ Person      *p1 = new Person();
  Angestellte *a  = new Angestellte();
  Person      *p2 = new Angestellte();
  Managerin   *m  = new Managerin();
  delete p2; p2 = m;}

// Dies geht auch, tut aber nicht das was man als erstes denkt…
{ Person      p1 = Person();
  Angestellte a  = Angestellte();
  Person      p2 = Angestellte();  // Nur gemeins. Daten (Typ Person) werden kopiert!
  Managerin   m  = Managerin();
  p2 = m;}                         // Nur gemeins. Daten (Typ Angestellte) werden kopiert!
```

Referenzen auf Instanzen

CODE

```
Person *a
    = new Person();

Person *b
    = new Angestellte();
```



Referenzen

- Referenzen auf „Angestellte“ und „Person“ brauchen gleich viel Speicher
- Instanzen unterschiedlich groß (bei „**new**“)
  - Einmal angelegt, kann eine „Angestellte“ wie eine „Person“ behandelt werden

Vererbung in C++

Vererbung

```
// Very simple example, we use “class”es this time, for a change…
class Person {…};                                  // name
class Angestellte : public Person {…};             // name, gehalt, abteilung
class Managerin : public Angestellte {…};          // name, gehalt, abteilung, bonus

// Polymorphie und Subtyping („richtig“, via Zeiger)
Person      *p  = new Person();
Angestellte *a  = new Angestellte();
Managerin   *m  = new Managerin();
Person      *p2 = a;
Person      *p3 = m;
cout << p->name;    // ok!
cout << a->name;    // ok!
cout << p2->name;   // ok!
cout << p3->name;   // ok!
cout << p->gehalt; // Typfehler! (Compilerfehler)
cout << a->gehalt; // ok!
cout << a->bonus;   // Typfehler! (Compilerfehler)
cout << ((Managerin*)p3)->bonus;  // ok (aber riskant; Compiler würde auch p2 erlauben)!
```

Statische Methoden

Vererbung

```
// Very simple example, we use “class”es this time, for a change…
class Person                              {…void doSomething();…};
class Angestellte : public Person         {…void doSomething();…};
class Managerin : public Angestellte      {…void doSomething();…};

// Standard-Aufruf, ohne Dynamic Dispatch
Person      *p  = new Person();
Angestellte *a  = new Angestellte();
Managerin   *m  = new Managerin();
Person      *p2 = a;
Person      *p3 = m;
// Aufruf genau nach Bezugstyp des Zeigers
cout << p->doSomething();  // Aufruf von Person::doSomething()
cout << a->doSomething();  // Aufruf von Angestellte::doSomething()
cout << m->doSomething();  // Aufruf von Managerin::doSomething()
cout << p2->doSomething(); // Aufruf von Person::doSomething()
cout << p3->doSomething(); // Aufruf von Person::doSomething()
cout << ((Managerin*)p3)->doSomething(); // Aufruf von Managerin::doSomething()
```

Member-Funktionen

Memberfunktionen / Methoden

- Aufruf in C++ standardmäßig nach *Bezugstyp*
- `Angestellte a; a.doSomething();`
  ruft
  `Angestellte::doSomething()` auf
- `Angestellte *a = new Managerin();`
  `a->doSomething();`
  ruft immer noch
  `Angestellte::doSomething()` auf!

Wenn man das nicht will → *Virtuelle Methoden*

Virtuelle Methoden

Vererbung

```
class Person {
   virtual void doSomething() {…} // Diese Methode virtuelle aufrufen!
   virtual ~Person() {} // Virtueller Destruktor ist notwendig – nicht automatisch!!!
};
class Angestellte : public Person {
   virtual void doSomething();   // überschriebene Mehode; automatisch immer virtuell
};                               // Daher: Destruktur bleibt auch automatisch virtuell
class Managerin : public Angestellte  {
   virtual void doSomething();   // überschriebene Mehode; automatisch immer virtuell
};

Person *p = new Person(); Angestellte *a  = new Angestellte();
Managerin *m  = new Managerin(); Person *p2 = a; Person *p3 = m;

