

Q : Bonjour ! Peux tu m'expliquer la notion de "public" et "private" dans un fichier .h pour un environnement arduino ?

Bonjour ! La notion de "public" et "private" est essentielle pour comprendre l'encapsulation en programmation orientée objet en C++, ce qui est aussi applicable pour les projets Arduino.

1. La Classe et ses Accès : Public et Private

Dans un fichier .h (header), tu définis généralement les classes qui serviront de "plans" pour créer des objets. Les mots-clés `public` et `private` permettent de contrôler l'accès aux données et aux fonctions de la classe, en indiquant ce qui est accessible à l'extérieur et ce qui est réservé à un usage interne.

2. public

Les membres (attributs ou méthodes) déclarés sous `public` sont accessibles depuis l'extérieur de la classe. Cela signifie que d'autres parties de ton programme (comme ton fichier .ino principal) pourront y accéder et les utiliser directement.

Exemple :

```
class MaClasse {
public:
    void maMethodePublique();
};
```

Dans ce cas, tu peux appeler `maMethodePublique()` depuis le fichier .ino en instanciant un objet de `MaClasse`, par exemple :

```
MaClasse objet;
objet.maMethodePublique();
```

3. private

Les membres déclarés sous `private` sont uniquement accessibles depuis l'intérieur de la classe elle-même. Cela signifie que le code en dehors de la classe ne peut pas y accéder directement. Le but est de protéger les données de modifications non voulues et de restreindre l'accès aux seules méthodes de la classe.

Exemple :

```
class MaClasse {
private:
    int valeurPrivee;

public:
    void setValeur(int valeur) {
        valeurPrivee = valeur;
    }

    int getValeur() {
        return valeurPrivee;
    }
};
```

Ici, `valeurPrivee` est privée, donc inaccessible directement depuis un fichier .ino. Cependant, tu peux l'utiliser indirectement via les méthodes publiques `setValeur` et `getValeur`.

4. Pourquoi les Utiliser ?

En structurant les données comme privées, tu contrôles mieux leur utilisation et minimises les risques d'erreurs. Les parties publiques de la classe fournissent une interface contrôlée pour interagir avec elle. Cela rend le code plus sûr, clair et plus facile à maintenir.

Exemple complet en Arduino

Imaginons que tu crées une classe pour gérer une LED :

```
// Led.h
class Led {
private:
    int pin;

public:
    Led(int pinLed);          // Constructeur
    void allumer();           // Méthode publique pour allumer la LED
    void eteindre();          // Méthode publique pour éteindre la LED
};
```

Et dans le fichier .cpp correspondant :

```
// Led.cpp
#include "Led.h"
#include <Arduino.h>

Led::Led(int pinLed) {
    pin = pinLed;
    pinMode(pin, OUTPUT);
}

void Led::allumer() {
    digitalWrite(pin, HIGH);
}

void Led::eteindre() {
    digitalWrite(pin, LOW);
}
```

Dans ce cas, pin est privé. L'utilisateur peut allumer ou éteindre la LED en appelant allumer et eteindre, sans accéder directement à pin.

Q : Tu pourrais me sortir un exemple de programme arduino "feux-tricolore.ino" avec un fichier .h et un fichier .cpp dans lequel j'aurais une classe "feuxTricolore" avec led verte sur pin 5, led jaune sur pin 6 et led rouge sur pin 7, avec les fonctions "allumer" et "eteindre" ?

Voici un exemple de programme Arduino pour un feu tricolore utilisant une classe FeuxTricolore définie dans des fichiers .h et .cpp.

Structure du projet

Le projet contiendra : 1. Un fichier feuxTricolore.h pour la déclaration de la classe. 2. Un fichier feuxTricolore.cpp pour l'implémentation des méthodes de la classe. 3. Un fichier principal feux-

tricolore.ino pour contrôler le comportement du feu.

