# Programmieren (T3INF1004)

DHBW Stuttgart
Christian Bader
christian.Bader@lehre.dhbw-stuttgart.de
Christian Alexander Holz
christian.holz@lehre.dhbw-stuttgart.de

# Kapitelübersicht

# Recap Aufgabe: Objektorientierung - Einführung

1) Erstellen Sie eine Klasse Rectangle.
   a) Dieses hat eine Länge und eine Breite.
   b) Durch eine Funktion soll die Länge und Breite verändert werden können.
   c) Der Flächeninhalt soll berechnet und ausgegeben werden können.
   d) Die Länge und Breite kann durch einen Konstruktor eingestellt werden.
2) Zusatzaufgabe (wird nicht besprochen):
   a. Erstellen Sie eine Klasse Triangle, welches eine „Base“ und eine „Height“ hat und ebenfalls den Flächeninhalt berechnen und ausgeben kann

Testen Sie Ihre Klasse durch Aufruf in der `main` Funktion.

# Kapitel 2: Objektorientierung

# Kapitel 2: Objektorientierung

21. Grundlagen

22. Vererbung

23. Polymorphismus

Exkurs: Git

C++ Basics: Header Files

C++ Basics: Pointer und Referenzen

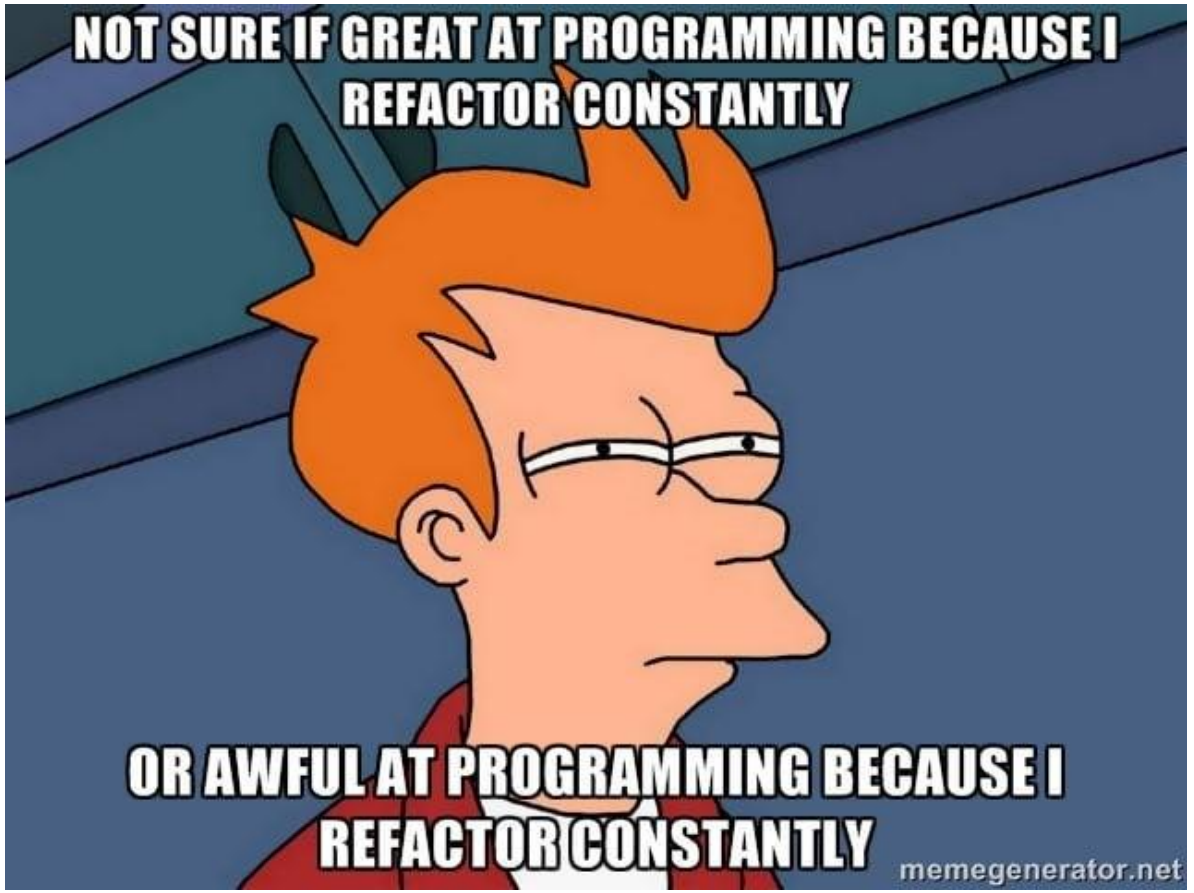
C++ Basics: Speicherverwaltung

# Interface Trennung

- Es ist guter Stil das Interface (also nach den nach außen sichtbaren Teil) von der Implementierung zu trennen
- → .hpp und .cpp Dateien.
- .hpp enthält Interface, d.h. Funktions-Signaturen (Deklaration)
- .cpp enthält alle Implementierungen (Definition)

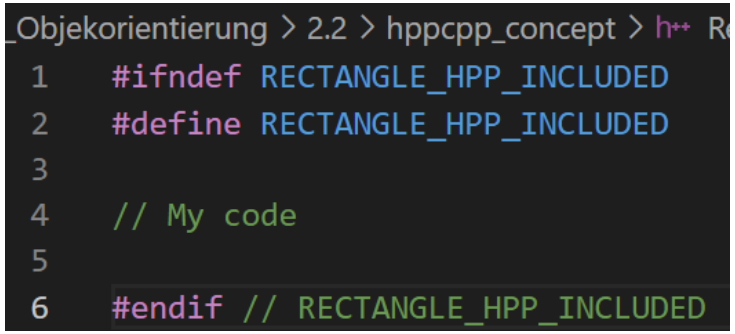
# Header Files: Einführung

Live



# Header Files: Best practices

- Jedes .hpp file enthält einen *include guard* (mit beachteter *naming convention*)




- Alternative: `#pragma once` (Preprocessor Direktive, viel verbreitet, kein cpp Standard)
- Jedes .hpp und .cpp file inkludiert alle benötigten Header selbst und nutzt keine über andere header inkludierten Funktionen und Objekte
- .hpp files dürfen sich **nicht gegenseitig inkludieren** (*cyclic includes*)
- `#include <…>` // für externe Header
- `#include „…“` // für selbst geschriebene Header

# VS-Code: task.json für das kompilieren von mehreren .cpp Dateien

- Dateien müssen in einem Ordner liegen

Rectangle.hpp | tasks.json | initialization_lists.cpp | task_vehi

.vscode > tasks.json > ...

```
{
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: g++.exe Aktive Datei kompilieren",
            "command": "C:\\msys64\\mingw64\\bin\\g++.exe",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${fileDirname}\\*.cpp",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Vom Debugger generierte Aufgabe."
        }
    ],
    "version": "2.0.0"
}
```

# enum class und switch cases

Live

# Aufgaben: Objektorientierung – .hpp Dateien, enum class, switch cases

Erstellen Sie ihre Klassen in .hpp und .cpp Files und rufen sie die Klassen aus einem separaten main.cpp file auf.

