Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner M. Dietl

Software Component Technology

Exercises 1: Introduction

Wintersemester 05/06

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1. Introduction for the Exercises

- Content
- Examination
- Contact
- Simulating OO in C – Alternatives

Content

- Exercises present…
- … more examples
- … other programming languages
- … research papers
- Exercises open for questions regarding the lecture

Examination

- 90 minute written exam in the exam period
- Content of lectures and exercises
- Type: 2V 1U
- Credits for Students: 5
- Credits for PhDs: 4 (with exam) or 3 (without exam)

Contact

- E-Mail for this course:

  `koop@inf.ethz.ch`

- Homepage:

  `http://sct.inf.ethz.ch/…`
  `…teaching/ws2005/koop/index.html`

- I am in room RZ F5

Systematic Look at Simulating OO in C

- Subtyping
- Attribute and Method Access
- Inheritance
- Dynamic Binding
- Overriding
- Calling Overridden Methods
- Multiple Subtyping

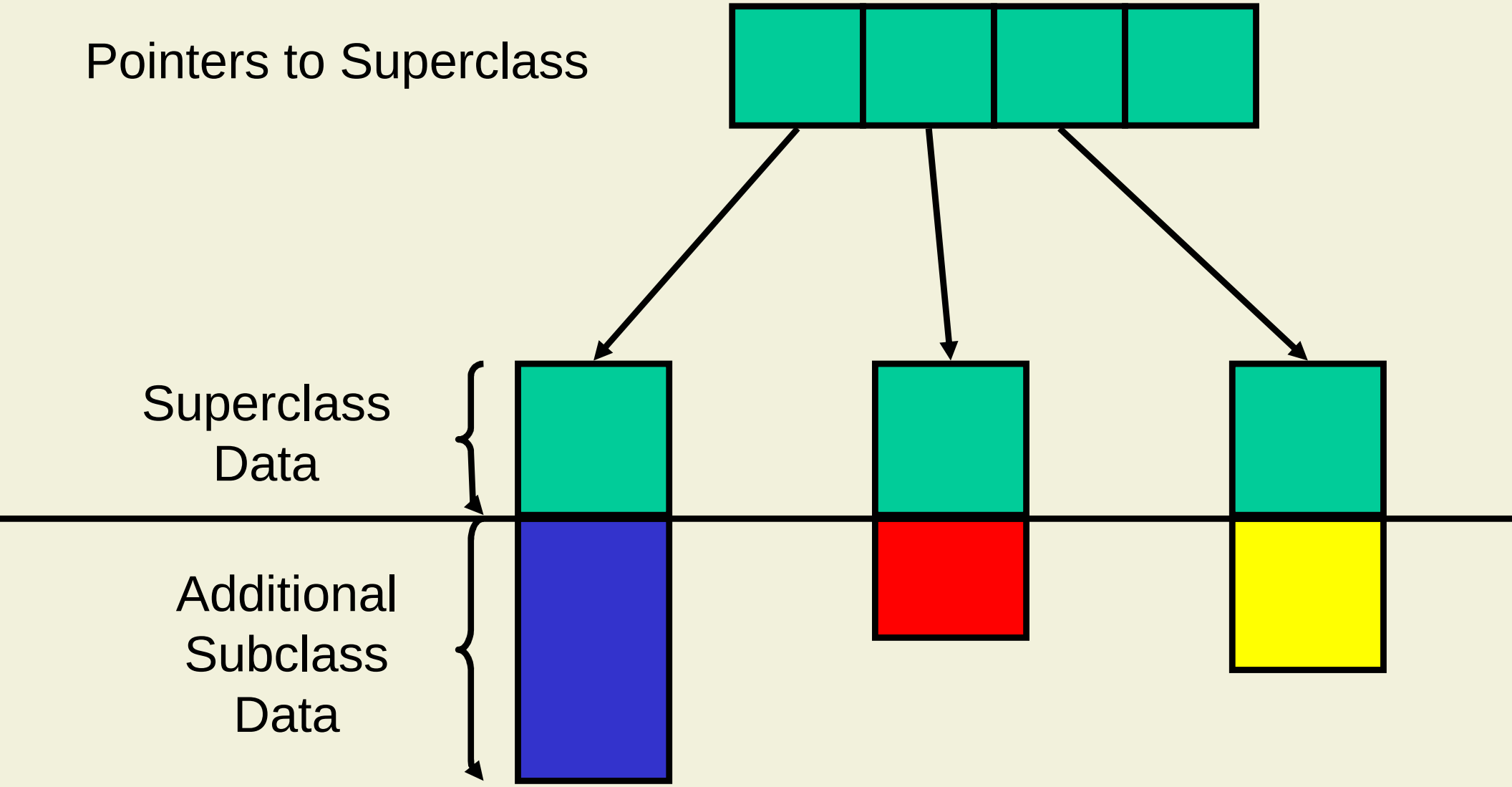
```
public class Student
   extends Person {

   public Student(String name,
                  int reg ) {
       super( name );
       this.reg_num = reg;
   }

   public void print() {
       super.print();
       System.out.println(
           "No: " + reg_num);
   }

   protected int reg_num;
}
```

