C++ Programming

Inheritance

M1 Math
Michail Lampis
Michail.lampis@dauphine.fr

In these slides

- Inheritance (héritage)
  - The ability to define classes that “derive” from other classes
- Basics
- Virtual functions, abstract classes
- Examples

Basic idea

- Inheritance: the ability to define a class that **extends** an already defined class
  - Motivation: we want to describe an “is special case of” relation for our objects
  - Examples:
    - A teacher is a special case of an employee
    - A square is a special case of a rectangle
    - A cat is a case of a mammal which is a special case of an animal

Basic idea

- Why do we need this? Typical scenario:
- We have already programmed the general class
  - The special case is almost the same...
  - → It could have some extra fields
  - → It could have some extra methods
  - → It could change some methods
- But the differences are so small that it doesn't make sense to re-program everything...

Example (from notes)

```
class Agent {
int _x, _y;
public:
Agent (int x = 0, int y = 0);
void set (int x, int y);
void affiche( );
};
```

- A class that implements an agent (e.g. in a game)
  - Data: x,y coordinates
  - Methods: set, print (affiche), implemented in the obvious way

Example (from notes)

- A special case: a mobile agent

```
class AgentMobile : public Agent {
int _xBut, _yBut;
public:
AgentMobile (int x, int y, int xBut, int yBut) {
    set (x, y);
    _xBut = xBut;
    _yBut = yBut;
}
};
```

Example explained

- Key line:

  class AgentMobile : public Agent

- Meaning: the AgentMobile class extends the Agent class
  - → All members (fields and methods) of Agent are automatically “inherited” into AgentMobile
  - But the AgentMobile class adds some new fields (_xBut, _yBut) as well as new methods (in this example only a constructor)

Example use:

```
Agent a1(2,3);
a1.affiche();
AgentMobile a2(2,3,4,5);
a2.affiche();
```

- This prints (2,3) twice
  - Why?
  - How does this work?

How it works

- Because AgentMobile inherits Agent, each time an AgentMobile object is created:
  - We allocate enough memory for an Agent object **plus** the extra fields of AgentMobile
  - The (default) constructor for Agent is called
- This is similar to an object that contains another object, but conceptually different (why?)
- AgentMobile objects also contain the same methods as Agent objects.

Demonstration

- Consider this version of the Agent constructor

```
Agent::Agent(int x, int y){
   cout << "x=" << x << " y=" << y << endl;
   _x = x;
   _y = y;
}
```

- Recall the constructor declaration inside the class

```
Agent (int x = 0, int y = 0);
```

- What is the result of the following code?

```
Agent a1(2,3);
AgentMobile a2(2,3,4,5);
```

Demonstration

- Output:

  x=2 y=3

  x=0 y=0

- Why?
  - First line is easy (constructor for Agent)
  - Second line: before constructor for AgentMobile, the **default** constructor for Agent is called.

Calling another constructor

- We can define the AgentMobile constructor to call the Agent constructor appropriately (with parameters)

```
AgentMobile::AgentMobile(int x, int y, int xBut, int yBut) :Agent(x,y) {
        _xBut = xBut;
        _yBut = yBut;
}
```

What we know so far

- We can define classes that “extend” previous classes.
  - These “child” classes inherit all fields and methods of their “parents”.
- Constructors/Destructors will be automatically called, in the order of inheritance to build new objects.
- Why do these things exist/make our life easier?
  - Main reason: polymorphism

Ignoring details

- The main advantage of using inheritance is that we can deal with child class object while **ignoring** their type.
  - We simple see them as objects of the more general class
  - This is the most basic example of **polymorphism**, the idea that an object can have “many forms”.

