C++ Course 4: Classes 1

2020 by Oleksiy Grechnyev

Class. Object.

C++ is an *object-oriented* language

Object (*instance*) includes *members*: fields and methods

Class is a type of an *object*

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

Access modifiers

public : Everybody can access
private : Only inside this class
protected : In this class and its subclasses

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

class and struct keywords, Ctor/Dtor

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 (fields included) are **public** (Data structure):

```
struct Point3D {
	double x, y, z;
};
```

Special methods:

Constructor (Ctor) is a method which initializes an object.

It typically initializes *class fields*.

Destructor (Dtor) is a method which cleans up object before it is deleted.

Dtor is called automatically when it is time for the object to die.

It is rarely used in user-defined classes.

E.g.: If Ctor allocates something with **new**, Dtor must delete it with **delete**!

Note: Ctor/Dtor do not create/delete object!

Class fields

Defined just like any (local) variable

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

In-place field initialization:

```
std::string name("Miriam");   // Error !!! Looks like method definition !
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* (Priority: Ctor overrides in-place !):

```
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 ?

this : Pointer to the current object
***this** : Current object (Reference)
name : Member
Warrior::name : Member (if ambiguous, preferred !)
this->name : Member (if ambiguous, old style)

/// Example : Setter (Change a private/protected field)

```
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) {
	...
}
// Warrior::within() is compiled only once into a file Warrior.o
// Methods defined in Warrior.h are compiled for each .cpp which uses them!
```

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 and initialization of the form

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

Good idea for constructors with 1 parameter !

Variable and field initialization

Variable initialization (default Ctor = Ctor with no parameters):

```
string s;                // Default constructor of string, if available, error otherwise !
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 (overrides in-place initialization !):

```
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)
Primitive types (**int**, **double**, ...) : local vars are *not initialized*,
fields and global vars are *zero-initialized*. But initialize everything yourself !

Can we initialize a field in the constructor body?

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

Can we initialize a field in the constructor body?

```
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 (a REAL copy/move assignment).

```
name = s;
```

For class types, this is either inefficient or impossible
(no default constructor and no initialization = error !)

Ctor field initialization, on the other hand, avoids the default ctor + assignment:
A single copy/move constructor call !

Creating objects

Local variable (Stack object, lives until the closing '}' of the block):

```
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 (Stack object, lives in the current code line only):

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

Smart pointer (Heap Object, dies when uPW dies or is reassigned):

```
unique_ptr<Warrior> uPW =
    make_unique<Warrior> ("Nel Zelpher", "Knives", 23 );
uPW->within(2300);   // Call a class method
```

Raw pointer (Heap Object, lives until you **delete** it!):

```
Warrior * pW = new Warrior("Nel Zelpher", "Knives", 23 );
pW->within(2300);   // Call a class method
delete pW;    // Don't forget to delete the object when not needed anymore !!!
```

const methods

Only method declared **const** can be called on **const** objects

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

const methods cannot modify class fields (except **mutable**) and call non-**const** methods.

As usual, **const** can be hacked by type casts, but don't do that (bad style) !

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** inside a method body:

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
```

Special constructors (sometimes auto-generated by the compiler):

```
Warrior();                           // Default (no-param) constructor
Warrior(const Warrior &  w);         // Copy constructor
Warrior(Warrior &&  w);              // Move constructor
```

Default (i.e. no-param) constructor is generated only if there are no other constructors defined

Default (i.e. no-param) constructor is needed to create variables/fields like

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

Using **default** and **delete** (NB: confusing term *default*):

```
Warrior() = default;                        // Create an empty ("default") default (no-param) constructor
Warrior(const Warrior &) = default;         // Create the default copy constructor
Warrior(const Warrior &) = delete;          // Delete the copy constructor
```

Delete copy constructors for objects which must not be copied ! Important!

Destructor

Destructor is called just before the object is destroyed
It cleans up the object, but does NOT "destroy the object" !!!

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

Note: destructors of all fields are called *after* the class destructor.

For correct polymorphism (if using inheritance) declare the destructor as **virtual**