Simulating OO in C – Lecture Example

- Use a `struct` for each class and function pointers for methods
- Each subclass needs to exactly copy the base class attribute and method layout
- We can change function parameters to the subtype structure
- Dynamic method binding works, because whichever method is at the named memory location will be executed

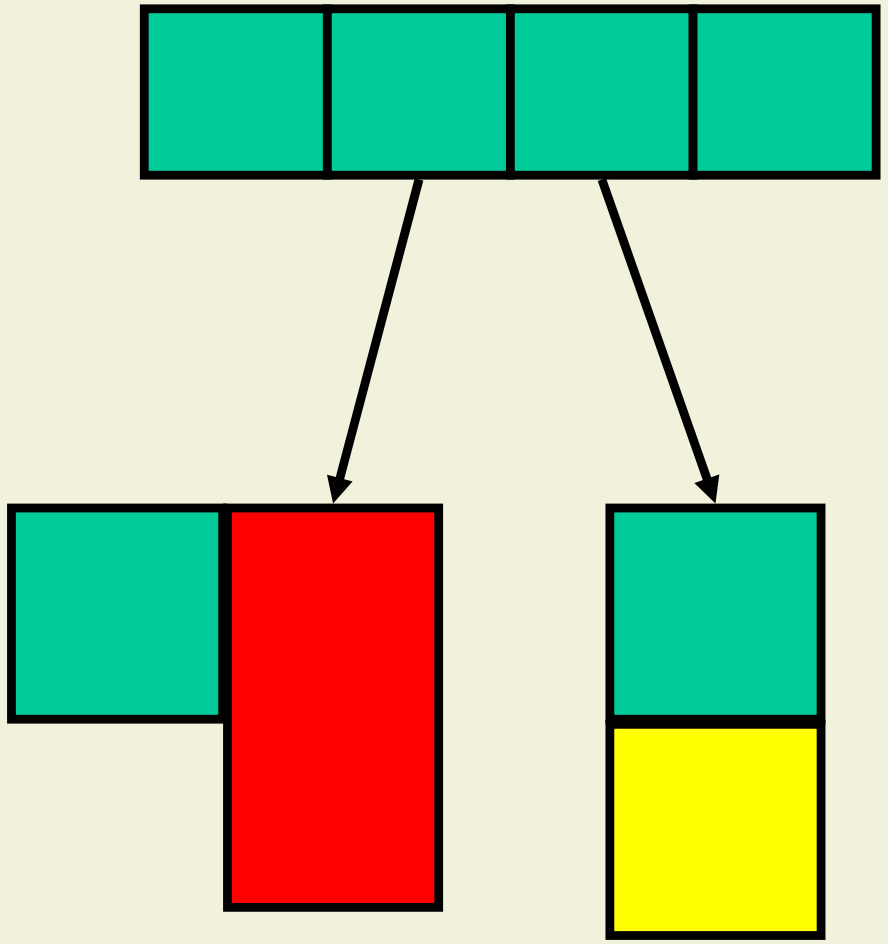
```
struct sPerson {
  String *name;

  void    (*print)( Person* );
  String* (*lastName)( Person* );
};
```

How does this work?



