Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Deep vs. shallow copy

Methods created by the compiler

- We have seen that the compiler creates for each class a default constructor and destructor…
- … but it creates also default copy constructor and assignment operator, to create a new object cloning another one or copying its attributes.
    - C++11 compiler may create other methods, not discussed in this lecture.

Copy methods

- C++ treats variables of user-defined types with value semantics. This means that objects are implicitly copied in various contexts.
- The **copy constructor** creates a fresh object based on the state of an existing object.
- The **assignment operator** copies the state of a **source** object onto an existing **target** object, thus it has to work a bit more to deal with the already existing target.

The compiler

- According to the C++ standard:
  - The compiler will implement copy constructor and assignment operator if they are used but not provided by the programmer (implicit definition)
  - The implicitly-defined copy constructor for a non-union class X performs a memberwise copy of its sub-objects.
  - The implicitly-defined copy assignment operator for a non-union class X performs memberwise copy assignment of its subobjects.

Shallow copy

- The implicit methods copy the content of the attributes of the class, i.e. they perform a bit-by-bit copy of their content (shallow copy).
- Is this enough to get a working copy of our object ?

Shallow copy example

```
class GameCharacter {

public:
    GameCharacter(std::string& name,
        int hp) : name(name), hp(hp)
        {  }

private:
    std::string name;
    int hp;

};
```

```
// Implicit methods created by the
// compiler. The default constructor
// is not created because of the
// constructor provided by us.

// 1. copy constructor
GameCharacter(GameCharacter& that) :
     name(that.name), hp(that.hp) { }

// 2. copy assignment operator
GameCharacter& operator=(GameCharacter&
                          that) {
    name = that.name;
    hp = that.hp;
    return *this;
}

// 3. destructor
~GameCharacter() {
}
```

Shallow copy example

```
class GameCharacter {

public:
    GameCharacter(std::string& name,
        int hp) : name(name), hp(hp)
        { }

private:


}
```

Memberwise copy. The attributes are copied bit-by-bit

```
// Implicit methods created by the
// compiler. The default constructor
// is not created because of the
// constructor provided by us.

// 1. copy constructor
GameCharacter(GameCharacter& that) :
    name(that.name), hp(that.hp) { }

// 2. copy assignment operator
GameCharacter& operator=(GameCharacter&
                          that) {
    name = that.name;
    hp = that.hp;
    return *this;
}

// 3. destructor
~GameCharacter() {
}
```

Shallow copy example

```
class GameCharacter {

public:
    GameCharacter(std::string&
        int hp) : name(name),
        {  }

private:

}
```

Typically these are "constant" arguments. We'll see what is `const` in a future lecture.

Memberwise copy. The attributes are copied bit-by-bit

```
// 1. copy constructor
GameCharacter(GameCharacter& that) :
    name(that.name), hp(that.hp) { }

// 2. copy assignment operator
GameCharacter& operator=(GameCharacter&
                          that) {
    name = that.name;
    hp = that.hp;
    return *this;
}

// 3. destructor
~GameCharacter() {
}
```

Deep copy

- If an attribute of the class is a pointer, e.g. to an array the bit-by-bit copy is not enough, since it results in copying the address and not the objects that are pinted.
- We need a **deep** copy that copies all the objects pointed, or we risk that the destruction of a copied object destroys the original.