1) Erstellen Sie eine Klasse Person.
   a) Die einen Namen und eine Nationalität hat (aus: `de, en, it, es` → `enum class Nationality`)
   b) Die eine `string getName()` Methode zum abrufen des privaten Namens hat.
   c) Die eine Member Funktion `void greet(Person greetedPerson)` hat, die eine Grußformel in der Muttersprache in der Konsole ausgibt, z.B. „Buongiorno Jose“ für `it` und `greetedPerson` = Jose (→ switch case).

# Die std::vector Klasse, neue for loops und Referenzen

Live

# Aufgaben: Objektorientierung – .hpp Dateien, enum class, switches und for loops

Erstellen Sie ihre Klassen in .hpp und .cpp Files und rufen sie die Klassen aus einem separaten main.cpp file auf. Verwenden Sie die Person Klasse aus der vorherigen Aufgabe.

1) Erstellen Sie eine neue `Vehicle` Klasse. Diese soll …
   a) Über eine für jede Instanz fixe Anzahl an Sitzen verfügen.
   b) Eine Liste an Personen haben die gerade im Fahrzeug sitzen.
   c) Eine Funktion `enter(Person person)` und eine Funktion `exit(int seatNumber)` haben.
2) Erweitern Sie die enter-Funktion, so dass alle im Fahrzeug anwesenden von der neu einsteigenden Person begrüßt werden und danach alle im Fahrzeug sitzenden Personen die neue Person begrüßen.
3) Zusatzaufgabe (wird nicht besprochen)
   a) Erweitern Sie die Klasse Vehicle um eine Farbe aus: `blau, grün, gelb, rot` → `enum class`
   b) Geben Sie die Begrüßung in der jeweiligen Farbe auf der Konsole aus
   c) Erstellen Sie eine Klasse `TollStation` mit einer funktion `control` und einer Person `cashier`, welche eine Liste von Fahrzeugen kontrolliert. Dabei begrüßen alle Insassen aus jedem Fahrzeug in der jeweiligen Farbe den cashier (füge dazu eine weitere funktion `greetAll` zur Klasse `Vehicle` hinzu). Außerdem wird der Preis der Mautstelle ausgegeben.

