C++

Programming Language

Fundamental Concepts of
Object-Oriented Programming

C++ for Embedded Systems Course

```
#include <iostream>
class Sensor {
 public:
   void ler();
};
```

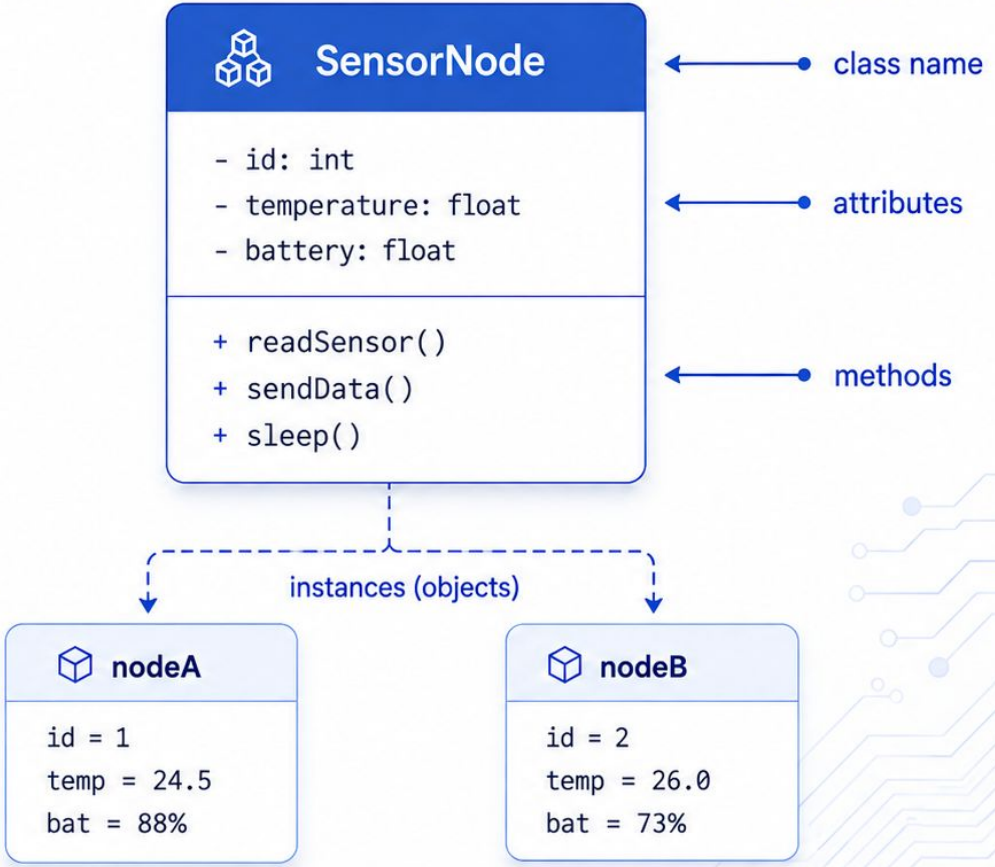
Federal University of Santa Catarina • Department of Electrical Engineering

Prof. Eduardo Augusto Bezerra

Classes

A class defines a common structure and a common behavior for its objects.

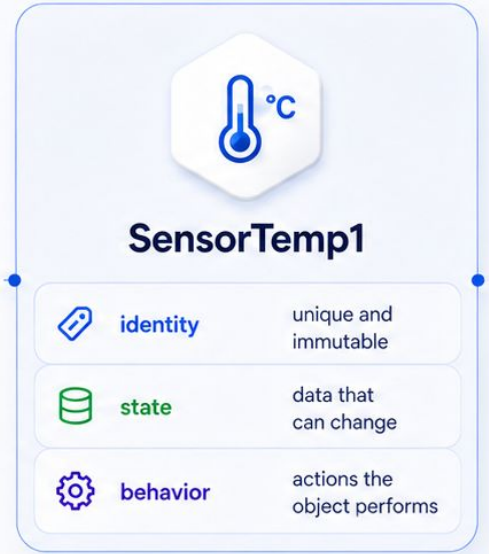
- brings together attributes and methods
- acts as a template
- allows multiple instances to be created



Objects

An object is an entity with **identity**, **state**, and **behavior**.

