📦 TP 2 — Module 4

POO & Architecture Embarquée

Classes · std::vector · Abstraction matérielle

📝 Changelog — V0.1.0

- Deck 4 : POO en C++ embarqué, classes Led/Button/LedBoard, std::vector, défi final.

🤔 Pourquoi la POO en embarqué ?

Code procédural (TP 1) :

```
const int LED1 = 18, LED2 = 19, LED3 = 21;

void allLedsOff() {
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, LOW);
    digitalWrite(LED3, LOW);
}
// Et si j'ajoute une 4e LED ? Je dois modifier allLedsOff() ...
```

Le problème : le code est **câblé** pour un nombre fixe de composants.

> ***Principe OCP :*** *une fonction devrait être ouverte à l'extension*
> *mais* ***fermée à la modification****.*

📐 Diagramme de classes

💡 La classe `Led`

```ino
#include <Arduino.h>

class Led {
  private:
    int pin;

  public:
    Led(int p) : pin(p) {}  // Constructeur avec liste d'initialisation

    void begin() {
        pinMode(pin, OUTPUT);
        off();
    }

    void on()     { digitalWrite(pin, HIGH); }
    void off()    { digitalWrite(pin, LOW);  }
    void toggle() { digitalWrite(pin, !digitalRead(pin)); }

    // PWM : luminosité variable (0-255)
    void setBrightness(int val) { analogWrite(pin, val); }
};
```

La classe `Button`

```
class Button {
  private:
    int pin;
    unsigned long lastDebounceTime = 0;
    bool lastState = HIGH;

  public:
    Button(int p) : pin(p) {}

    void begin() {
        pinMode(pin, INPUT_PULLUP);
    }

    // Retourne true si le bouton est actuellement pressé
    bool isPressed() {
        return digitalRead(pin) == LOW;
    }

    // Retourne true une seule fois par clic (anti-rebond 50ms)
    bool wasClicked() {
        bool current = digitalRead(pin) == LOW;
        if (current && !lastState &&
            (millis() - lastDebounceTime > 50)) {
            lastDebounceTime = millis();
            lastState = current;
            return true;
        }
        lastState = current;
        return false;
    }
};
```

⏱️ La classe `Timer`

```
class Timer {
  private:
    unsigned long startTime = 0;
    unsigned long duration  = 0;
    bool running = false;

  public:
    void start(unsigned long d) {
        duration  = d;
        startTime = millis();
        running   = true;
    }

    void stop() { running = false; }

    bool isExpired() {
        if (!running) return false;
        return (millis() - startTime) >= duration;
    }

    void reset() { startTime = millis(); }
};
```

📋 `std::vector` : le tableau dynamique

```cpp
#include <vector>

std::vector<int> nombres;         // Vecteur vide
nombres.push_back(42);            // Ajout d'un élément
nombres.push_back(17);

std::cout << nombres.size();      // → 2
std::cout << nombres[0];          // → 42

// Boucle "range-based for" (C++11)
for (int n : nombres) {
    Serial.println(n);
}
```

Avantage : la taille s'ajuste automatiquement à l'exécution.
Sur ESP32 : disponible nativement (512 KB RAM suffisante).

> *Note : Sur Arduino Uno (2 KB RAM), préférez un tableau statique* `Led* leds[10]`.

🎛️ La classe LedBoard

```
#include <vector>

class LedBoard {
  private:
    std::vector<Led*> leds;  // Tableau de pointeurs Led

  public:
    void addLed(Led* l) {
        leds.push_back(l);   // Ajouter une LED au tableau
        l->begin();
    }

    void allOff() {
        for (Led* l : leds) { l->off(); }
    }

    void allOn() {
        for (Led* l : leds) { l->on(); }
    }

    // S'adapte automatiquement si on ajoute des LEDs !
    void runChenillard(int speed) {
        for (Led* l : leds) {
            l->on();
            delay(speed);
            l->off();
        }
    }
};
```