# Kapitel 2: Objektorientierung

21. Grundlagen

22. Vererbung

23. Polymorphismus

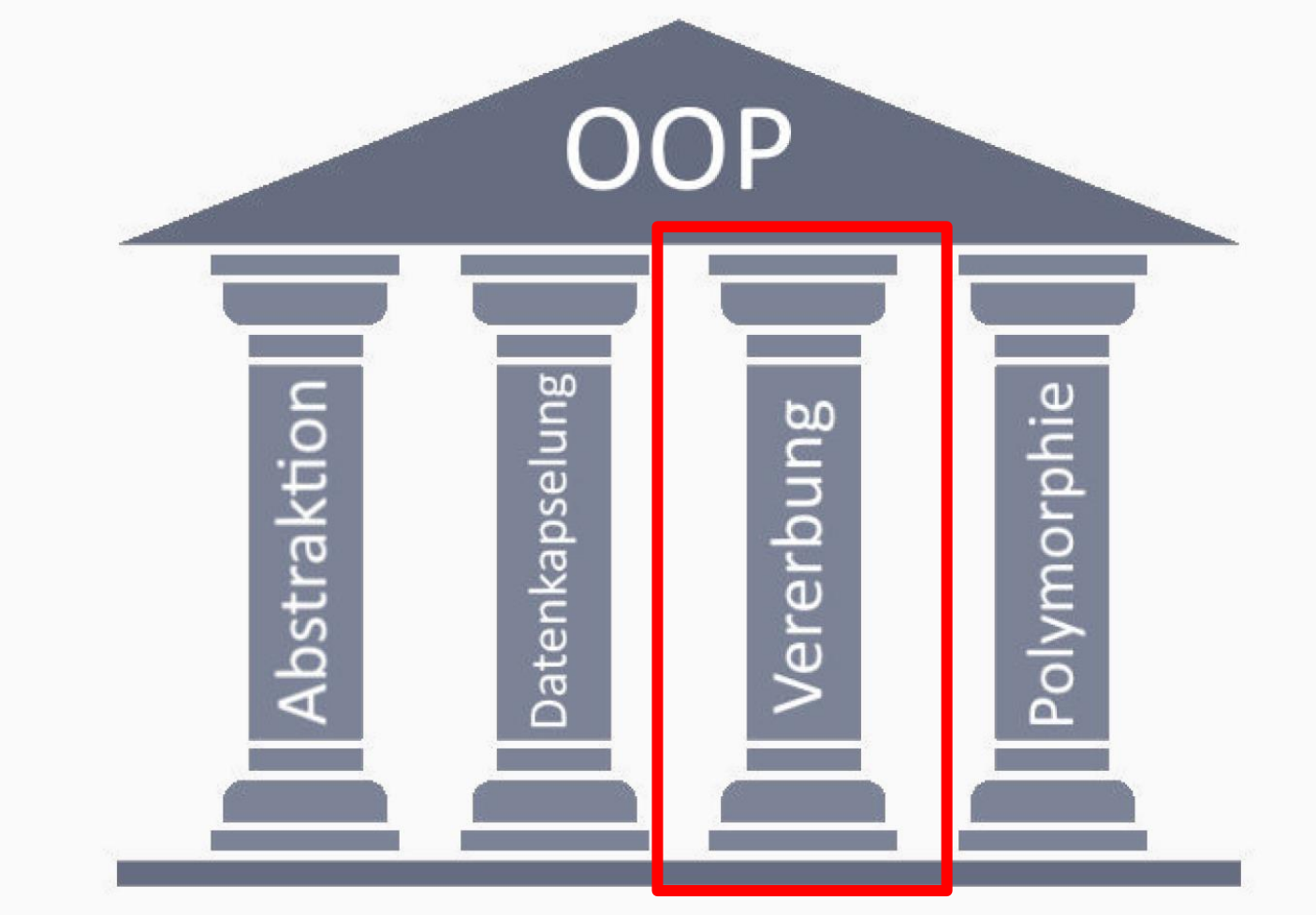
Exkurs: Git

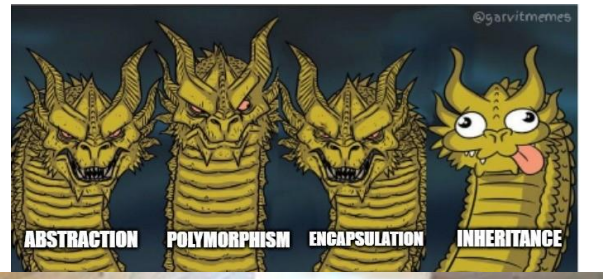
C++ Basics: Header Files

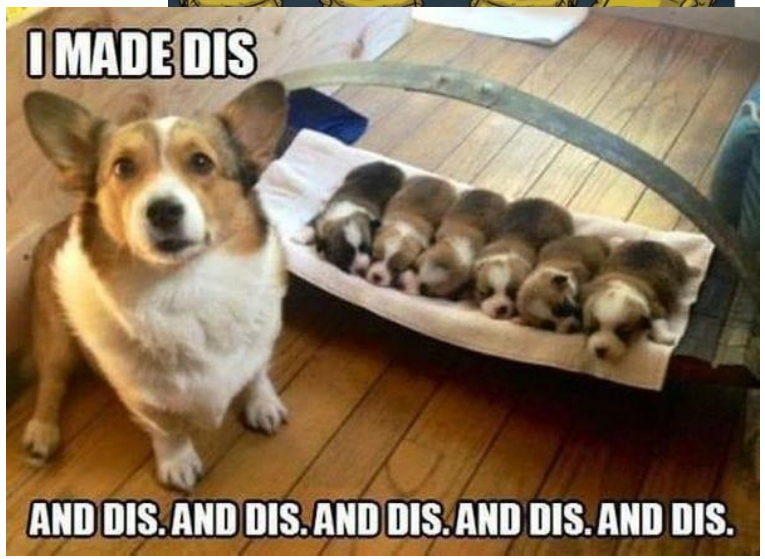
C++ Basics: Pointer und Referenzen

C++ Basics: Speicherverwaltung

# Die vier Säulen der objektorientierten Programmierung

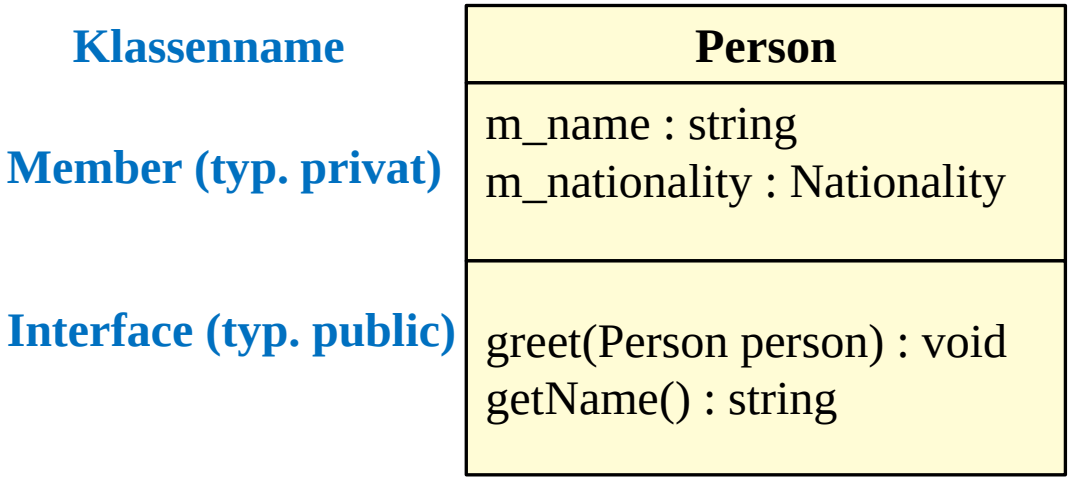


# Einschub: UML – Unified Modelling Language

- Zur Visualisierung von Klassen und deren Zusammenhängen: UML-Klassendiagramme
- Bsp.: Klasse Person

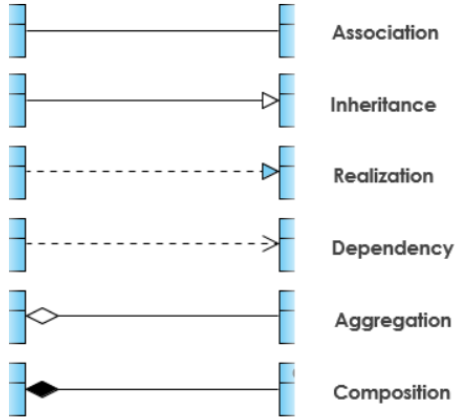


# UML - Motivation

- Modellierung von **has-a-Beziehung: Membervariablen!**
  - Bsp.: Eine Person hat einen Namen
- Modellierung von **is-a-Beziehung: Vererbung!**
  - z.B. ein Golf ist ein Auto, ein Student ist eine Person, …

- Überblick UML Klassendiagramme [UML Class Diagram Tutorial]
- Vielzahl an Tools zur Modellierung verfügbar
  - Beispiel: draw.io (diagrams.net), www.visual-paradigm.com
  - Manche können funktionalen Code erzeugen

