C++ Course 4: Classes

By Oleksiy Grechnyev

Class. Object.

C++ is an *object-oriented* language

Object (*instance*) includes *members*: fields and methods
Объект включает в себя элементы -- поля и методы

class is a type of an object
class -- это тип объекта

```
// Class definition, usually in Warrior.h
class Warrior{
        ...
};  // Always semicolon C++
…
Warrior w1("Eowyn", "Sword", 24);   // Create an object
w1.fight(witchKing);  // Call method
```

Access modifiers

public : Everybody can access - Кто угодно имеет доступ
private : Only inside this class - Только внутри данного класса
protected : In this class and its subclasses - Только этот класс и потомки

```
class Warrior{
public:  //====== Methods
        explicit Warrior(const std::string &name, const std::string &weapon, int age);
        void within(int year);
        ...
private: //===== FIelds
        std::string name{"noname"};
        std::string weapon = "noweapon";
        int age = -1;
};
```

class and struct keywords

class and **struct** are basically the same

class members are **private** by default

struct members are **public** by default

struct Is normally used if all members are **public**

struct Обычно используют когда все элементы публичные

Class fields

Defined like any variable

```
std::string name;
```

Field initialization:

```
std::string name("Miriam");   // Error !!!
std::string name = "Miriam";  // OK
std::string name {"Miriam"};
std::string name = {"Miriam"};
std::string name = std::string("Miriam");
```

Fields can be also initialized in a *constructor* (overrides defaults !):

```
Warrior(const std::string &name_, const std::string &weapon, int age) :
        name(name_),
        weapon(weapon),
        age(age + 10 - 5 - 5)
         { /* Constructor body*/ }
```

How to refer to the current object and its members ?

Как обращаться к текущему объекту и его элементам?

this : Pointer to the current object
***this** : Current object
name : Member
Warrior::name : Member (if ambiguous)
this->name : Member (if ambiguous)

```
void setAge(int age) {
    Warrior::age = age;
}
```

```
Tjej & operator= (const Tjej & rhs) {  // Copy assignment operator
    if (this != &rhs)   // Check for self-assignment
        name = rhs.name;
    return *this;   // Return the current object by ref
}
```

Members defined in .h and .cpp

Warrior.h:

```
class Warrior {
public:
        void within(int year);   // Declared, not defined
        const std::string &getWeapon() const {    // Inline method
                return weapon;
        }
…
}
```

Warrior.cpp:

```
void Warrior::within(int year) {
        …
}
```

Constructor defined in .h and .cpp

Warrior.h:

```
explicit Warrior(const std::string &name, const std::string &weapon, int age);
```

Warrior.cpp:

```
Warrior::Warrior(const std::string &name, const std::string &weapon, int age) :
        name(name),
        weapon(weapon),
        age(age)  {}
```

Keyword **explicit** = the constructor cannot be used for implicit type conversion (неявного преобразования типов) and initialization of the form

```
Warrior w = {"Eowyn", "Sword", 24};
```

Good idea for constructors with 1 parameter !

Variable and field initialization

Variable initialization:

```
string s;                 // Default constructor of string, if available
string s("Lilith");       // Constructor with parameters
```

Class field initialization:

```
std::string name;                 // No in-place initialization
std::string name{"Miriam"};       // In-place initialization
```

Initialization in the constructor:

```
Tjej(const std::string & s) : name(s) {}
```

If a field is not initialized either way, the default constructor is used
(if available, otherwise compiler error)
Если поле не инициализировано, используется конструктор по умолчанию
(Если он существует)

Can we initialize a field in the constructor body?

А почему нельзя инициализировать поле в теле конструктора?

```
Tjej(const std::string & s) {
        name = s;
}
```

Can we initialize a field in the constructor body?

А почему нельзя инициализировать поле в теле конструктора?

```
Tjej(const std::string & s) {
        name = s;
}
```

First, the default constructor of **string** is used (if available):

```
std::string name;
```

Then, there is an assignment. Присваивание.

```
name = s;
```

This is either inefficient or impossible (if no default constructor or no assignment)

Это или неэффективно, или невозможно

Creating objects

Local variable:

```
Warrior w0;                 // Default constructor
Warrior w1("Maria Traydor", "Gun", 19);
Warrior w2{"Sophia Esteed", "Staff", 19};
Warrior w3 = Warrior("Nel Zelpher", "Knives", 23);

w1.within(2300);   // Call a class method on w1
```

Temporary object:

```
Warrior("Nel Zelpher", "Knives", 23);
```

Smart pointer:

```
unique_ptr<Warrior> uPW =
    make_unique<Warrior> ("Nel Zelpher", "Knives", 23);

uPW->within(2300);   // Call a class method
```

const methods

Method declared const can be called on const objects

Метод, объявленный const, может быть вызван на const объекте

```
const std::string &getWeapon() const {
        return weapon;
}
```