How can this go wrong?

- Lack of Type Safety!
- If there is an incorrect structure in the array (e.g. order of elements was changed)
- It is interpreted as though it is of the superclass type
- Lots of trouble possible: access to invalid data, arbitrary function calls



Example of incorrect behavior

- We have a new Animal structure:

```
struct sAnimal {
  void (*print)(Animal*);
  void (*bite)(Animal*);
  String *name;
};
```

- It also has a print method
- Therefore we try to put it into the Person array and pass it to the printAll method
- The bite method is called!
- A lot worse could happen!

```
printall result:
---- 0 ----
Name: Max Gans
No: 777
---- 1 -----
Name: Karl Heinrich Huber
---- 2 ----
Name: Meister Klug
No of PhDs: 2
---- 3 ----
Animal Fifi bites your arm
   off!
Bye.
```

Example of wrong behavior:

What we think to have:

```
struct sPerson {
 String *name;
 void (*print)(Person*);
 String* (*lastName)(Person*);
};
```

What there really is:

```
struct sAnimal {
 void (*print)(Animal*);
 void (*bite)(Animal*);
 String *name;
};
```

Why didn't the compiler catch this?

```
d = ProfC("Meister Klug", 2);
a = AnimalC("Fifi");
test[2] = (Person*) d;
test[3] = (Person*) a;
```

Note: It's not simply that print is the second item in Person, therefore we call the second item in Animal!
The data layout in memory is important.

Analysis of this Solution

- Subtyping only by imposing the same data layout
- Member access via struct
- Inheritance only by explicitly calling other functions
- Dynamic Binding because of same data layout
- Multiple subtyping not possible

- Very error prone and tedious to write

OO in C – Alternative Implementation

- Basic idea: reuse the struct of the superclass
- Person is the same as in the last example
- Child classes simply use the superclass structure as the first data member:

```
struct sStudent {
  Person pers;

  int reg_num;
};
```

- Easy for multiple levels of single subtyping

How do we access the components?

