Développement logiciel

Introduction à la Programmation Orienté Objet

Le langage C++

OBJECTIFS

À l'issue de cette séquence, vous devrez être capable de :

- savoir ce qu'est un objet, une classe une instance
1. de connaître le vocabulaire attribut, méthode, et de l'utiliser
- de savoir ce qu'est un constructeur et pouvoir l'utiliser pour initialiser les attributs d'une classe
- de savoir reconnaître un destructeur.
- de mettre en place une relation de composition entre deux classes
- de mettre en place une relation d'héritage ente deux classes
- de lire un diagramme de classes UML, et le vocabulaire associé : cardinalité, rôle

Niveau de maîtrise attendu pour le BTS Systèmes Numérique option Informatique et Réseaux :

S4. Développement logiciel		IR

SNir1 Version 1.0

Sommaire

Développement logiciel.....3

1. Orienté objet.....2

1.1. Notion d'objet.....2
1.2. Objectifs.....2
1.3. Définition.....2
1.4. Distinction entre classes et instances.....2

2. Représentation de la classe.....3

2.1. Représentation graphique UML.....3
2.2. Représentation en langage C++.....4

3. Liens entre les classes.....6

3.1. Représentation graphique de la composition sous UML.....6
3.2. Implémentation de la relation de composition en C++.....6

4. Compléments sur le codage en C++.....7

4.1. Le constructeur.....7
4.2. Utilisation des attributs.....8
4.3. Appel des méthodes en dehors de la classe.....8
4.4. Gestion des chaînes de caractères en C++.....9

5. Relation d'héritage.....10

5.1. Introduction.....10
5.2. Représentation UML de l'héritage.....10
5.3. Codage en C++ de la relation d'héritage.....11

6. Interaction entre les objets.....12

6.1. Diagramme de séquence.....12
6.2. Codage C++ du diagramme de séquence.....12

7. Comportement des objets.....13

7.1. Diagramme états-transitions.....13
7.2. Codage en C++ du diagramme états-transitions.....13

8. Allocation dynamique.....15

8.1. Variables automatiques.....15
8.2. Variables dynamiques.....15
8.3. Tableaux dynamiques.....16

9. Complément sur la notion d'héritage.....17

9.1. Accès aux membres de la classe de base.....17
9.2. Ordre d'appel des constructeurs et destructeurs.....18
9.3. Appel d'une méthode surchargée de la classe de base.....19

1. Orienté objet

1.1. Notion d'objet

Ce mode de programmation permet le développement de logiciels fondés sur la modélisation du monde réel. Il met en œuvre une collection d'objets dissociés comprenant à la fois une structure de données et un comportement.

Exemple : Une horloge possède des données telles que :

- les heures,
- les minutes,
- les secondes

Elle dispose de fonctionnalités permettant :

- de faire évoluer ses données,
- d'effectuer les réglages nécessaires pour la mise à l'heure

Dans une approche fondée sur le monde réel, on remarque que plusieurs objets peuvent être liés entre eux pour former un objet plus complexe et réutilisable dans d'autres circonstances. Ainsi, une horloge possède en plus de sa mécanique d'un clavier et d'un cadran, un thermomètre se compose également d'un capteur de température et d'un afficheur.

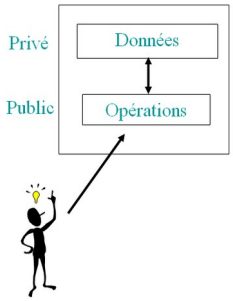
1.2. Objectifs

Ce style de programmation a pour objectif de ***maîtriser la complexité*** d'applications de plus en plus volumineuses et de faire évoluer l'industrie du logiciel vers ***plus de réutilisabilité***.

Ce style de programmation a pour effet de ***réduire le poids de la maintenance*** dans le coût du logiciel et de structurer les ***logiciels en couches***.

1.3. Définition

Un **objet** est un élément unique. Chaque objet possède sa propre identité grâce à un identifiant pour un ensemble fortement couplé contenant des données et des opérations permettant de les manipuler.



Afin de protéger les données et les garder intègres tout le long du programme, elles ne seront accessibles que par les opérations mises à disposition. On parle alors d'***encapsulation des données***. Elles composent la ***partie privée*** de l'objet.

D'une façon générale, le programmeur utilisant un objet n'a pas à se préoccuper de la manière dont sont implémentées les données encapsulées. Il aura accès uniquement à la ***partie publique*** de cet objet permettant de les manipuler.

Afin d'être cachées du reste du programme, certaines opérations peuvent également être privées. Elles permettent de réaliser des traitements intermédiaires qui ne sont pas nécessaires pour l'utilisation de l'objet vu de l'extérieur.

On parle d'**attributs** pour les données et de **méthodes** pour les opérations ou fonctions agissant sur les données.

1.4. Distinction entre classes et instances

Une ***classe*** est un ensemble d'objets ayant les mêmes attributs ou données et disposant des mêmes méthodes. Cet ensemble forme un moule qui sera utilisé comme le modèle pour les objets qui la composent. Cela correspond à la définition d'un type en langage C.

Une ***instance*** est un exemplaire de la classe. C'est l'équivalent d'une variable dans la terminologie du langage C. On parle alors d'instanciation de la classe.

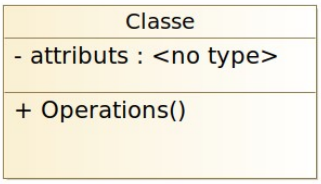
Deux instances sont distinctes, même si tous leurs attributs ou données ont des valeurs identiques.

2. Représentation de la classe

2.1. Représentation graphique UML

UML pour Unified Model Langage est un outil d'analyse, utilisé pour modéliser les applications informatiques. Cette modélisation permet une représentation graphique des composants du système. Différentes vues sont nécessaires pour en déterminer le fonctionnement et l'architecture. Chaque vue se compose de diagrammes eux-mêmes constitués de différents éléments.

La représentation graphique d'une classe qui constitue un de ces éléments en UML est un rectangle divisé en trois parties :



- Le nom de la classe
- La liste des attributs
- La liste des méthodes

Le signe « **-** » indique le caractère ***privé*** de l'élément, le signe « **+** » indique, quant lui, le caractère ***public***.

Par convention, le nom d'une classe commence par une majuscule. Les attributs sont des noms de données commençant par une minuscule et le nom d'une opération est un verbe généralement à l'infinitif suivi éventuellement d'un complément et commençant par une majuscule.

Chaque opération peut recevoir des paramètres et retourner une valeur comme pour les fonctions du langage C.

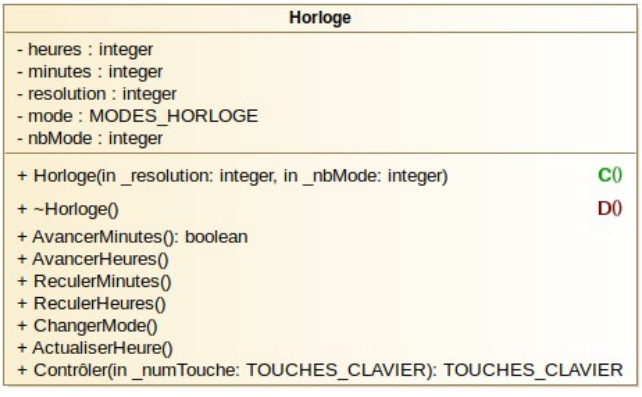
Chaque attribut possède un type particulier comme ceux rencontrés en langage C ou même une autre classe au besoin. Les attributs seront des données qui seront partagées entre toutes les méthodes de la classe. Il ne sera donc pas nécessaire de les passer en paramètre.

Certaines opérations jouent un rôle particulier pour l'objet :

Il s'agit par exemple du ***constructeur*** dont le rôle est d'initialiser les attributs de la classe. Éventuellement, il permet également d'allouer dynamiquement de la mémoire ou placer l'objet dans un certain contexte, choix de police de caractères, couleur de fond… Le constructeur est facilement reconnaissable, il porte le ***même nom que la classe***.