```
virtual ~Tjej(){   // In the parent class !
	std::cout << "Dtor " << name << std::endl;
}
```

Then **delete** *always* calls the correct destructor. Otherwise wrong destructor can be called!

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");**

The rule of five (previously "three")

Normally, class has the following default methods (auto-created by the compiler)

- copy constructor (copy all fields)
- move constructor (move all fields)
- copy assignment (copy all fields)
- move assignment (move all fields)
- destructor (empty)

Rule of five: If you implement *any* of these methods, the defaults are no longer generated, you will have to implement *all* of them. Idea: you might have custom memory management, or some external resources (streams, sockets), thus default behavior is wrong.

Use **default** + **delete** to override this behavior if needed.

Note: This has nothing to do with the default (no-param) Ctor, like **Warrior()**.
It is generated if and only if there are no other (explicitly defined) constructors.

Static class fields

Static class field belongs to *class*, not *object*

Warior.h :

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

Warrior.cpp (static field must be always defined in some .cpp file, and only one !):

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

Note: Static fields are essentially *global variables*, avoid them like plague! But:

```
static constexpr double xi = 0.123;   // Define constant in a class, OK
```

Static class methods

Static methods cannot access class fields or **this** ! There is no object!

Warrior.h :

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

Warrior.cpp :

```
// Defined, no 'static' keyword
void Warrior::printWarriorCount(){
	// We can use static field 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) {   // Can access 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**

Class templates : A trivial container

Let us create a trivial container, which can store a value of any type **T** :

```
template <typename T>
class Box{
public:
    Box(const T &val) : val(val) {}    /// Ctor
    void setVal(const T &val) {        /// Setter
        Box::val = val;
    }
    const T & getVal(){            /// Getter
        return val;
    }
private:
    T val;    /// The value
};
```

Template instantiation : Create a class for particular **T**, e.g. **Box<int>**

This happens at compile time, separate binary code for every **T** in each .cpp file

All methods must be fully implemented in a .h file ! No .cpp for templates !

Class templates : A trivial container

We can now create **Box<T>** objects for different **T** :

```
// Box of ints
Box<int> a(13);
a.setVal(17);
cout << "a.getVal() = " << a.getVal() << endl;

// And a box of strings
Box<string> b("Rutabaga");
cout << "b.getVal() = " << b.getVal() << endl;
b.setVal("Turnip");
cout << "b.getVal() = " << b.getVal() << endl;

// Note: Type T must be always specified (unlike for function templates)
```

C++ containers **std::array**, **std::vector** etc. are built like this!

(But a bit more advanced)

Inheritance.

Monster inherits **Entity** (all public members of **Entity** are inherited as public) :

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

Entity has a **protected** field **name** and a public 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)

Constructors are NEVER inherited !

virtual and abstract (pure virtual) methods

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

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

virtual method is defined, but can be overridden

Abstract (pure virtual) method is not defined, must be implemented by non-abstract subclasses

Class with an abstract method is an abstract class, cannot create objects of it

Classes with inheritance (any **virtual** method !) introduce time/space overheads!