1. Fichier `feuxTricolore.h`

```
// feuxTricolore.h

#ifndef FEUXTRICOLE_H
#define FEUXTRICOLE_H

class FeuxTricolore {
private:
    int pinVerte;
    int pinJaune;
    int pinRouge;

public:
    FeuxTricolore(int pinVerte, int pinJaune, int pinRouge);
    void allumer(int pin);
    void eteindre(int pin);
    void eteindreTout();
};

#endif
```

2. Fichier `feuxTricolore.cpp`

```
// feuxTricolore.cpp

#include "feuxTricolore.h"
#include <Arduino.h>

// Constructeur : initialise les pins des LEDs et les configure en sortie
FeuxTricolore::FeuxTricolore(int pinVerte, int pinJaune, int pinRouge) {
    this->pinVerte = pinVerte;
    this->pinJaune = pinJaune;
    this->pinRouge = pinRouge;
    pinMode(pinVerte, OUTPUT);
    pinMode(pinJaune, OUTPUT);
    pinMode(pinRouge, OUTPUT);
}

// Méthode pour allumer la LED spécifiée par le paramètre pin
void FeuxTricolore::allumer(int pin) {
    digitalWrite(pin, HIGH);
}

// Méthode pour éteindre la LED spécifiée par le paramètre pin
void FeuxTricolore::eteindre(int pin) {
    digitalWrite(pin, LOW);
}

// Méthode pour éteindre toutes les LEDs
void FeuxTricolore::eteindreTout() {
    eteindre(pinVerte);
    eteindre(pinJaune);
    eteindre(pinRouge);
}
```

3. Fichier principal `feux-tricolore.ino`

```
#include "feuxTricolore.h"

// Définition des pins pour les LED du feu tricolore
const int pinVerte = 5;
const int pinJaune = 6;
const int pinRouge = 7;

FeuxTricolore feux(pinVerte, pinJaune, pinRouge);

void setup() {
    // On éteint toutes les LEDs au démarrage
    feux.eteindreTout();
}

void loop() {
    // Simule un cycle de feu tricolore

    // Allume la LED verte pendant 5 secondes
    feux.allumer(pinVerte);
    delay(5000);
    feux.eteindre(pinVerte);

    // Allume la LED jaune pendant 2 secondes
    feux.allumer(pinJaune);
    delay(2000);
    feux.eteindre(pinJaune);

    // Allume la LED rouge pendant 5 secondes
    feux.allumer(pinRouge);
    delay(5000);
    feux.eteindre(pinRouge);
}
```

Explications

1. **Classe FeuxTricolore** : La classe FeuxTricolore gère les trois LEDs du feu tricolore (verte, jaune, rouge). Elle possède trois méthodes principales :
2. `allumer(int pin)`: allume la LED correspondant à la pin spécifiée.
3. `eteindre(int pin)`: éteint la LED correspondant à la pin spécifiée.
4. `eteindreTout()`: éteint toutes les LEDs en appelant `eteindre` sur chaque pin.
5. **Cycle de feu tricolore** : Dans la boucle principale (`loop`), chaque LED est allumée et éteinte avec un délai, simulant le comportement d'un feu tricolore standard.

Q : Merci beaucoup ! Il t'a fallu 2 secondes pour me proposer ce programme, il va me falloir quelques minutes, voir plus pour comprendre tout cela !

Q : Le programme fonctionne super ! Merci ! Dans le fichier `.ccp` que veut dire "this-->" ?

Super que le programme fonctionne bien !

Dans le fichier .cpp, le mot-clé `this->` fait référence à l'instance courante de la classe. C'est une manière d'accéder aux membres (variables ou méthodes) de l'objet en cours d'utilisation.

Dans notre programme, `this->pinVerte = pinVerte;` signifie que la variable membre `pinVerte` de l'objet est assignée à la valeur du paramètre `pinVerte` passé dans le constructeur. En d'autres termes :

- `this->pinVerte` fait référence à la variable membre `pinVerte` de la classe `FeuxTricolore`.
- `pinVerte` sans `this->` fait référence au paramètre passé au constructeur.

