Inheritance in C++

Computer Vision 2018
Marco Carraro
Pietro Zanuttigh
Stefano Ghidoni

Concepts & definitions from Savitch, "Absolute C++"
Some material cplusplus.com and Eunsuk Kang's online slides

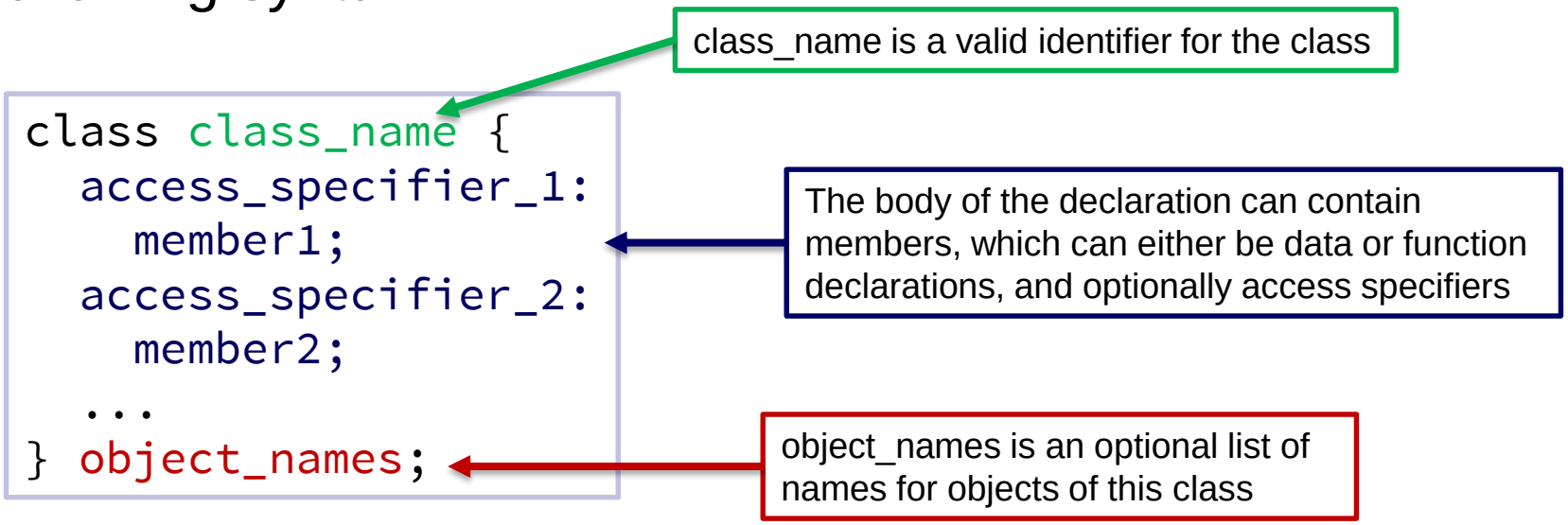
Agenda

1. Brief recap of classes
2. Inheritance basics
3. Method overriding
4. Polimorphism & virtual functions

- A few prerequisites
  - Classes
  - Class member functions
  - Public & private members

1. Recap: Classes

- Classes are an expanded concept of data structures
- Their members can be both data and functions
- An object is an instantiation of a class. In term of variables
  - the class is the type
  - the object is the variable
- Classes are defined using the keyword "*class*" with the following syntax:



Recap : Access Control

- An access specifier is one of the following three keywords: *private*, *public* or *protected*.
- These specifiers modify the access rights for the members that follow them

1. *private* members of a class are accessible only from within other members of the same class
2. *protected* members are accessible from other members of the same class and also from members of their derived classes
3. *public* members are accessible from anywhere where the object is visible

- By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other access specifier has private access automatically.