// Virtuelle Methoden: Aufruf nach dynamischem Typ!
cout << p->doSomething();   // Aufruf von Person::doSomething()
cout << m->doSomething();   // Aufruf von Managerin::doSomething()
cout << p2->doSomething(); // Aufruf von Angestellte::doSomething()
cout << p3->doSomething(); // Aufruf von Managerin::doSomething()
```

Abstrakte Methoden

Vererbung

```
class Person {
   virtual void doSomething() = 0; // Diese virtuelle Methode ist abstrakt!
   virtual ~Person() {}
};
class Angestellte : public Person {
   virtual void doSomething() {…}  // überschriebene Methode, jetzt konkret
};
class Managerin : public Angestellte  {
   virtual void doSomething() {…}  // überschriebene Methode
};

Person *p = new Person(); Angestellte *a  = new Angestellte();
Managerin *m  = new Managerin();

// Virtuelle Methoden: Aufruf nach dynamischem Typ!
cout << p->doSomething();  // Laufzeitfehler! – Methode ist abstrakt
cout << m->doSomething();  // Aufruf von Managerin::doSomething()
cout << p2->doSomething(); // Aufruf von Angestellte::doSomething()
cout << p3->doSomething(); // Aufruf von Managerin::doSomething()
```

Vererbung und virtuelle Methoden in C++

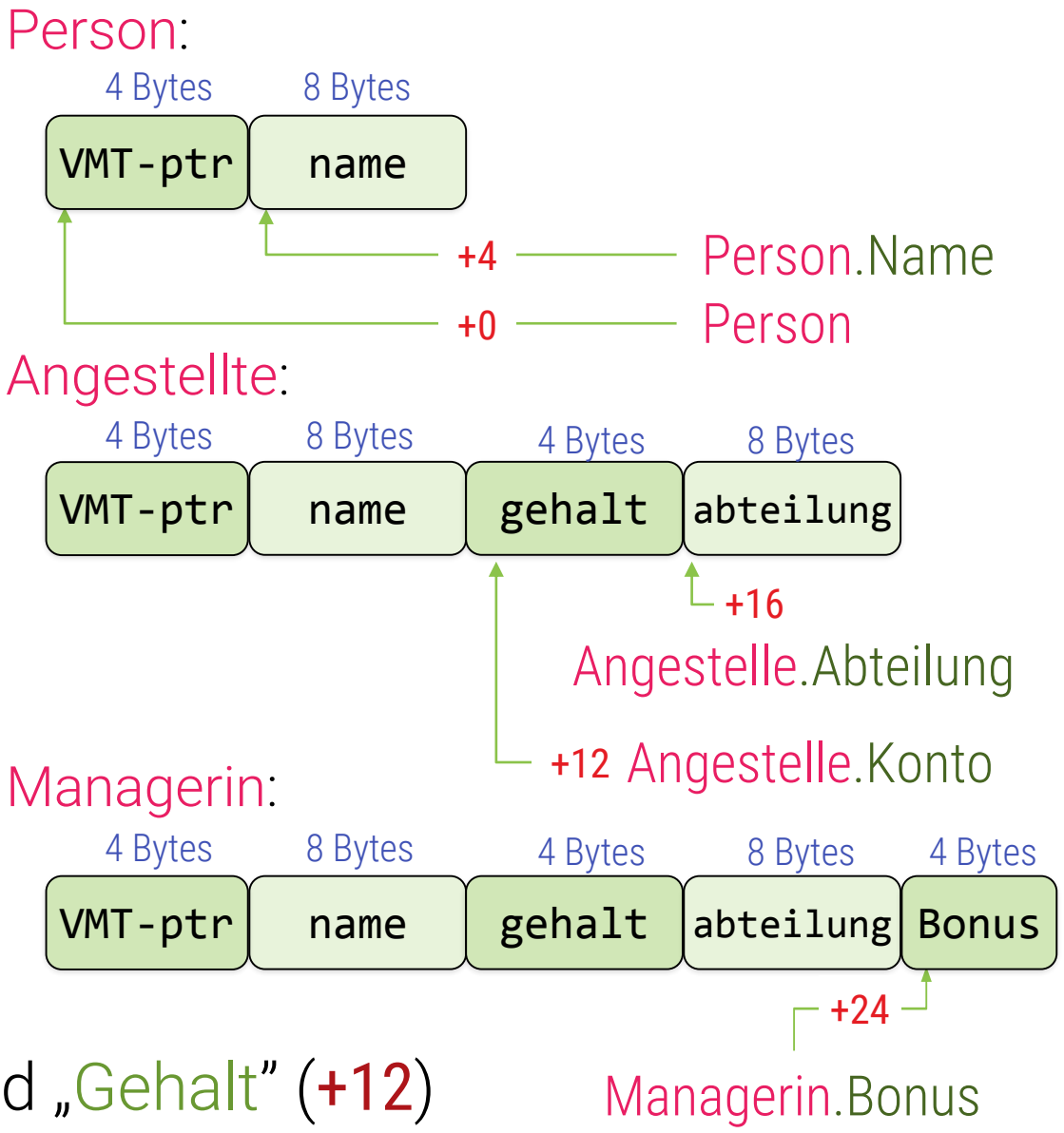
```
class Person {
 private:
   std::string name;
 public:
   virtual std::string toString() {return name;}
};