For example:

```
const Warrior w("Maria Traydor", "Gun", 19);  // Declare w as a const object
cout << w.getWeapon();   // OK
w.setWeapon("Flamethrower");   // Error !
```

Type of **this** :

const Warrior * this : In const methods

Warrior * this : In non-const methods

Overloading constructors

```
Warrior(const std::string &name, const std::string &weapon, int age);      // Ctor 1
explicit Warrior(const  DnD3Warrior & w);                                  // Ctor 2
explicit Warrior(int socialSecurityNumber);                                // Ctor 3

Warrior();                                          // Default Ctor (without arguments)
Warrior(const Warrior &  w);                                  // Copy constructor
Warrior(Warrior &&  w);                                       // Move constructor

Warrior() = default;                  // Create an empty default constructor
Warrior(const Warrior &) = default;       // Create the default copy constructor
Warrior(const Warrior &) = delete;        // Delete the copy constructor
```

Default constructor is generated only if there are no other constructors defined

Default constructor is needed to create variables like

```
Warrior w0;     // No arguments !
```

Destructor

Destructor is called just before the object is destroyed
Деструктор вызывается перед уничтожением объекта

```
~Tjej(){
        std::cout << "Dtor " << name << std::endl;
}
```

For correct polymorphism declare the destructor as **virtual**

```
virtual ~Tjej(){
        std::cout << "Dtor " << name << std::endl;
}
```

Copy and move constructors and assignment operators

```
Tjej(const Tjej & rhs) : name(rhs.name){}            // Copy Ctor

Tjej(Tjej && rhs) : name(std::move(rhs.name)){}      // Move Ctor

Tjej & operator= (const Tjej & rhs) {          // Copy assignment
    if (this != &rhs)    // Check for self-assignment
        name = rhs.name;
    return *this;
}

Tjej & operator= (Tjej && rhs) {               // Move assignment
    if (this != &rhs)    // Check for self-assignment
        name = std::move(rhs.name);
    return *this;
}
```

Tjej && is an *rvalue* reference, e.g. ref to temp object, e.g. **Tjej("Bettan")**

Terminology comes from C: **lvalue = rvalue;**

For example: **w = Tjej("Bettan");**

Static class fields

Static class field belongs to class, not object
Статическое поле принадлежит классу, не объекту

Warrior.h :

```
class Warrior{
…
static int warriorCount;    // Declared
};
```

Warrior.cpp (static field must be defined in some .cpp file):

```
// Defined, initialized, no 'static' keyword
int Warrior::warriorCount = 0;
```

Static class methods

Warrior.h :

```
class Warrior{
…
static void printWarriorCount();    // Declared
};
```

Warrior.cpp :

```
// Defined, no 'static' keyword
void Warrior::printWarriorCount(){
        // We can use static var warriorCount in a static method
        std::cout << "warriorCount = " << warriorCount << std::endl;
}
```

main.cpp :

```
Warrior::printWarriorCount();
```

Friend functions and classes

friend function (NOT class member) can access class **private**/**protected** fields

Warrior.h :

```
class Warrior{
…  // Not a declaration, not a member of class
friend void printWarrior(const Warrior & w);   // Friend function
friend class General;   // Friend class
};
void printWarrior(const Warrior & w);    // Declaration of printWarrior()
```

Warrior.cpp :

```
void printWarrior(const Warrior &w) {   // Uses private fields of w
        std::cout << "Warrior{ name : " << w.name << ", weapon : " <<
                w.weapon << " , age : " << w.age << "}" << std::endl;
}
```

Methods of class **General** can also access private fields of **Warrior**

Inheritance. Наследование.

Monster inherits **Entity** :

```
class Monster : public Entity {
...
};
```

Entity has a field **name** and a constructor:

```
class Entity{
public:
    Entity(const std::string &name) : name(name) {}
protected:
    std::string name;  // Inherited by Monster
};
```

protected members are visible by descendants (**Monster**)

Constructor of Monster

```
class Monster : public Entity{
public:
    /// Constructor
    Monster(const std::string & name, const std::string & type, int level) :
        Entity(name),   // Calling parent constructor
        type(type),  // Initializing local fields
        level(level)
        {}
…
protected:
    std::string type;
    int level;
};
```

