Lecture

11_6.3

# Copy Constructor & Other Advanced features

**Alessandro Savino**

**Politecnico di Torino (Italy)**

*alessandro.savino@polito.it*

*www.testgroup.polito.it*

# License Information

# Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

# Goal

- This lecture presents a deeper view about C++ classes and objects

# Prerequisites

– **A basic knowledge about classes**

# Homework

– **None**

# Outline

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Copy Constructor

- A copy constructor is a special constructor that makes possible defining an object as a copy of an existing object of the same class.
- A copy constructor has only one formal parameter that is the type of the class (the parameter may be a reference to an object).

# Copy Constructor

- **A copy constructor is a special constructor that makes possible defining an object as a copy of an existing object of the same class.**
- **A copy constructor has only one formal parameter that is the type of the class (the parameter may be a reference to an object).**

```
Rectangle(const Rectangle &to_copy);
```

# Copy Constructor

- **In the definition it is possible to refer to any private data of the object-to-copy directly.**
  - **You must program what has to be copied!**

```
Rectangle::Rectangle(const Rectangle &to_copy) {
  this->m_width = to_copy.m_width;
  this->m_length = to_copy.m_length;
}
```

# Copy Constructor

- **The invocation requires then to pass the object to be copied as parameter of the constructor**

```
int main() {
    ...
   Rectangle r3(2,8)
   Rectangle r4(r3);
    ...
}
```

After this operation r4 has the same width and length of r3

# Outline

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Composition: Objects as member of classes

- **Composition**
  - **Sometimes referred to as a *has-a* relationship**
  - **A class can have objects of other classes as members**
  - **Example**
    - **AlarmClock object with a Time object as a member**

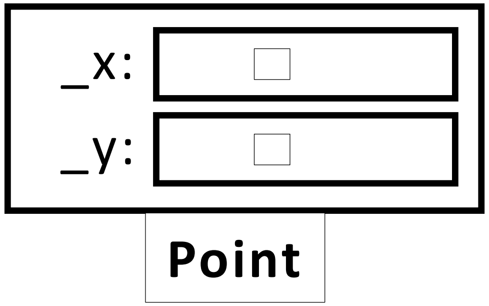
# *Composition – 2*

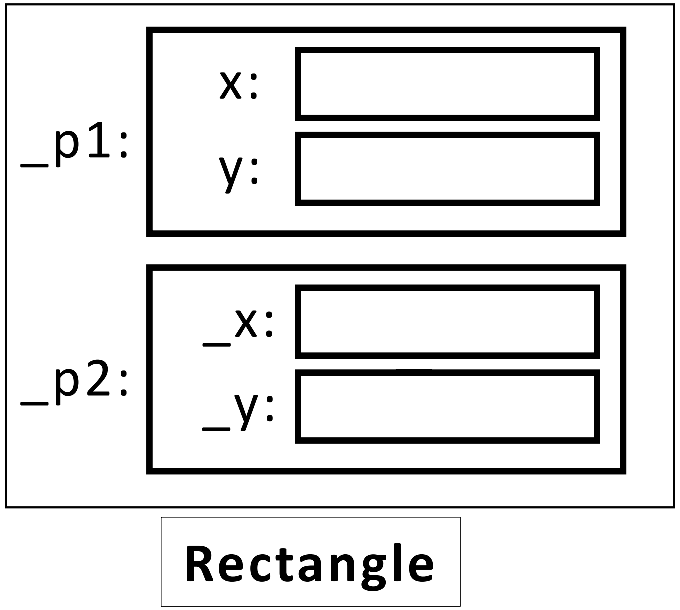
- **Initializing member objects**
  - **Member initializers pass arguments from the object's constructor through the *member initializer list* to member-object constructors**
  - **Member objects are constructed in the order in which they are declared in the class definition**
  - **If a member initializer is not provided**
    - **The member object's default constructor will be called implicitly**

# Composed objects

```
class Point {
public:
  Point(int x, int y);
private:
  int _x, _y;
};

class Rectangle {
public:
  Rectangle(
       int x1, int y1,
      int x2, int y2);
private:
  Point _p1,_p2;
};
```





# Outline

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Using the `this` pointer

- **Member functions know which object's data members to manipulate.**
  - **Every object has access to its own address through a pointer called `this` (a C++ keyword).**
  - **An object's `this` pointer is not part of the object itself.**
  - **The `this` pointer is passed (by the compiler) as an implicit argument to each of the object's `non-static` member functions.**