Deep copy

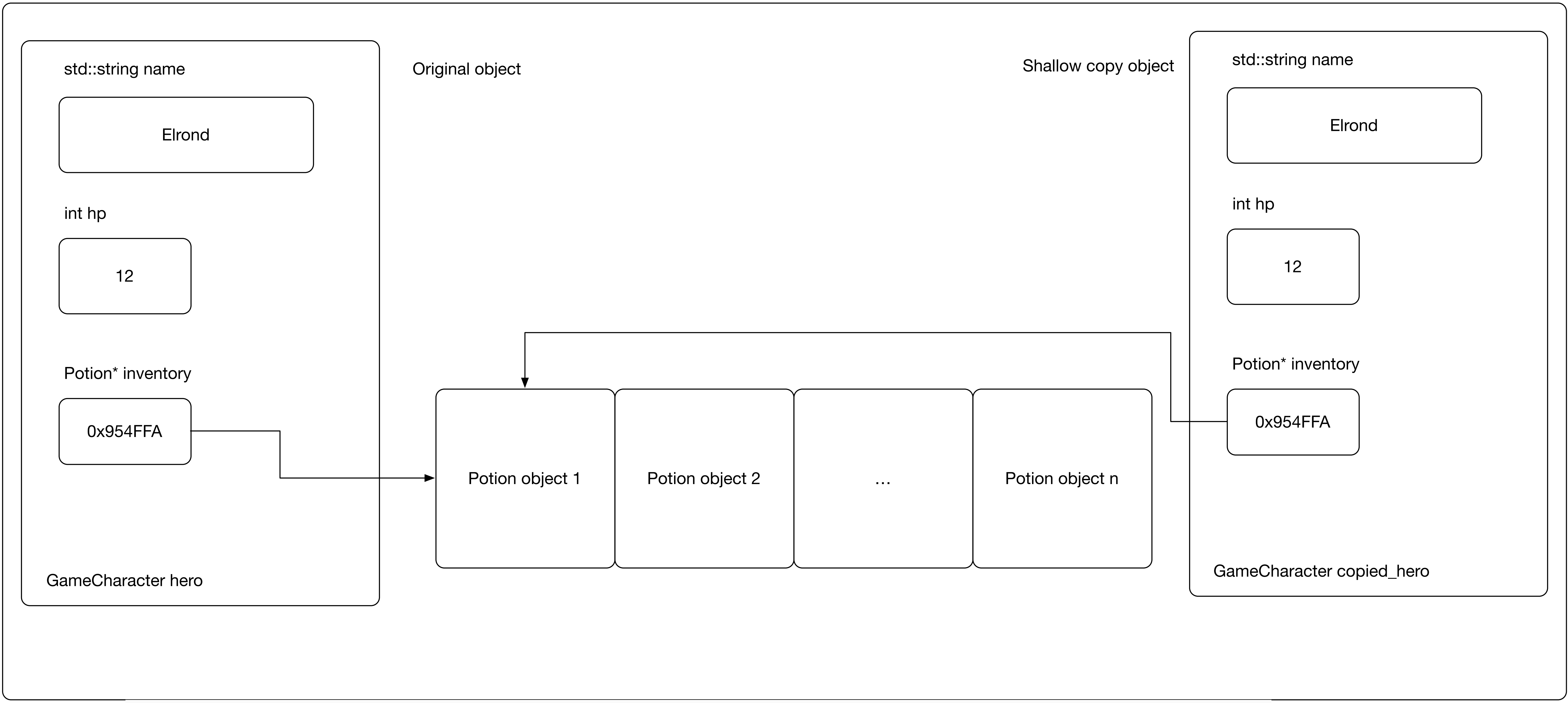
- If an attribute of the class is a pointer, e.g. to an array the bit-by-bit copy is not enough, since it results in copying the address and not the objects that are pinted.
- We need a **deep** copy that copies all the objects pointed, or we risk that the destruction of a copied object destroys the original.

Think of a bad photocopier that creates magical copies that once destroyed cause automatic destruction of the original…