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

```
rect.set_values (3,4);
myarea = rect.area();
w = rect.width;
```

Recap: Constructors

- Class members need to be initialized before being used
- A class can include a special function called its constructor, which is automatically called whenever a new object of this class is created
  - It allows the class to initialize member variables or allocate storage
- The constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type
- Constructors cannot be called explicitly as if they were regular functions
  - They are only executed once, when a new object of that class is created

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};
Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}
```

```
int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Same results:

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

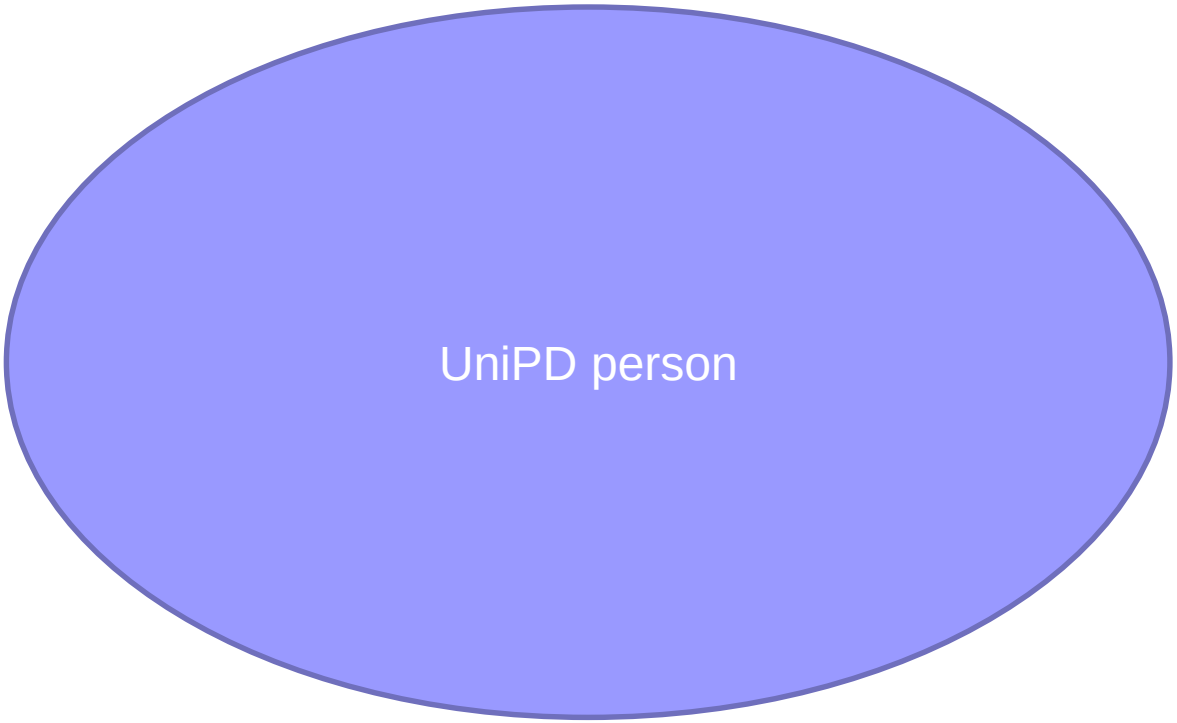
Recap: Pointers, Members, Address

expression	***can be read as***
*x	pointed to by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed to by x
(*x).y	member y of object pointed to by x (equivalent to the previous one)
x[0]	first object pointed to by x
x[1]	second object pointed to by x
x[n]	(n+1)th object pointed to by x