De même, chaque classe possède un ***destructeur***, son rôle est de restituer le contexte, de libérer la mémoire… Il porte également le même nom que la classe précédée du signe « **~** ».

Exemple : La classe **Horloge** peut être définie graphiquement par la figure ci-dessous :



Elle se compose de cinq attributs :

heures et **minutes**, deux entiers.

resolution, un entier prenant la valeur 12 ou 24.

mode est une variable du type **MODES_HORLOGE** représentant un des trois modes de fonctionnement de l'horloge (**AUCUN_REGLAGE**, **REGLAGE_HEURES**, **REGLAGE_MINUTES**).

nbMode est un entier indiquant le nombre de modes de l'horloge ici 3.

Elle dispose d'un constructeur, **Horloge()**, avec un paramètre d'entrée **_resolution** permettant d'initialiser entre autres l'attribut **resolution**, et un second **_nbMode** également un entier pour initialiser l'attribut **nbMode**. Elle possède un destructeur, **~Horloge()**.

La méthode **AvancerMinutes()** retourne un booléen indiquant que les minutes sont repassées à 0, donc une heure s'est écoulée. Les autres Methodes **AvancerHeures()**, **ReculerHeures()**, **ReculerMinutes()** ne possèdent pas de paramètre et ne retournent rien, elles servent à faire évoluer les attributs **heures** et **minutes**. La méthode **ChangerMode()** permet de passer d'un mode à l'autre de l'horloge. La méthode ActualiserHeure() est chargée de réaliser l'affichage des informations sur le cadran. La méthode **Controler()** reçoit en paramètre d'entrée **_numTouche**, le numéro de la touche enfoncée sur le clavier et la restitue en paramètre de retour.

Les paramètres d'entrée sont repérés avec la mention **in** précédant le nom du paramètre. Il existe également les termes **out** pour sortie et **inout** pour entrée-sortie.

2.2. Représentation en langage C++

La déclaration de la classe se fait dans un fichier d'entête .h. Il porte par convention le nom de la classe sans la majuscule.

Déclaration de la classe Horloge **horloge.h**

```
#ifndef HORLOGE_H
#define HORLOGE_H
enum TOUCHES_CLAVIER {        // Déclaration des touches du clavier dans une constante énumérée,
     AUCUNE = -1,             // juste ici pour les besoins de l'exemple par la suite dans clavier.h
     MODE,
     PLUS,
     MOINS,
     FIN
};
class Horloge
{
public:
    Horloge(const short _resolution = 24, const short _nbMode = 3);
    ~Horloge();
    bool AvancerMinutes();
    void AvancerHeures();
    void ReculerMinutes();
    void ReculerHeures();
    void ChangerMode();
    void ActualiserHeure();
    TOUCHES_CLAVIER Controler(const TOUCHES_CLAVIER numTouche);
    enum MODES_HORLOGE {      // déclaration des modes de l'horloge
        AUCUN_REGLAGE,
        REGLAGE_HEURES,
        REGLAGE_MINUTES
    };
private:
    int heures;
    int minutes;
    int resolution;
    const short nbMode;
    MODES_HORLOGE mode;
};
#endif /* HORLOGE_H */
```

C'est le mot clé **class** qui permet la déclaration d'une classe, il est suivi du nom de la classe ici **Horloge**. La déclaration se poursuit par les accolades ouvrante et fermante et se termine par un point-virgule. Apparaît ensuite les deux sections, publique et privée repérées par les mots clés **public:** et **private:** Une section se termine lorsqu'une nouvelle commence ou lors de l'accolade fermante.

Le constructeur et le destructeur ne retournent rien, mais ne sont pas précédés du mot clé **void**. Le destructeur ne possède jamais de paramètre. De manière facultative, un paramètre peut posséder une valeur par défaut, dans ce cas le programmeur n'est pas obligé de préciser la valeur du paramètre lors de l'instanciation de la classe.

Instanciation de la classe Horloge **main.cpp**

```
#include "horloge.h"
int main()
{
    Horloge uneHorloge ;      // ici les paramètres ont pris les valeurs par défaut
    TOUCHES_CLAVIER laTouche = AUCUNE;
    do
    {
        laTouche = uneHorloge.Controler(laTouche);
    } while (laTouche != FIN);

    return 0;
}
```

C'est la méthode **Controler()** qui assure le fonctionnement de l'horloge, elle est appelée à chaque tour de boucle jusqu'à l'appui sur la touche **FIN**. L'appel de cette méthode se fait à la manière d'une structure en langage C en utilisant l'opérateur **.** lorsqu'il s'agit d'une instance par valeur, le cas présent, et une flèche **->** si l'instance était désignée par un pointeur.

L'instanciation de la classe aurait très bien pu être paramétrée

Variante pour l'instanciation de la classe Horloge **main.cpp**

```
#include "horloge.h"
int main()
{
    Horloge uneHorloge(12);   // les heures varient ici de 0 à 11 ou 12 en fonction du matin ou du soir
    ...
}
```

Dans ce cas, le paramètre **_resolution** possède la valeur **12**. Le paramètre **_nbMode** garde la valeur par défaut **3,** car il n'y a pas de deuxième valeur précisée.

Remarque : Pour affecter une nouvelle valeur au deuxième paramètre, il est impératif de donner une valeur au premier, sinon le compilateur lui affecte la valeur.

L'implémentation de la classe se fait dans un fichier .cpp portant le même nom que la classe sans la majuscule.

Implémentation du constructeur de la classe Horloge **horloge.cpp**

```
#include "horloge.h"
Horloge::Horloge(const short _resolution, const short _nbMode):
heures(0),
minutes(0),
resolution(_resolution),
nbMode(_nbMode),
mode(AUCUN_REGLAGE)
{
}
```

L'initialisation des attributs est réalisée par la ***liste d'initialisation***, c'est la méthode à privilégier. Pour l'instant rien d'autre n'est prévu dans le corps du constructeur, le bloc entre accolades. C'est notation est équivalente à la notation qui va suivre. On perçoit, ici, le fait de mettre le _ devant le paramètre afin de ne pas confondre avec l'attribut de la classe, **_resolution** le paramètre et **resolution** l'attribut.

Variante pour l'implémentation du constructeur de la classe Horloge **horloge.cpp**

```
#include "horloge.h"
Horloge::Horloge(const short _resolution, const short _nbMode)
{
        heures = 0;
        minutes = 0;
        resolution = _resolution;
        nbMode = _nbMode;
        mode = AUCUN_REGLAGE;
}
```

La première apparition du mot **Horloge** désigne le nom de la classe. Il est suivi de l'opérateur de déréférenciation **::** il permet d'indiquer que le constructeur **Horloge** est dans la classe **Horloge**. Cette notation permet de faire la différence avec une simple fonction du langage C.

Lors de l'entête du constructeur dans le fichier **horloge.cpp**, les valeurs par défaut ont disparu. Elles ne sont présentes uniquement dans le fichier .h.

Implémentation de la méthode AvancerMinutes() de la classe Horloge **horloge.cpp**

```
bool Horloge::AvancerMinutes()
{
    bool retour = false;
    if (++minutes == 60)      // pré-incrémentation puis test d'égalité avec la valeur 60
    {
        minutes = 0;
        retour = true;        // on est arrivé à la fin de l'heure
    }
    return retour;
}
```

Le type du paramètre de retour est placé devant le nom de la classe.

Exercice : Codez en C++ les trois autres méthodes permettant de faire évoluer les heures et les minutes.