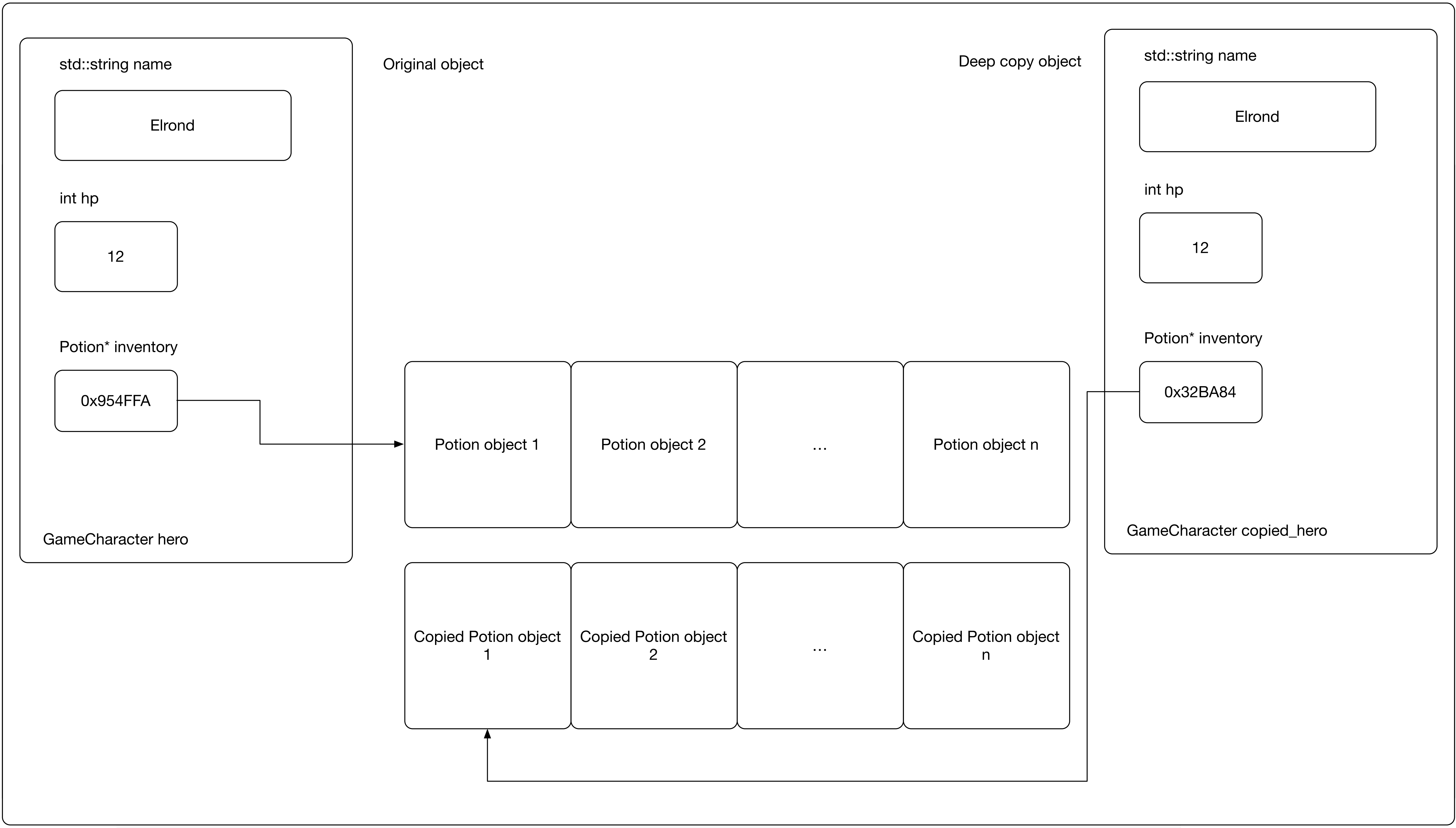
Shallow copy problem

```
class GameCharacter {
public:
   // ...
   ~GameCharacter() {
      delete[] inventory;
   }

private:
   std::string name;
   int hp;
   Potion* inventory;

};
```

Shallow copy problem



If the shallow copy object `copied_hero` gets destructed then also the original object `hero` loses its inventory



A deep copied object causes no issue to the source: it has its own copy of the resources.

“Rule of three”

- When our class manages a resource, i.e. when an object of the class is responsible for that resource, then we need to declare explicit methods for copying and creating objects from other objects.
  Typically the resource is acquired in the constructor (or passed into the constructor) and released in the destructor.
- Implement copy constructor, destructor and assignment operation.

How to create the methods ?

- Both copy constructor and assignment operator receive a reference to the original (source) object.
    - Actually a `const` reference…
    - The methods share a lot of code. Think about factorizing it in an helper method.
- The destructor should release the resource
- The operator returns a reference, to allow multiple assignments. The operator must handle the existing resources of the target object and avoid self assignment.

Disable copy/cloning

If you do not want to allow copying an object disable the copy constructor and assignment operators with C++11 =`delete` method syntax:

```
class Foo {
public:
    Foo& operator=(const Foo&) = delete;
      // disallow use of assignment operator
    Foo(const Foo&) = delete;
      // disallow copy construction
};
```

Reading material

- L.J. Aguilar, “Fondamenti di programmazione in C++. Algoritmi, strutture dati e oggetti” - cap. 12, pag. 381-383
- D.S. Malik, “Programmazione in C++” - cap. 10, pag. 510-512
- Thinking in C++, 2nd ed. Volume 1, cap. 11, pag. 479-497

Credits

- These slides are based on the material of:
  - D.S. Malik, “Programmazione in C++”