🔍 Pointeurs : `*` et `&`

```
Led rouge(21);        // Objet Led en mémoire locale

// & = "prendre l'adresse de"
Led* ptr = &rouge;    // ptr contient l'adresse de rouge
                      //  ex: 0x3FFB1234 (quelque part en RAM)

// * = "accéder à la valeur à cette adresse"
(*ptr).on();          // Équivalent de rouge.on()
ptr->on();            // Notation abrégée (flèche) → identique

// Dans LedBoard
monTableau.addLed(&rouge); // On passe l'adresse, pas une copie
```

Pourquoi des pointeurs ?
Car `std::vector<Led>` ferait des **copies** des objets.
`std::vector<Led*>` stocke les **adresses** — pas de copie, pas de gaspillage.

🏗️ Assemblage dans `main.cpp`

```
// Instanciation des LEDs individuelles
Led vert(18), jaune(19), rouge(21), bleu(22);

// Le tableau de bord
LedBoard board;
Button bouton(12);

void setup() {
    bouton.begin();
    // On enregistre les LEDs dans le tableau
    board.addLed(&vert);
    board.addLed(&jaune);
    board.addLed(&rouge);
    board.addLed(&bleu);
}

void loop() {
    // Le chenillard ne tourne QUE si le bouton est maintenu
    if (bouton.isPressed()) {
        board.runChenillard(200);
    } else {
        board.allOff();
    }
}
```

✅ Point clé : l'évolutivité

Avant (procédural) : ajouter une LED = modifier 5 fonctions
Après (POO + vector) : ajouter une LED = 2 lignes

```
Led blanc(23);            // 1. Déclarer la nouvelle LED
board.addLed(&blanc);     // 2. La déclarer dans le tableau
// Fin. runChenillard() fonctionne automatiquement avec 5 LEDs !
```

> *C'est le principe **Open/Closed** en pratique :*
> *ouvert à l'extension, fermé à la modification.*

🏋️ Défi TP 2 — Énoncé complet

Tâches à réaliser :

1. Implémenter `Led`, `Button`, `Timer` et `LedBoard` dans `lib/`
2. **Interaction** : le chenillard démarre uniquement si le bouton est maintenu
3. **Évolutivité** : ajoutez une 5e LED en **tirant exactement 2 lignes de code**
4. **(Bonus)** Méthode `runChenillardBounce()` : chenillard aller-retour
5. **(Bonus)** Méthode `blink(int interval)` dans `Led` utilisant `Timer`

Critères d'évaluation :

- Utilisation correcte des pointeurs (`&` et `*`)
- Boucle range-based : `for (Led* l : leds)`
- Code propre, pas de variables globales inutiles

💡 Note sur PlatformIO & la STL

ESP32 / ESP8266 : `<vector>`, `<map>`, `<functional>` disponibles nativement.

Arduino Uno / Nano (ATmega328) :
La STL n'est pas incluse par défaut (seulement 2 KB de RAM).

```
// Alternative pour Uno : tableau statique
class LedBoard {
  private:
    Led* leds[10];    // Tableau fixe de 10 pointeurs max
    int  count = 0;
  public:
    void addLed(Led* l) { leds[count++] = l; l->begin(); }
    void allOff() {
        for (int i = 0; i < count; i++) leds[i]->off();
    }
};
```

🎯 Synthèse du TP 2

Concept	Retenir
Classe	Encapsule données + comportement
Constructeur	Initialise la broche au moment de la création
`std::vector<Led*>`	Tableau dynamique de pointeurs
`push_back()`	Ajouter un élément au vecteur
`for (T* x : vect)`	Parcourir le vecteur (range-based for)
Pointeur `&` / `*`	Adresse et déréférencement

Prochain TP → Machines à États (FSM) — contrôle d'automates

👨‍💻 Questions ?

TP 2 — POO & Architecture Embarquée

Réda BOUREBABA & Sébastien Antonico