Université de Paris

Programmation avancée
3 – OOP in C++

Sylvain Lobry

03/10/2022

Université de Paris

Sondage: https://www.wooclap.com/PROGAC3

Université de Paris

Menu of the day:
- Very basics of OOP
- OOP in C++

Object's concepts

Reminders on object-based approach

Definition of an **object**: An object is an entity referenced by an identifier. It is often tangible.

- An object has a set of attributes (structure) and methods (behaviour)

Object's concepts

Reminders on object-based approach

- Definition of a **class**: set of similar objects (i.e. having the same attributes and the same methods).
- An object from a class is an *instance of this class*.

- Defintion of the **abstraction**: principle of selecting the relevant properties of an object for a given problem

Object's concepts

Reminders on object-based approach

- Definition of **encapsulation**: to bundle in a single unit data (attributes) and the methods to manipulate them. Additionally, to hide some attributes or methods to other objects. Note that this in an abstraction.

- **Specialization**: a new class A can be created as a subclass of another class B, in which case class A specializes the class B.
- **Generalization** is the opposite (superclass B is a generalization of subclass A).

- **Inheritance**: the fact that a subclass gets the behaviour and the structure of the superclass
- This is a **consequence** of specialization

Object's concepts

Reminders on object-based approach

- **Abstract** and **concrete** classes: abstract classes are classes that do not have instances (e.g. Mammal). Concrete classes do (e.g. Human).
- Abstract classes allow for class hierarchies and to group attributes and methods. They should have subclasses.

- **Polymorphism**: behavior from objects of a same class (in general abstract) can be different as they are instances of different subclasses.

Object’s concepts

Reminders on object-based approach

- **Composition**: complex objects can be composed of other objects.
- It is defined at the class level, but we only compose actual instances
- It can be:
  - a strong relationship: components cannot be shared; destruction of the composed object implies destruction of the components
  - a weak relationship (a.k.a. aggregation): components can be shared

C++ 101

Oriented object programming

- A class definition in C++ looks like

```
class name
{
    access_specifier:
        attribute or method;
        attribute or method;
        ...
    access_specifier:
        attribute or method;
        attribute or method;
        ...
};
```

C++ 101

Oriented object programming

- A class definition in C++ looks like

```
class name
{
    access_specifier:
        attribute or method;
        attribute or method;
        ...
    access_specifier:
        attribute or method;
        attribute or method;
        ...
};
```

- access_specifier can either be:
  - public: everyone has access
  - private: only member of the same class (and friends)
  - protected: only member of the same class, but also derived classes (and friends)
  - By default: private
- attribute: variable inside a class
- method: function inside a class

C++ 101

Example

```
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float, float);
        float dist(Point &p);
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

```cpp
#include <iostream>
#include <math.h>

class Point
{
    public:
        void set(float, float);
        float dist(Point &p);
    private:
        float x, y;
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

```
exClass.cpp:30:18: error: 'x' is a private member of 'Point'
        std::cout << p1.x << std::endl;
                        ^
exClass.cpp:6:6: note: implicitly declared private here
        int x, y;
            ^
1 error generated.
```

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float, float);
        float dist(Point &p);
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
    std::cout << p1.x;
}
```

C++ 101

Example

- Back to something that works…

```
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float, float);
        float dist(Point &p);
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

- Back to something that works…
- Let’s improve the signature of dist. How ?

```
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float, float);
        float dist(Point &p);
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

- Back to something that works…
- Let's improve the signature of dist. How ?
- Adding const qualifier

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float, float);
        float dist(const Point &p);
};

void Point::set(float a, float b)
{
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << "Distance between points = " << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

- Back to something that works…
- Let’s improve the signature of dist. How ?
- Adding const qualifier
- Inline definition of set

```
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float a, float b) {x = a; y = b;};
        float dist(const Point &p);
};

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Example

- First line of main: variables declaration
- What about initialization?

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        void set(float a, float b) {x = a; y = b;};
        float dist(const Point &p);
};

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1, p2;
    p1.set(1, 2);
    p2.set(4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

What is the value of p1.x?