# `this` Example

```
#include <iostream>
using namespace std;

class Test
{
public:
    Test( const int &value = 0 ); // default constructor
    void print() const;
private:
    int _x;
};
```

# this Example

```
Test::Test( const int &value )
{
    x = value;
} // end constructor Test

void Test::print() const
{
    cout << "        x = " << x;
    cout << "\n  this->x = " << this->x;
    cout << "\n(*this).x = " << ( *this ).x << endl;
}

int main()
{
    Test testObject( 12 ); // instantiate and
    testObject.print();   // initialize testObject
    return 0;
} // end main
```

# Outline

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Polymorphism

- **When a function in a derived class overrides a function in a base class, the function to call is determined by the type of the object.**
  - **This decision is taken at run-time.**
- **In programming languages, polymorphism means that some code or operations or objects behave differently in different contexts.**

# Polymorphism

- As example, refer to the + (plus) operator in C++:

- 4 + 5 <-- integer addition
- 3.14 + 2.0 <-- floating point addition
- “foo” + "bar" <-- string concatenation!

# Exercise

- We want to represent 2 types of employees as classes in C++: a generic employee (class Employee) and a manager (class Manager).
  - For these employees, we want to store data, like their name and salary.
  - We require the functionality to expose the employee's salary and name.
  - Salaries are calculated to employees’ bank accounts by an external officer.
    - A manager is an employee, with a higher salary

# Exercise

- W… …epresent 2 types of employ…
  …+: a generic employee (cl… …e)
  a… …er (class Manager)
  - … …ke
    t…
  - W…
    e…
  - …
    … …ry

Take your time and solve it!

# Solution

```
class Employee {
public:
  string getName() const;
  virtual float getSalary() const;
  void setNameAndSalary(const string &name,
                    const float &salary);

protected:
  string _name;
  float _salary;
};
```

# Solution

```
class Employee {
public:
  string getName() const;
  virtual float getSalary() const;
  void setNameAndSalary(const string &name,
                        const float &salary);

protected:
  string _name;
  float _salary;
};
```

protected: → Less Restrictive

string _name; float _salary; → Accessible by derived classes

# Solution

```
string Employee::getName() const
{
  return _name;
}

float Employee:: getSalary() const
{
  return _salary;
}
```

# Solution

```
void Employee::setNameAndSalary(const string
&name, const float &salary) {
     _name = name;
     _salary = salary;
}
```

# Solution

```
#include “Employee.h”

class Manager: public Employee{
public:
     float getSalary() const;
};
```

**No need to define again properties getName() and setNameAndSalary(): they are inherited!**

# Solution

```
#include “Manager.h”

float Manager::getSalary() const
{
  return 3.5*_salary;
}
```

# Exercise

- Program a function that calculates pays for 160 hours of work per month.
  - Can we use write one function working either fro Employees and Managers?

# Exercise

- P function that calculates pays f 100 h s
  month.
  e write one function work o

Take your time and solve it!

# Solution

```
float calculatePay(Employee &e)
{
     float pr = e.getSalary();
     return pr*160;
}
```

**Can we use this function too for Managers?**

# Solution

```
Employee emp;
Manager man;
float empPay, manPay;
…
empPay = calculatePay(emp);
manPay = calculatePay(man);
```

# Solution

- **How it works?**

```
manPay = calculatePay(man);
                          ↓ "IS A" relationship
float calculatePay(Employee &e)
{                    ↓
     pr = e.getSalary(); → man.getSalary();
     return pr*160;
}
```

**A perfect match between the two virtual functions exist, so they can be exchanged!**

# Example

- **What if the getSalary() function is not virtual?**

# Example

```
class Employee {
public:
  string getName() const;
  float getSalary() const;
  void setNameAndRate(const string &name,
                    const float &salary);

protected:
  string _name;
  float _salary;
};
```

# Example

- **What happens?**

```
manPay = calculatePay(man);
                              "IS A" relationship
float calculatePay(Employee e)
{
     pr = e.getSalary(); X ~~man.getSalary();~~
     return pr*160;
}
```

**We will always get the lower pay rate!**

# Outline

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Functions Overloading

- You can have multiple definitions for the same function name in the same scope.
  - The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
  - The idea is the same applied to multiple constructors
- You can not overload function declarations that differ only by return type.