class Angestellte : public Person {
 private:
   int gehalt;
   std::string abteilung;
 public:
   virtual std::string toString() {
      return Person::toString()+" verdient "+std::to_string(berechneGehalt());
   }
   virtual int berechneGehalt() {return gehalt;}

};

class Managerin : public Angestellte {
 private:
   double bonus;
 public:
   virtual int berechneGehalt() {return int(gehalt*bonus);}

};
```

Technischer Hintergrund

Zugriff auf Datenfelder

Offsets

- Implementations-abhängig!
  - Hier nur das Prinzip!
- Offset 0:
  VMT Zeiger
- Offset 4:
  Person.Name
  ≙ Zeiger + 4
  - Auch für Angestellte „Zeiger + 4“
- Angestellte: neues Feld „Gehalt” (+12)

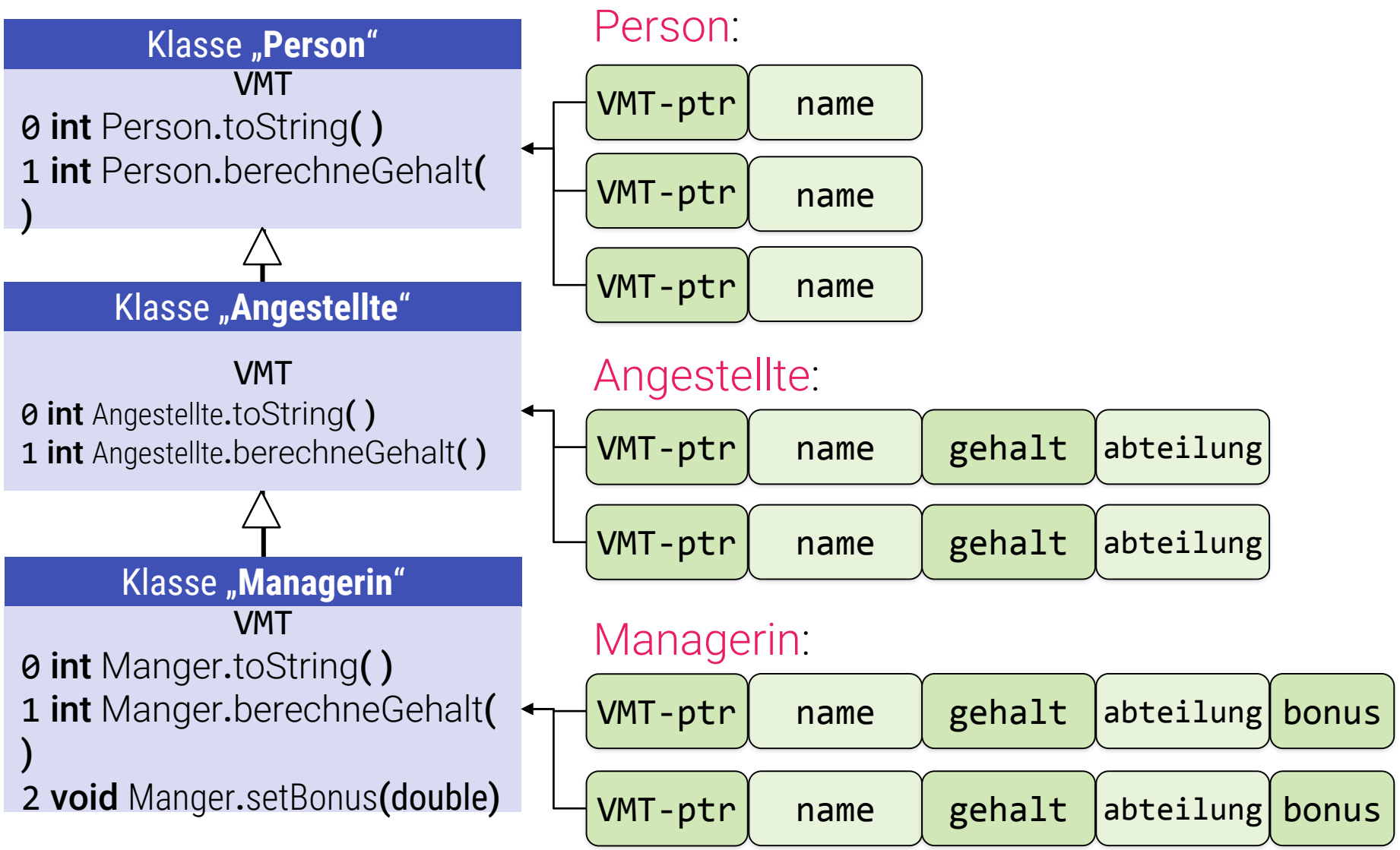


Virtuelle Methodentabellen

VMTs- Virtuelle Methoden Tabellen

- Englisch: „virtual method tables“
- Analog zu Python
  - Aber Offsets statt Hash-Tables *String → Function*
- Wesentlich schneller
- Statisches Subtyping nötig, um korrekte Funktion zu garantieren
- Weniger flexibel
  - Kein Duck-Typing via Offsets möglich!

Methoden: Dynamic Dispatch



Vererbung in C++