C++ 101

Constructor

- First line of main: variables declaration
- What about initialization?

```
Buidling a point.
Buidling a point.
4.24264
```

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
{
    std::cout << "Building a point." << std::endl;
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2 (4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Constructor

- First line of main: variables declaration
- What about initialization?
- If not defined: default constructor (no parameters)

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
{
    std::cout << "Building a point." << std::endl;
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2 (4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Constructor

- First line of main: variables declaration
- What about initialization?
- If not defined: default constructor (no parameters)
- If defined, no default constructor

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
{
    std::cout << "Building a point." << std::endl;
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2;

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Constructor

```
exClass.cpp:27:8: error: no matching constructor for initialization of 'Point'
        Point p2;
              ^
exClass.cpp:4:7: note: candidate constructor (the implicit copy constructor) not
      viable: requires 1 argument, but 0 were provided
class Point
      ^
exClass.cpp:12:8: note: candidate constructor not viable: requires 2 arguments,
      but 0 were provided
Point::Point(float a, float b)
       ^
1 error generated.
```

Recommendation: always define a constructor

```cpp
#include <iostream>
#include <math.h>

class Point
{
    float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
{
    std::cout << "Building a point." << std::endl;
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2;

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Member initializer list

- Possible to initialize elements before the body of the constructor.

```
#include <iostream>
#include <math.h>

class Point
{
    const float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
    : x(a), y(b)
{
    std::cout << "Building a point." << std::endl;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2 (4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Member initializer list

- Possible to initialize elements before the body of the constructor.
- Useful in case of const attributes for instance

```
exClass.cpp:12:8: error: constructor for 'Point' must explicitly initialize the
      const member 'x'
```

```
#include <iostream>
#include <math.h>

class Point
{
    const float x, y;
    public:
        Point(float, float);
        float dist(const Point &p);
};

Point::Point(float a, float b)
{
    std::cout << "Building a point." << std::endl;
    x = a;
    y = b;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1, 2);
    Point p2 (4, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Constructor overloading

- Possible to define several constructors with different parameters lists.

```
Building a point using default constructor.
Buidling a point.
5
```

```cpp
#include <iostream>
#include <math.h>

class Point
{
    const float x, y;
    public:
        Point();
        Point(float, float);
        float dist(const Point &p);
};

Point::Point()
    : x(0), y(0)
{
    std::cout << "Building a point using default constructor." << std::endl;
}

Point::Point(float a, float b)
    : x(a), y(b)
{
    std::cout << "Building a point." << std::endl;
}

float Point::dist(const Point &p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1;
    Point p2 (0, 5);

    std::cout << p1.dist(p2) << std::endl;
}
```

C++ 101

Destructor

- Similarly, you can define a destructor (called at the end of life of an object)
- In general: useful for dynamically allocated memory

exClass.cpp

```
#include <iostream>
#include <math.h>

class Point
{
    float *x, *y;
    public:
        Point(float, float);
        ~Point();
        float dist(const Point& p);
};

Point::~Point()
{
    delete x;
    delete y;
    std::cout << "Deleting a point." << std::endl;
}

Point::Point(float a, float b)
{
    x = new float (a);
    y = new float (b);
    std::cout << "Buidling a point." << std::endl;
}

float Point::dist(const Point& p)
{
    return sqrt((*p.x - *x) * (*p.x - *x) + (*p.y - *y) * (*p.y - *y));
}

int main()
{
    Point p1 (1,2);
    Point p2 (3,4);
    std::cout << p1.dist(p2) << std::endl;
    return 0;
}
```

C++ 101

Struct is a class

- A struct in C++ is actually a class (can hold methods as well as attributes)
- Only difference, default access is public

exClass.cpp

```cpp
#include <iostream>
#include <math.h>

struct Point
{
    Point();
    Point(float, float);
    float dist(const Point& p);
    private:
        float x, y;
};

Point::Point(float a, float b)
    : x(a), y(b)
{
    std::cout << "Buidling a point." << std::endl;
}

float Point::dist(const Point& p)
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1,2);
    Point p2 (3,4);
    std::cout << p1.dist(p2) << std::endl;
    return 0;
}
```

C++ 101

Const methods

- A method can be marked const
- Will not be able to modify the object
- Can be called by const objects

exClass.cpp

```cpp
#include <iostream>
#include <math.h>

struct Point
{
    Point();
    Point(float, float);
    float dist(const Point& p) const;
    private:
        float x, y;
};

Point::Point(float a, float b)
    : x(a), y(b)
{
    std::cout << "Buidling a point." << std::endl;
}

float Point::dist(const Point& p) const
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1,2);
    Point p2 (3,4);
    std::cout << p1.dist(p2) << std::endl;
    return 0;
}
```

C++ 101

Operator overloading

- Operators can be overloaded for objects of user-designed classes
- Friend method if there is a need to access private/protected attributes.
- Most operators can be overloaded.

exClass.cpp