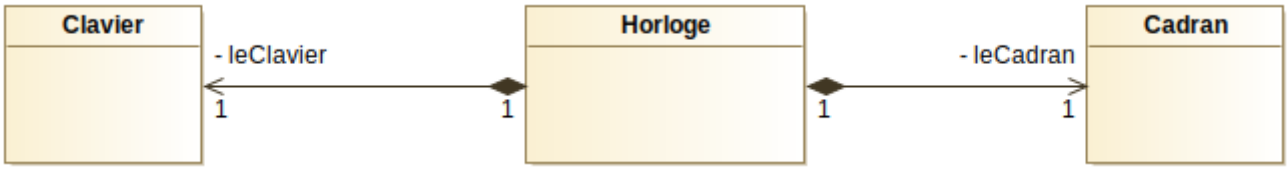
3. Liens entre les classes

Dans la première partie du document, le fait de lier plusieurs objets entre eux pour former un objet plus complexe a été évoqué. Effectivement, pour que notre horloge fonctionne, il lui faut un cadran pour afficher les heures et les minutes et des boutons formant un clavier pour effectuer les réglages nécessaires à son bon fonctionnement.

Dans notre projet, l'existence du clavier et du cadran est liée à la durée de vie de l'horloge. Sans classe **Horloge** nul besoin d'une classe **Clavier** ou d'une classe **Cadran**. Ce type de relation est une **composition**, la classe **Horloge** est composée d'une classe **Clavier** et d'une classe **Cadran**. Lorsque la classe **Horloge** est instanciée, les classes **Clavier** et **Cadran** sont créées. Lorsque le programme s'arrête ou que la classe **Horloge** est détruite, ses deux composants le sont aussi.

3.1. Représentation graphique de la composition sous UML

Le diagramme de classes sous UML permet de représenter les relations entre les différentes classes de l'application.



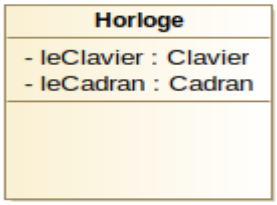
La relation de composition est représentée par un losange plein. Elle indique que la classe **Horloge** possède une instance de la classe **Clavier** nommée **leClavier** et une instance de la classe **Cadran** nommée **leCadran**. Ces deux instances, on parle également de **rôles,** sont privées, le signe – devant le nom l'indique. La cardinalité, les nombres aux extrémités des relations, indique le nombre de classes liées. Ici, une **Horloge** est en relation avec un **Clavier** et un **Cadran**. De même, un **Clavier** n'est en relation qu'avec une **Horloge**, la cardinalité est identique pour le **Cadran**.

La cardinalité est très importante pour l'implémentation en C++ de la relation, elle indique le nombre d'instances nécessaires.

On remarque également que les relations sont orientées, la flèche à l'extrémité de la composition, cela signifie que seule la classe **Horloge** utilise les méthodes des classes **Clavier** et **Cadran**. Ces deux dernières classes n'ont pas besoin de faire appel aux méthodes de la classe **Horloge**. La relation est unidirectionnelle.

3.2. Implémentation de la relation de composition en C++

La relation de composition s'implémente de deux manières en C++, soit en ajoutant un simple attribut dans la classe contenant, soit de manière dynamique en utilisant un pointeur et une allocation mémoire. La deuxième méthode sera étudiée dans un second temps.



Le premier cas de figure revient donc à ajouter les deux attributs, aux autres déjà présents, réalisant ainsi les deux compositions représentées dans le diagramme de classes ci-dessus. C'est une autre manière de représenter la composition. Il faut simplement choisir une des deux représentations.

Dans les deux cas, cela indique que la durée de vie des instances des classes **Clavier** et **Cadran** est liée à celle de la classe **Horloge**. Si l'instance de la classe **Horloge** est détruite, les deux instances de **Clavier** et **Cadran** le sont aussi.

Les instances sont ajoutées dans la section privée de la classe **Horloge** comme l'indique le signe moins précédent le nom des instances. La relation de composition indique qu'une instance d'une classe fait partie d'une autre.

Déclaration de la classe Horloge (suite) **horloge.h**

```
class Horloge
{
public:
    // Déclaration du constructeur et autres méthodes
private:
    // Déclaration des autres attributs
    Clavier leClavier ;       // il ne faut pas oublier #include "clavier.h"
    Cadran  leCadran ;        // il ne faut pas oublier #include "cadran.h"
};
```

Du côté des composants, il n'y a rien de particulier à faire, la relation est ici unidirectionnelle. Le constructeur du contenant doit juste fournir les éventuels paramètres qui sont nécessaires aux composants.

L'exemple de la classe **Clavier** illustre ce propos :

Déclaration de la classe Clavier — Clavier.h

```
#ifndef CLAVIER_H
#define CLAVIER_H
#include <Arduino.h>
class Clavier
{
public:
    enum TOUCHES_CLAVIER {
         AUCUNE = -1,
         MODE,               // Les valeurs suivantes vont croissantes ici MODE = 0, PLUS = 1 ...
         PLUS,
         MOINS,
         FIN
    };
    Clavier(const uint8_t _bps[], const int _nbPoussoirs=4);
    TOUCHES_CLAVIER Scruter();
private:
    uint8_t lesBoutonsPoussoirs[4];
    int nbPoussoirs;
};
#endif // CLAVIER_H
```

Le constructeur de la classe **Clavier** attend un tableau d'entiers non signés sur 8 bits uint8_t et éventuellement le nombre de touches avec une valeur par défaut égale à 4. Le code du constructeur de la classe **Horloge** devient donc :

Implémentation du constructeur de la classe Horloge (suite) — horloge.cpp

```
#include <Arduino.h>
#include "horloge.h"
uint8_t boutonsPousoirs[4] = {36,39,34,35};
Horloge::Horloge(const short _resolution, const short _nbMode):
heures(0),
minutes(0),
resolution(_resolution),
nbMode(_nbMode),
mode(AUCUN_REGLAGE),
leClavier(boutonsPousoirs)
{ }
```

Il est impératif d'utiliser la liste d'initialisation, pour passer les paramètres au constructeur de la classe **Clavier**. Ici, les numéros contenus dans le tableau boutonsPousoirs correspondent aux broches du GPIO de la carte ESP32.

4. Compléments sur le codage en C++

4.1. Le constructeur

L'exemple précédent montre les initialisations réalisées dans le constructeur. La liste d'initialisation fonctionne correctement pour les variables simples, des entiers, des caractères, des réels… Cependant, lorsqu'il s'agit d'un tableau cela ne peut se faire que dans le corps du constructeur, par exemple pour la classe **Clavier** :

Implémentation du constructeur de la classe Clavier — clavier.cpp

```
#include "clavier.h"
Clavier::Clavier(const uint8_t _bps[],const int _nbPoussoirs):
    nbPoussoirs(_nbPoussoirs)        // initialisation avec la liste d'initialisation
{
    if(nbPoussoirs > 4)              // initialisation dans le corps du constructeur
        nbPoussoirs = 4;
    if(nbPoussoirs > 0)              // Le nombre de boutons-poussoirs doit être compris entre 1 et 4
        for(int indice = 0 ;indice < nbPoussoirs ; indice++)
        {
            lesBoutonsPoussoirs[indice] = _bps[indice]; // Recopie du paramètre dans l'attribut
            pinMode(lesBoutonsPoussoirs[indice],INPUT); // Initialisation des broches du GPIO en entrée
        }
}
```

4.2. Utilisation des attributs

L'exemple suivant nous montre l'intérêt des attributs, ils seront partagés par tous les membres de la classe.

Implémentation du constructeur de la classe Clavier — clavier.cpp

```
Clavier::TOUCHES_CLAVIER Clavier::Scruter()
{
    TOUCHES_CLAVIER retour = AUCUNE;
    for(int indice = 0 ;indice < nbPoussoirs ; indice++)
    {
        if(!digitalRead(lesBoutonsPoussoirs[indice]))
            retour = static_cast <TOUCHES_CLAVIER> (indice); // si appui sur plusieurs touches, seule la
    }                                                        // dernière est conservée.
    delay(200);       // pour limiter les rebonds sur les boutons-poussoirs
    return retour;
}
```

Comme le tableau lesBoutonsPoussoirs et la variable nbPoussoirs sont attributs de la classe, ils sont utilisables dans toutes les fonctions de la classe.