# Functions Overloading

```
class Rectangle {
public:
  Rectangle();
  Rectangle(const double &w,
               const double &l);
  Rectangle(const double &w_l);
  ~Rectangle() {};
   void setW(const double &w);
   void setW(const int &w);
   void setL(const double &l);
   void setL(const int &l);
...
```

overloaded functions

# Functions Overloading

```
int main() {
  ...
  Rectangle r5, r6;
  r5.setW(2);
  r5.setL(4);
}
```

# *Outline*

- Copy Constructor
- Composition: Objects as member of classes
- The this keyword
- Polymorphism sets to practice
- Functions Overloading
- Operators Overloading

# Operators Overloading

- **What is an operator?**
  - **For each basic types you (might) have already seen:**
    1. **Assignment operator (=)**
    2. **Arithmetic operators ( +, -, *, /, % )**
    3. **Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)**
    4. **Increment and decrement (++, --)**
    5. **Relational and comparison operators ( ==, !=, >, <, >=, <= )**
    6. **Logical operators ( !, &&, || )**
    7. **Conditional ternary operator ( ? )**
    8. **Comma operator ( , )**
    9. **Bitwise operators ( &, |, ^, ~, <<, >> )**
    10. **....**

# Operators Overloading

- **If I need that for my own classes, would it make sense?**

```
int main() {
   ...
   Rectangle r4(r3);
   …
   Rectangle r5, r6;
   ...
   r6 = r5 + r4;
   r6.uguale(r5.somma(r4));
}
```

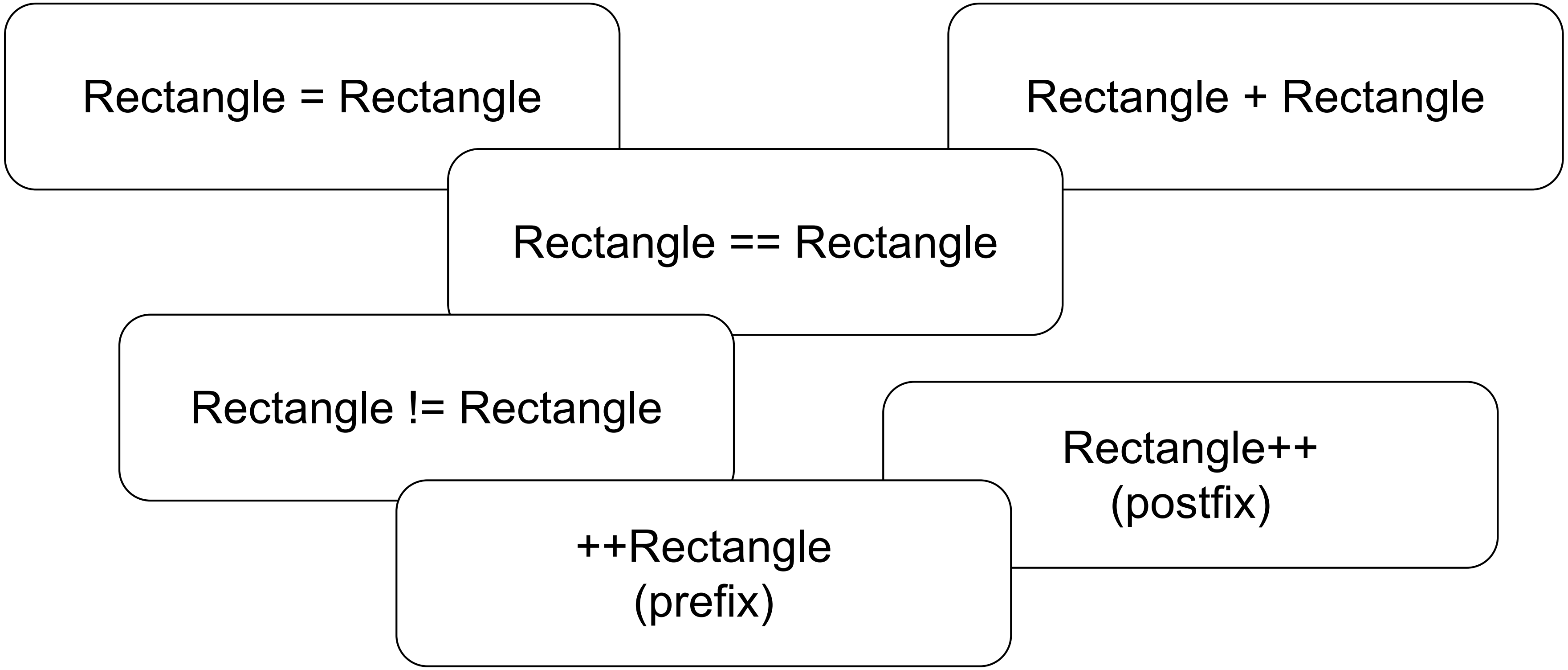
# Operators Overloading

- **If I need that for my own classes, would it make sense?**

```
int main() {
   ...
   Rectangle r4(r3);
   …
   Rectangle r5, r6;
   ...
   r6 = r5 + r4;        ?
}
```