On utilise `this->` pour différencier la variable membre de la variable locale (paramètre) qui a le même nom, ce qui aide à éviter les ambiguïtés.

Quelle différence avec "extern" ?

`extern` est un concept différent de `this->` et joue un rôle important, notamment en C et C++, pour déclarer des variables ou des fonctions définies ailleurs (dans un autre fichier, par exemple).

Voici comment ils diffèrent :

1. `this->` : Utilisé dans une classe pour référencer l'instance courante

- `this->` est spécifique aux classes et aux objets en C++.
- Il est utilisé pour accéder aux membres (variables et méthodes) de l'objet actuel de la classe.
- `this->` est souvent utilisé dans les méthodes d'une classe pour clarifier quand on parle des variables membres de cette instance.

Exemple :

```
class Exemple {
private:
    int valeur;

public:
    Exemple(int valeur) {
        this->valeur = valeur; // Assigne le paramètre `valeur` à la variable membre `valeur`
    }
};
```

2. `extern` : Utilisé pour déclarer des variables ou des fonctions définies ailleurs

- `extern` permet de déclarer l'existence d'une variable ou d'une fonction définie dans un autre fichier ou un autre emplacement dans le programme.
- Cela indique au compilateur que cette variable ou fonction existe quelque part, même si elle n'est pas définie dans le fichier actuel.
- On utilise souvent `extern` dans des programmes qui s'étendent sur plusieurs fichiers .cpp, pour partager des variables globales ou des fonctions.

Exemple : Imaginons que tu as une variable définie dans `fichier1.cpp` :

```
// fichier1.cpp
int maVariablePartagee = 10;
```

Pour pouvoir utiliser cette variable dans `fichier2.cpp`, on utilise `extern` :

```
// fichier2.cpp
extern int maVariablePartagee; // Déclare que la variable existe ailleurs
void setup() {
    Serial.begin(9600);
    Serial.println(maVariablePartagee); // Utilisation de la variable
}
```

Résumé :

- **this** -> est spécifique aux classes et permet d'accéder aux membres de l'objet en cours.
- **extern** est utilisé pour déclarer des variables ou fonctions globales définies ailleurs, afin de les partager entre plusieurs fichiers.

Ils ont donc des utilisations et des contextes différents, mais chacun est utile pour structurer et organiser un programme complexe.

Dans un fichier .h je peux déclarer une variable "extern" ?

Oui, tu peux tout à fait déclarer une variable avec le mot-clé `extern` dans un fichier .h. C'est même une pratique courante pour partager des variables globales entre plusieurs fichiers .cpp.

Comment et pourquoi utiliser `extern` dans un fichier .h

Déclarer une variable `extern` dans un fichier .h permet de rendre cette variable accessible depuis plusieurs fichiers de ton projet. La déclaration avec `extern` indique au compilateur que la variable est définie ailleurs (dans un fichier .cpp, par exemple), sans en créer une copie.

Exemple d'utilisation

Imaginons un projet avec les fichiers suivants : - `variables.h` : Contient la déclaration de la variable globale partagée. - `fichier1.cpp` : Définit la variable globale. - `fichier2.cpp` : Utilise la variable globale.

Étape 1 : Déclaration dans le fichier `variables.h`

```
// variables.h
#ifndef VARIABLES_H
#define VARIABLES_H

extern int compteurGlobal; // Déclaration de la variable globale

#endif
```

Étape 2 : Définition dans `fichier1.cpp`

```
// fichier1.cpp
#include "variables.h"

int compteurGlobal = 0; // Définition de la variable globale
```

Étape 3 : Utilisation dans `fichier2.cpp`

```
// fichier2.cpp
#include "variables.h"
```

```
void incrementerCompteur() {
    compteurGlobal++; // Utilisation de la variable définie dans fichier1.cpp
}
```

Pourquoi utiliser extern dans le .h ?