Ils ont été initialisés dans le constructeur de la classe Clavier, et sont disponibles pour chaque méthode. La classe **Horloge** ne peut pas agir sur ces attributs hormis lors de l'initialisation du constructeur, car ils sont privés.

4.3. Appel des méthodes en dehors de la classe

La portion de code suivant va permettre de vérifier le bon fonctionnement de la classe **Clavier** avant de l'intégrer à la classe **Horloge**.

Programme de test de la classe Clavier — testClavier.cpp

```
#include <Arduino.h>
#include "clavier.h"
uint8_t leds[4] = {13,12,14,27};     // broches pour les LED 1 à 4
uint8_t bps[4]  = {36,39,34,35};     // broches pour les BP  1 à 4

Clavier leClavier(bps);

void setup()
{
    for (int i=0;i < 4 ;i++ )
        pinMode(leds[i],OUTPUT);
}
void loop()
{
     Clavier::TOUCHES_CLAVIER touche = leClavier.Scruter(); // appel de la méthode
    if (touche != Clavier::AUCUNE)
       digitalWrite(leds[touche],!digitalRead(leds[touche])); // on relit d'abord la valeur de la LED
}
```

Dans l'environnement de développement pour l'ESP32 basé sur celui de l'Arduino. La fonction **setup()** initialise ici, les broches réservées aux LEDs 1 à 4. La fonction **loop()** boucle indéfiniment en effectuant une lecture des touches du clavier et en allumant ou éteignant la LED correspondant au numéro du bouton poussoir.

Pour ne pas instancier à chaque tour de boucle la classe **Clavier**, elle est déclarée en amont de ces deux fonctions. L'appel de la méthode **scruter()** se fait en utilisant le nom de l'instance de la classe **Clavier** suivi d'un point, il ne s'agit pas ici d'un pointeur, sinon le point se serait transformé en flèche.

Dans l'environnement Arduino, il y a un **main**, comme dans tout programme en C ou C++, mais il est caché.

Il se présente ainsi :

mainArduino.cpp

```
void main()
{
    setup();          // initialisation
    while(1)
        loop();       // boucle infinie
}
```

Ce code est utilisé lors de l'édition de lien avec le reste du code qui a été produit , mais n'est à aucun moment à coder lors d'un développement.

4.4. Gestion des chaînes de caractères en C++

Les chaînes de caractères peuvent toujours être gérées à la manière du C avec un tableau de caractères avec une marque de fin de chaîne, le caractère '\0'. Il existe cependant une classe **String** qui facilite la programmation.

La classe **String** est présente dans un environnement C++ classique gcc sous Linux ou autre et avec quelques particularités dans l'environnement Esp32 – Arduino. C'est dans cet environnement que se situe cette partie de cours. Lien vers la documentation : https://www.arduino.cc/reference/en/language/variables/data-types/stringobject/

Avec la classe **String**, il n'est pas nécessaire de définir à l'avance la taille de la chaîne. Le signe égal « = » correspond à l'affectation de la chaîne.

Exemple de déclaration d'une instance de String

```
String chaine1;
chaine1 = "Bonjour"
```

Il existe de nombreux constructeurs pour la classe **String**, ils diffèrent simplement par leurs paramètres. On nomme cette caractéristique du C++ la ***surcharge d'opération***.

Exemple de déclaration d'une instance de String

```
String(const char *cstr = "");                  // init à partir d'un tableau de caractères
String(const char *cstr, unsigned int length);  // idem en indiquant le nombre de caractères utiles
String(const String &str);                      // Constructeur de copie
String(char c);                                 // init à partir d'un caractère
String(unsigned char, unsigned char base = 10); // Pour les suivantes init à partir d'une valeur
String(int, unsigned char base = 10);           // numérique en précisant la base pour les entiers
String(unsigned int, unsigned char base = 10);  // DEC par défaut. Il existe aussi HEX pour hexa
String(long, unsigned char base = 10);          // et BIN pour binaire.
String(unsigned long, unsigned char base = 10);
String(float, unsigned char decimalPlaces = 2); // et le nombre de décimales pour les réels.
String(double, unsigned char decimalPlaces = 2);
```

Les principaux opérateurs d'affectation et de comparaison ont été surchargés pour cette classe, ainsi :

	Description	**Exemple :** String s1, s2
[ ]	Accès à un élément, comme pour un tableau de caractères	s1[3]
+	Concaténation de deux chaînes	s1 + s2
+=	Ajout	s1 += s2
==	Comparaison, les deux chaînes sont-elles identiques ?	s1 == s2
>	La chaîne 1 est-elle après la chaîne 2 dans l'ordre lexicographique	s1 > s2
>=	La chaîne 1 est-elle après la chaîne 2 dans l'ordre lexicographique ou identique	s1 >= s2
<	La chaîne 1 est-elle avant la chaîne 2 dans l'ordre lexicographique	s1 < s2
<=	La chaîne 1 est-elle avant la chaîne 2 dans l'ordre lexicographique ou identique	s1 <= s2
!=	Comparaison, les deux chaînes sont-elle différentes ?	s1 != s2

Remarque : la concaténation et l'ajout peuvent se faire avec une chaîne de caractère tableau de caractères ou String, un caractère, un nombre entier, quelle que soit sa taille, un nombre réel double ou float.

Des méthodes permettent la conversion de **String** en un autre type :

toCharArray()	Recopie la chaîne dans un tableau
toInt()	Conversion en entier
toFloat()	Conversion en float
toDouble()	Conversion en double
toLowerCase()	Conversion de la chaîne en minuscules
toUpperCase()	Conversion de la chaîne en majuscules

Exemples de conversion

```
char tabCar[20];
String s1("Bonjour");
s1.toCharArray(tabCar,7);

String s2(152);
int val2 = s2.toInt();

s1.toUpperCase();
```

D'autres méthodes réalisent des traitements classiques sur les chaînes de caractères comme en langage C.

length()	Retourne la longueur de la chaîne en caractère.
remove()	Modifie la chaîne en supprimant tout ou partie des caractères à partir d'un index
replace()	Modifie la chaîne en remplaçant toutes les sous-chaînes définies par une autre
substring()	Permet le découpage en sous-chaîne à partir d'un index ou d'un index à un autre
reserve()	Pre-alloue de la mémoire pour une chaîne
startsWith()	Vérifie qu'une chaîne commence bien par un caractère ou une chaîne donnée
endsWith()	Vérifie qu'une chaîne termine bien par un caractère ou une chaîne donnée
c_str()	Converti une chaîne de caractère en une chaîne constante de type C, se terminant par '\0'
equalsIgnoreCase()	Compare l'égalité de deux chaînes sans se préoccuper des majuscules et des minuscules
indexOf()	Localise le caractère ou la chaîne dans une autre en retournant la position.
lastIndexOf()	Idem en partant de la fin de la chaîne
trim()	Supprime tous les espaces et tabulations au début et à la fin de chaîne

Exercice : Complétez le code permettant de formater la chaîne de caractère sous la forme d'une String pour afficher les heures et les minutes. L'affichage des heures et des minutes doit toujours se faire sur 2 chiffres. Les heures et les minutes sont séparées par le caractère « : ».