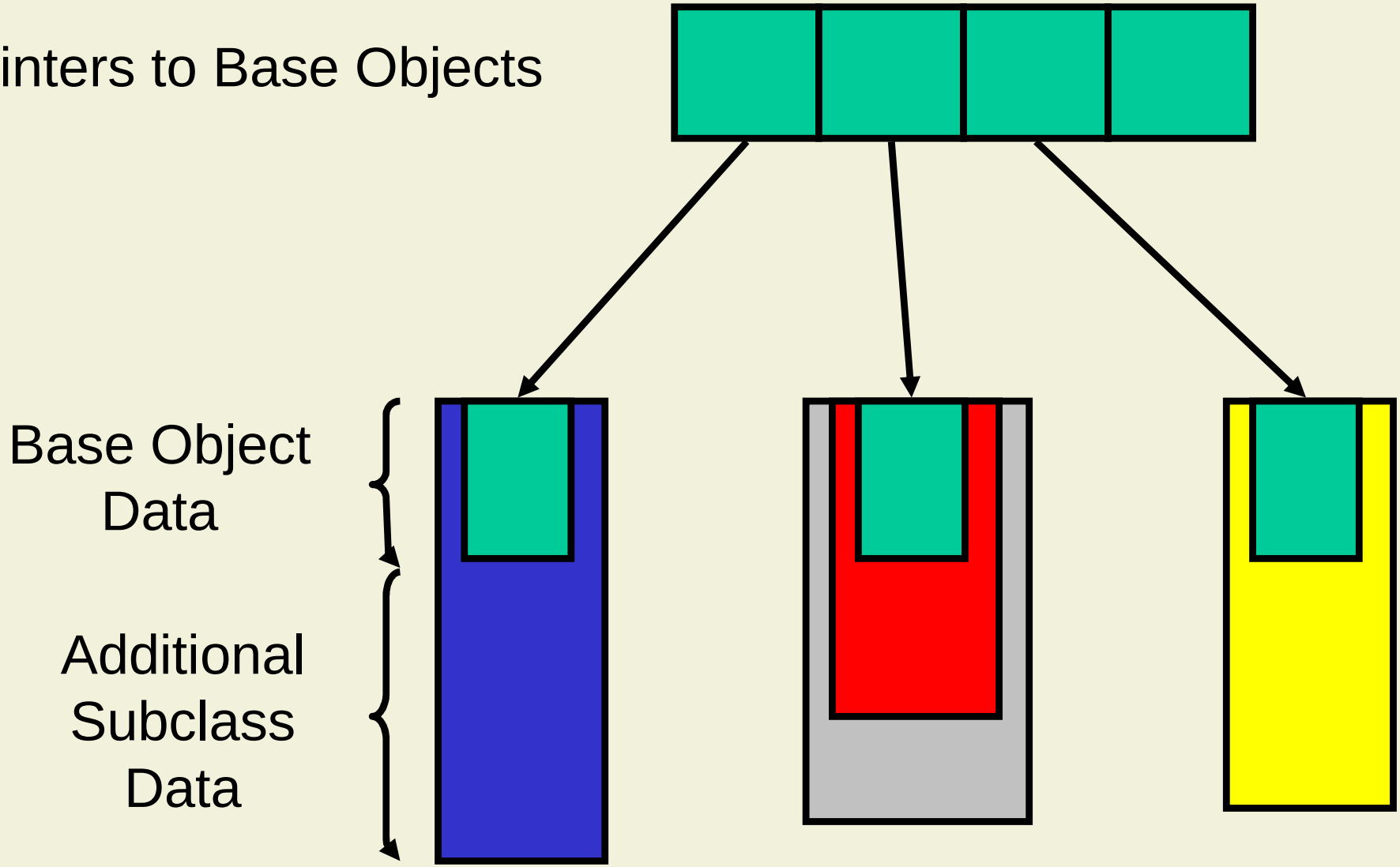
- Nice and clean with component access:

```
initStudent( &(this->stud), n, reg);
this->stud.pers.print =
     (void (*)(Person *)) printGraduate;
```

- Flatten the hierarchy by a cast:

```
initStudent( (Student *) this, n, reg );
((Person *) this)->print =
     (void (*)(Person *)) printBachelor;
```

How does this work?



Analysis of this Solution

- Subtyping still only by imposing the same data layout
- But by reusing the existing structures we do not have to type as much and are less likely to make errors
- Member access via struct, either with casts or subcomponent access
- Inheritance only by explicitly calling other functions
- Dynamic Binding because of same data layout
- Multiple subtyping not possible
- Basic lack of type safety, structures are cast around as programmer thinks is best

Other Solutions

- The mapping of OO into C could be spun even further
- The first C++ and Objective-C compilers created C-code instead of binary code
- The Eiffel compiler also creates C code

- For some historical comments also see OOSE 34.4 pages 1106 ff.

Java Solution Revisited

- Java provides all the language constructs needed for Object-Oriented programming
- Keyword `class` introduces a new data structure consisting of attributes and methods
- Classes can inherit from each other:

```
public class Graduate extends Student {
```

- Similar to including the structure of the superclass in the new class
- Safe types with subtyping, no need for insecure casting

Java Solution Revisited

- Polymorphic method calls
- Encapsulation
- Information Hiding
- Many other advantages that we will hear of later

Questions?