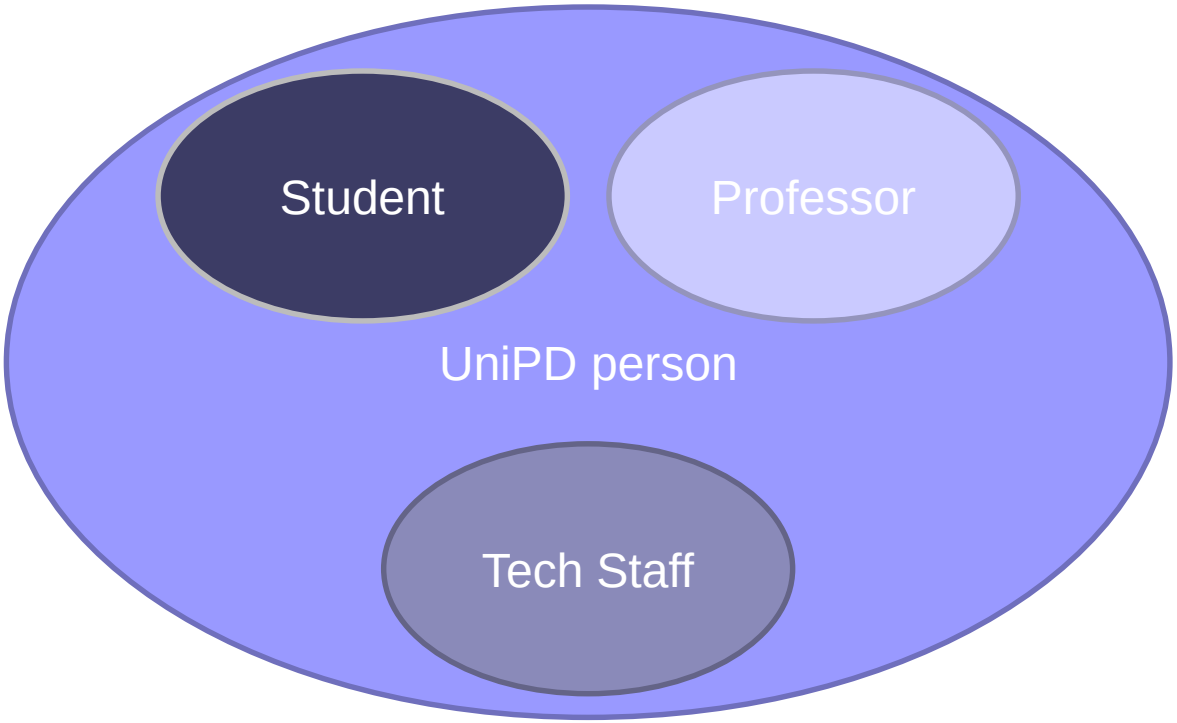
2. Inheritance Basics: Hierarchy

- Sometimes, a natural hierarchy exists in a data structure
- Person @ UniPD
  - Student
  - Professor
  - Technical staff member
- Geometric transform
  - Affine
    - Rotation
    - Translation
- Geometric shape
  - Quadrilateral
    - Rectangle
      - Square
- Image Filter (→LAB3)
  - Bilateral
  - Gaussian
  - Median