```cpp
#include <iostream>
#include <math.h>

struct Point
{
    Point();
    Point(float, float);
    float dist(const Point& p) const;
    friend std::ostream & operator << (std::ostream &o, const Point &p);
    private:
        float x, y;
};

std::ostream & operator << (std::ostream &o, const Point &p)
{
    o << "(" << p.x << ", " << p.y << ")";
    return o;
}
Point::Point(float a, float b)
    : x(a), y(b)
{
    std::cout << "Buidling a point." << std::endl;
}

float Point::dist(const Point& p) const
{
    return sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y));
}

int main()
{
    Point p1 (1,2);
    Point p2 (3,4);
    std::cout << p1.dist(p2) << std::endl;
    std::cout << p1 << std::endl;
    return 0;
}
```

C++ 101

Static members

- Same value/method for all objects of the class
- Needs to be initialized outside of the class

```
Number of points 1
Number of points 2
Number of points 1
Number of points 0
```

exClass.cpp

```cpp
#include <iostream>
#include <math.h>

class Point
{
    public:
        Point();
        ~Point();
        Point(float, float);
        static int count;
    private:
        float x, y;
};

int Point::count = 0;

Point::Point(float a, float b)
    : x(a), y(b)
{
    ++count;
    std::cout << "Number of points " << count << std::endl;
}

Point::~Point()
{
    --count;
    std::cout << "Number of points " << count << std::endl;
}

int main()
{
    Point p1 (1,2);
    Point p2 (3,4);

    return 0;
}
```

C++ 101

Inheritance

- Public inheritance
- Most common case
- Everything is inherited except:
  - constructor/destructor
  - operator "="
  - friends
- Can inherit from multiple class:

class A: public B, public C

exDeriv.cpp

```cpp
#include <iostream>

class Base
{
    private:
        int a;
    protected:
        int b;
    public:
        int c;
        Base() {a=5; b=5; c=5;};
};

class DerivedPub: public Base
{
    //a is not accessible, b is protected, c is public
    public:
        DerivedPub() {};
};

int main()
{
    Base test;
    DerivedPub testPub;
}
```

C++ 101

Inheritance

- Other access specifiers possible

exDeriv.cpp

```
#include <iostream>

class Base
{
    private:
        int a;
    protected:
        int b;
    public:
        int c;
        Base() {a=5; b=5; c=5;};
};

class DerivedPub: public Base
{
    //a is not accessible, b is protected, c is public
    public:
        DerivedPub() {};
};


class DerivedPro: protected Base
{
    //a is not accessible, b is protected, c is protected
    public:
        DerivedPro() {};
};

class DerivedPri: private Base
{
    //a is not accessible, b is private, c is private
    public:
        DerivedPri() {};
};

int main()
{
    Base test;
    DerivedPri testPri;
    DerivedPro testPro;
    DerivedPub testPub;
}
```

C++ 101

Polymorphism

- Pointer to a derived class and pointer to base class: type compatible
  - We can access members of the base class and derived class through pointers
  - Although, need pointers of different types
  - Area methods have to be called on the derived class

```
Modifying
In rectangle
12
Modifying
In triangle
6
```

poly.cpp

```cpp
#include <iostream>
class Shape {
    protected:
        int width, height;

    public:
        Shape(int a, int b) : width(a), height(b) {};
        void modify(int a, int b) {
            std::cout << "Modifying" << std::endl;
            width = a;
            height = b;
        };
};
class Rectangle: public Shape {
    public:
        Rectangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In rectangle" << std::endl;
            return (width * height);
        };
};
class Triangle: public Shape {
    public:
        Triangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In triangle" << std::endl;
            return (width * height * 1.0 / 2);
        };
};
int main() {
    Rectangle rec(5,3);
    Triangle tri(5,3);

    Shape *shape = &rec;
    shape->modify(3, 4);
    std::cout << rec.area() << std::endl;

    shape = &tri;
    shape->modify(3, 4);
    std::cout << tri.area() << std::endl;

    return 0;
}
```

C++ 101

Polymorphism

- Pointer to a derived class and pointer to base class: type compatible
  - We can access members of the base class and derived class through pointers
  - Although, need pointers of different types
  - Area methods have to be called on the derived class
  - Way around: keyword ”virtual”
  - Indicate that the method can be re-defined in derived classes

```
Modifying
In rectangle
12
Modifying
In triangle
6
```

poly.cpp

```cpp
#include <iostream>
class Shape {
    protected:
        int width, height;

    public:
        Shape(int a, int b) : width(a), height(b) {};
        void modify(int a, int b) {
            std::cout << "Modifying" << std::endl;
            width = a;
            height = b;
        };
        virtual float area() {return 0;}
};
class Rectangle: public Shape {
    public:
        Rectangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In rectangle" << std::endl;
            return (width * height);
        };
};
class Triangle: public Shape {
    public:
        Triangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In triangle" << std::endl;
            return (width * height * 1.0 / 2);
        };
};
int main() {
    Rectangle rec(5,3);
    Triangle tri(5,3);