```
class Person {
 public:
   std::string name;
   virtual std::string toString() {
      return name;
   }
};

class Angestellte : public Person {

   int gehalt;
   virtual std::string toString() {
      return Person::toString()+" verdient "+std::to_string(berechneGehalt());
   }
   int annotherMethod() {...}

};

...

// somewhere else
Person *p1;
p1 = new Angestellte();
p1->name = "Herr Meier";
std::cout << p1->toString();
```

Headerfile *.h

```
class Person {                    // Praxisempfehlung: Nur Eine Klasse pro Header/Source File!
 public:
   std::string name;
   virtual std::string toString();
};

class Angestellte : public Person {

   int gehalt;
   virtual std::string toString();
   int annotherMethod();

};
```

Sourcefile *.cpp

```
#include "MyHeaderfile.h"


std::string Person::toString() {  // virtual nicht nochmal wiederholen!
   return name;
}

std::string Angestellte::toString() {
   return Person::toString()+" verdient  "+std::to_string(berechneGehalt());
}

int Angestellte::annotherMethod() {...}
```

Wie funktioniert das? – Emulation in reinem C

```
// Funktionszeiger – gibt es auch (genauso) in C++!
// Wir können annehmen, dass diese alle gleich groß sind (typ. 32/64-Bit Zeiger)
// Signatur ist nur Hinweis an Compiler für Aufruf; gespeichert wird immer das gleiche.
typedef void  (*Function)(void*);          // Pointer to simple function with pointer arg.
typedef char* (*ToStringFunction)(void*);  // Pointer to function returning char*
typedef int   (*BerGehaltFunction)(void*); // Pointer to function returning int

struct Person {
   Function *vmt; // C-Array! (Pointer to multiple functions)
   char *name;
};

struct Angestellte {
   Person parent; // Enthält VMT-Zeiger
   int gehalt;
};

char *Person_toString(void* self) {
   Person *typed_self = (Person*)self;
   return typed_self.name;
}
char *Angestellte_toString(void* self) {…}
int Angestellte_berechneGehalt(void* self) {…}
```

Wie funktioniert das? – Emulation in reinem C