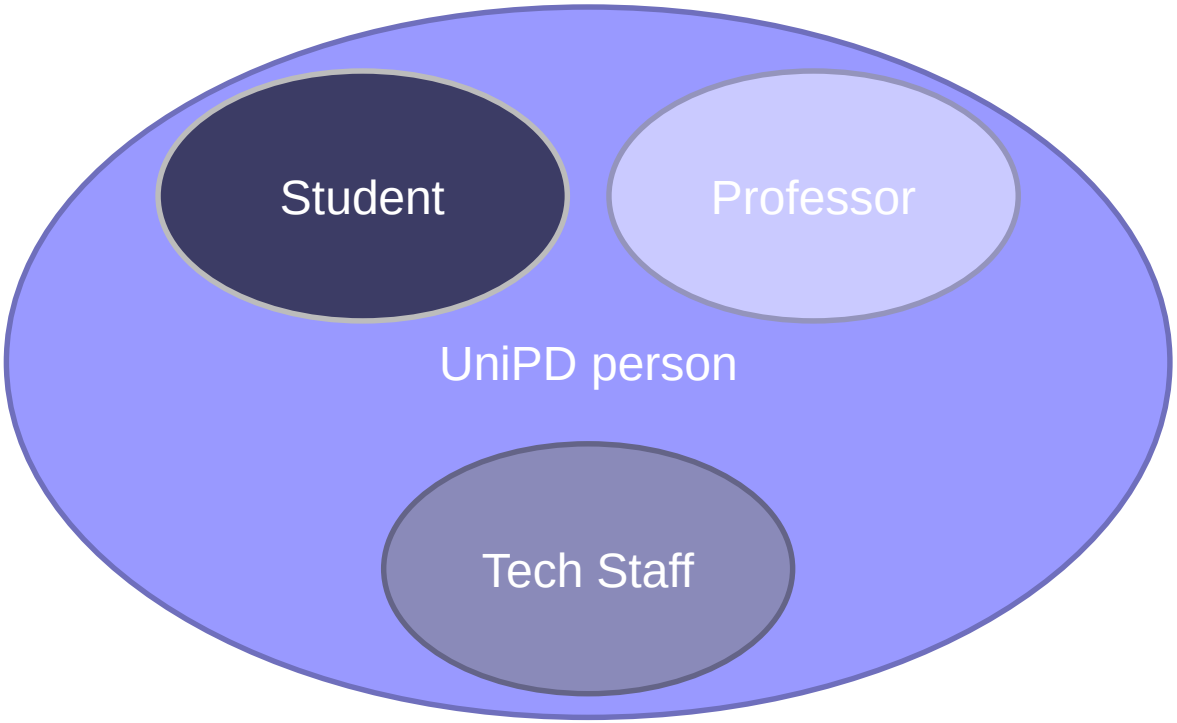
Visualization



Visualization

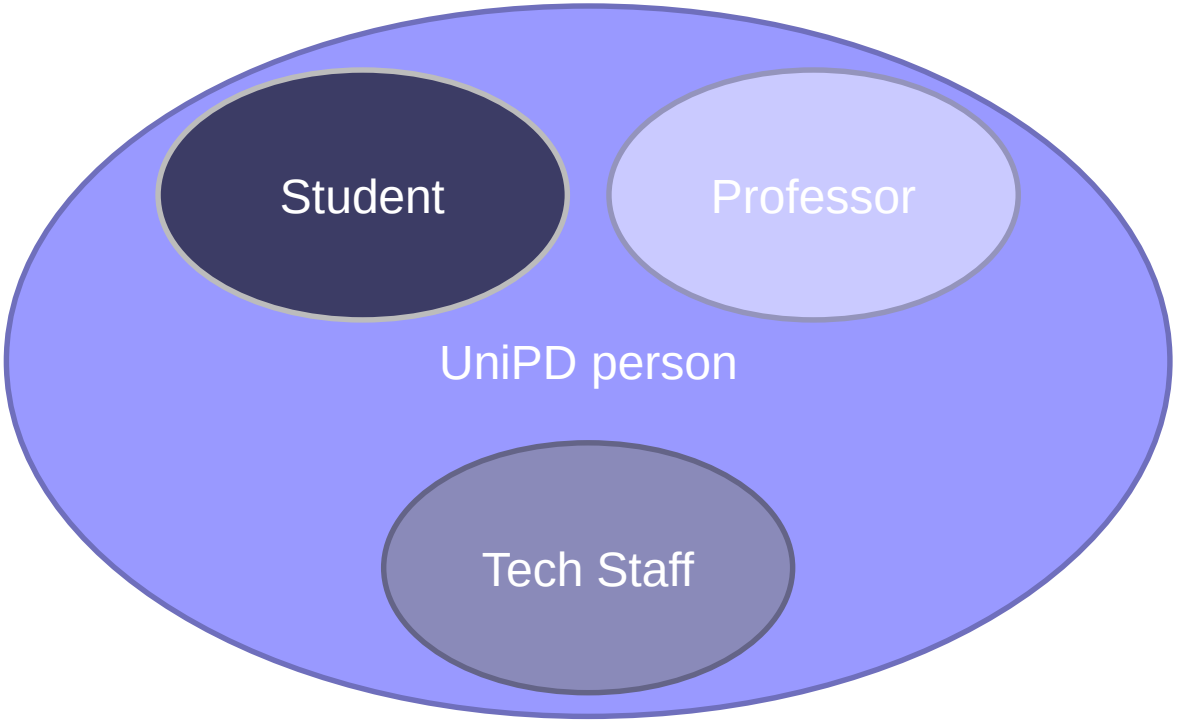