Formatage d'une String

```
int heures = 10;
int minutes = 5;
String heuresMinutes;

// poursuivez...
```

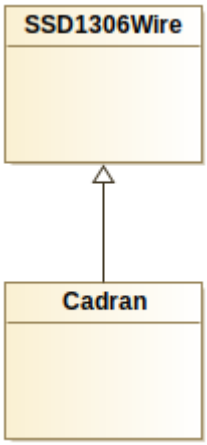
5. Relation d'héritage

5.1. Introduction

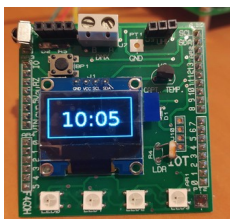
Afin d'accroître le modèle de développement en couche, le C++ introduit la notion d'héritage. Cela consiste à fabriquer une nouvelle classe à partir d'une classe de base pour y ajouter de nouvelles fonctionnalités sans pour autant avoir besoin d'accéder au code de la classe de base.

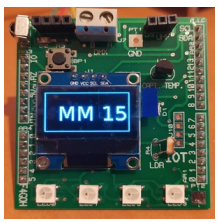
Dans notre exemple d'horloge, on peut très bien imaginer une version pour un ESP32 avec un cadran basé sur un afficheur OLED et une version Linux avec un cadran basé sur un affichage avec un écran d'ordinateur. Il y a la couche de l'Horloge, Cadran, Clavier et une couche inférieure avec le pilote de l'écran OLED. L'essentiel est de ne pas avoir à réécrire la classe horloge si son cadran change.

5.2. Représentation UML de l'héritage



La relation d'héritage se représente par un triangle dans le diagramme de classe UML. La classe **Cadran** hérite de la classe **SSD1306Wire** qui gère l'écran OLED SSD1306 sur le bus I2C. La classe SSD1306Wire est la ***classe de base***, Le Cadran est une ***classe dérivée***.
Le cadran peut utiliser directement les membres publiques de la classe de base, car le Cadran est ***une sorte de*** SSD1306Wire spécialisé pour afficher les heures et les minutes d'une horloge. On parle également de ***spécialisation***.





5.3. Codage en C++ de la relation d'héritage

Classe Cadran **cadran.h**

```
#ifndef CADRAN_H
#define CADRAN_H
#include <Arduino.h>
#include <SSD1306Wire.h>
class Cadran : public SSD1306Wire // Relation d'héritage
{
public:
    Cadran ();
    void Afficher (const int _heures, const int _minutes);
    void Afficher (const String _chaine, const int _valeur);
};
#endif // CADRAN_H
```

La définition du constructeur de la classe **Cadran** doit initialiser celui de la classe SSD1306Wire et préparer l'affichage sur cet écran.

Constructeur de la classe Cadran **cadran.cpp**

```
#include "cadran.h"
Cadran::Cadran():
    SSD1306Wire(0x3c,SDA,SCL, GEOMETRY_128_64)
{
    init();
    flipScreenVertically();
}
```

On remarque que les méthodes de la classe **SSD1306Wire** sont appelées directement sans passer par une instance, la classe **Cadran** est une sorte d'écran OLED. Elle possède le même comportement

Exercice : Réalisez le codage des deux méthodes Afficher() ayant fait l'objet d'une surcharge, vous pourrez utiliser la classe String pour vous aidez à la mise en forme de l'affichage.

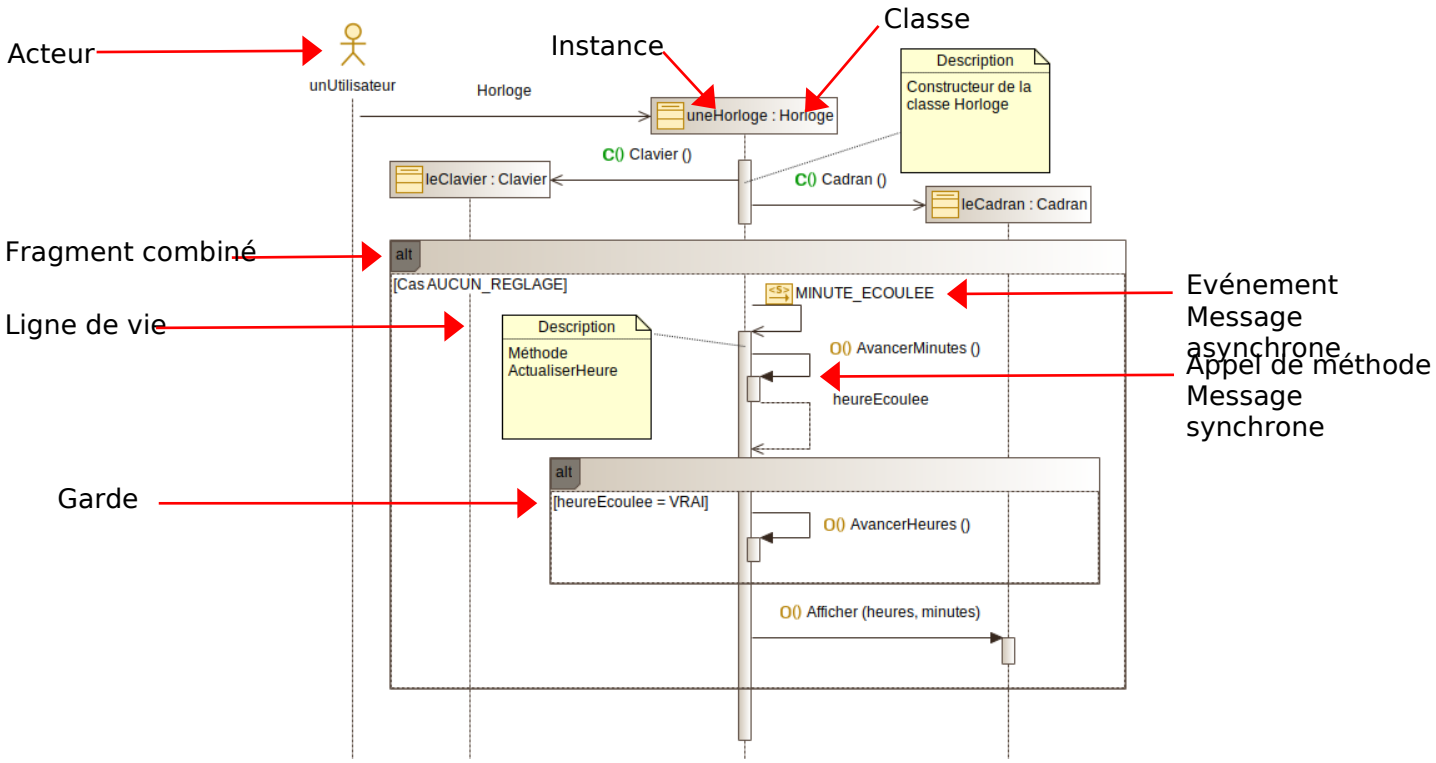
- La première affiche les heures et les minutes sur l'écran OLED en police 32 sous la forme hh:mm.
- La seconde permet l'affichage pour le mode réglage des heures et des minutes. Sous la forme HH xx ou MM yy, la police des lettres majuscules est une police de 16
- L'affichage des deux méthodes sur l'écran se fait dans un rectangle sur les bords de l'afficheur OLED.

6. Interaction entre les objets

D'autres diagrammes UML permettent de montrer les interactions entre les différents objets d'une application. C'est le cas du diagramme de séquence. Le diagramme fait apparaître un acteur ici unUtilisateur, il joue un rôle actif pour le système, ici il met en fonctionnement l'horloge et effectuera des réglages.

6.1. Diagramme de séquence

L'application possède une instance de la classe **Horloge** nommée **uneHorloge**. Son constructeur crée l'instance de la classe **Clavier leClavier** et l'instance de la classe **Cadran leCadran**. Il s'agit bien des instances qui ont été définies par les relations de composition vues dans le diagramme de classes.



Ce diagramme se lit de haut en bas. Il montre l'enchaînement des différents appels de méthodes et le nom de chaque instance. Les fragments combinés sont ici des alternatives ALT. Ils représentent le schéma conditionnel pour un diagramme de séquence.