- **Éviter les duplications** : Sans extern, si tu définis une variable dans le .h, chaque fichier .cpp qui inclut ce .h créerait sa propre copie de la variable, ce qui mènerait à des erreurs de compilation pour "multiple definition".
- **Partage de variables globales** : En utilisant extern, la variable est définie dans un seul fichier .cpp, mais elle est accessible dans tout autre fichier incluant le .h.

Résumé

- **Déclaration avec extern dans le .h** : Indique que la variable existe quelque part.
- **Définition dans un fichier .cpp** : Alloue la mémoire pour cette variable.
- Ensuite, les autres fichiers peuvent inclure le .h pour utiliser la variable sans créer de copie.

Q : Donc l'intérêt serait au départ de faire un fichier variables.h contenant toutes les variables de mon programme .ino ? Dois-je les déclarer en "extern" ?

Oui, c'est une bonne pratique de créer un fichier variables.h pour centraliser les déclarations de variables globales que tu souhaites partager dans différents fichiers de ton projet Arduino. Cela rend le code plus organisé et facilite les modifications, car tu n'as besoin de mettre à jour les variables qu'à un seul endroit.

Quand utiliser extern dans variables.h

Si ton projet est structuré en plusieurs fichiers .cpp et .h, alors oui, tu devras déclarer tes variables avec extern dans variables.h. Voici la règle générale :

- **Déclare les variables globales avec extern dans le fichier .h** : Cela indique aux autres fichiers que ces variables existent quelque part sans les définir plusieurs fois.
- **Définis les variables dans un seul fichier .cpp ou .ino** : C'est là où la mémoire est allouée pour ces variables.

Exemple : Fichier variables.h

```
// variables.h
#ifndef VARIABLES_H
#define VARIABLES_H

extern int compteur;          // Variable entière globale
extern float temperature;    // Variable de température globale
extern bool etatSysteme;     // Indicateur d'état global

#endif
```

Dans ton fichier principal .ino

Dans le fichier .ino, tu inclues variables.h et définis les variables sans extern :

```
#include "variables.h"

int compteur = 0;           // Définition de la variable compteur
```

```
float temperature = 0.0; // Définition de la variable temperature
bool etatSysteme = true; // Définition de la variable etatSysteme

void setup() {
    // Code d'initialisation
}

void loop() {
    // Code principal
}
```

Dans un autre fichier .cpp

Quand tu as besoin d'utiliser ces variables dans un autre fichier .cpp, tu inclues simplement `variables.h` :

```
#include "variables.h"

void afficherEtat() {
    if (etatSysteme) {
        Serial.println("Système actif");
    } else {
        Serial.println("Système inactif");
    }
}
```

Pourquoi faire cela ?

1. **Meilleure organisation** : Toutes les variables globales sont centralisées dans un seul fichier .h.
2. **Évite les erreurs de "multiple definition"** : Tu declares avec `extern` dans le .h, mais la variable n'est définie qu'une seule fois dans le .ino.
3. **Partage simplifié** : Tu peux inclure `variables.h` dans tout fichier qui a besoin de ces variables sans créer de copies.

Donc oui, utiliser `extern` dans `variables.h` est une bonne pratique pour des projets plus structurés et complexes, surtout quand tu commences à séparer les fonctionnalités dans plusieurs fichiers .cpp et .h.

Q : Ça peut aussi entraîner des tensions si des personnes n'ont pas la même approche de cette manière de programmer :) Quels peuvent être les derniers arguments ?

Ah, je comprends bien ! Les approches en programmation peuvent en effet provoquer des débats, surtout quand il s'agit de bonnes pratiques ou d'organisations du code. Pour convaincre des amis qui pourraient être sceptiques, voici quelques arguments supplémentaires :

1. Lisibilité et Maintenance à Long Terme

- **Argument** : Centraliser les variables globales dans un fichier unique rend le code plus facile à lire et à maintenir.
- **Explication** : Dans des projets complexes, rechercher les déclarations dans tout le code devient fastidieux. Un fichier `variables.h` permet de voir rapidement toutes les variables globales du projet.