# Operators Overloading

- **Not all operators make sense applied to classes (and objects).**
  - **Still you can implement what you might need**



Rectangle = Rectangle

Rectangle + Rectangle

Rectangle == Rectangle

Rectangle != Rectangle

Rectangle++
(postfix)

++Rectangle
(prefix)

# Operators Overloading (How to)

- **You need to declare them in the Class definition as (public) methods.**

```
class Rectangle {
public:
   ...
   Rectangle operator+(const Rectangle &to_be_added);
   void operator=(const Rectangle &to_be_assigned);
   const Rectangle& operator++(); // prefix
   const Rectangle operator++( int ); // postfix
   bool operator==(const Rectangle &to_be_compared);
   bool operator!=(const Rectangle &to_be_compared);
   ...
}
```

# Operators Overloading (How to)

- **You need to declare them in the Class definition as (public) methods.**

Wait... This can be further generalized!

# Operators Overloading (How to)

- **A generic T class can implement its own operators to fulfil any design requirements.**

```
class T {
public:
   ...
   T operator+(const T &to_be_added);
   void operator=(const T &to_be_assigned);
   const T& operator++(); // prefix
   const T operator++( int ); // postfix
   bool operator==(const T &to_be_compared);
   bool operator!=(const T &to_be_compared);
   ...
}
```

# Operators Overloading (How to)

- **A generic T class can implement its own operators to fulfil any design requirements.**

T is a general class you are willing to create!

# Operators Overloading (How to)

- **Their form is (almost) forced to the semantic and syntax already defined by the language**

```
class T {
public:
   ...
   T operator+(const T &to_be_added);
   void operator=(const T &to_be_assigned);
   const T& operator++(); // prefix
   const T operator++( int ); // postfix
   bool operator==(const T &to_be_compared);
   bool operator!=(const T &to_be_compared);
   ...
}
```

# Operators Overloading (How to)

- **Notice the const  keyword in the parameters...**
- **... And the referenced parameters...**
- **... And all the return types**

```
class T {
public:
   ...
   T operator+(const T &to_be_added);
   void operator=(const T &to_be_assigned);
   const T& operator++(); // prefix
   const T operator++( int ); // postfix
   bool operator==(const T &to_be_compared);
   bool operator!=(const T &to_be_compared);
   ...
}
```

# Operators Overloading (How to)

- **In the implementation private members of parameters can be accessible.**

```
void Rectangle::operator=(const Rectangle
&to_be_assigned) {
 this->m_width = to_be_assigned.m_width;
 this->m_length = to_be_assigned.m_length;
}

Rectangle Rectangle::operator+(const Rectangle
&to_be_added) {
  Rectangle output;
  output.m_width = this->m_width + to_be_added.m_width;
  output.m_length = this->m_length +
to_be_added.m_length;
  return output;
}
```

# Operators Overloading (How to)

- **Methods can call each other.**

```
bool Rectangle::operator==(const Rectangle
&to_be_compared) {
   return ((m_width == to_be_compared.m_width) &&
           (m_length == to_be_compared.m_length));
}

bool Rectangle::operator!=(const Rectangle
&to_be_compared) {
   return !(*this == to_be_compared);
}
```

Notice the (*this) usage and how != is implemented through ==

# Operators Overloading (How to)

- **They should mimic the original operator behavior as much as possible**

```
const Rectangle& Rectangle::operator++() {
   m_width++;
   m_length++;
   return *this;
}

const Rectangle Rectangle::operator++( int ) {
   Rectangle R(*this);
   ++(*this);
   return R;
}
```

Notice the **this* usage here: you are returning a new object ”copy” of the actual one

# Operators Overloading (How to)

- **They should mimic the original operator behavior as much as possible**

```
const Rectangle& Rectangle::operator++() {
   m_width++;
   m_length++;
   return *this;
}

const Rectangle Rectangle::operator++( int ) {
   Rectangle R(*this);
   ++(*this);
   return R;
}
```

Notice both the ”copy before increment” and the re-usage of prefix version to shorten up the code

Малые Автюхи, Калинковичский район, Республики Беларусь