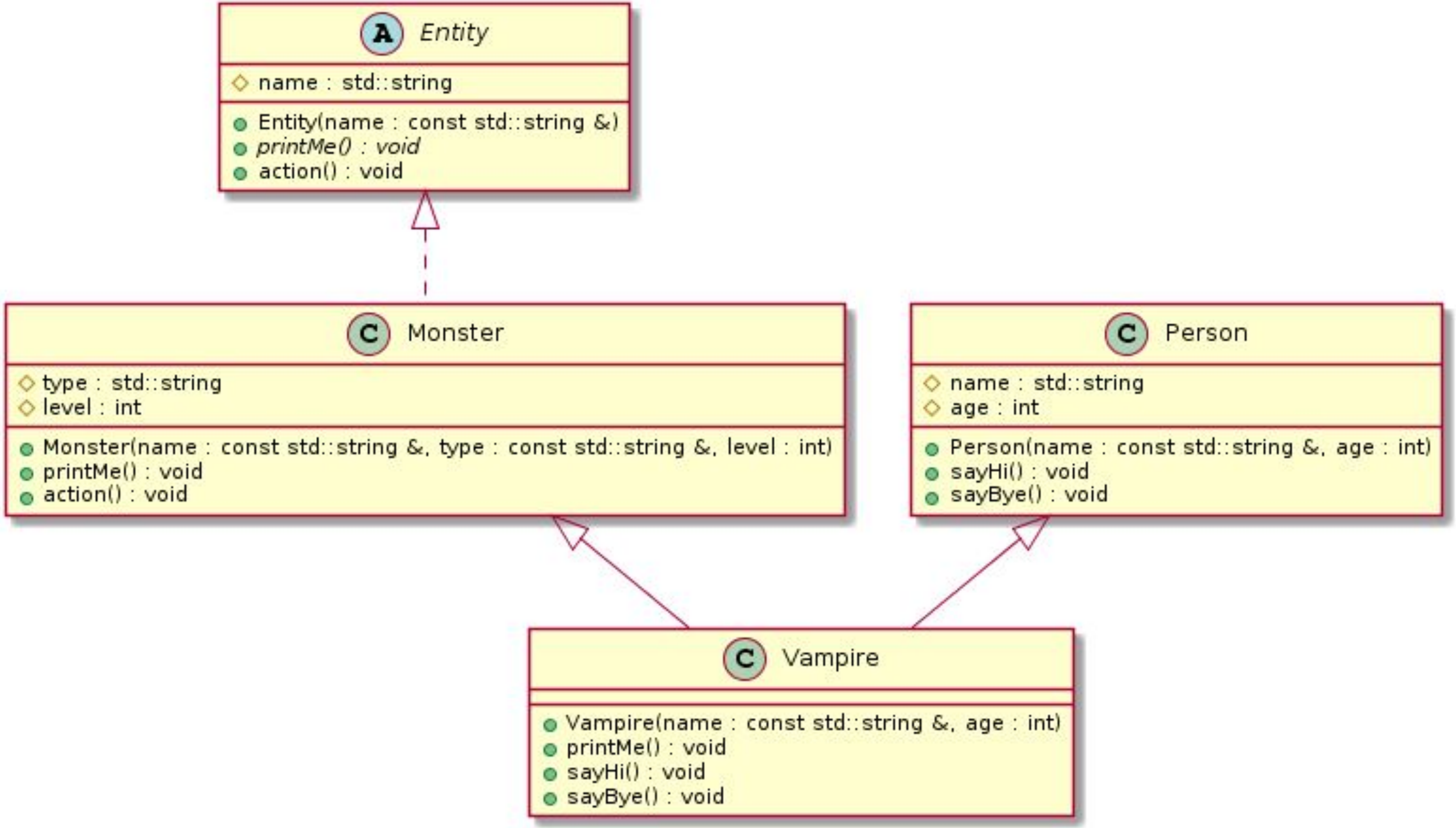
Overriding methods

```
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 !

Tool of the day: PlantUML : Draw UML class diagrams



Tool of the day: PlantUML : *.puml file

```
@startuml

abstract class Entity{
# name : std::string
+ Entity(name : const std::string &)
+ {abstract} printMe() : void
+ action() : void
}

class Monster {
# type : std::string
# level : int
+ Monster(name : const std::string &, type : const std::string &, level : int)
+ printMe() : void
+ action() : void
}
...
@enduml
```

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;
    }
…
};
```

Extra Slides: Diamond problem, virtual inheritance.

Polymorphism: references, pointers or smart pointers

Polymorphism in object oriented programming:
A **Child** object is also a **Parent** object.
Functions which operate on **Parent** can also operate on **Child**.

In C++: copy operations are no good for polymorphism.
Polymorphism works for references, pointers, and **shared_ptr**.

Vampire v("Lucius", 1234); // Vampire is both Monster and Person
Monster & m = v; // m is a Monster & ref to v ! Every vampire is a monster!
Person & p = v; // p is a Person & ref to v ! Every vampire is a person!

C++ actually knows that object **m** is actually **Vampire**, and not **Monster**!
This means there is a hidden class field like "class ID" (dynamic type system),
which introduces minor space+time overheads.
Only classes with inheritance (with **virtual** methods) have those overheads.

Polymorphism and virtual methods

C++ knows that object **m** (type **Monster &**) is actually **Vampire**, and not **Monster**!
virtual methods can be overridden. The correct (**Vampire**) method is called even from parent class reference (**Monster** or **Person**) !

m.printMe(); // printMe() is virtual, calls Vampire::printMe(), not Monster::PrintMe() !!!
m.action(); // action() is virtual, calls Monster::action() as Vampire does not override it
p.sayHi(); // sayHi() is virtual, Vampire::setHi() is called !

Virtual methods: method is *always* determined by the class *of the object* (**Vampire**).
But what about non-virtual methods?
Non-virtual methods: method is *always* determined by the class *of the reference* (**Person**).

Suppose **Vampire** replaces the non-virtual method **Person::setBye()**, it is not override !
void sayBye() {...} // In Vampire, and in Person, no virtual/override
p.sayBye() calls the method of **Person**, not **Vampire** !
p.sayBye(); // sayBye() is NOT virtual, Person::setBye() is called !

When to use and when not to use classes?

When NOT to use classes?