Relationship types

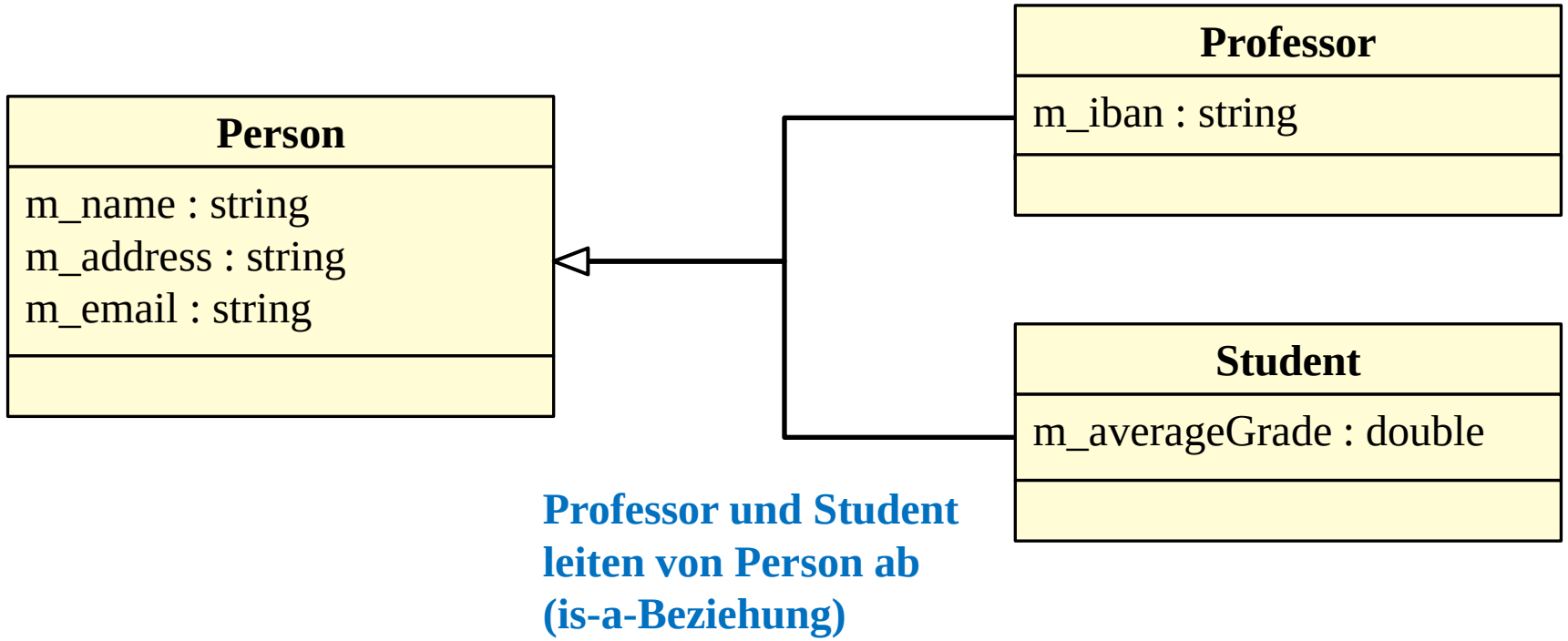


# Bsp: Personenverwaltung für die DHBW

- Zwei Arten von Personen: Studenten und Professoren

# Bsp: Personenverwaltung für die DHBW

- Basis-Klasse: **Person** ← Kind-Klassen: **Professor**, **Student**

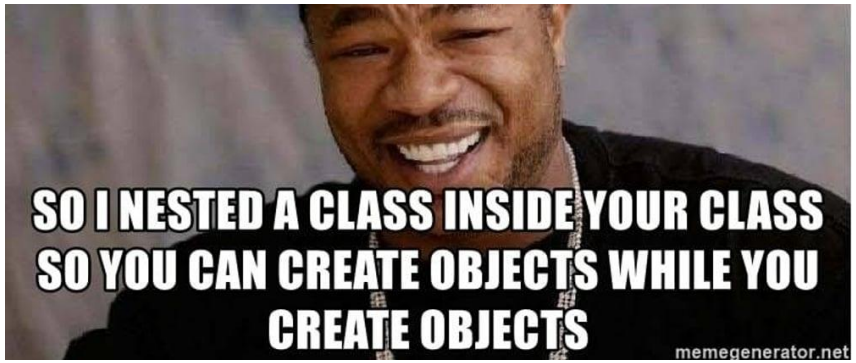


# Möglichkeiten: Vererbung

- Verhindern von Code-Duplizierung: Überschneidende Funktionalität in eigene Klasse
- **Aber:** nur bei **echter is-a-Beziehung** Vererbung nutzen!
- Komplexität reduzieren:
  - Übersichtlichkeit durch kleinere Klassen
  - Neue Abstraktionsebene durch Verallgemeinerung (und damit kleinere Klassen)
- Ermöglicht erstellen eines gemeinsamen **Interfaces** (von außen sichtbarer Teil) von Klassen (kommt nochmal ausführlicher)

# Nicht ohne Grund anwenden

- Kann die **Übersichtlichkeit** bei verschachtelter Vererbung verschlechtern
- Kann die **Wartbarkeit** verschlechtern → Elternklasse hat Einfluss auf Kinderklassen
- Am besten nicht ohne guten Grund anwenden: **KISS: *„Keep it stupid simple“***

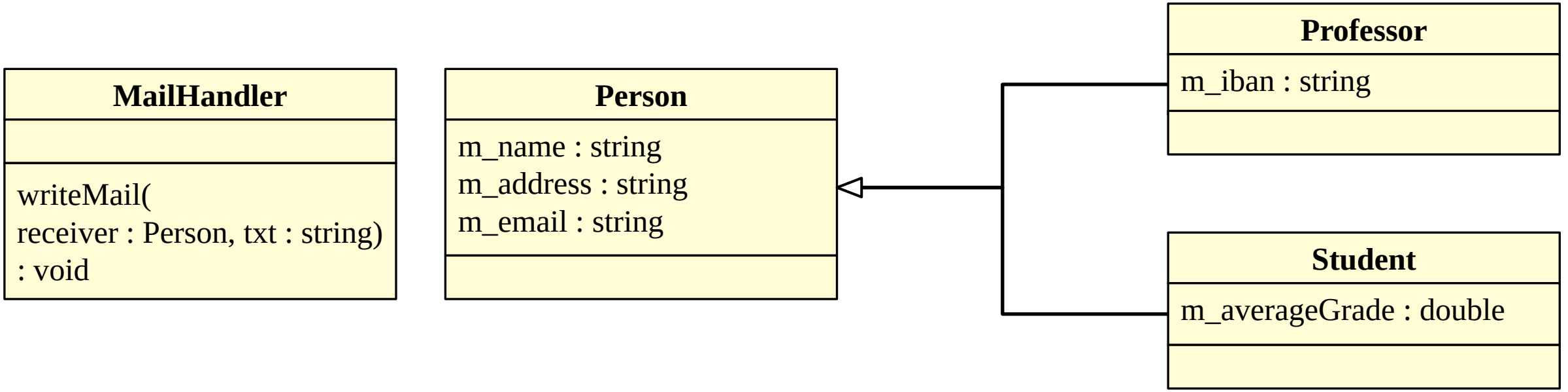


# Was sind korrekte Ableitungen?

Fragen Sie sich „Jede/r <Kind-Klasse> ist ein <Basis-Klasse>.“

- Auto ← Sportwagen | Van | Combi
- Nahrungsmittel ← Süßigkeiten | Fleisch | Obst | Gemüse | Teigwaren | Getränke
- Viereck ← Parallelogramm ← Rechteck ← Quadrat
- Viereck ← Parallelogramm | Rechteck | Quadrat
- Quadrat ← Rechteck
- Bier ← Pils | Weizen | Export | Radler

# Vererbung zur Generierung von Interfaces



# Vererbung

Live

# Zugriffsmodifikationen: public/protected/private

## Zugriffsmodifikationen

| Access | public | protected | private |
|---|---|---|---|
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

- Konstruktoren und Destruktoren werden bei Vererbung **nicht** übertragen

## Art der Vererbung

- Public Inheritance (base class / child class):
  - `public` → `public`
  - `protected` → `protected`
  - `private` → **Nie** in der abgeleiteten Klasse verfügbar
- Protected Inheritance (fast nie genutzt)
  - `public`, `protected` → `protected`
- Private Inheritance (fast nie genutzt)
  - `public`, `protected` → `private`

Zugriffsmodifikationen dienen der Kapselung

# Beispiele: Zugriffsmodifikationen

Frage: Was funktioniert?

```
2   class Base
3   {
4   public:
5       void a(){};
6   protected:
7       void b(){};
8   private:
9       void c(){};
10  };
11
12
13  int main()
14  {
15      Base base;
16      base.a();
17      base.b();
18      base.c();
19      return 0;
20  }
```

# Beispiele: Zugriffsmodifikationen

Frage: Was funktioniert?

```
1   class Base
2   {
3   public:
4       void a(){};
5   protected:
6       void b(){};
7   private:
8       void c(){};
9   };
10
11  class Derived : public Base
12  {
13  };
14
15
16  int main()
17  {
18      Derived derived;
19      derived.a();
20      derived.b();
21      derived.c();
22      return 0;
23  }
```

# Beispiele: Zugriffsmodifikationen

Frage: Was funktioniert?

```
1   class Base
2   {
3   public:
4       void a(){};
5   protected:
6       void b(){};
7   private:
8       void c(){};
9   };
10
11  class Derived : public Base
12  {
13      void test()
14      {
15          a();
16          b();
17          c();
18      }
19  };
20
21
```

# Aufgaben: Vererbung

1) Implementieren Sie die Klassen Person, Student, Professor und MailHandler aus diesem Kapitel.
2) Keine der Membervariablen der Klassen darf `public` sein. Falls sie auf eine Member-Variable zugreifen müssen, dann über einen Getter.
3) Testen Sie ihre Implementierung durch Aufrufe in der main.cpp

# Zusatzaufgaben: Vererbung

1) Implementieren Sie eine Klasse `PerceptionSensor` als Basisklasse von Radar und Lidar
2) `PerceptionSensor` hat eine maximale Detektionsreichweite und kann diese mit `printProperties` ausgeben
3) Erstellen Sie eine Klasse Objekt mit Distanz d und material (`kunststoff` oder `metall`, `enum`)
4) Erstellen Sie für Radar und Lidar die Funktion `detectObject`, welche prüft ob das Objekt detektiert wird
   a) Der Lidar detektiert das Objekt immer innerhalb der detektionsreichweite.
   b) Der Radar Detektiert Objekte aus Metall immer, hat aber eine Detektionswahrscheinlichkeit als Membervariable, mit der Objekte aus Kunststoff detektiert werden.