Visualization



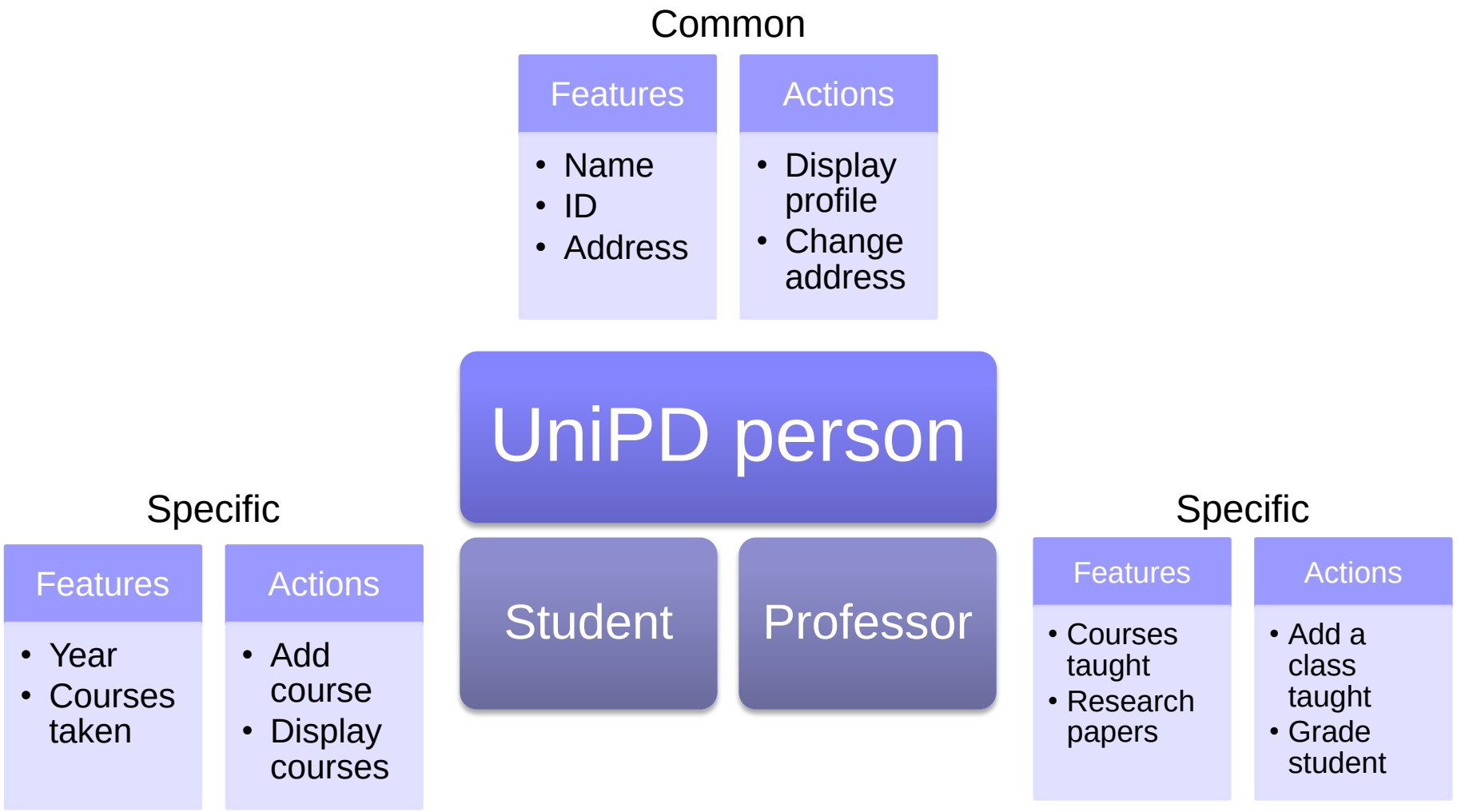
Student, Professor and Tech Staff are subtypes of UniPD person

