Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger
LS 11 Algorithm Engineering

Lecture 9 • Winter 2011/12 • Dec 13

technische universität dortmund
department of computer science

Access Control

- Recall our example for structure **`point`**:

```
struct point {
    int x;
    int y;
};
```

- We ensured the validity of the data (x ∈ [0..1919] and y ∈ [0..1079]) by using constructors, an assign member function etc.
- However, we can still write code like this:

```
point p;
p.x = -5;
```

Code that depends on the validity of data might not work on such points!

Access Specifications

- Access specifications describe the permitted access to a data member or member function.
- `private`
  - Access is only allowed from within a member function, not from outside (e.g. the main program)
  - Private members are only visible inside the structure
- `public`
  - Access is allowed from everywhere
  - Public members are also visible from outside the structure

Example: Points with access control

```
struct point {
private: // everything from here on is private
    int x;
    int y;
public: // everything from here on is public
    // add all the old stuff here...
};
```

- Now writing code like this

```
point p;
p.x = -5;
```

results in an error at compile time!

Getter Member Functions

- How can we access private data members?
- One solution: Provide a "getter" member function

```
struct point {
private:
     int x;
     int y;
public:
     int get_x() const {
        return x;
     }
     ...
};
```

- The get member function returns just a copy of **x**
  → **x** cannot be changed

The const modifier for member functions

- Member functions can be declared as `const`
- For constant instances (including const references) of this structure, only such member functions may be called
- Declare all your getter member functions as `const`!

Setter Member Functions

- Similarly, we can add "setter" member functions for modifying private data members

```
struct point {
private:
    int x;
    int y;
public:
    void set_x(int new_x) {
        if(0 <= new_x && new_x < 1920)
            x = new_x;
    }
    ...
};
```

- Now we can modify **`x`** from outside the structure and still ensure its validity

Interfaces vs. Implementation

- `private` and `public` allow us to separate the interface of our structures from the actual implementation
  - interface: all publicly visible methods and data
  - implementation: everything "under the hood" that makes the custom type work. Should be kept private.
- Why should we hide the implementation?
  - We can change it later without changing any other code
  - Users of the data type (structure) do not need to worry about the actual implementation
- Example: vectors
  - We do not know about the internals of an **`std::vector`** and should not really care
  - Knowing its interface is enough for using vectors

The Keyword class

- The keyword `class` is (almost) a synonym of `struct`
- We could also write:

```
class point {
    // ...
};
```

instead of

```
struct point {
    // ...
};
```

- The only difference is the default visibility:
  - for `struct` it is `public`
  - for `class` it is `private`

Derived Classes

- Suppose we want to model employees of a company
- We could write a data structure as follows:

```
class Employee {
     string name;
     int    salary;
     // further data

public:
     Employee();
     void print();
     int get_salary();
     // further methods
};
```

Example "Employees" continued

- Suppose now our program should also handle managers
- Managers are also employees, but they also have
  a) a group of employees they manage
  b) a level of competence

Modeling managers: first try

- We copy all the data members from **`Employee`** to **`Manager`**:

```
class Manager {
    string name;
    int    salary;

    vector<Employee> group;
    int    level;
    // further data members and methods
};
```

Data members copied from **`Employee`**

New data members

- **Disadvantages:**
  - This doubles the code and we can introduce new errors
  - Whenever we change the implementation of **`Employee`**, we also have to change the implementation of **`Manager`**

Modeling managers: second try

- We use a data member of type **`Employee`**:

```
class Manager {
    Employee emp_data;

    vector<Employee> group;
    int     level;
    // further data members and methods
};
```

Data member of type **`Employee`**

- **Discussion:**
  - This is better. But still we have to write some obscure code like this:

```
int Manager::get_salary() {
    return emp_data.get_salary();
}
```

Modeling managers using inheritance

- With the features we know so far we cannot express that a manager is a special kind of employee
- We can achieve this using inheritance:

```
class Manager : public Employee
{
    vector<Employee> group;
    int     level;
    // further data members and methods
};
```

`Manager` inherits all data members and methods from **`Employee`**

- This declaration expresses that a **`Manager`** is an **`Employee`** with some additional data

Inheritance

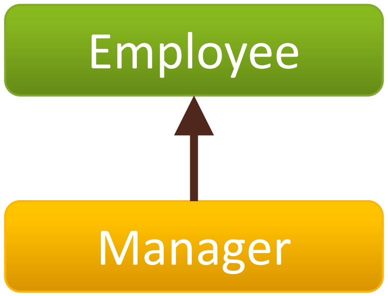
- Given a declaration like this

```
class Manager : public Employee {
    // ...
};
```

  we say that

  - **Manager** is derived from **Employee**
  - **Employee** is a base class of **Manager**

- **Manager** is said to inherit from its base class:



Why is inheritance useful?

- Given this declaration of class Manager

```
class Manager : public Employee {
    // ...
};
```

we can use a Manager wherever an Employee is expected
→ Polymorphism

```
Employee bill;
Manager adam;

Employee &emp_one = bill;
Employee &emp_two = adam;

emp_one.print();
emp_two.print();
```

Calling inherited methods

- Recall:

```
class Employee {
    // ...
    void print();
    // ...
};
```

- What happens in this situation?

```
Manager adam;
adam.print();
```

- **`adam`** is a **`Manager`**, hence also an **`Employee`**
  $\rightarrow$ the **`print()`** method of **`Employee`** is invoked
- But how can we (also) print the additional data of **`adam`**?