**Implementieren Sie alle Klassen in .hpp und .cpp Files!**

# Kapitel 2: Objektorientierung

21. Grundlagen

22. Vererbung

23. Polymorphismus

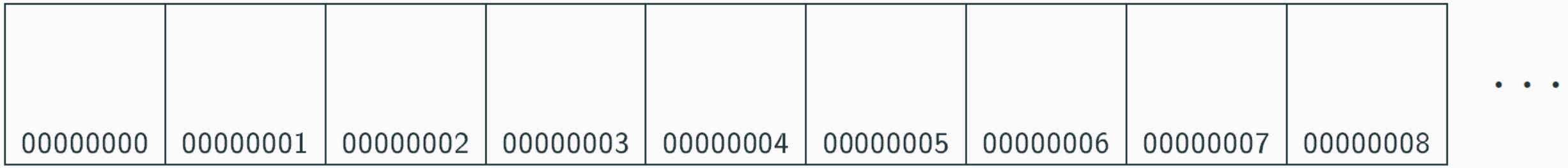
Exkurs: Git

C++ Basics: Header Files

C++ Basics: Pointer und Referenzen

C++ Basics: Speicherverwaltung

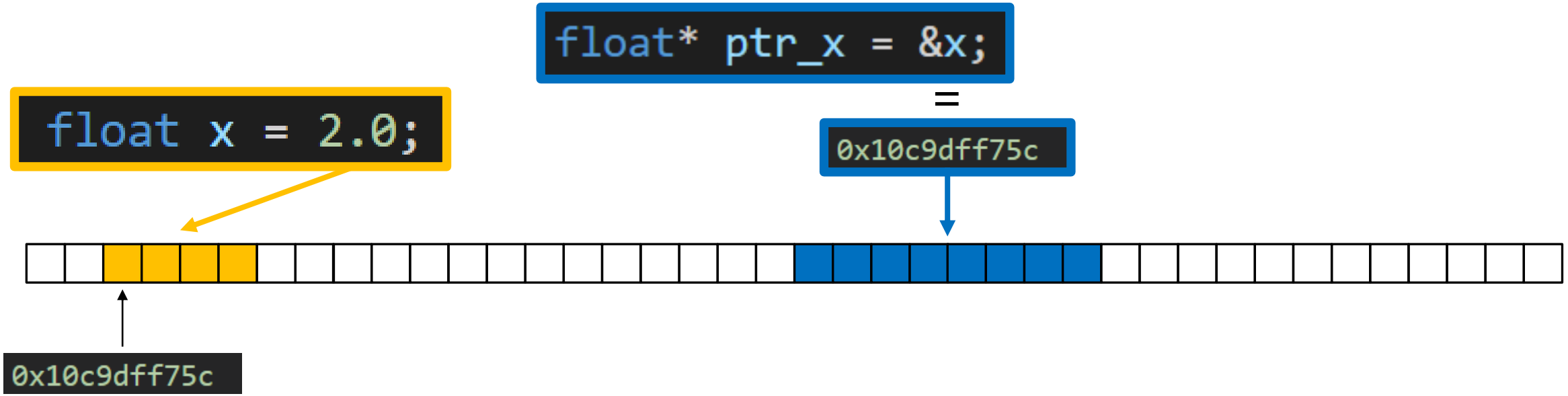
# Visualisierung auf Speicherebene




- Ausschnitt aus Speicherbereich (RAM)
- Jeder Speicherblock (ein Byte = 8 Bit) hat eine eindeutige Adresse

# Visualisierung auf Speicherebene

- Objekt vom Type `float` (4 Byte = 32 Bit) wird hinter Adresse `0x10c9dff75c` gespeichert
- Diese Adresse (64 Bit auf 64 Bit Maschine, also 8 Byte) wird im Pointer `ptr_x` gespeichert. Der Pointer selbst hat wieder eine eigene Adresse.



# C++: Pass by refrence / pass by value: Speed-up

```
2   int main()
3   {
4       SomeBigObject x;
5       foo(x)
6
7       return 0;
8   }
9
10  void foo(SomeBigObject x)
11  {
12      // do something with Object
13  }
```

Objekt wird kopiert!

```
33  int main()
34  {
35      SomeBigObject x;
36      foo(x)
37
38      return 0;
39  }
40
41  void foo(SomeBigObject& x)
42  {
43      // do something with Object
44      x.getPrivateValues();
45  }
```

Adresse von Objekt wird übergeben!

# C++: Referenzen

Live

# Vergleich: Pass by reference C/C++

```
void foo(SomeBigObject* x)
```

**C:**

- Pointer wird übergeben
- Vor Aufruf von Funktion: Referenzierung
- In Funktion: Dereferenzierung oder direkt mit pointer auf Objekt arbeiten SomeBigObject->doSomething();
- Nicht Nullpointer-safe!

```
void foo(SomeBigObject& x)
```

**C++:**

- & legt fest, dass Referenz übergeben wird.
- Vor Aufruf von Funktion: Nichts.
- In Funktion: Nichts (SomeBigObject.doSomething();)
- Nullpointer-safe!

# Kapitel 2: Objektorientierung



# New/delete, formally known as malloc/free

- Keywords `new` und `delete` werden genutzt um Speicher zu allokieren und wieder frei zu geben.
- Auch `new` und `delete` sollten soweit möglich vermieden werden (mehr dazu später)
- Regel: Für jedes `new` ein `delete`

```
double* ptr = new double;          // reserve memory for pointer (on heap)
delete ptr;                        // free reserved memory
ptr = nullptr;                     // set ptr to NULL


ptr = new double(7.0);             // reserve memory for pointer and set value to 7.0
delete ptr;
```

# Stack vs. Heap

## Stack

- Verwaltung abhängig von Programmiersprache, OS, System-Architektur
- Nicht direkt vom Entwickler beeinflussbar (aber z.B. in Assembler)
- Freigabe von Speicher erfolgt automatisch
- Größe wird bei Programmstart festgelegt
- Ohne Pointer nutzbar

VS

## Heap

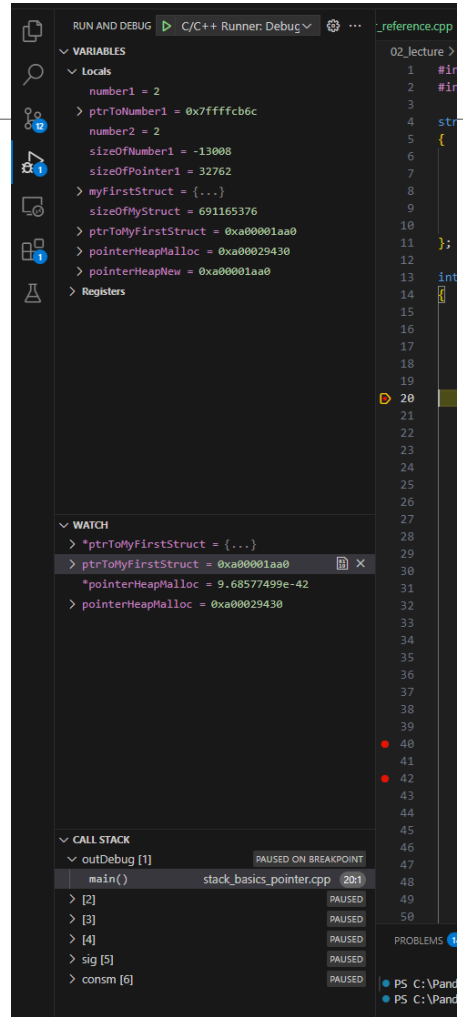
- Muss vom Entwickler manuell verwaltet werden
- Viele Programmiersprachen bringen Garbage-Kollektoren mit, die obsoleten Speicher finden und frei geben
- Manuell vergrößerbar
- Kann nur über Pointer angesprochen werden

```
Class myClass1;                    // Stack
Class* myClass2 = new Class();     // Heap
```