Visualization

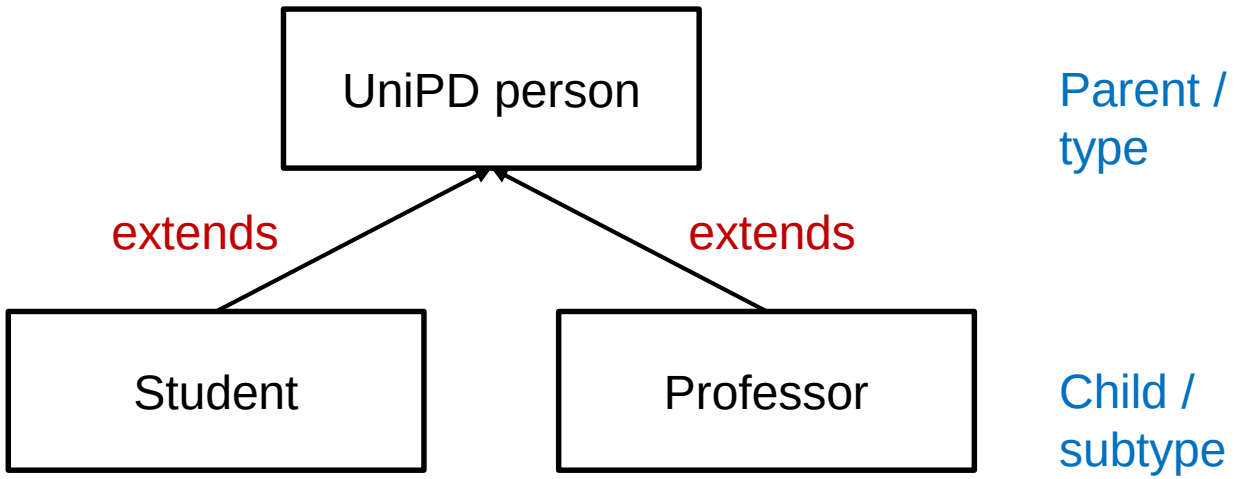


- Which characteristics do they share?
- What does specialize each type?

Common and Specific Features



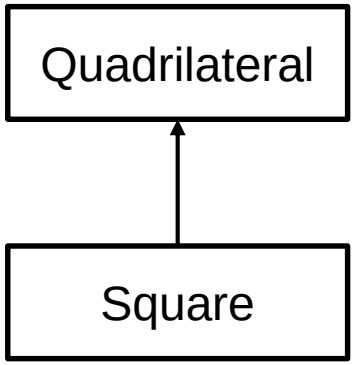
Inheritance Basics



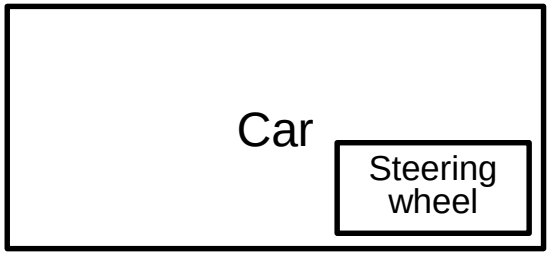
Inheritance is the process by which a new class – known as a derived class – is created from another class, called the base class

"Is a" vs "has a"

- Inheritance is based on the is a paradigm
- A square *is a* quadrilateral



- A car *has a* steering wheel



Access Control

- Public
  - Accessible from outside
- Protected
  - Accessible from inside the class and its descendants
- Private
  - Accessible only inside the class – descendants are excluded
    - otherwise, deriving a class would be enough to break data protection

UnipdPerson (*base*) Class

```
#include <string>

class UnipdPerson
{
public:
  UnipdPerson(int id, const std::string& name,
              const std::string& address);

  void DisplayProfile(void);
  void ChangeAddress(const std::string& newAddress);

protected:
  int m_id;
  std::string m_name;
  std::string m_address;
};
```