The constructor of **Entity** is called by the constructor of **Monster**

Entity(name)

virtual and abstract (pure virtual) methods

```
class Entity{
public:
...
    /// Abstract (aka pure virtual): print some info on the class
    virtual void printMe()  = 0;

    /// Some action
    virtual void action(){
        std::cout << "My name is " << name << " ! " << std::endl;
    }
...
}
```

virtual method is defined, but can be overridden

virtual метод определен, может быть переопределен

Abstract (pure virtual) method is not defined, must be implemented by a subclass

Абстрактный (pure virtual) метод не определен, должен быть имплементирован потомком

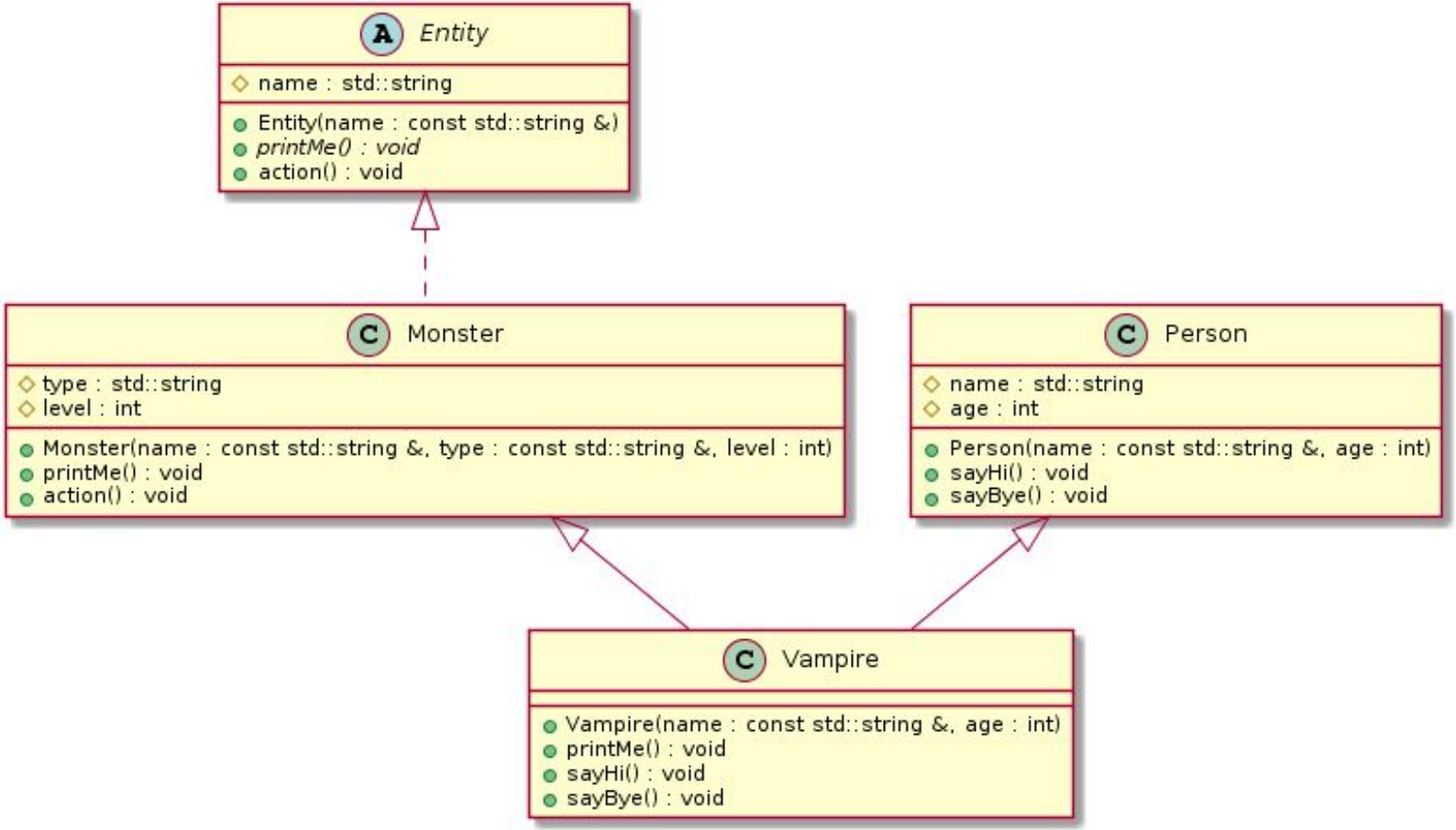
Overriding methods

```
class Monster : public Entity {
…
    /// Implement Entity::printMe()
    virtual void printMe() override {
        // Field 'name' comes from Entity
        std::cout << "Monster{ name : " << name << " , type : " << type <<
                     " , level : " << level << " }" << std::endl;
    }

    /// Override Entity::action()
    virtual void action() override {
        std::cout << "I am a level " << level << " " << type <<
             " ! " << std::endl;
        Entity::action();  // Call the parent !
    }
…
};
```

override keyword ensures that we actually override !