# Stack

Live

- Variablen nur lokal verfügbar (Locals)
- Call Stack zeigt alle auf dem Stack liegenden Funktionen



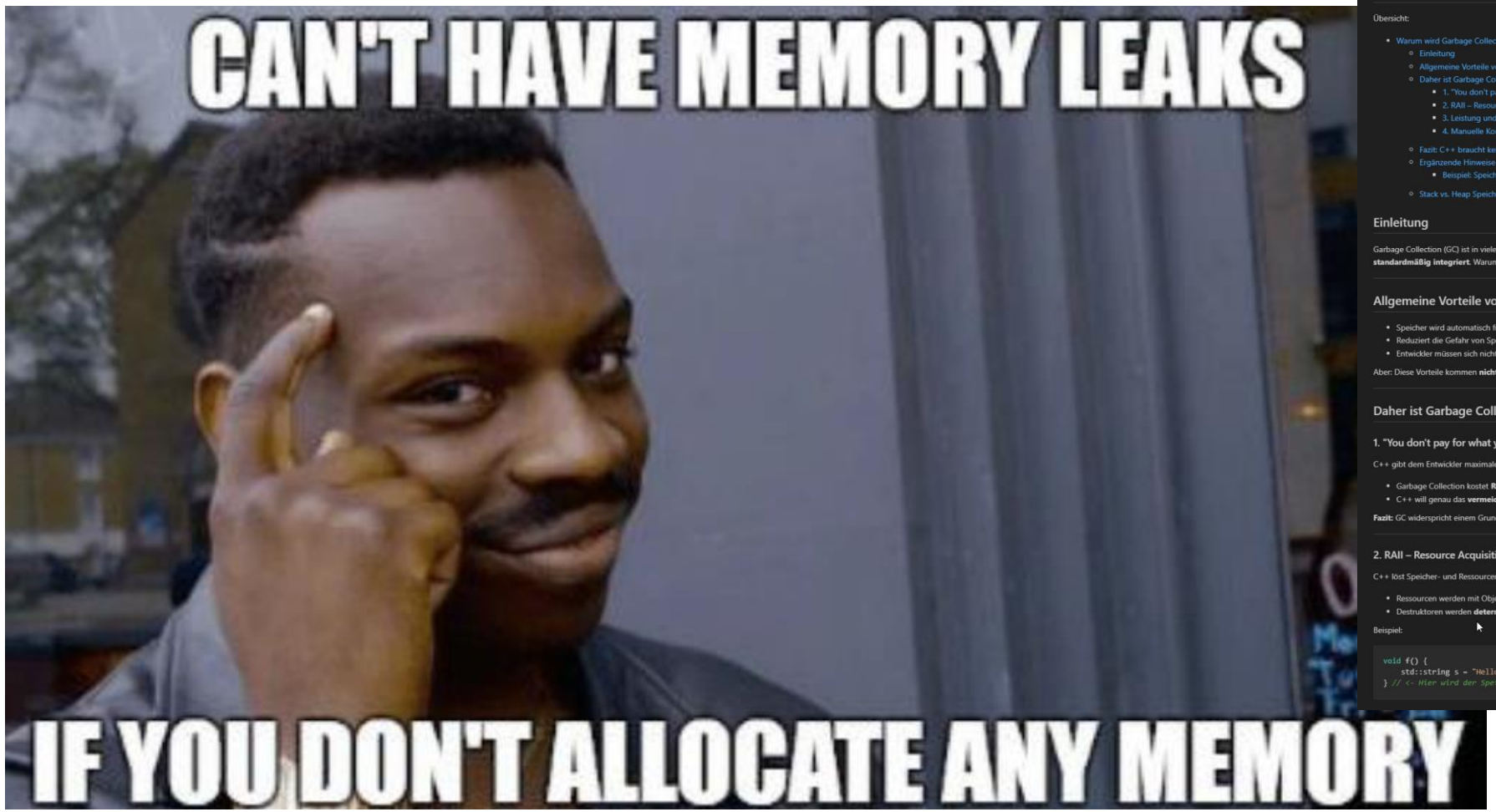
# Stack – Fragen

- Was passiert mit dem Stack bei einer Endlosrekursion?
- Was passiert bei einem Infinite-Loop?
- Wenn eine Schleife 10.000 mal durchläuft und darin eine Variable benötigt wird. Wie viel weniger Speicher wird initialisiert, wenn diese davor 1 mal deklariert wird?

# Fazit – Pointer und Speicherverwaltung

- **KISS!**
- Pointer nur benutzen wenn zwingend notwendig (in Legacy code oft nicht vermeidbar)
- Vermeiden Pointer gegenseitig zuzuweisen
- Call-by-Reference immer nutzen falls Argumente nicht trivial
- Embedded Software: Oft gibt es überhaupt keinen Heap!
- Wenn doch mit Pointern gearbeitet werden soll, dann gibt es smart Pointer
  `std::unique_ptr<Class> std::shared_ptr(new Class())`
  Diese kümmern sich um Speicherverwaltung (hier nicht im Detail besprochen).

# Fazit – Pointer und Speicherverwaltung




## Warum wird Garbage Collection in C++ kaum verwendet?

Übersicht:



### Einleitung

Garbage Collection (GC) ist in vielen modernen Programmiersprachen wie Java oder C# ein selbstverständlicher Bestandteil. In C++ hingegen ist sie bewusst **nicht standardmäßig integriert**. Warum?

### Allgemeine Vorteile von Garbage Collection

- Speicher wird automatisch freigegeben
- Reduziert die Gefahr von Speicherlecks
- Entwickler müssen sich nicht um Speicherfreigabe kümmern

Aber: Diese Vorteile kommen **nicht ohne Kosten.**

### Daher ist Garbage Collection in C++ ungewöhnlich

#### 1. "You don't pay for what you don't use" – Das C++ Prinzip

C++ gibt dem Entwickler maximale Kontrolle und vermeidet unnötige Kosten:

- Garbage Collection kostet **Rechenzeit und Speicher**, selbst wenn nichts bereinigt werden muss
- C++ will genau das **vermeiden**, wenn die Funktion nicht benötigt wird

**Fazit:** GC widerspricht einem Grundprinzip von C++

#### 2. RAII – Resource Acquisition Is Initialization

C++ löst Speicher- und Ressourcenmanagement durch RAII:

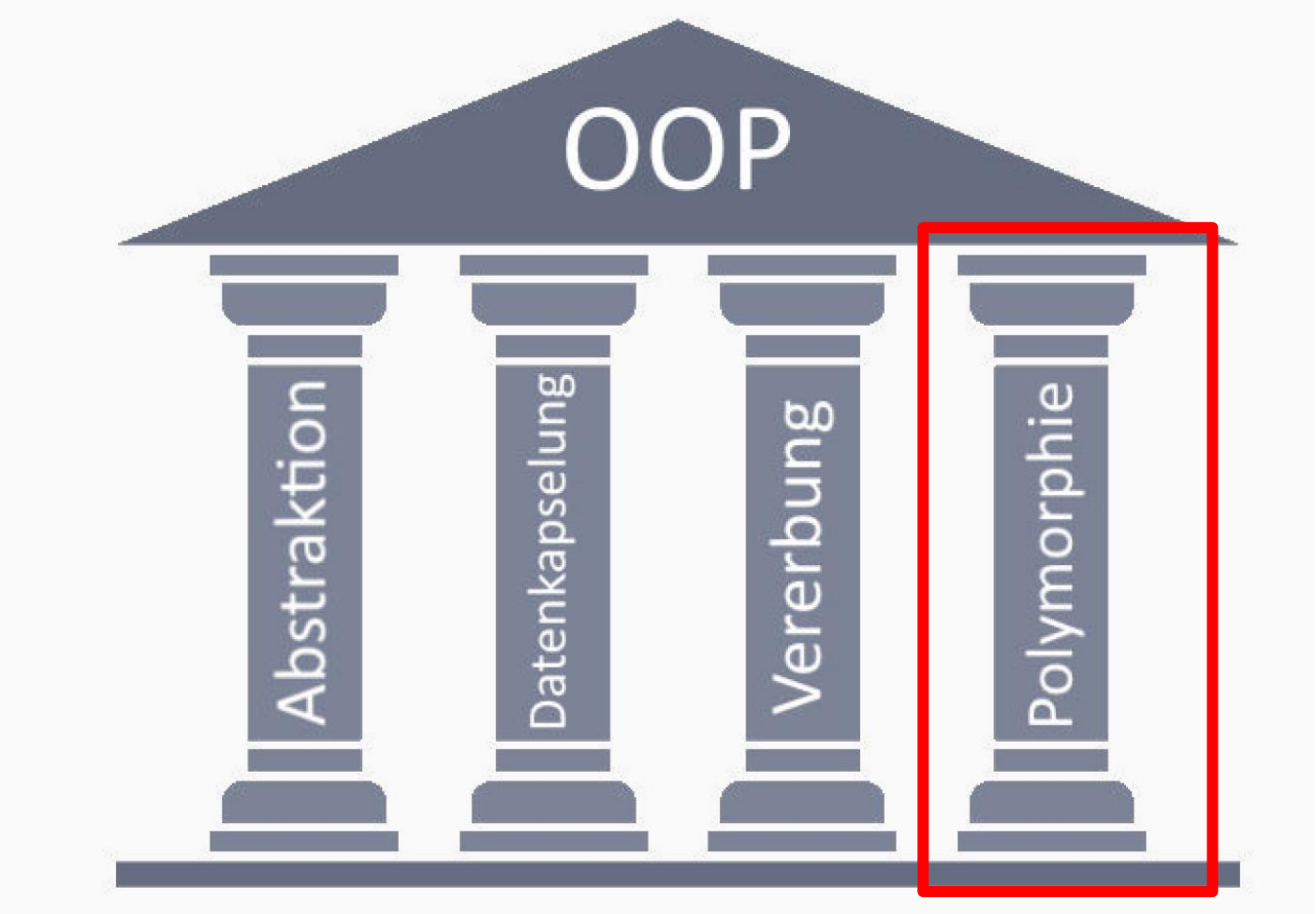
- Ressourcen werden mit Objektlebensdauer verwaltet
- Destruktoren werden **deterministisch** (zu einem klar definierten Zeitpunkt) aufgerufen

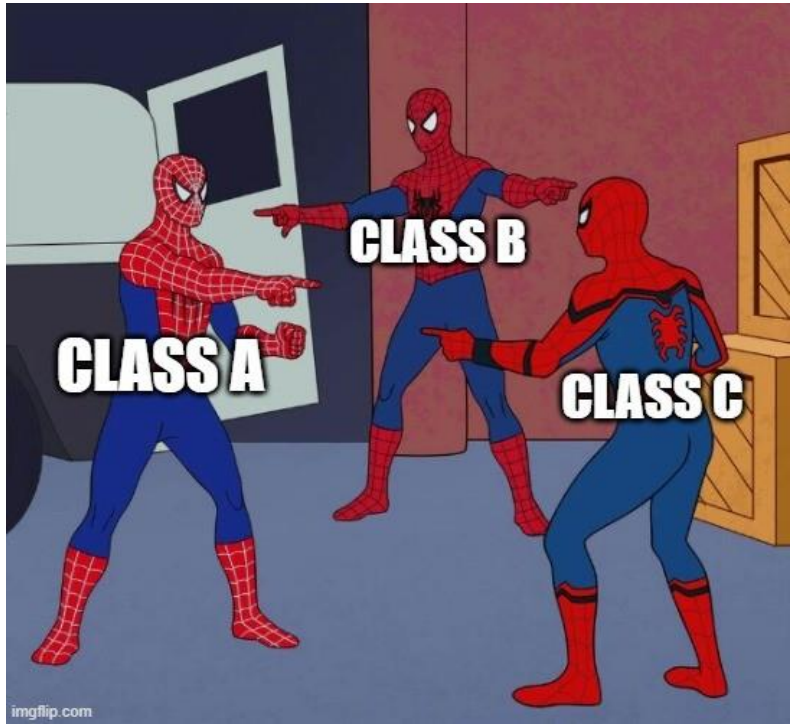
Beispiel:

```
void f() {
    std::string s = "Hello"; // Speicher wird automatisch freigegeben sobald wir den Scope verlassen
} // <- Hier wird der Speicher freigegeben
```

# Kapitel 2: Objektorientierung

# Die vier Säulen der objektorientierten Programmierung



# Polymorphismus

Live

# Wichtigster Vorteil von Vererbung: Polymorphismus

- **Definition**
  - Objekte mit der selben Basis-Klasse haben unterschiedliches Verhalten
  - Konkrete Implementierung von Methode nicht in Basis-Klasse sondern kann in den Kind-Klassen überschrieben werden.
- **Vorteile**
  - Wir können mit Objekten arbeiten von denen wir lediglich den Basis-Typ kennen
  - Ermöglicht Erstellen von Interfaces, die festlegen wie etwas genutzt wird (also welche Funktionen mit welcher Signatur `public` sind), aber nicht wie diese genau implementiert werden.
  - Damit entsteht eine **„Dependency-Firewall“**: Alles was unserer Klassen benutzt ist nur abhängig von der Interface-Basis-Klasse aber **komplett unabhängig** von der konkreten Implementierung der Methoden
  - Zudem kann dadurch eine neue Abstraktionsebene geschaffen werden → Komplexitäts Reduzierung

# Keyword: `override`

- Seit C++11 wird `override` unterstütz
- Gibt an, dass diese Funktion eine andere überlädt

## Vorteile

- Bessere Lesbarkeit
- Compiler Error, falls keine Überladung stattfindet

```
63  // Derived class
64  class Triangle : public Shape
65  {
66  public:
67      // Leave out the constructor for the first try
68      Triangle(double width, double height)
69      : Shape(width, height)
70      {}
71
72      double getArea() override
73      {
74          return (m_width * m_height * 0.5);
75      }
```