- represents something from the real or conceptual world
- has attributes and methods
- is an instance of a class



Examplos of Objects

temperature sensor
Measures and provides the current ambient temperature.

configuration file
Stores system parameters and preferences.

scheduling policy
Defines how processes will be selected and executed.

Examples of Classes

with their attributes and methods

IoT Sensor

- id: int
- temperature: float
- battery: float

+ read()
+ calibrate()
+ sendData()

Drone

- altitude: float
- speed: float
- battery: int

+ takeOff()
+ land()
+ navigate()

Motor Controller

- rpm: int
- dutyCycle: float
- temperatura: float

+ start()
+ setPWM()
+ stop()

Abstraction

Abstraction is the process of focusing on an object's essential characteristics while ignoring unnecessary details for the current purpose.

- focuses on what is essential
- hides irrelevant details
- depends on context

Real system

Physical implementation with many components and internal details.

TemperatureSensor Class

Attributes (essential)

- temperature in °C

Behaviors (essential)

- measure temperature

Interface

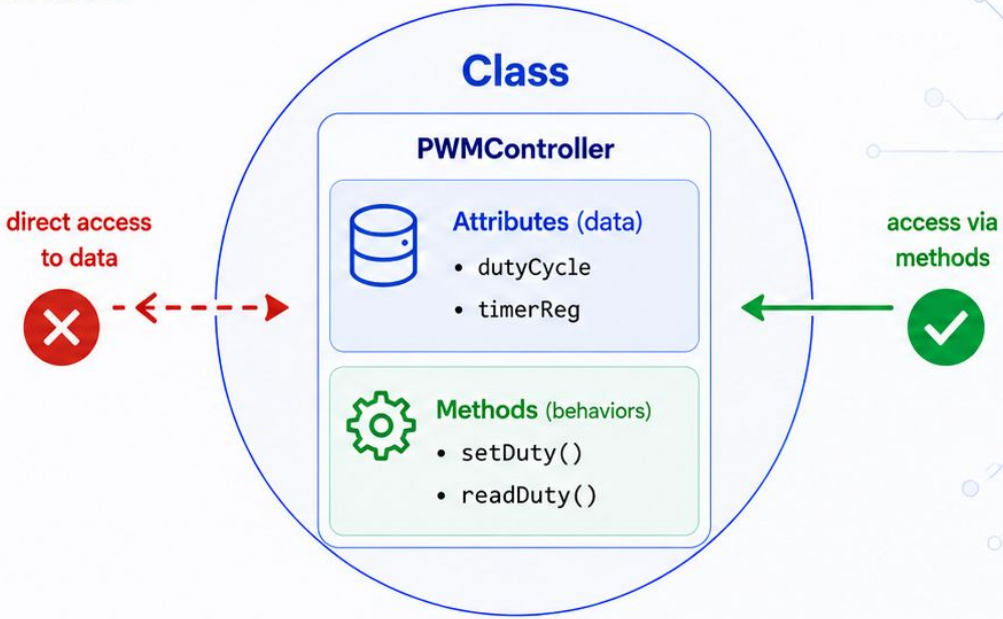
```
+ begin()
+ readCelsius()
+ calibrate()
```

In embedded systems, abstraction makes it possible to use complex hardware through a simple interface.

Encapsulation

Encapsulation is the packaging of attributes and methods within the same class, protecting data and hiding internal details.

- protects data
- reduces complexity
- exposes only the necessary interface



Encapsulation protects data and simplifies the use of the object.