```
#define PERSON_toString_INDEX 0

Person *Person_constructor() {
   Person *result = malloc(sizeof(Person));
   result.vmt = malloc(sizeof(Function));
   result.vmt[0] = (Function)&Person_toString();
   return result;
}

#define ANGESTELLTE_toString_INDEX 0
#define ANGESTELLTE_berechneGehalt_INDEX 1

Angestellte *Angestellte_constructor() {
   Angestellte *result = malloc(sizeof(Angestellte));
   result.vmt = malloc(sizeof(Function)*2);
   result.parent.vmt[0] = (Function)&Angestellte_toString();
   result.parent.vmt[1] = (Function)&Angestellte_toString();
   return result;
}

// Aufruf: Keine Typsicherheit in reinem C
Person *p1 = Angestellte_constructor();
p1->name = "Frau Meier";
char* result = ((ToStringFunction)p1.VMT[PERSON_toString_INDEX])(p1);
```

Virtueller Methodenaufruf

Virtueller Methodenaufruf

- Eine (effiziente) Form von „Dynamic Dispatching“

Idee (Pseudocode)

- Deklaration

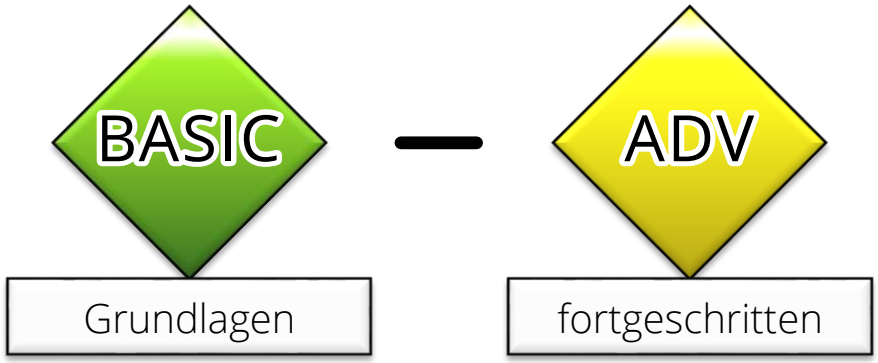
```
class ObjektTyp {
   ...
   Array of Function-pointers VMT[Anzahl Methoden];
   ...
};
```

- Aufruf:

```
ObjektTyp a;
...
call(a.VMT[INDEX_DER_METHODE]);
...
```

Werkzeuge für OOP in C++

(Fortsetzung)



Werkzeuge

Werkzeuge

- Klassen (`struct`/`class`)
- Memberfunktionen / -datenfelder
- Kontrolle der Sichtbarkeit (neu!)
- Vererbung (wie in Python) / Subtyping (neu!)
- Explizite Zeiger (für Identitätsobjekte; optional)
- Virtueller Methodenaufruf
  für dynamic Dispatch (neu, optional!)
- **`dynamic_cast<…>(…)`** (neu!)
- Mehrfachvererbung (auch in Python, dort einfacher)

Best Practices

Getrennte Aspekte in C++

- Virtueller Methodenaufruf
  - Schlüsselwort „virtual“ bei Methodendeklaration
- Instanzen via Zeiger anlegen
- Vererbung
- Wertsemantik via Operatoren ==, =, Copy-Constr., etc…

Frei kombinierbar

- Aber nicht alles macht Sinn

Empfehlungen

„Echte“ Objekte wie in Python oder JAVA

- Immer per Zeiger zugreifen „`Typ*`“
- Immer auf dem Heap anlegen (via **`new`**, **`delete`**)
- Alle Methoden, die überschrieben werden können, „**`virtual`**“ deklarieren
- Virtuellen Destruktor definieren!
  - Kann leer sein, z.B. `virtual ~MyClass() {}`
- Vererbung benutzen, um Untertypen zu definieren
- Polymorpher Aufruf via Zeiger vom Typ „`Oberklasse*`“ möglich

Empfehlungen

Methoden, die oft nicht überschrieben werden

- z.B. inline „getter“/„setter“ zur Kapselung privater Felder