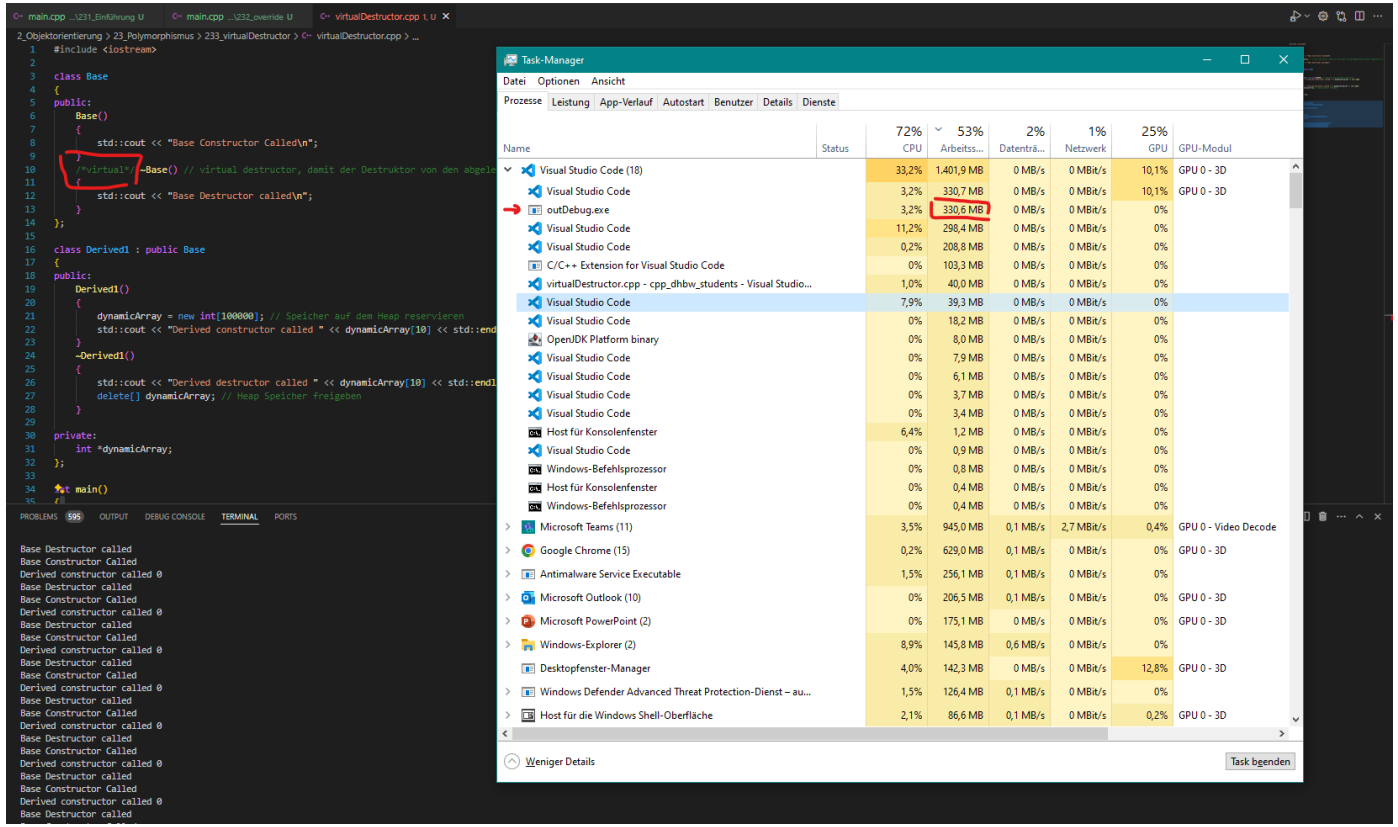
# Polymorphismus: virtueller Destruktor

Live

# Polymorphismus virtueller Destruktor

Live

- Ohne virtual destructor:
  - Derived Destruktor wird nicht aufgerufen
  - Cleanups werden nicht durchgeführt
  - Memory leak entsteht im Beispiel



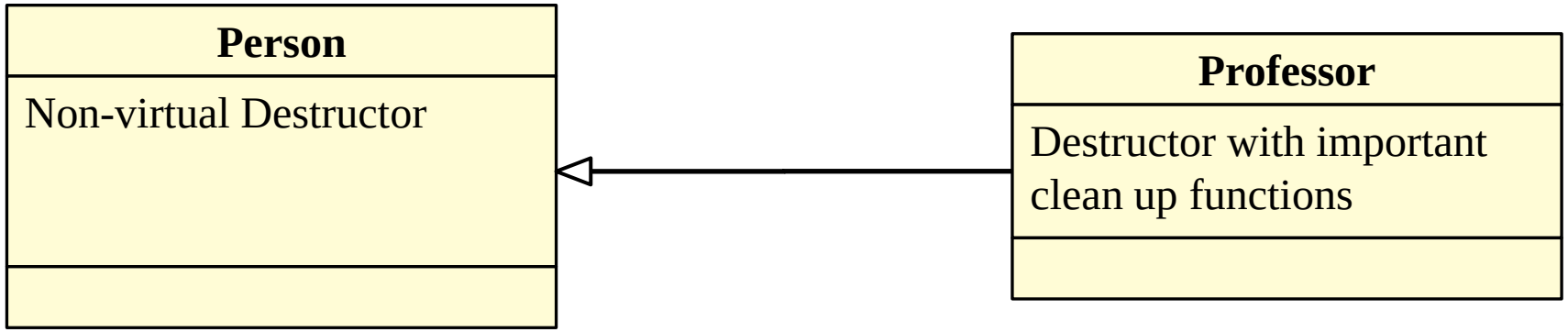

(Wenn Sie in VS Code Ihr C++ Programm kompilieren, wird der Quellcode (.cpp-Dateien) durch den Compiler in eine ausführbare Datei übersetzt.

In der Standardkonfiguration wird diese ausführbare Datei oft **outDebug.exe** genannt, wenn Sie auf einem Windows-System arbeiten.)

# Destruktoren und Vererbung

**Regel: Sobald von einer Klasse geerbt wird muss Destruktor in Basis Klasse virtuell sein!**

- Sonst:




- Destruktor von Person wird nicht überschrieben und das clean-up funktioniert nicht so wie gehofft → Vermeidbare Fehlerquelle!
- Am einfachsten: **Alle Destruktoren virtuell!**

```
virtual ~Person(){}
```

# Recap: Alle Säulen werden genutzt!


# Aufgaben: Polymorphismus, Places

1) Implementieren Sie eine Klasse Place.
   - Ein Ort hat einen `name` und eine (integer) x- und y-Koordinate. Beides soll zum Erstellungszeitpunkt gesetzt werden können.
   - Orte können besucht werden (`visit()`). Dabei geben sie Ihren Namen und ihre Position aus
2) Neben normalen Orten gibt es noch `Sights` (Sehenswürdigkeiten, z.B. „Eifel Tower“). Diese erhalten zum Erstellungszeitpunkt noch einen String, der beschreibt, was es zu sehen gibt (z.B. „All of Paris“).
   - Beispielausgabe: „Eifel tower (5, 10). Here you can see: All of Paris.“
3) Außerdem existieren noch `Toilets`.
   - Diese erhalten analog zu den Sehenswürdigkeiten einen String mit einem `Smell` („z.B. Sulfur“).
   - Beispielausgabe: „The devils toilet (23, 15). Here it smells like: Sulfur“
4) Ihre `main` soll einen Typ von jeder Klasse in einem `vector` erstellen und über alle drei Orte iterieren und dabei jeden besuchen.

**Implementieren Sie alle Klassen in .hpp und .cpp Files!**

```
17    for (Place* place : myPlaces)
18    {
19        place->visit();
20    }
```