6.2. Codage C++ du diagramme de séquence

Chaque minute, la méthode **ActualiserHeure()** est appelée. La définition de la méthode peut se traduire du diagramme de séquence ci-dessus.

Exercice : Complétez le codage de cette méthode.

Méthode ActualiserHeure **Horloge.cpp**

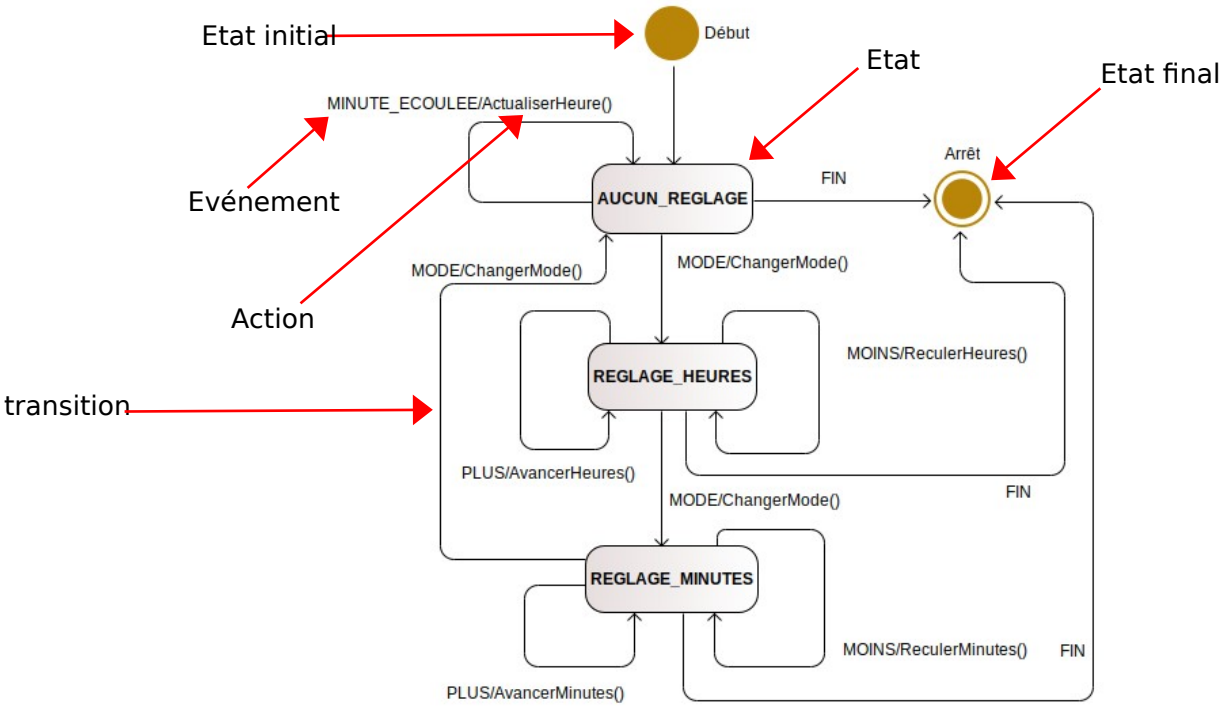
```
void Horloge::ActualiserHeure()
{




}
```

7. Comportement des objets

7.1. Diagramme états-transitions

Le comportement d'un objet est représenté sous UML par un diagramme états-transitions. Il montre comment l'objet réagit aux sollicitations extérieures en provenance des acteurs et éventuellement aux événements internes au système.

L'exemple ci-dessous montre le comportement de la classe Horloge.



Les états sont représentés par des rectangles avec des coins arrondis. Les transitions par des flèches montrant le sens du passage d'un état à l'autre. Les transitions sont la conséquence d'événements qui déclenche éventuellement des actions qui sont des méthodes de la classe.

Seules les classes faisant l'objet de sollicitations, comme des événements, nécessitent d'expliciter sous forme de diagramme états-transitions leur comportement. Bien souvent, seul le comportement de la classe principale est représenté ainsi.

7.2. Codage en C++ du diagramme états-transitions

Le codage en C++ d'un diagramme états-transitions peut être réalisé de manière systématique.

- Une constante énumérée reprend tous les états du système.
- Une variable contient l'état courant de l'objet
- Une instruction switch() … case … explicite les actions à mener pour chaque état.

Dans notre exemple, c'est la méthode **Controler()** de la classe **Horloge** qui va regrouper cette suite d'instruction. Elle sera appelée cycliquement pour faire évoluer le comportement de l'horloge en fonction du temps qui s'écoule et des sollicitations de l'utilisateur par l'appui sur les touches du clavier.

Définition des modes de la classe Horloge **Horloge.h**

```
enum MODE_HORLOGE {
    AUCUN_REGLAGE,
    REGLAGE_HEURES,
    REGLAGE_MINUTES
} ;
```

La constante énumérée est à ajouter à la section publique dans la déclaration de la classe Horloge.

La structure de la méthode Controler() est proposée ici, elle est ajoutée au fichier horloge.cpp. C'est la structure type pour le codage de chaque diagramme états-transitions. La méthode retourne ici le code de la touche du clavier si l'utilisateur le sollicite pour la gestion de fin de programme. C'est un choix arbitraire de programmation.

Reste à compléter la séquence correspondant à chaque état en utilisant les différentes transitions présentes sur le diagramme UML.

Méthode controler **Horloge.cpp**

```
TOUCHES_CLAVIER Horloge::Controler(TOUCHES_CLAVIER numTouche)
{
    switch (mode)
    {
        case AUCUN_REGLAGE:

            break;
        case REGLAGE_HEURES:

            break;
        case REGLAGE_MINUTES:

            break;
    }

    return leClavier->ScruterClavier();
}
```

Exercice : À partir du diagramme états-transitions, complétez le codage de cette méthode.

Par la suite, le programme principal de l'application ressemblera à la suite d'instructions suivantes :

Programme principal dans l'environnement ESP32 **main.cpp**

```
#include <Arduino.h>
#include "horloge.h"
#include "clavier.h"

Horloge uneHorloge;
Clavier::TOUCHES_CLAVIER touche ;

void setup()
{
        touche = Clavier::AUCUNE ;
}
void loop()
{
  touche = uneHorloge.Controler(touche);
}
```

Dans l'environnement ESP32, le programme ne s'arrête pas, la gestion de l'état final n'a pas été implémentée. Dans un autre environnement, cela pour faire l'objet de la condition de sortie de la boucle.

8. Allocation dynamique

8.1. Variables automatiques

Jusqu'à présent, la déclaration de variables ou l'instanciation de classes s'est faite de manière ***automatique***. Lors de la déclaration dans un bloc, c'est-à-dire entre les accolades, le programme demande au système d'exploitation la possibilité d'utiliser une zone mémoire pour le type de données à stocker, le système réserve cet emplacement et indique au programme l'adresse de la zone mémoire. À la fin du bloc, la variable est automatiquement supprimée et la mémoire est libérée.

Allocation mémoire automatique — **main.cpp**

```
uint8_t leds[4] = {13,12,14,27};     // broches pour les LEDs 1 à 4
void setup()
{
    int numLed ;
    for (numLed=0;numLed < 4 ;numLed++ )
        pinMode(leds[i],OUTPUT);
}
```

L'entier **numLed** est ici déclaré au début de la fonction **setup()**, le système alloue automatiquement la mémoire pour stocker l'entier. Il est accessible dans le bloc constituant le corps de la fonction. À la fin de la fonction, il est détruit et la mémoire est libérée.

Remarque : Le tableau **leds** est déclaré de manière globale, en dehors de tout bloc. Il existe jusqu'à la fin du programme. Il est accessible pour l'ensemble des fonctions du fichier. Ce type de déclaration est à proscrire de manière générale, la variable n'est pas protégée, son intégrité est menacée. C'est souvent la seule façon de faire dans le cadre de la programmation type ESP32, car la fonction **main()** n'est pas accessible.