```
privat:
  int a;
public:
  inline getA(){return a;}
  inline setA(int a) {
    if (a<=42) this->a = a;
    else       a = 42;
  }
```

- Im Zweifel einfach immer virtual
  (da nur etwas langsamer)

Empfehlungen

Empfehlung für „echte“ Objekte wie in Python oder JAVA

- Gemeinsame Oberklasse definieren
  - C++ hat keine Standardoberklasse wie Python, JAVA, C#
- Keine Operatoren überschreiben
  - Zuweisung via „`assign()`“ oder ähnlichem (wie in JAVA)
  - In C++-11 kann man in der Oberklasse die Operatoren abschalten

Diese beiden Empfehlungen

- Geschmackssache – nur ein Vorschlag von mir

```
class IdentityObject {                    // Echtes Beispiel: GeoX 6 Oberklasse (in dev.)

    // no copy constructor - use assign()!
    IdentityObject(const IdentityObject &) = delete;
    // no move constructor - use assign()!
    IdentityObject(const IdentityObject &&) = delete; // move semantics, see docs

    // no operators= use assign()!
    virtual void operator=(const IdentityObject &) = delete;   // copy semantics
    virtual void operator=(const IdentityObject &&) = delete; // move semantics

    // Assign the data of another object to this instance. Used automatically for copy() below.
    virtual void assign(const IdentityObject *otherObject);

    /// creates a deep copy of this instance. Calls assign.
    virtual IdentityObject *copy() const final {

        IdentityObject *result = this->getClass()->newInstance();
        result->assign(this);
        return result;

    }

    /// Get type information. Not provided by Standard C++ (unlike Python). Details omitted here.
    virtual ClassType *getClass() const {…}

    /// virtual destructor - C++ quirk requires explicit introduction
    virtual ~IdentityObject() { }

};
```

Empfehlungen

Empfehlung für Werttypen

- Keine Vererbung benutzen
  - Wenn, dann ohne polymorphe Funktionen
- Nutzung als Werte
  - Zeiger trotzdem möglich, und manchmal sinnvoll
  - z.B. für Geschwindigkeit
- Alle wichtigen Operatoren überschreiben
  - Copy-Constructor, assignment operator „=“
  - ggF. +, -, *, /, oder „!=“, „<“, „>“, oder „[]“, usw.
  - Speicherverwaltung beachten! (typischerweise: RAII)
- Keine virtuellen Methoden benutzen

Was ist was?

Was sind typische Werttypen?

- Objekte, wo nur deren Wert relevant ist
- Zahlen (Bruch, Complex, etc.)
- Mathematische Typen (Vektoren, Matrizen, etc.)
- Listen, Arrays, andere Container (wie im Beispiel)
- Complexere Datencontainer (z.B. „NumPy-Arrays“)

Was sind typische Identitätstypen?

- Alle Objekte, deren Identität eine Bedeutung hat
- GUIS: Widget, Window, Button
- OS/HW: Server-Objekt, Devicedriver, Prozess, Message

Werkzeuge

Werkzeuge

- Klassen (`struct`/`class`)
- Memberfunktionen / -datenfelder
- Kontrolle der Sichtbarkeit (neu!)
- Vererbung (wie in Python) / Subtyping (neu!)
- Explizite Zeiger (für Identitätsobjekte; optional)
- Virtueller Methodenaufruf
  für dynamic Dispatch (neu, optional!)
- **`dynamic_cast<…>(…)`** (neu!)
- Mehrfachvererbung (auch in Python, dort einfacher)

dynamic_cast

dynamic_cast

```
// Very simple example, we use “class”es this time, for a change…
class Person                          {…virtual void personMethod();…};
class Angestellte : public Person     {…virtual void angestMethod();…};
class Managerin : public Angestellte  {…virtual void managerMethod();…};

// Standard-Aufruf, ohne Dynamic Dispatch
Person      *p  = new Person();
Managerin   *m  = new Managerin();
Person *p1 = p;
Person *p3 = m;

// Aufruf genau nach Bezugstyp des Zeigers
cout << ((Managerin*)p3)->managerMethod(); // Aufruf ok!
cout << ((Managerin*)p1)->managerMethod(); // Aufruf stürzt ab! (Compiler: ok!)