    Shape *shape = &rec;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    shape = &tri;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    return 0;
}
```

C++ 101

Abstract class

- Virtual function = can be re-defined.
- Does it make sense to define a null area?
- Shape can be made abstract
- Abstract class can not be initialized.

poly.cpp

```cpp
#include <iostream>
class Shape {
    protected:
        int width, height;

    public:
        Shape(int a, int b) : width(a), height(b) {};
        void modify(int a, int b) {
            std::cout << "Modifying" << std::endl;
            width = a;
            height = b;
        };
        virtual float area()=0;
};
class Rectangle: public Shape {
    public:
        Rectangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In rectangle" << std::endl;
            return (width * height);
        };
};
class Triangle: public Shape {
    public:
        Triangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In triangle" << std::endl;
            return (width * height * 1.0 / 2);
        };
};
int main() {
    Rectangle rec(5,3);
    Triangle tri(5,3);

    Shape *shape = &rec;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    shape = &tri;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    return 0;
}
```

C++ 101

Abstract class

- Virtual function = can be re-defined.
- Does it make sense to define a null area?
- Shape can be made abstract if it contains at least one virtual pure function (”=0” after declaration)
- Abstract class can not be initialized.

```
poly.cpp:36:2: error: allocating an object of abstract class type 'Shape'
        Shape(3, 4);
        ^
poly.cpp:13:23: note: unimplemented pure virtual method 'area' in 'Shape'
        virtual float area()=0;
                      ^
1 error generated.
```

poly.cpp

```cpp
#include <iostream>
class Shape {
    protected:
        int width, height;

    public:
        Shape(int a, int b) : width(a), height(b) {};
        void modify(int a, int b) {
            std::cout << "Modifying" << std::endl;
            width = a;
            height = b;
        };
        virtual float area()=0;
};
class Rectangle: public Shape {
    public:
        Rectangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In rectangle" << std::endl;
            return (width * height);
        };
};
class Triangle: public Shape {
    public:
        Triangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In triangle" << std::endl;
            return (width * height * 1.0 / 2);
        };
};
int main() {
    Rectangle rec(5,3);
    Triangle tri(5,3);
    Shape(3, 4);

    Shape *shape = &rec;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    shape = &tri;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    return 0;
}
```

C++ 101

Abstract class

- Virtual function = can be re-defined.
- Does it make sense to define a null area?
- Shape can be made abstract
- Abstract class can not be initialized.

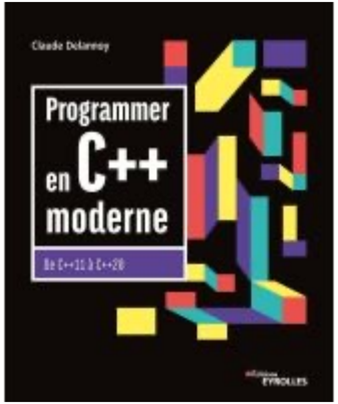
poly.cpp

```cpp
#include <iostream>
class Shape {
    protected:
        int width, height;

    public:
        Shape(int a, int b) : width(a), height(b) {};
        void modify(int a, int b) {
            std::cout << "Modifying" << std::endl;
            width = a;
            height = b;
        };
        virtual float area()=0;
};
class Rectangle: public Shape {
    public:
        Rectangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In rectangle" << std::endl;
            return (width * height);
        };
};
class Triangle: public Shape {
    public:
        Triangle(int a, int b) : Shape(a, b) {};

        float area () {
            std::cout << "In triangle" << std::endl;
            return (width * height * 1.0 / 2);
        };
};
int main() {
    Rectangle rec(5,3);
    Triangle tri(5,3);

    Shape *shape = &rec;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    shape = &tri;
    shape->modify(3, 4);
    std::cout << shape->area() << std::endl;

    return 0;
}
```

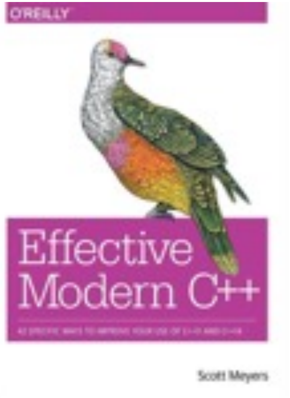
Conclusion

- We have seen very basics of C++
- Should be capable of reading a C++ code, writing simple ones
- Going further:

http://www.cplusplus.com/files/tutorial.pdf



Claude Delannoy, Eyrolles



After some time, any book by Scott Meyers