2. Facilité de Modification

- **Argument** : Avec un fichier centralisé, il est plus rapide et sûr de modifier les variables globales.
- **Explication** : Si tu veux changer un type ou une valeur par défaut, tu modifies simplement

variables.h. Sinon, tu risques d'oublier de changer la variable partout, ce qui peut créer des bugs difficiles à repérer.

3. Évolutivité pour les Projets Plus Grands

- **Argument** : Utiliser extern et structurer les variables globales dans un fichier .h est une technique standard pour les projets plus avancés.
- **Explication** : Même dans les grands projets Arduino ou industriels, cette approche est souvent utilisée. Cela permet au projet de grandir et d'intégrer plus de fonctionnalités sans ajouter de complexité inutile.

4. Bonne Pratique dans l'Industrie

- **Argument** : L'approche avec extern et des fichiers .h fait partie des bonnes pratiques de développement en C et C++.
- **Explication** : Si les amis prévoient de travailler dans des environnements où le code doit être partagé et maintenu par une équipe, adopter ces pratiques dès maintenant leur facilitera la transition.

5. Meilleure Détection des Erreurs de Compilation

- **Argument** : Centraliser les déclarations avec extern peut révéler rapidement les erreurs de déclaration ou de définition.
- **Explication** : Avec un fichier unique pour les déclarations, il est facile de voir quelles variables sont partagées. Cela évite des erreurs de compilation liées aux redéfinitions multiples, qui sont souvent frustrantes.

Ces arguments peuvent aider à montrer que cette approche est là pour rendre les projets plus fiables, collaboratifs et simples à maintenir, même si elle demande un peu de discipline.

Merci beaucoup ! J'ai oublié : je peux mettre une variable "static" dans variables.h ?

Non, tu ne peux pas déclarer une variable **static** dans un fichier variables.h de la même manière que tu utiliserais extern pour une variable globale. Voici pourquoi :

Rappel sur static

- **static** signifie que la variable ou la fonction n'est visible que dans le fichier où elle est définie, ce qui la rend **locale** à ce fichier.
- Si tu utilises static dans un fichier .h, chaque fichier .cpp qui inclut ce .h créera sa propre copie de la variable, ce qui va à l'encontre du comportement que tu souhaites pour une variable globale.

Exemple de problème avec static dans un .h

Si tu fais cela dans variables.h :

```
// variables.h
#ifndef VARIABLES_H
#define VARIABLES_H

static int compteur = 0; // Déclaration de variable statique

#endif
```

- **Problème** : Chaque fichier .cpp qui inclut variables.h aura sa propre version de la variable compteur. Ce n'est pas ce que tu veux, car static empêche de partager cette variable entre

plusieurs fichiers.

Utilisation correcte de static

Si tu veux une variable **static**, il faut la déclarer et la définir dans un fichier .cpp (et non dans un .h) pour qu'elle soit **locale** à ce fichier, tout en conservant son état entre les appels de fonction. Par exemple :

```
// fichier1.cpp
#include "variables.h"

static int compteur = 0; // La variable est locale à fichier1.cpp

void incrementerCompteur() {
    compteur++;
}

int getCompteur() {
    return compteur;
}
```

Conclusion :

- **Dans variables.h** : Tu ne peux pas utiliser static si tu veux que la variable soit partagée entre différents fichiers.
- **Dans .cpp** : Utilise static si tu veux que la variable soit locale au fichier, mais si tu veux la rendre accessible à d'autres fichiers, il vaut mieux éviter static et utiliser extern.

Si tu veux une variable globale accessible partout, privilégie l'utilisation de extern dans le .h et la définition dans un seul fichier .cpp.