Constructor

```
// in unipd_person.cpp
UnipdPerson::UnipdPerson(int id, const std::string& name,
const std::string& address)
{
    m_id = id;
    m_name = name;
    m_address = address;
}
```

Student (*derived*) Class

```
#include <string>
#include "unipd_person.h"

class Student : public UnipdPerson
{
public:
    Student(int id, const std::string& name,
            const std::string& address, int year);
    void DisplayProfile(void);
    void AddCourse(const std::string& new_course);

private:
    int m_year;
    std::vector<Course> m_courses;
};
```

Derived Class Constructor

```
// in student.cpp
Student::Student (int id, std::string name,
                  std::string address, int year) :
                        UnipdPerson(id, name, address)
{
   m_year = year;
}
```

Constructor of the base class
(It is called first)

MANDATORY!!

- m_id
- m_name
- m_address
- m_year

Initialized in inherited constructor from the base class

Specific of constructor of the student class

```
// in unipd_person.cpp
UnipdPerson::UnipdPerson(int id, const std::string& name,
                         const std::string& address)
{
    this->m_id = id;
    this->m_name = name;
    this->m_address = address;
}
```

Objects are constructed from the bottom up (base before member and member before derived) and destroyed top-down (derived before member and member before base);

Inheritance Types

- Public
- Protected
- Private

```
class SuperPippo : public Pippo
{
  . . . .
};
```

```
class Pippo
{
public:
    int p1;
    int p2;

private:
    int p3;
};
```

Base member type	Inheritance type: Public	Inheritance type: Protected	Inheritance type: Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	Not accessible	Not accessible

2. Method Overriding

DisplayProfile is defined inside both UnipdPerson and Student

```
class UnipdPerson
{
public:
    UnipdPerson(int id, std::string name,
std::string address);

    void DisplayProfile(void);
    void ChangeAddress(std::string newAddress);

private:
    int m_id;
    std::string m_name;
    std::string m_address;
};
```

```
class Student : public UnipdPerson
{
public:
    Student(int id, std::string name,
            std::string address, int year);
    void DisplayProfile(void);
    void AddCourse(std::string new_course);

private:
    int m_year;
    std::vector<std::string> m_courses;
};
```

DisplayProfile Methods

```
void UnipdPerson::DisplayProfile(void)
{
    cout << "ID: " << m_id << "\nName: " << m_name <<
            "\nAddress: " << m_address << "\n";
}
```

```
void Student::DisplayProfile(void)
{
    cout << "ID: " << m_id << "\nName: " << m_name <<
            "\nAddress: " << m_address << "\nYear: " <<
            m_year << "\n";

    cout << "Courses:\n";

    for (int i = 0; i < m_courses.size(); ++i)
        cout << m_courses[i] << "\n";
}
```

Which one will be called

Method Overriding

- Defining a method inside a derived class hides the definition in the base class
  - Different from function overloading
- Methods that are not redefined are available in derived classes
- Not all methods are inherited – must be defined if needed
  - Constructors
  - Destructors

Polimorphism and Virtual Functions

Recall the *is a* paradigm:

- An object of a derived class has more than one type
  - A Student *is a* UnipdPerson also in C++ code
- You can assign an object of class student to a UnipdPerson variable
  - Not vice-versa

Actual vs Declared Type

```
UnipdPerson *Alice = new UnipdPerson(63, "Alice", "10 Narrow Street");
```

```
UnipdPerson *Bob = new Student(91, "Bob", "100 Large Avenue", 2);
```

Upcasting

```
Student *John = new Student(44, "John", "50 Huge Highway", 1);
```

- What types are Alice and Bob?
  - What are the declared types?
  - What are the actual types?
- What if we call:
  - Alice->DisplayProfile() `UnipdPerson::DisplayProfile()`
  - Bob->DisplayProfile() `UnipdPerson::DisplayProfile()`
  - John->DisplayProfile() `Student::DisplayProfile()`