Managerin* m3 = dynamic_cast<Managerin*>(p3); // Liefert Zeiger m3 = p3
if (m3) m3->managerMethod(); // sicher! Nur Ausführen, wenn Konvertierung möglich.
if (m3!=nullptr) m3->managerMethod(); // Lange schreibweise – equivalent

Managerin* m1 = dynamic_cast<Managerin*>(p1); // Liefert Zeiger m1 = nullptr
if (m1!=nullptr) m1->managerMethod(); // sicher! nix passiert.
```

dynamic_cast

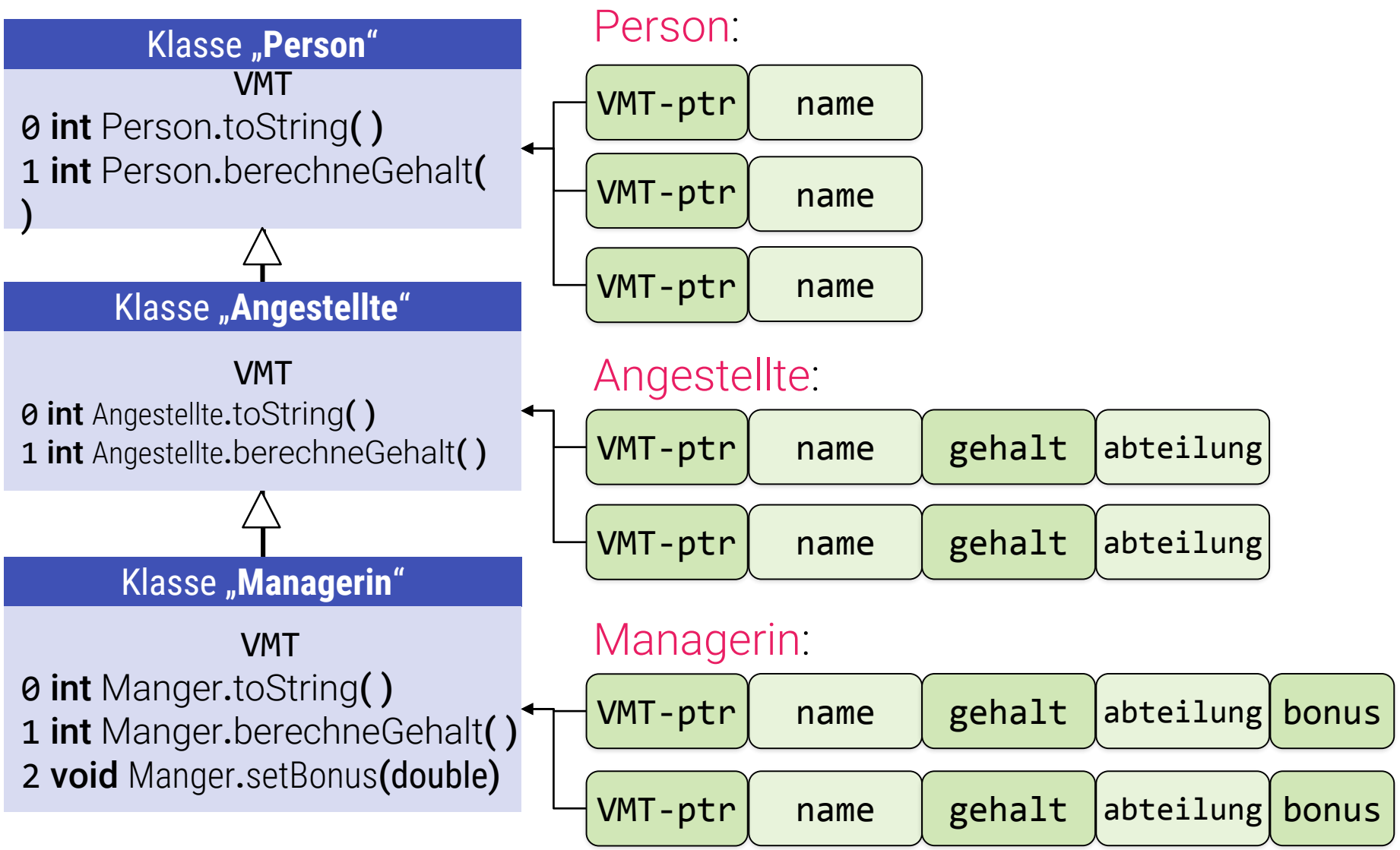
Sichere Typumwandlung

- **`dynamic_cast<…NeuerTyp…>(…Objekt…)`** führt eine sichere Typumwandlung durch
  - Ergebnis ist **`0`** (aka. „**`nullptr`**“ oder „**`NULL`**“) wenn Objekt nicht von „NeuerTyp“ abstammt
  - Anderenfalls: Ergebnis ist der gleiche Pointer „Objekt“
  - Statischer Bezugstyp ändert sich

Hinweise

- Nur auf Zeiger auf Objekte anwenden
- Nur bei Vorhandensein von virtuellen Methoden möglich (nutzt VMTs für Test!)

Methoden: Dynamic Dispatch



Werkzeuge

Werkzeuge

- Klassen (`struct`/`class`)
- Memberfunktionen / -datenfelder
- Kontrolle der Sichtbarkeit (neu!)
- Vererbung (wie in Python) / Subtyping (neu!)
- Explizite Zeiger (für Identitätsobjekte; optional)
- Virtueller Methodenaufruf
  für dynamic Dispatch (neu, optional!)
- dynamic_cast<…>(…) (neu!)
- Mehrfachvererbung (auch in Python, dort einfacher)

Mehrfachvererbung

Allgemein

- C++-Klassen (wie auch Python-Klassen) können mehrere Vorfahren haben
- Subtyping entsprechend (zu allen Vorfahren kompatibel)
- Doppelte Namen machen Probleme
  - In Python einfach feste Suchreihenfolge
  - In C++ komplizierte Regeln (z.B. „virtuelle Basisklassen“)

Mehrfachvererbung

Empfehlung

- Auf Mehrfachvererbung möglichst verzichten
- Persönliche Erfahrung:
  - In 19 Jahren C++ Entwicklung so gut wie nie gebraucht
  - Geschmackssache, wie immer

Sonderfall

- „Interfaces“ wie in JAVA
  - Klasse ohne Datenelemente,
  - Nur virtuelle Methoden, kein Code
- Dies kann durchaus nützlich sein

Interface Klassen

Vererbung in C++

