DTU

02393 Programming in C++

Module 8: Inheritance

Alceste Scalas <alcsc@dtu.dk>, **Giovanni Meroni** <giom@dtu.dk>

Slides based on previous versions by Andrea Vandin, Alberto Lluch Lafuente, Sebastian Mödersheim

25 October 2022

Course plan

Module no.	Date	Topic	Book chapter*
0 and 1	30.08	Welcome & C++ Overview	1
2	06.09	Basic C++ and Data Types	2.1, 2.3 - 2.5, 11.1, 11.3
3	13.09	Enumerations and Structures & *LAB DAY*	1.5
4	20.09	Memory Management	12.1, 11.2, 11.3
5	27.09	Libraries and Interfaces	2.2, 2.6 - 2.8, 3.1 - 3.3, 4.1 - 4.3
6	04.10	Classes and Objects	5.1, 6.1, 11.2, 12.1, 12.4, 12.7
7	11.10	Templates	5.1, 14.2
		Autumn break	
8	25.10	Inheritance	19.1 - 19.3
9	01.11	*LAB DAY*	*Previous exams*
10	08.11	Recursive Programming	7
11	15.11	Linked Lists	12.2, 12.3
12	22.11	Trees	16.1 - 16.4
13	29.11	Conclusion & *LAB DAY*	*Exam preparation*
	22.12	**Exam (held physically, all aids allowed)**	

* Recall that the book uses some ad-hoc libraries (e.g., for vectors). We will use standard libraries

Outline

A recap of the previous lectures

- **The structure of a C++ program**
  - `#include` and `#define` directives, the `main` function, user-defined functions and methods (including constructors, destructors, operators), `template`s
- **Simple input/output**
  - `cin`, `cout`
- **Variables, values, and types**
  - `string`, `int`, `double`, `float`, arrays (statically and dynamically allocated), pointers, `enum`, `struct`, `vector`, `ifstream`, `ofstream`, `class`, `this`
- **Expressions**
  - Some numeric and boolean operators and math functions, conditional expressions
- **Statements**
  - `if`, `while`, `for`, `switch`

Recap: OOP in C++

- A `class` is similar to a `struct`, but its members can be both **variables** and **methods**
  - a method is bit like a function
- An **object** is an instance of a class
- Class members can be `public` or `private`
  - users of a class can only access `public` members (**data encapsulation**)
- Classes can have some **special methods**:
  - **constructor**: called when an object is created
    - either statically, or dynamically using `new`
  - **destructor**: called when an object is destroyed
    - either statically by exiting a scope, or dynamically using `delete`
  - **assignment**: one can customise the behaviour of operator `=`
    - e.g., when the class internally uses dynamic allocation

Recap: Abstract Data Types

Use C++ encapsulation to write code that **abstracts from implementation details**

- Specify allowed operations on an ADT, by making them `public`
- Hide everything else, by making it `private`
- Instances of an ADT can only be **constructed** and **used** via `public` operations

Programs that use a well-designed ADT **do not need to be changed** when the ADT's (`private`) implementation details are changed

We can use `template`s to make our code **generic and reusable** (e.g., for containers)

Inheritance: from subtypes to subclasses

A **subtyping relation** says: every instance of a subtype **is (also) an** instance of a supertype

- in arithmetic, every **integer number is a real number**
- in geometry, every **square is a rectangle**
- in a program for managing salaries, every `HourlyEmployee` **is an** `Employee`
- . . .

The supertype supports **general operations**; the subtype may have **specialised operations**

Inheritance: from subtypes to subclasses

A **subtyping relation** says: every instance of a subtype **is (also) an** instance of a supertype

- in arithmetic, every **integer number is a real number**
- in geometry, every **square is a rectangle**
- in a program for managing salaries, every `HourlyEmployee` **is an** `Employee`
- ...