8.2. Variables dynamiques

L'allocation dynamique est une allocation ***manuelle*** de la mémoire. En fonction des besoins du programme, la mémoire est allouée par l'opération **new**. La zone mémoire obtenue est disponible et utilisable par le programme, dès l'instant ou l'adresse de la zone est accessible, et ce, jusqu'à la demande la libération par l'opération **delete**. Ainsi, il est nécessaire de déclarer un ***pointeur***, variable contenant l'adresse, permettant la localisation de la zone mémoire allouée.

Les pointeurs sont typés en fonction de la donnée à stocker. Cette caractéristique est utile pour l'arithmétique des pointeurs, ainsi lorsqu'un pointeur est incrémenté, celui-ci désigne la variable suivante dans un ensemble de données, l'adresse a été augmentée de la taille occupée en mémoire par la variable.

Allocation dynamique et restitution de la mémoire — **horloge**

```
#include "cadran.h"
#include "clavier.h"

class Horloge {
public:
    Horloge();
    ~Horloge();
private:
    Clavier *leClavier;       // déclaration des pointeurs en tant qu'attributs de la classe
    Cadran *leCadran;         // Les instances seront utilisables par toutes les méthodes de la classe
};

Horloge::Horloge() // Allocation dynamique de la mémoire, instances disponibles jusqu'à la libération
{
    leCadran = new Cadran(5,5);
    leClavier = new Clavier;
}

Horloge::~Horloge()   // Libération de la mémoire, Les instances sont détruites
{
    delete leCadran;
    delete leClavier;
}
```

Remarque : Dans le cadre de la programmation type ESP32, cette technique permet de retarder l'appel du constructeur, laissant le temps au matériel de terminer son initialisation.

8.3. Tableaux dynamiques

Un autre avantage propre au langage C++ du mécanisme d'allocation dynamique de mémoire, est la possibilité de dimensionner dynamiquement la taille d'un tableau. Ainsi la réservation de la mémoire pour un tableau peut se faire en cours de programme, elle peut donc s'adapter au besoin et non pas être figée au moment de la compilation.

L'allocation, dans ce cas, est réalisée avec l'opérateur **new [ ]** et la libération doit **impérativement** être effectuée avec l'opérateur **delete [ ]**. Le contenu du tableau est alors détruit dans l'ordre décroissant des indices. Le fait de ne pas utiliser les [ ] lors de la destruction rend le programme instable.

La classe **Clavier** peut donc être revue de la manière suivante :

Nouvelle déclaration de la classe Clavier — **Clavier2.h**

```
#ifndef CLAVIER_H
#define CLAVIER_H
#include <Arduino.h>
class Clavier
{
public:
    enum TOUCHES_CLAVIER {
         AUCUNE = -1,
         BP1,                // Les valeurs suivantes vont croissantes ici MODE = 0, PLUS = 1 ...
         BP2,
         BP3,
         BP4
    };
    Clavier(const uint8_t _bps[], const int _nbPoussoirs=4);
    ~Clavier();                      // Il faut prévoir un destructeur pour libérer la mémoire
    TOUCHES_CLAVIER Scruter();
private:
    uint8_t *lesBoutonsPoussoirs;    // Pointeur pour désigner le tableau
    int nbPoussoirs;
};
#endif // CLAVIER_H
```

Le tableau des broches pour les boutons-poussoirs s'adapte en fonction du nombre.

Implémentation du nouveau constructeur de la classe Clavier — **clavier2.cpp**

```
#include "clavier2.h"
Clavier::Clavier(const uint8_t _bps[],const int _nbPoussoirs):
    nbPoussoirs(_nbPoussoirs)
{
    if(nbPoussoirs > 4)
        nbPoussoirs = 4; // Limite le nombre de boutons poussoirs à 4
    if(nbPoussoirs > 0)
    {
        lesBoutonsPoussoirs = new uint8_t[nbPoussoirs]; // allocation dynamique du tableau

        for(int indice = 0 ;indice < nbPoussoirs ; indice++)
        {
            lesBoutonsPoussoirs[indice] = _bps[indice];
            pinMode(lesBoutonsPoussoirs[indice],INPUT);
        }
    }
    else lesBoutonsPoussoirs = nullptr; // On fixe une valeur au pointeur si l'allocation n'a pas lieu
}
Clavier::~Clavier()
{
    if(lesBoutonsPoussoirs != nullptr)  // la destruction ce fait uniquement si l'allocation a eu lieu
        delete [] lesBoutonsPoussoirs;
}
```

9. Complément sur la notion d'héritage

9.1. Accès aux membres de la classe de base

Précédemment, lorsque la relation d'héritage a été abordée, il ne s'agissait que d'utiliser les membres publics de la classe de base. Le **Cadran** faisait uniquement appel aux méthodes de la classe **SSD1306Wire** pour fonctionner.

Maintenant, on peut imaginer qu'***un réveil est une sorte d'horloge***. La relation d'***héritage*** paraît tout à fait bien se prêter à la situation. Un réveil doit pouvoir gérer les heures et les minutes de la même manière qu'une horloge, il est juste nécessaire de lui ajouter les fonctionnalités liées à la gestion d'une alarme.

Par exemple, si **heureAlarme** et **minuteAlarme** correspondent à l'heure et à la minute ou l'alarme du réveil doit être déclenchée, il est nécessaire de pouvoir les comparer aux attributs **heures** et **minutes** de la classe **Horloge**. Or, ceux-ci ne sont pas accessibles, car ils sont privés. Seuls les membres de la classe **Horloge** peuvent le faire. Une première solution consisterait à les placer dans la section publique de la classe. La notion d'objet perdrait alors son intérêt. Les concepteurs du langage C++ ont prévu une notion intermédiaire, **protégé**. Un membre placé dans la section protégée de la classe est accessible aux membres de la classe où il est déclaré et dans les classes dérivées. Il reste invisible pour l'extérieur de la classe.

La déclaration de la classe Horloge doit être revue de la manière suivante :

Classe Horloge version avec Reveil **horloge.h**

```
#ifndef HORLOGE_H
#define HORLOGE_H
#include "clavier.h"
#include "cadran.h"
class Horloge {
public:
    Horloge(const short _nbMode = 3, const short _resolution = 24);
    ~Horloge();

    enum MODE_HORLOGE {     /// Différents modes de l'horloge
        AUCUN_REGLAGE,      /// Fonctionnement normal
        REGLAGE_HEURES,
        REGLAGE_MINUTES
    };

    void ActualiserHeure();
    bool AvancerHeures();
    bool AvancerMinutes();
    void ReculerHeures();
    void ReculerMinutes();
    Clavier::TOUCHES_CLAVIER Controler( Clavier::TOUCHES_CLAVIER numTouche);
    void ChangerMode();

protected:              // les attributs sont rendus accessibles aux classes dérivées
    int heures;         /// Valeur des heures courantes
    int minutes;        /// Valeurs des minutes courantes
    int resolution;     /// Résolution de l'horloge 12 ou 24 (valeur par défaut)
    const short nbMode; /// Nombre de modes de l'horloge
    short mode;         /// Mode courant de l'horloge
    uint64_t tempsPrec;  /// Valeur précédente du temps

    Clavier *leClavier; /// Composition de la classe Clavier
    Cadran  *leCadran;  /// Composition de la classe Cadran
};
```

C'est l'unique modification à apporter à la classe **Horloge** pour qu'elle puisse être dérivée et devenir un réveil par exemple. Ce réveil aura également accès au **Clavier** et au **Cadran** qui sont liés à la classe **Horloge** puisqu'ils sont dans la section privée.

Remarque : Si on souhaite rendre inaccessibles aux classes dérivées certains membres de la classe de base, il faut absolument les laisser dans la section privée de la classe de base.