```
class HasAValue {  // This is an interface class
 public:
   virtual double calculateValue() = 0;   // Abstrakte Methode – nicht aufrufbar
};


class Angestellte : public Person, public HasAValue {
 private:
   int gehalt;
   std::string abteilung;
 public:
   virtual std::string toString() {
      return Person::toString()+" verdient  "+std::to_string(berechneGehalt());
   }
   virtual int berechneGehalt() {return gehalt;}
   virtual double calculateValue() {return gehalt;}

};
```

Ein paar Problemchen…

(Unerwartete ?) Schwierigkeiten

Achtung in C++

- Konstruktoren sind nie virtuell
  - Konstruktoren werden vor Erzeugung der VMTs ausgeführt
  - Daher ist jeder Methodenaufruf in einem Konstruktor statisch gebunden (kein dynamic dispatch)
  - Das macht oft Ärger in der Praxis!
- Probleme mit templates
  - Templates und virtuelle Methoden sind nicht gut aufeinander abgestimmt
  - Anschauung/Regel: templates = Code entspr. kopieren
  - `vector<Person*>` und `vector<Angestellte*>` sind zwei völlig verschiedene Typen
  - Moderne Sprachen (z.B. Scala): Co-/Kontra-/Invarianz

(Unerwartete ?) Schwierigkeiten

Achtung in C++ (Verbesserung evtl. in C++2X)

- Es gibt keine Typen für Klassen!
  - Anders als in Python, JAVA, C#, Delphi
  - Ärgerlich, da es sehr nützlich wäre
  - Spart (etwas) Speicherplatz (für Reifikation der Klassen)
- Statische Methoden sind (daher) nie virtuell
  - Man bräuchte dazu Klassentypen
  - „Factories“ (Objekte, die andere Objekte erzeugen) sind komplizierter umzusetzen
- In C++ gibt es kein Introspection / Reflection
  - In JAVA, Python kann man die Struktur von Klassen zur Laufzeit ansehen. Das ist für bestimmte Anwendungen sehr nützlich.