The supertype supports **general operations**; the subtype may have **specialised operations**

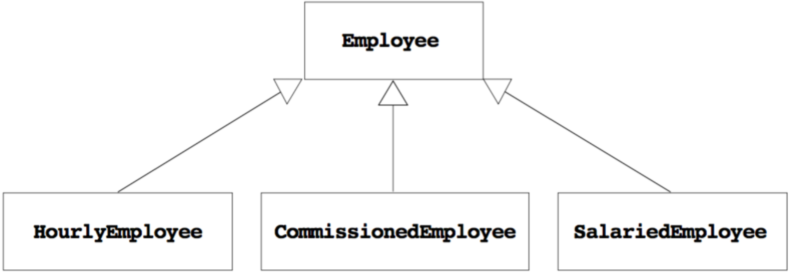
When is this useful?

- **Bottom-up perspective (generalisation)**
  - *"We have many employee classes with shared functionalities: let's group them together"*
- **Top-down perspective (specialisation)**
  - *"An `Employee` class distinguishes different kinds of employees: let's make separate classes"*

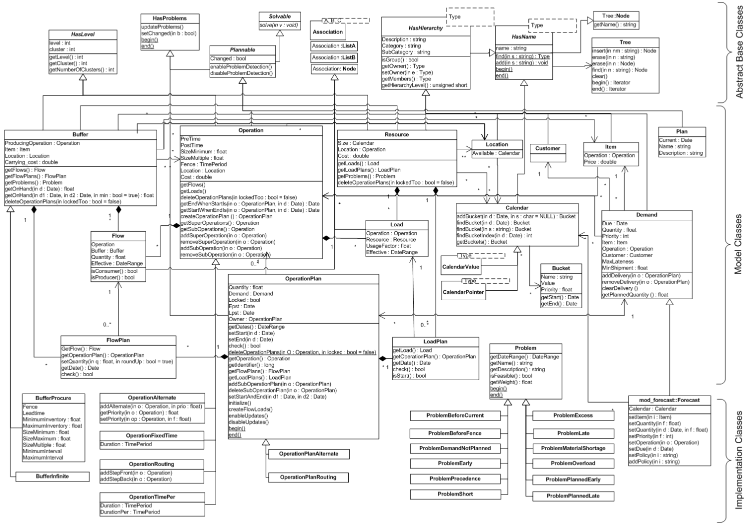
Advantages: **modularity**, **clarity**, **maintainability**

From the "is-a" relations to class diagrams

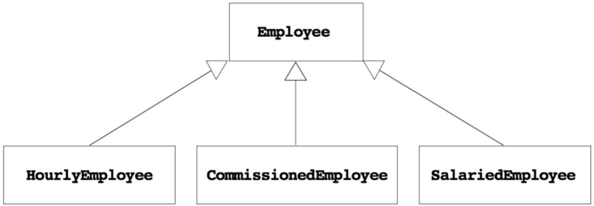
"Every `HourlyEmployee` **is an** `Employee`"
"Every `CommissionedEmployee` **is an** `Employee`"
"Every `SalariedEmployee` **is an** `Employee`"



Class diagrams in real life



From class diagrams to code (live coding!)



In C++ we implement this class diagram as:

```cpp
class Employee {
    // Interface and implementation methods for all employees
}

class HourlyEmployee : Employee {
    // Specialised code for hourly employees
}

// More code for other kinds of employee
```

Encapsulation

We can **control the access** to class fields and methods:

- `private` members are accessible by objects of the **class and no one else** (default)
- `protected` members are accessible by objects of the **class and derived classes**
- `public` members are accessible by **everyone**

Useful to **hide implementation details** and **prevent unintended use**

Inheritance: `class B : A {...}`

What is actually inherited?

- `B` inherits all `public` and `protected` member variables
- `B` does **not** inherit `private` methods of `A`
- `B` **cannot access** the `private` member variables of `A`

Inheritance: `class B : A {...}`

What is actually inherited?

- `B` inherits all `public` and `protected` member variables
- `B` does **not** inherit `private` methods of `A`
- `B` **cannot access** the `private` member variables of `A`

What happens to the interface of `A`?

- **It depends!** We can write: `class B : <modifier> A`
  where `<modifier>` is either `public`, `protected` or `private` (default)
- Details on the next slide. . .

