UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

INTRODUCTION TO MODERN C++

LECTURE 5

Rémi Géraud

February 25, 2016

École Normale Supérieure de Paris

Lecture 5
Structs/Classes: Members, Inheritance, and the Rule of 3/0/5

A WORD OF CAUTION

We are now entering the world of classes.

We'll stay there for some time.

Then we'll leave for a better place.

TABLE OF CONTENTS

STRUCTS AND MEMBERS

STRUCTS

In last week's lab, we encountered *compound types* such as `structs`.

```
int main() {
    struct Point2D {
        double x;    // First field
        double y;    // Second field
    };

    Point2D v {4, 7};

    std::cout << "v = ("
              << v.x
              << ", "
              << v.y
              << ")"
              << std::endl;
}
```

Structs as special-purpose types: fields

`Struct`s are very practical ways to create special-purpose types, e.g.:

- `Mario` (fields: position, score, etc.)
- `Koopa` (fields: health, level, etc.)
- `Sky` (fields: color, position, etc.)

or, closer to earth,

- `Option` (fields: price, volatility, quantity, etc.)
- `Country` (fields: population, GDP, name, etc.)

STRUCT METHODS

In fact, we can do more. Here's an example:

```
int main () {
    struct Point2D {
        double x, y;            // Fields

        double Norm2() {        // Norm2 method
            return x*x + y*y;
        };
    };

    Point2D v {10, 2};
    Point2D w {3, 4};

    std::cout << "||v||^2 = " << v.Norm2() << std::endl;
    std::cout << "||w||^2 = " << w.Norm2() << std::endl;
}
```

Struct fields and methods: members

General rule of thumbs:

- Fields = Properties
- Methods = Responsibilities

Example: `struct Mario`

- Fields: position, score, etc.
- Methods: move, jump, die, grow, etc.

PUBLIC AND PRIVATE PARTS

STRUCTS ARE PUBLIC BY DEFAULT

By default,

- **All** the fields of a `struct` can be read and modified,
- **All** the methods of a `struct` can be called,

by *anyone* (i.e. any part of the program).

Sometimes, you don't want that: IP, correctness, etc.

CLASSES ARE PRIVATE BY DEFAULT

We can use `class` instead of `struct`. By default,

- Only the `class` itself can read or modify its fields,
- Only the `class` itself can call its methods.

and *no one else* (unless explicitly specified)

Classes and structs are otherwise equivalent. But in practice, almost everyone uses classes.

Here is a typical C++ class example:

```
class Square {
  double x, y, w, h;       // Private
  double area, perimeter; // Private

  public:  // Everything that follows is public
     double getArea() { ... };
     double getPerimeter() { ... };
     double resize(double newW, double newH) { ... };
}
```

CLASSES AND HEADER FILES

For the sake of clarity, it is better to separate the class definition from its implementation. To that end we use *header files*. Example:

```
// File: Point2D.h
class Point2D {
  public:
    double x, y;
    double Norm2();
};
```

```
// File: Point2D.cpp
double Point2D::Norm2() {
  return x*x + y*y;
}
```

This makes it easier to separate *specification* from *implementation*. To use the class `Point2D` you must add `#include "Point2D.h"` to your program.

INHERITANCE

Inheritance: main idea

There is not much difference between a square and a rectangle. Is there a way to avoid coding the same things twice?

Yes: Inheritance.

INHERITANCE: COPYING EVERYTHING FROM YOUR PARENTS

```
class Rectangle {
  public:
    double x, y, w, h;
    double getArea();
};

class Square : public Rectangle {
  // All public fields are copied from Rectangle
  // All public methods are copied from Rectangle
};
```

We say that `Square` is a “child” of `Rectangle`. Or that `Rectangle` is a “parent” of `Square`.

INHERITANCE: PROTECTED MEMBERS

What about `private` fields and methods? Those don't get copied.

But we can share something within a family by using `protected`:

```
class Rectangle {
  protected:
    double x, y, w, h;
  public:
    double getArea();
};

class Square : public Rectangle { };
```

`Square` will have access to `x, y, w, h`. But someone who isn't part of the family will not have access.

HIDING INHERITANCE

Inheritance can be embraced or denied:

```
class Shape1 : public Rectangle { ... };    // Recognized
class Shape2 : protected Rectangle { ... }; // Family secret
class Shape3 : private Rectangle { ... };   // Unrecognized
```

By default, inheritance of classes is `private`.

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming

OOP is a software design paradigm developed in the 1970's.