Class Declaration in C++

A class can contain attributes, methods, and access levels.

```
class SensorNode {
  private:
     int id;
     float temperature;
  public:
     void update(float t);
     float reading() const;
};
```

```
SensorNode node;
node.update(25.3f);
```

Use classes to organize and encapsulate data and behaviors in a safe and reusable way.

private

public

protected

attributes store data

methods define behaviors

access levels control visibility

Visibility and Constructors

Fundamental Concepts of Object-Oriented Programming

Visibility

public	part of the object's interface
protected	accessible in the class and derived classes
private	accessible only inside the class

Constructors

A constructor initializes the object at creation time.

```
class Motor {
public:
    Motor(int initialRPM);
    void setRPM(int rpm);
private:
    int rpm;
};
```

```
Motor m(1200);
```

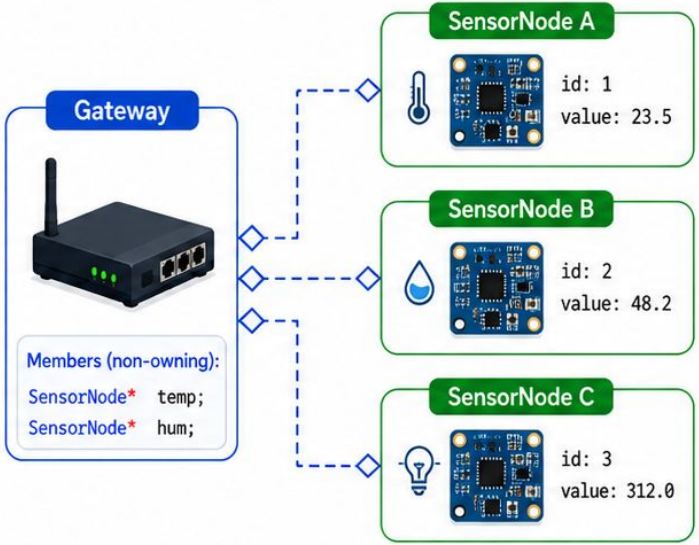
Initializes object `m` with initialRPM = `1200`

Aggregation in C++

Aggregation is a weaker has-a relationship: a class uses other existing objects without necessarily owning them.

```cpp
class SensorNode {
private:
  int id;
  float value;
public:
  SensorNode(int i, float v) : id(i), value(v) {}
};

class Gateway {
private:
  SensorNode* temp;   // non-owning pointer
  SensorNode* hum;    // non-owning pointer
public:
  Gateway(SensorNode* t, SensorNode* h)
      : temp(t), hum(h) {}
};
```



1. Weak has-a relationship
2. Parts can exist independently of the aggregate
3. The aggregate usually does not own the parts
4. Often implemented with references or non-owning pointers

Takeaway: Aggregation groups objects that can live independently from the aggregate.

Composition in C++

Composition is a strong part-of relationship: a class **contains** and **owns** objects from other classes.

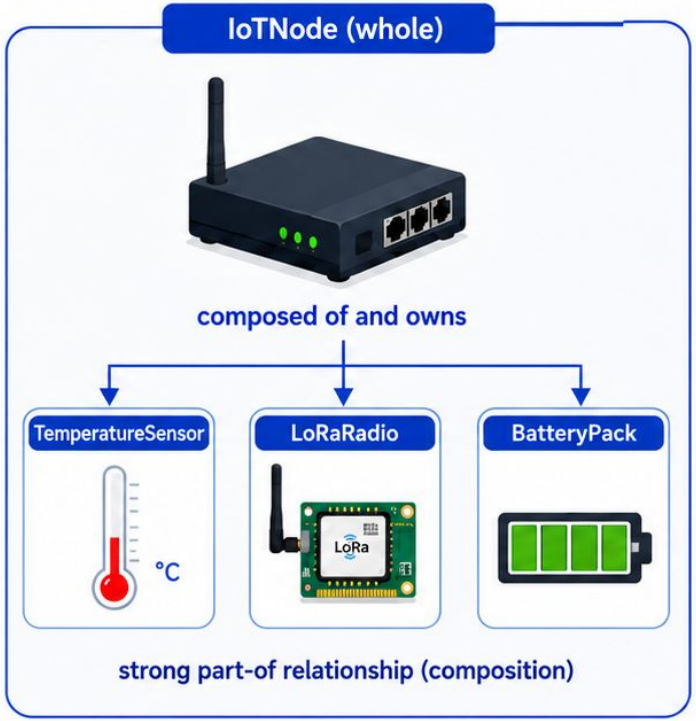
C++

```
class TemperatureSensor {
public:
   float readCelsius() const;
};

class LoRaRadio {
public:
   void send(const std::string& msg);
};

class BatteryPack {
public:
   float levelPercent() const;
};

class IoTNode {
private:
   TemperatureSensor temp;
   LoRaRadio radio;
   BatteryPack battery;
public:
   void start();
};
```



1. Strong part-of relationship

The parts are integral to the whole and have no independent existence.