Encapsulation and inheritance

```
class B: public A { ... }
```

- B inherits `public` members, which remain `public`
- B inherits `protected` members, which remain `protected`

Encapsulation and inheritance

```
class B: public A { ... }
```

- B inherits `public` members, which remain `public`
- B inherits `protected` members, which remain `protected`

```
class B : protected A { ... }
```

- B inherits `public` members, **which become** `protected`**!**
- B inherits `protected` members, which remain `protected`

Encapsulation and inheritance

```
class B: public A { ... }
```

- B inherits `public` members, which remain `public`
- B inherits `protected` members, which remain `protected`

```
class B : protected A { ... }
```

- B inherits `public` members, **which become** `protected`**!**
- B inherits `protected` members, which remain `protected`

```
class B : private A { ... }
```

- B inherits `public` members, **which become** `private`**!**
- B inherits `protected` members, **which become** `private`**!**

Encapsulation and inheritance (live coding)

```
class A {
public:
    int x; // accessible to everyone
protected:
    int y; // accessible to all derived classes (A, B, C, D)
private:
    int z; // accessible only to A
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A {
    // x is private
    // y is private
    // z is not accessible from D
};
```

Overriding methods

We can **override** (i.e., refine) inherited methods, so we can **specialise their code**

```
class A {
public:
    void f(); // Original method
};

class B: public A {
public:
    void f(); // Overridden method
};
```

Overriding methods

We can **override** (i.e., refine) inherited methods, so we can **specialise their code**

```
class A {
public:
    void f(); // Original method
};

class B: public A {
public:
    void f(); // Overridden method
};
```

```
void main() {
    B *b = new B();
    A *a = b;
    b->f();
    a->f();
}
```

- **Which `f()` is invoked by `b.f()`?**
- **Which `f()` is invoked by `a.f()`?**

Overriding methods

We can **override** (i.e., refine) inherited methods, so we can **specialise their code**

```
class A {
public:
    void f(); // Original method
};

class B: public A {
public:
    void f(); // Overridden method
};
```

```
void main() {
    B *b = new B();
    A *a = b;
    b->f();
    a->f();
}
```

- **Which** `f()` **is invoked by** `b.f()`**?** Answer: `B::f()`
- **Which** `f()` **is invoked by** `a.f()`**?**

Overriding methods

We can **override** (i.e., refine) inherited methods, so we can **specialise their code**

```
class A {
public:
    void f(); // Original method
};

class B: public A {
public:
    void f(); // Overridden method
};
```

```
void main() {
    B *b = new B();
    A *a = b;
    b->f();
    a->f();
}
```

- **Which `f()` is invoked by `b.f()`?** Answer: `B::f()`
- **Which `f()` is invoked by `a.f()`?** **Answer: `A::f()`!**

This is because the C++ uses (very fast) **static method dispatch** based on `a`'s type

To ensure that `B::f()` is always called for objects of class `B`, we mark `f()` as `virtual` in `A`.
Result: slower (but usually more intuitive) **dynamic method dispatch**

Refining methods (live coding)

```
class father {
public:
    void f(void) = { ... };
    virtual void g(void) = { ... };
};

class son : public father {
public:
    void f(void) = { ... };
    void g(void) = { ... };
};

int main(void){
    son *b = new son();
    father *p = b;

    b->f(); // calls son::f()
    p->f(); // calls father::f(), due to static dispatch
            // (based on p's type)

    b->g(); // calls son::g()
    p->g(); // calls son::g(), due to dynamic dispatch
}
```

Abstract classes

A class is **abstract** if it contains at least one "**pure** `virtual`" method, marked with "`= 0`"

For example:

```
class Employee {
public:
    string name(void);
    virtual double salary(void) = 0; // Pure virtual method
    ...
};

class HourlyEmployee : public Employee {
public:
    double salary(void);
};
```

An abstract class **cannot be instantiated**: it only defines an **interface** for derived classes

A derived class can only be instantiated if it **overrides all pure `virtual` methods**

Constructors and inheritance

```
class B: A { ... }
```

Constructors and inheritance can be tricky, because **constructors are not inherited!**

- `B` may need to define its own constructors
- `B`'s constructors may need to **explicitly invoke** one of `A`'s constructors

Lab

Today's lab begins now. Tasks:

- make sure C++ works on your computer, request help if it doesn't
- begin working on **Assignment 8**
- ask questions if something is unclear (including previous assignments)