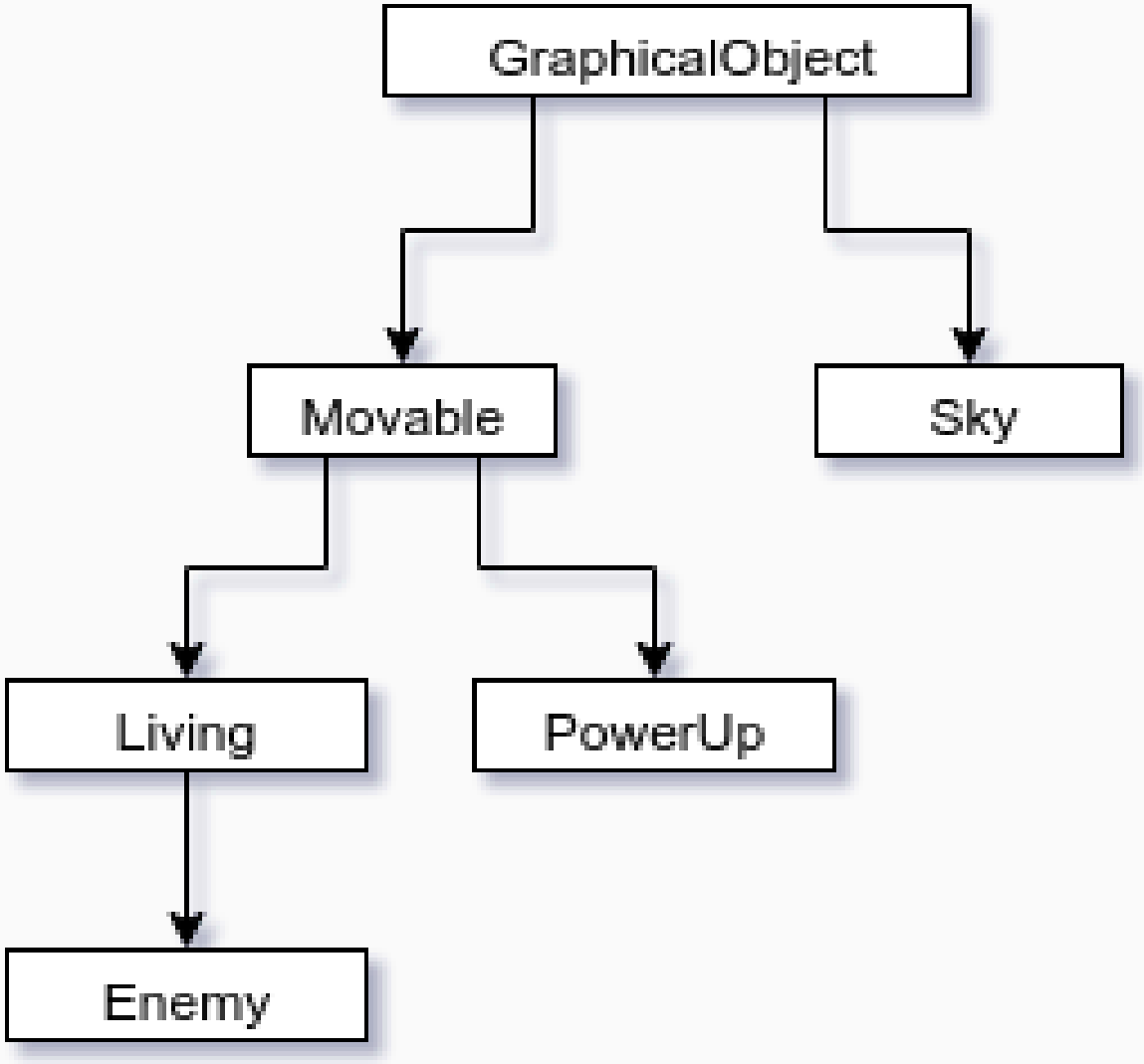
Main ideas:

- Construct objects (= classes)
- Specify their properties (= fields), responsibilities (= methods), and visibility (= private/public)
- Use dependencies (= inheritance) to avoid rewriting code

Main goals:

- Separation of concerns (= team work)
- Encapsulation (= how I work is not your business).

EXAMPLE: PLATFORM GAME CLASS DIAGRAM



OOP: THE GOOD, THE BAD, AND THE UGLY

- The Good: Fast development, easy teamwork, easy to learn
- The Bad: multithreading, resource management
- The Ugly: ...

EXAMPLE: VIOLATING THE LISKOV SUBSTITUTION PRINCIPLE

- Create a class `Ellipse`. Create a class `Circle`.
- A circle "is an" ellipse, therefore `Circle` inherits from `Ellipse`.
- Assume that `Ellipse` has a `stretchX` method.
- This method is inherited by `Circle`.
- But if we use `stretchX` on a `Circle`, it is no longer a circle...

Bottom line:

An OO-model of a circle should not be a sort of OO-model of an ellipse

CONSTRUCTORS & DESTRUCTORS

INITIALISATION: CONSTRUCTORS

If we want a `class` to be initialised in some way, we can use a special method called a *constructor*.

```
class Point2D {
    double x, y;
    public:
       Point2D(double newX, double newY); // Constructor
};

Point2D::Point2D (double newX, double newY) {
    x = newX;
    y = newY;
}

int main() {
    Point2D myPoint (27, 35);
    ...
}
```

INITIALISATION: CONSTRUCTORS

You can have several constructors, as long as they don't overlap.
Most useful ones are:

- Default constructor;
- Copy constructor — necessary for complicated classes;
- Move constructor — if you want to move without copying.

CLEANING-UP: DESTRUCTORS

A class should clean after itself. The cleaning-up code is taken care of in a *destructor*:

```
class MyStorage {
   ...
   public:
     MyStorage( ... ); // Constructor: Opens a file
     ~MyStorage();     // Destructor: Closes the file
};
```

Important: Any resources acquired during creation should be freed upon destruction.

The rule of 3 (or 0 or 5)

You should use only one of these combinations:

0. No destructor, copy or move constructor, no assignment operator;
3. Destructor, copy constructor and copy assignment operator;
3. Destructor, move constructor and move assignment operator;
5. Destructor, copy and move constructors, copy and move assignment operators.

Remember:

Respect the rule of 3 (or 0 or 5).

If you don't, your program might behave unexpectedly.

Questions?

Lab Session
Const, Virtual, and Move Semantics