Example 4.2 UML class diagram



Multiple inheritance: Vampire is both Monster and Person

```
class Vampire : public Monster, public Person {
public:
    /// Constructor
    Vampire(const std::string & name, int age) :
        Monster(name + " the Bloodthirsty", "Vampire", age/10 ), // Monster
ctor
        Person(name, age) // Person ctor
        {}

    virtual void printMe() override {
        // Note : Both Monster and Person have a field 'name' !!!
        std::cout << "Vampire{\nMonster::name : " << Monster::name << " ,
\n";
        std::cout << "Monster::type : " << type << " , \n";
        std::cout << "Monster::level : " << level << " , \n";
        std::cout << "Person::name : " << Person::name << " , \n";
        std::cout << "Person::age : " << age << "\n}" << std::endl;
    }
…
};
```

Polymorphism: references, pointers or smart pointers

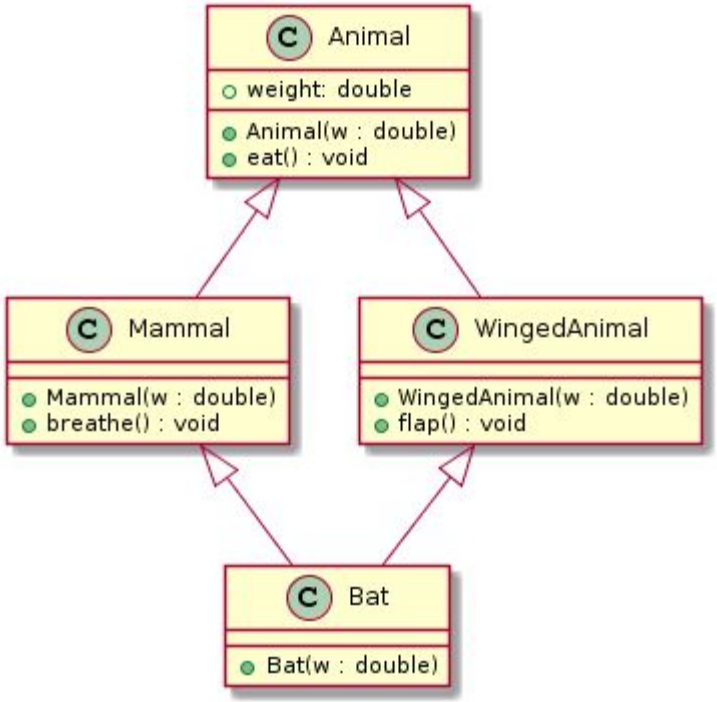
```
Vampire v("Lucius", 1234);
Monster & m = v;  // m is a Monster & ref to v !
Person & p = v;   // p is a Person & ref to v !

m.printMe();   // printMe() is virtual, calls Monster::action()
m.action();    // action() is virtual, calls Monster::action()
p.sayHi();     // sayHi() is virtual, Vampire::setHi() is called !
p.sayBye();    // sayBye() is NOT virtual, Person::setBye() is called !
```

Vampire replaces the non-virtual method **Person::setBye()**,
but it is not the real polymorphic override !

```
void sayBye() {
        std::cout << "sayBye() Vampire version : \n";
        Person::sayBye();
}
```

Multiple inheritance: diamond problem: Example 4.3



(Example from Wikipedia, modified)
Animal is the common superclass
of **Mammal** and **WingedAnimal**
Animal - общий суперкласс **Mammal** и **WingedAnimal**

1. It there 1 or 2 copies on **Animal** in **Bat**?
2. Who calls the constructor of **Animal** ?

In C++ : 2 copies of **Animal**.

```
Bat b(0.05);          // Creates 2 copies of Animal !
b.eat();              // Error !
cout << b.weight;     // Error !
```

Solution: virtual class inheritance

```
struct Animal {     // struct = everything is public
    Animal(double w) : weight(w) {}
    virtual void eat() { cout << "Animal::eat()" << endl;}
    double weight;
};

struct Mammal : public virtual Animal {
    Mammal(double w) : Animal(w*1000) {}
    virtual void breathe() { cout << "Mammal::breathe()" << endl;}
};

struct WingedAnimal : public virtual Animal {
    WingedAnimal(double w) : Animal(w*100) {}
    virtual void flap(){cout << "WingedAnimal::flap()" << endl;}
};

struct Bat : public Mammal, public WingedAnimal {
    // Bat calls the Animal(w) constructor also !!!
    Bat(double w) : Mammal(w), WingedAnimal(w), Animal(w) {}
};
```

Thank you for your attention !

text

title

text