C++ is not Java, don't use classes if there is no persistent state (no memory)!

```
class MyClass{
public:   // No fields !
        static cv::Mat method1(...);   // All methods are static !
        static cv::Mat method2(...);   // All methods are static !
};
```

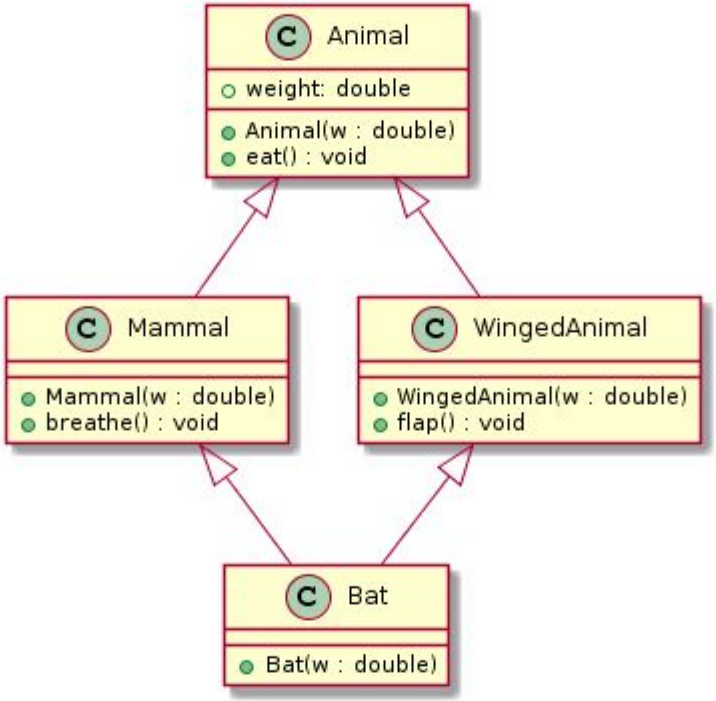
When to use classes?

When there is a natural persistent state/memory (including config, caches, ...) !

```
Config config;      // Some structure
MyData data;        // Another structure
init(data, config);
for (;;) {
        Mat result = process(data, config, frame);
        ...
```

Thank you for your attention !

Multiple inheritance: diamond problem: Example 4.3



(Example from Wikipedia, modified)
Animal is the common superclass
of **Mammal** and **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

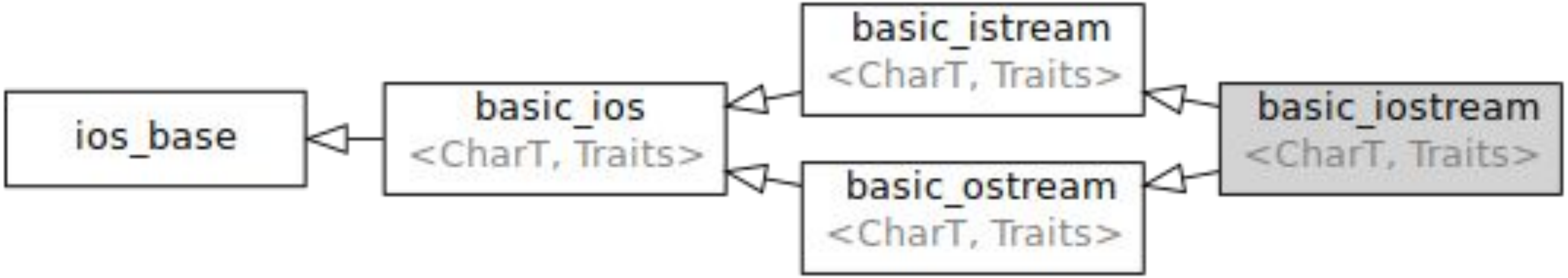
```
struct Animal {     // struct = everything is public
    Animal(double w) : weight(w) {}
    virtual void eat() { cout << "Animal::eat()" << endl;}
    double weight; // Which Ctor will set up weight ?
};

struct Mammal : public virtual Animal {
    Mammal(double w) : Animal(w*1000) {} // Mammal-only, Ignored for Bat
    virtual void breathe() { cout << "Mammal::breathe()" << endl;}
};

struct WingedAnimal : public virtual Animal {
    WingedAnimal(double w) : Animal(w*100) {} // WA-only, Ignored for Bat
    virtual void flap(){cout << "WingedAnimal::flap()" << endl;}
};

struct Bat : public Mammal, public WingedAnimal {
    // Bat must call the Animal(w) constructor also !!!
    Bat(double w) : Mammal(w), WingedAnimal(w), Animal(w) {} //weight(w) is
used
};
```

Diamond pattern in C++ standard library: IO streams



title

text