Inherited Methods

- The declared type selects the method to be called
  - Often, an undesired feature
- Class hierarchies are often used to store a collection of objects with the same base class
  - Quite a common case

Late/Dynamic Binding

- Wait until runtime to determine the implementation of a function
- Tool (C++): virtual functions
- The right implementation is got from the object instance
- Efficiency concerns
  - Late binding costs CPU at runtime
  - C++ gives the programmer control over this aspect

3. Virtual functions

Declaring the overridden function as virtual in the base class changes the behavior

```
class UnipdPerson
{
public:
    UnipdPerson(int id, std::string name, std::string
address);

    virtual void DisplayProfile(void);
    void ChangeAddress(std::string newAddress);

//…
};
```

Overriding vs redefining

- A virtual function redefined in a derived class is said to be overridden
- A non-virtual function redefined in a derived class is said to be redefined
- In practice, "*overriding*" is often used in both cases
- Differentiating is nevertheless meaningful, as the compiler treats the two cases differently
- The virtual property is inherited!

Overriding vs Redefining

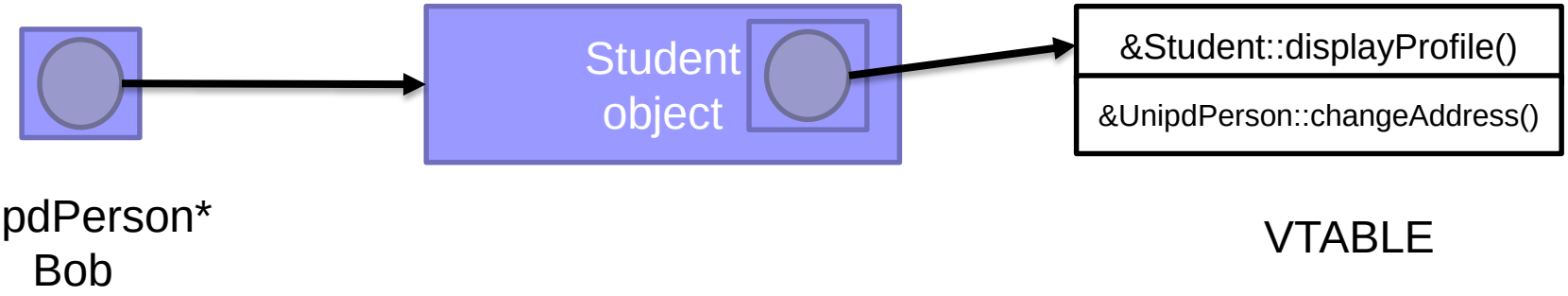
What goes on under the hood?

Overriding vs Redefining

What goes on under the hood?

VIRTUAL TABLE

- Stores pointers to all virtual functions
- Created per each class
- Lookup during the function call



Actual vs Declared Type

(Virtual Function)

```
UnipdPerson *Alice = new UnipdPerson(63, "Alice", "10 Narrow Street");

UnipdPerson *Bob = new Student(91, "Bob", "100 Large Avenue", 2);
```

Upcasting

```
Student *John = new Student(44, "John", "50 Huge Highway", 1);
```

```
class UnipdPerson
{
public:
    UnipdPerson(int id, std::string name, std::string address);

    virtual void DisplayProfile(void);
    void ChangeAddress(std::string newAddress);

//…
};
```

This time the function is virtual

Two differences

- ❑ The function to be used is selected at runtime
- ❑ The derived functions is selected independently of the declaration

- What if we call:
  - Alice->DisplayProfile()
  - Bob->DisplayProfile()
  - John->DisplayProfile()

```
UnipdPerson::DisplayProfile()
Student::DisplayProfile()
Student::DisplayProfile()
```

Constructors and Destructors

- Constructors cannot be declared virtual
  - C++ needs to know what exact type an object is in order to build it
- Destructors might be declared virtual
  - This is needed e.g., when memory management in the derived class should be performed