02393 Programming in C++
Module 8: Inheritance

Teacher: Alberto Lluch Lafuente

Sebastian Mödersheim (slides author)

April 4, 2016

Lecture Plan

#	Date	Topic	Chapter *
1	1.2	Introduction	1
2	8.2	Basic C++	1
3	15.2	Data Types	2
4	22.2		
5	29.2	Libraries and Interfaces	3
6	7.3	Classes and Objects	4.1, 4.2 and 9.1, 9.2
7	14.3	Templates	4.1, 11.1
		Påskesferie	
8	4.4	Inheritance	14.3, 14.4, 14.5
9	11.4	Recursive Programming	5-7
10	18.4	Lists and Trees	10.5, 11, 13.1
11	25.4	Trees	13
12	2.5	Graphs	16
13	9.5	Summary	
	17.5	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g. strings and vectors).

Recap

- Generic Programming
  - ★ Templates (e.g. type-parametric functions, etc.)
  - ★ Implicit and explicit specialization
- Object Oriented Programming
  - ★ Classes and objects;
  - ★ Encapsulation (private/protected/public attributes/methods)
  - ★ Methods (constructors, destructors, etc.)
  - ★ Inheritance (today)
- OO + GP
  - ★ Class templates (e.g. containers)

Inheritance: from subtypes to subclasses

Example of subclass/subtype relations:

- Every integer is a real;
- Every square is a rectangle;
- Every `HourlyEmployee` is an `Employee`;
- etc.

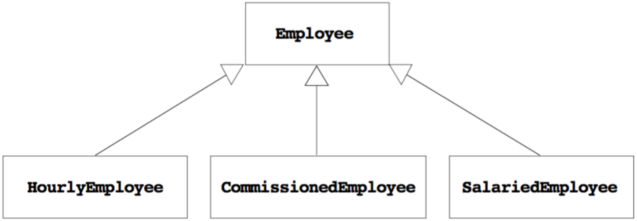
When is this useful?

- Bottom-up perspective (generalization)
  "we have classes for different kinds of Employees which share some functionalities... let us group them together."
- Top-down perspective (specialization)
  "the class of employees is full of specialized code for particular kinds of employees... let us separate them in different classes."

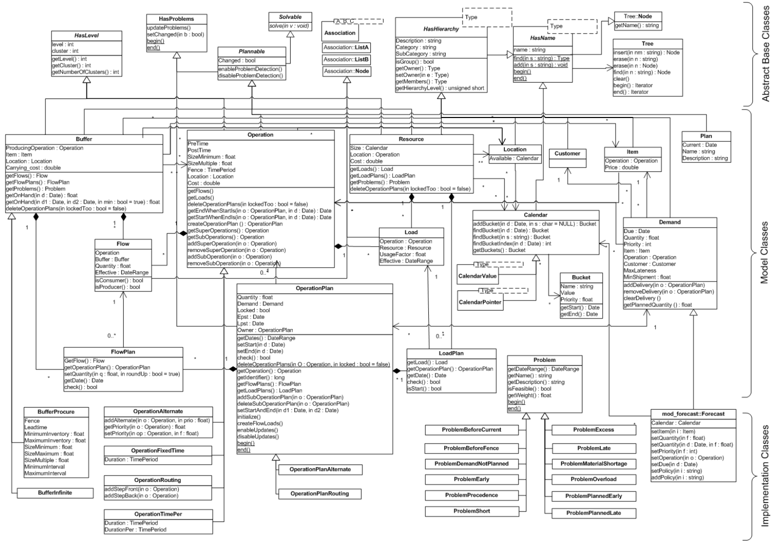
Advantages: modularity, clarity, maintanability, etc.

From "is-a" relations to class diagrams

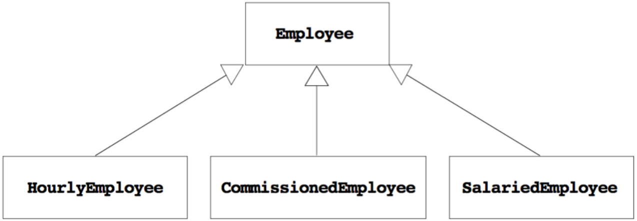
"Every `HourlyEmployee` is an `Employee`."
"Every `CommissionedEmployee` is an `Employee`."
"Every `SalariedEmployee` is an `Employee`."



Diagrams in real life



From diagrams to code



In C++ we write something like

```
class Employee {
    ...
}

class HourlyEmployee : Employee {
    ...
}

...
```

Inheritance: more

```
class B : A ...
```

What is actually inherited?

- `B` inherits (almost) all `public` and `protected` members.
- `B` does not inherit `private methods` of `A`.

What happens to the interface?

- It depends. Actually, we can write `class A :` *p* `B` with *p* being either `public`, `protected` or `private` (default).
- Depending on the choice of *p*, the members of *B* will be `public`, `protected` or `private`.

What happens to the methods?

- In some cases we can override/specialise them.
- In some cases we must override/specialise them.

Will my specialised method be used? ... not always!

Encapsulation

The access to members of a class can be controlled:

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

This is useful to hide implementation details and also to protect the implementation from unintended or malicious use.

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

```
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
};
```

Refining methods

A method `f` inherited from `A` can be refined if we need to write specialized code for the subclass `B`.

If we want the *intuitive* behaviour (call `B::f` for objects of class `B`) `f` must be marked as `virtual` in `A`. This is realised by so-called *dynamic* dispatch.

Otherwise, the method dispatch is *static* (i.e. decided at static time by the compiler), which means that sometimes `A::f` may be called for objects of class `B`!

Static dispatch is more performant.

Refining Methods

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

class B : public A {
public:
    void f(void) = { ... };
    void g(void) = { ... };
};

int main(void){
    B b;
    A* p = &b;

    b.f();  // calls B::f()
    p->f(); // calls A::f()
            // due to static binding based on p's type

    b.g();  // calls B::g()
    p->g(); // calls B::g()
            // due to dynamic binding
}
```

Abstract Classes

Abstract classes

- Cannot be instantiated;
- Define an abstract interface for derived classes;
- Are specified by at least one *pure* virtual function
  `virtual void someMethod() = 0;`
  which *must* be overriden by derived classes

Example

```
class Employee {
public:
    String name(void);
    virtual double salary(void) = 0 ;
    ...
};

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

Constructors and Inheritance

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

Constructors and Inheritance can be tricky:

- Constructors are not inherited, B may need to define its own constructors;
- Before constructing an object `B` the constructor of class `A` needs to be invoked;