2. The whole owns its parts

The whole is responsible for creating, managing, and destroying its parts.

3. Parts share the lifetime

When the whole is destroyed, its parts are destroyed too.

4. Often implemented as member objects

Stored as members inside the owning class.

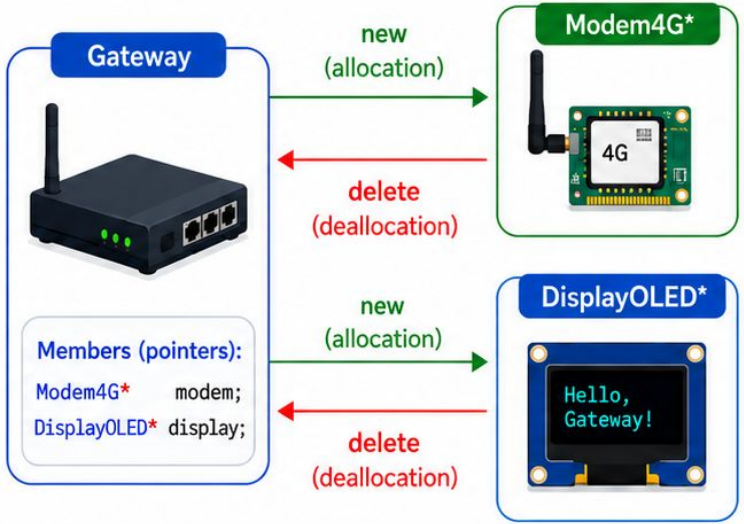
Takeaway: Composition models objects that are integral parts of a larger object.

Dynamic Composition in C++

The **Gateway** class dynamically creates and owns its components.

```cpp
class Gateway {
private:
  Modem4G* modem;
  DisplayOLED* display;

public:
  Gateway() {
    modem = new Modem4G();
    display = new DisplayOLED();
  }
  ~Gateway() {
    delete display;
    delete modem;
  }
};
```



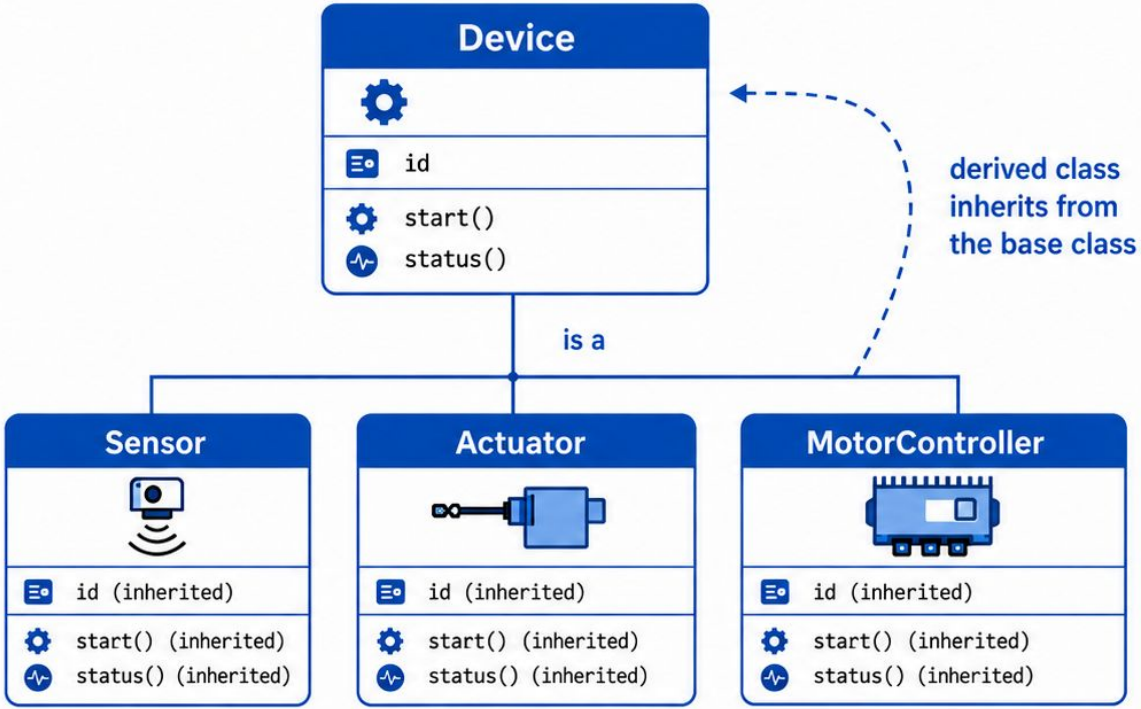
1. **Gateway owns** the components
2. **Lifetime is controlled** by Gateway
3. **Dynamic allocation can** still represent composition
4. **Pointers do not define** the relationship. Ownership does.

