02393 Programming in C++

02393 Programming in C++
Module 9: Inheritance

Lecturer:
Alceste Scalas

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

3 November 2020

Lecture Plan

<table>
<tr><th>#</th><th>Date</th><th>Topic</th><th>Book chapter *</th></tr>
<tr><td>1</td><td>01.09</td><td>Introduction</td><td></td></tr>
<tr><td>2</td><td>08.09</td><td>Basic C++</td><td>1</td></tr>
<tr><td>3</td><td>15.09</td><td rowspan="2">Data Types</td><td rowspan="2">2</td></tr>
<tr><td rowspan="2">4</td><td rowspan="2">22.09</td></tr>
<tr><td rowspan="2">Libraries and Interfaces</td><td rowspan="2">3</td></tr>
<tr><td>5</td><td>29.09</td></tr>
<tr><td>6</td><td>06.10</td><td>Classes and Objects</td><td>4.1, 4.2 and 9.1, 9.2</td></tr>
<tr><td colspan="4">Autumn break</td></tr>
<tr><td>7</td><td>20.10</td><td>Templates</td><td>4.1, 11.1</td></tr>
<tr><td>8</td><td>27.10</td><td>LAB DAY</td><td>Old exams</td></tr>
<tr><td>9</td><td>03.11</td><td>Inheritance</td><td>14.3, 14.4, 14.5</td></tr>
<tr><td>10</td><td>10.11</td><td>Recursive Programming</td><td>5</td></tr>
<tr><td>11</td><td>17.11</td><td>Linked Lists</td><td>10.5</td></tr>
<tr><td>12</td><td>24.11</td><td>Trees</td><td>13</td></tr>
<tr><td>13</td><td>01.12</td><td>Summary & Exam Preparation</td><td></td></tr>
<tr><td></td><td>07.12</td><td>Exam</td><td></td></tr>
</table>

* 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 (GP)**
  - ⋆ Templates (e.g. type-parametric functions, etc.)
  - ⋆ Implicit and explicit specialization
- **Object Oriented Programming (OOP)**
  - ⋆ Classes and objects;
  - ⋆ Encapsulation (private/protected/public attributes/methods)
  - ⋆ Methods (constructors, destructors, etc.)
  - ⋆ **Inheritance** (today)
- **Combination of OOP + GP**
  - ⋆ Class templates (e.g., used to define and implement containers)

Inheritance: from subtypes to subclasses

Example of **subclass** (or subtype) relations:

- Every integer number *is a* real
- Every square *is a* rectangle
- Every `HourlyEmployee` *is an* `Employee`
- ...

Inheritance: from subtypes to subclasses

Example of **subclass** (or subtype) relations:

- Every integer number *is a* real
- Every square *is a* rectangle
- Every `HourlyEmployee` *is an* `Employee`
- ...

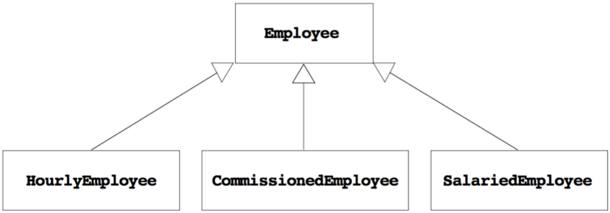
When is this useful?

- Bottom-up perspective (generalisation)
  - *"We have classes for different kinds of Employees which share some functionalities... let us group them together"*
- Top-down perspective (specialisation)
  - *"The class of employees is full of specialized code for particular kinds of employees... let us separate them in different classes"*

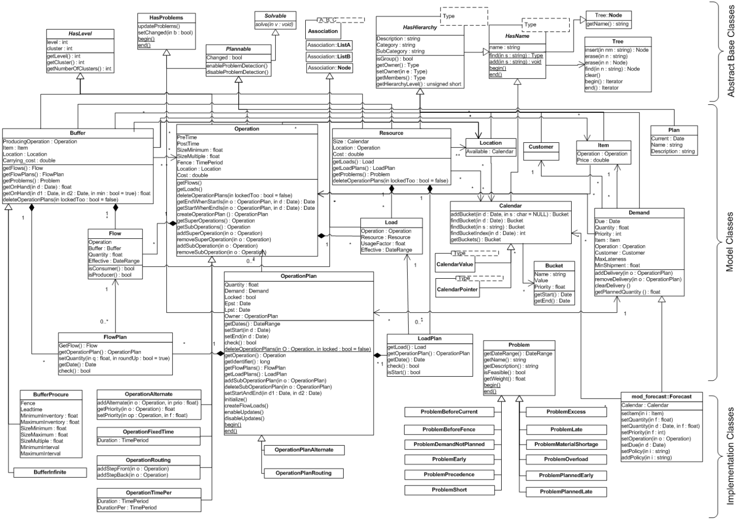
Advantages: modularity, clarity, maintanability

From "is-a" relations to class diagrams

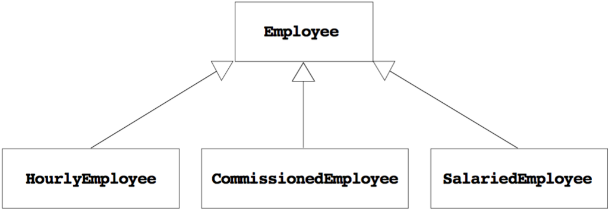
"Every `HourlyEmployee` *is an* `Employee`"
"Every `CommissionedEmployee` *is an* `Employee`"
"Every `SalariedEmployee` *is an* `Employee`"



Diagrams in real life



From diagrams to code (LIVE)



In C++ we write something like:

```
class Employee {
    ...
}

class HourlyEmployee : Employee {
    ...
}

...
```

Encapsulation

The access to the 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 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 A :` *p* `B`
  where *p* 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)

```cpp
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 want to write specialized code for a subclass `B`

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

Refining methods

A method `f()` inherited from `A` can be **refined** if we want to write specialized code for a subclass `B`

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

```
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()`?**

Refining methods

A method `f()` inherited from `A` can be **refined** if we want to write specialized code for a subclass `B`

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

```
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()`?**

Refining methods

A method `f()` inherited from `A` can be **refined** if we want to write specialized code for a subclass `B`

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

```
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 *static method dispatch* (very fast)

To ensure that `B::f()` is *always* called for objects of class `B`, we must mark `f()` as `virtual` in `A` (result: slower *dynamic* dispatch)

Refining Methods (LIVE)

```cpp
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

An `abstract class` is a class that 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:
    void double salary(void);
};
```

An abstract class **cannot be instantiated**: it only defines an **abstract 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

02393 Programming in C++