Calling inherited methods

- **First solution:** We could add a second method for printing:

```
class Manager : public Employee {
    // ...
    void print_manager_data();
```

  and then print a manager like this:

```
adam.print();
adam.print_manager_data();
```

- **Disadvantages:**
  - This is again error-prone. What if we forget that **`adam`** is a manager?

Redefining Inherited Methods

- **Second solution:** We can redefine the `print()` method:

```
class Manager : public Employee {
    vector<Employee> group;
    int     level;
public:
    void print();
};

void Manager::print()
{
    Employee::print();
    cout << "level = " << level << '\n';
    // further output ...
}
```

Redefine the **`print()`** method of **`Employee`**

Implementation outside of the class declaration

Call **`print()`** method of the base class

Redefining Inherited Methods

- Given the following declarations:

```
Employee bill;
Manager adam;
```

- **`bill.print()`**
  invokes the **`print()`** method of **`Employee`**
- **`adam.print()`**
  invokes the redefined **`print()`** method of **`Manager`** (whose implementation will then also invoke the **`print()`** method of **`Employee`**)

Derived Classes and Constructors

- Given that **`Employee`** has the following constructor:

```
Employee::Employee(string n, int s)
        : name(n), salary(s) { }
```

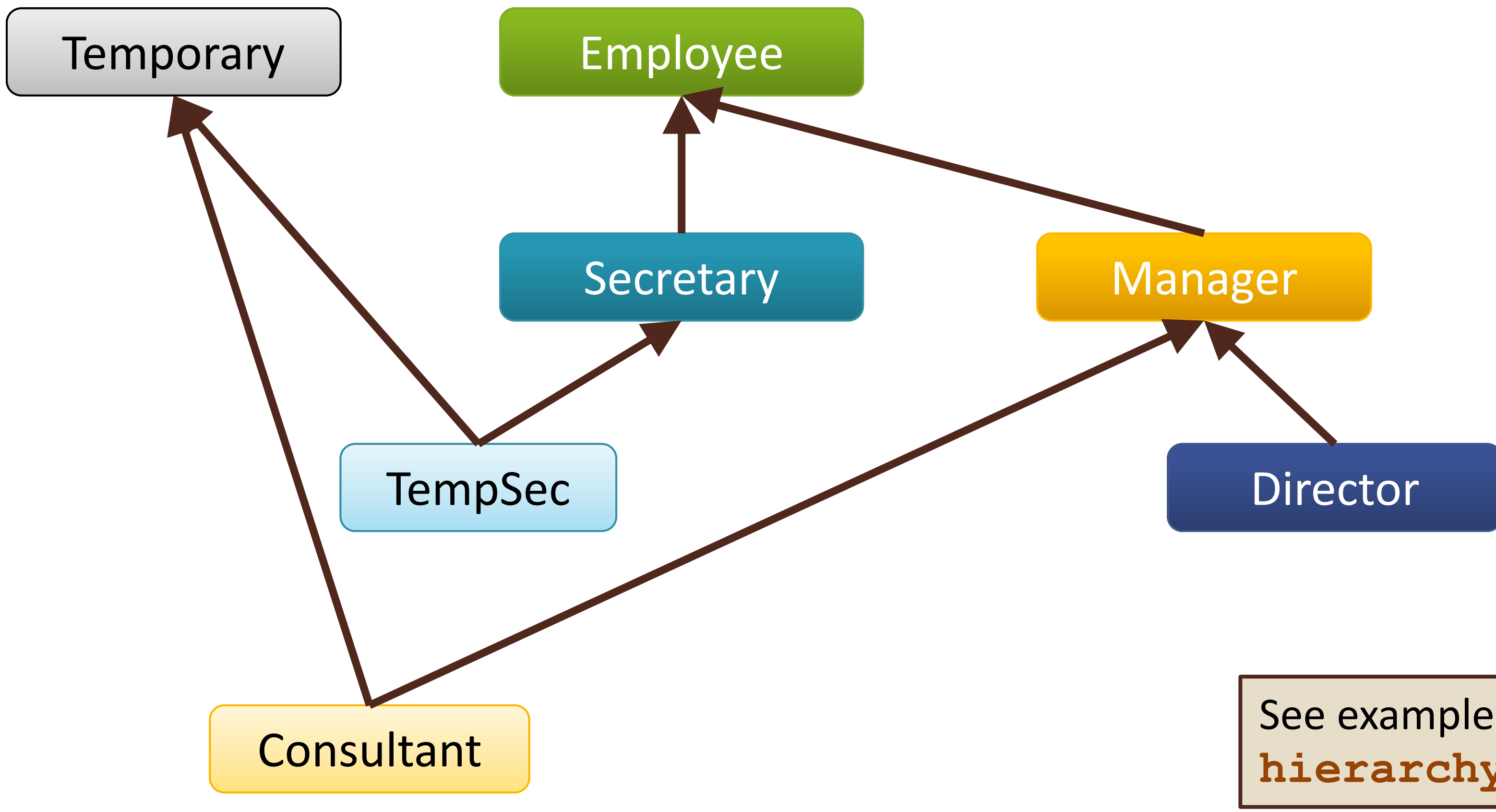
- We define the constructor of **`Manager`** as follows:

```
Manager::Manager(string n, int s, int l)
        : Employee(n,s), level(l) { }
```

Calls the constructor of **`Employee`**

- Order of construction:
  1. the base class
  2. the data members
  3. the derived class itself (the code in the constructor)
- Objects are destroyed in the opposite order

Class Hierarchies



See example
`hierarchy.cpp`

- Classes can also inherit from several base classes
  - We will not make use of this in this course!

Preparations for next week

- Constructors, destructors, and assignment
- Pointers
- Virtual and purely virtual functions