constructor allocates resources (*new*) → **use** uses the components → **destructor** releases resources (*delete*)

Takeaway: this is composition because the whole creates, owns, and destroys its parts.

C++ FOR EMBEDDED SYSTEMS

Inheritance



- reuse of attributes and methods
- class specialization
- from generic to specific

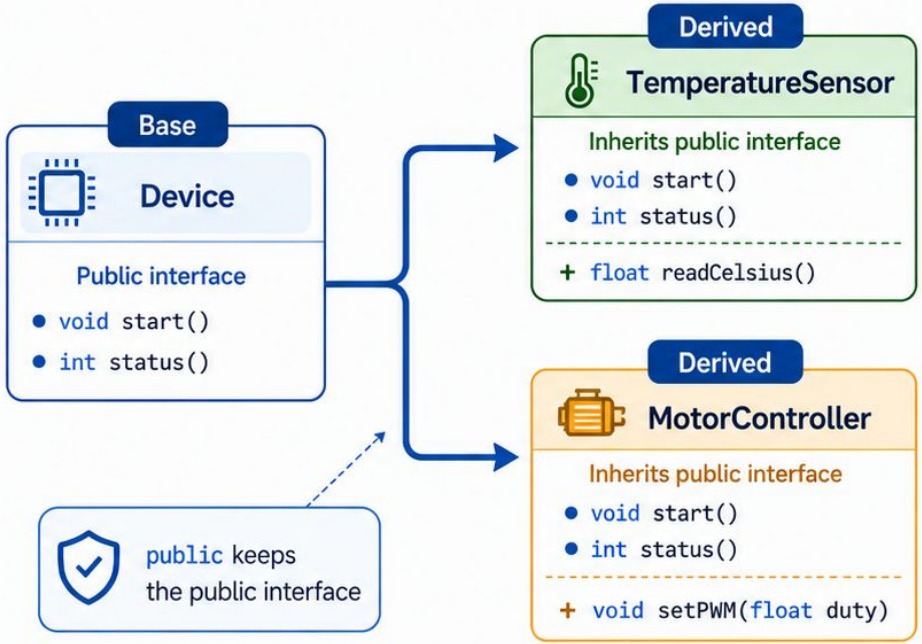
Reusable code. Better organized systems. Less duplication.

Inheritance in C++

Inheritance allows a class (derived) to reuse and extend another class (base), promoting reuse and better organization.

```
class Derived : public Base
```

```
1  class Device {
2  public:
3      void start();
4      int status();
5  };
6
7  class TemperatureSensor : public Device {
8  public:
9      float readCelsius();
10 };
11
12 class MotorController : public Device {
13 public:
       void setPWM(float duty);
   };
```



Inheritance Specifiers

- Inherited access depends on the **original access level** and on the **inheritance type.**

INHERITANCE TYPE (as declared in the derived class)

BASE CLASS ACCESS	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	private	private	inaccessible

public
publicly accessible

protected
accessible by derived classes and by the class itself

private
accessible only within the class itself

inaccessible
not accessible in the derived class

B

```
class B : public A {
   // ...
};
```

C

```
class C : protected A {
   // ...
};
```

D

```
class D : private A {
   // ...
};
```

Practical rule: the more restrictive the inheritance, the less access there is to inherited members.