Example

- Consider this function

```
void test(Agent a){  a.affiche(); }
```

- This function takes an Agent parameter. However, the following is legal.

```
Agent a1(2,3);
AgentMobile a2(2,3,4,5);
test(a1);
test(a2); //Here we treat a2 as an Agent
```

Example

- This is more commonly used with pointers

```
Agent *p;
Agent a1(2,3);
AgentMobile a2(2,3,4,5);
p = &a1;
p->affiche();
p = &a2; //This is OK!!
p->affiche();
```

Overriding functions

- There is one serious problem with what we have done so far
  - AgentMobile inherits the affiche( ) method
  - But it doesn't work as we would like!
- Solution: override it! (that is, redefine it)

First attempt

- Add affiche( ) declaration to AgentMobile

```
class AgentMobile : public Agent {
…  public:
  void affiche(); … };
```

- Implement it

```
void AgentMobile::affiche() {
        cout << "(" << _x << "," << _y << ")"  << "->" ;
        cout << "(" << _xBut << "," << _yBut << ")"  << endl;
}
```

Access restrictions

- The previous program does not compile!
  - A member of the AgentMobile class (affiche( )), cannot access a private member of the Agent class (x, y)
  - Solution: declare fields which are mean to be accessed by child classes **protected** instead of private.

Second attempt

- Change Agent class

```
class Agent {
        protected:   int _x, _y; … };
```

- Add affiche( ) method to AgentMobile class

```
void AgentMobile::affiche() {
        cout << "(" << _x << "," << _y << ")"  << "->" ;
        cout << "(" << _xBut << "," << _yBut << ")"  << endl;
}
```

Using the overridden method

```
Agent a1(2,3);
AgentMobile a2(2,3,4,5);
a1.affiche();
a2.affiche();
```

- The first call is for Agent::affiche( )
- The second for AgentMobile::affiche( )
- How does the compiler know?
  - From the type of a2

Override + Polymorphism

- Consider the following:

```
Agent *p;
AgentMobile a2(2,3,4,5);
p = &a2;
p -> affiche();
```

- What does this print?
- Which affiche( ) is called?

Static binding

- The compiler decides which method to call by looking at the **declared** type of the object, not the actual **run-time** type the object has.
- In the previous example:

  p -> affiche();

- p has been declared as pointer to Agent
  - → Agent::affiche( ) is called
  - Even though p is actually pointing to an AgentMobile!

Virtual functions

- The previous behavior can be avoided by using **virtual** functions
- A virtual function is a function that will be resolved dynamically
  - When the program runs, to execute the command p->affiche( ), the program first examines what type p **really** has
  - Then, the most specific affiche( ) found is called

Example

- First, change the Agent class

  class Agent { ...

  virtual void affiche ( ); … }

- This means that we intend to **dynamically override** this method
- If a child class also defines it, call to affiche will be resolved at run-time

Example

- Now, we don't need to change anything in AgentMobile class
  - Adding the virtual keyword there is recommended

```
Agent *p;
AgentMobile a2(2,3,4,5);
p = &a2;
p -> affiche();
```

- This code now calls AgentMobile::affiche( )

Virtual destructors

- Consider the following

```
class A { … };
class B : public A { … };
A *p = new B; //Is this OK? Which constructor?
delete p; //Is this OK? Which destructor?
```

Explanation

- The first line is OK
  - We are allocating an object of type B. An A pointer can point to it (polymorphism). B's constructor is used.
- The second line is a problem
  - Since p is pointing to A, this will call ~A( )
  - This will not clean up the object properly!
  - Solution: make A's destructor virtual

Pure virtual functions

- We can also define functions which **must** be overridden
- This is common when designing an **interface**
  - An abstract base class which does not yet contain enough information to do useful things
  - But its special cases do
  - We want them all to have the same uniform methods

An example

```
class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) { }
    virtual int area (void) =0; //CANNOT calculate this!!
    void printarea( )
      { cout << this->area() << '\n'; }
};
```

Example (continued)

```
class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) { }
    int area() //Override!
      { return width*height; }
};
```

Example (continued)

```
class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area() //Override!
      { return width*height/2; }
};
```

Example (continued)

- What will this print?

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
ppoly1->printarea();
ppoly2->printarea();
```

Example (continued)

- This?

  ```
  Polygon * ppoly1 = new Polygon (4,5);
  ppoly1 → printarea( );
  ```

- Compiler error!

Summary

- Inheritance:
  - Create derived classes which keep methods/fields
  - Intended to model “is special case of”
- Polymorphism
  - Write code for general objects, treat special ones
  - Methods can be overridden to work correctly for the child classes
  - This can be done dynamically on run-time