9.2. Ordre d'appel des constructeurs et destructeurs

Lorsqu'une classe dérivée est instanciée, son constructeur est appelé après celui de la classe de base. En effet, il est judicieux de laissé la classe dérivée terminer la construction de l'objet, car elle peut-être amené à modifier certains paramètres de la classe de base.

Lors de sa destruction c'est le contraire, l***e destructeur de la classe dérivée est d'abord appelé ensuite celui de la classe de base***. Le premier rétablit la situation engendrée par la classe dérivée, le second termine. De toute manière, il ne sait pas ce qu'a fait la classe dérivée.

Classe Reveil **reveil.h**

```
#ifndef REVEIL_H
#define REVEIL_H
#include "horloge.h"
#include "sonnerie.h"
class Reveil : public Horloge
{
public:
    Reveil(const int _nbMode=6,const int _resolution=24);
    ~Reveil();
    enum MODE_REVEIL
    {
        REGLAGE_HEURE_ALARME = 3, // Les modes du réveil font suite à ceux de l'horloge
        REGLAGE_MINUTE_ALARME,
        ACTIVATION_ALARME
    };
    void AvancerHeuresAlarme();
    void AvancerMinutesAlarme();
    void ReculerHeuresAlarme();
    void ReculerMinutesAlarme();
    void SurveillerAlarme();
    Clavier::TOUCHES_CLAVIER Controler(const Clavier::TOUCHES_CLAVIER _numTouche);
private:
    int heureAlarme;
    int minuteAlarme;
    bool alarme;
    bool enAlarme;  // pour pouvoir arrêter l'alarme
    Sonnerie laSonnerie;
};
#endif // REVEIL_H
```

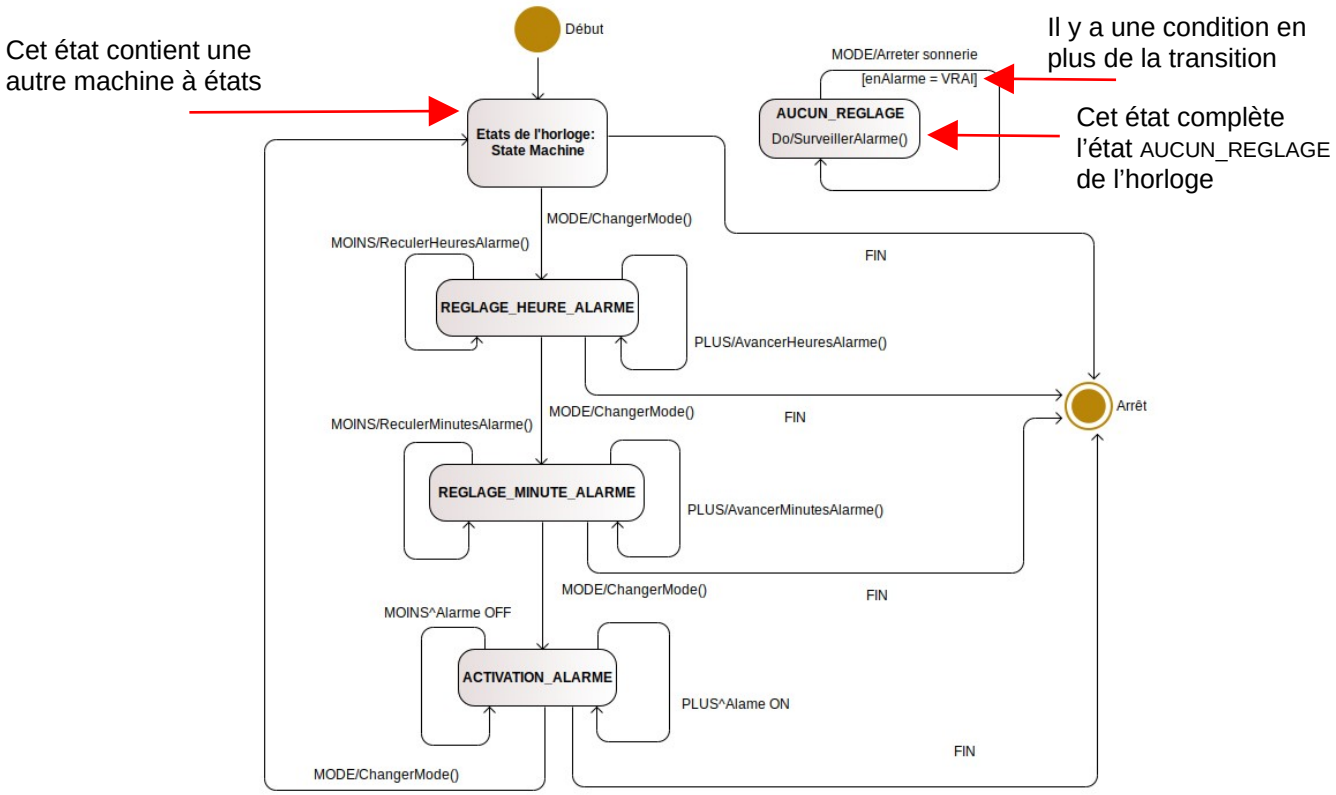
Le constructeur de la classe Reveil transmet en premier lieu les paramètres au constructeur de la classe Horloge

Constructeur de la classe Reveil **reveil.cpp**

```
Reveil::Reveil(const int _nbMode, const int _resolution):
    Horloge(_nbMode,_resolution),    // Apple du constructeur de la classe de base en 1er.
    heureAlarme(0),
    minuteAlarme(0),
    alarme(false),
    enAlarme(false)
{
}
```

9.3. Appel d'une méthode surchargée de la classe de base

Parfois, il est nécessaire d'appeler une méthode de la classe de base qui a été surchargée dans la classe dérivée. C'est le cas avec la méthode **controler()** dans notre réveil. Comme le montre le diagramme états-transitions, les états de l'horloge font partie des états du réveil.



Du point de vue du codage, cela va se traduire de la manière suivante :

Extrait de la méthode Controler de la classe Reveil **reveil.cpp**

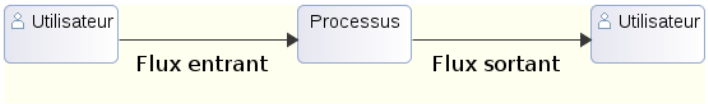
```
Clavier::TOUCHES_CLAVIER Reveil::Controler(const Clavier::TOUCHES_CLAVIER _numTouche)
{
    Clavier::TOUCHES_CLAVIER touche = Horloge::Controler(_numTouche);
    switch(mode)
    {
    case AUCUN_REGLAGE:
        SurveillerAlarme();
        if(touche == Clavier::MODE && enAlarme) // la touche mode arrête la sonnerie si on est en alarme
        {
            laSonnerie.Arreter();
            touche = Clavier::AUCUNE;
            enAlarme = false;
        }
        break;
    case REGLAGE_HEURE_ALARME:
                // code pour le réglage de l'heure de l'alarme
        break;
    case REGLAGE_MINUTE_ALARME:
                // code pour le réglage de la minute de l'alarme
        break;
    case ACTIVATION_ALARME:
                // code pour activer ou pas l'alarme du réveil
        break;
    }
    return static_cast<Clavier::TOUCHES_CLAVIER> (touche);
}
```

Pour l'appel, il est nécessaire de faire précéder le nom de la méthode du nom de la classe de base suivit des **::**, l'opérateur de déréférentiation, sinon on appelle la méthode de la classe dérivée.

10. Entrées / Sorties en utilisant les flux

10.1. Généralités

Tout comme le langage C, le C++ ne dispose pas de mots clés spécifiques pour réaliser les entrées / sorties avec l'utilisateur ou plus généralement avec le système. Il dispose de librairies standards qui permettent de mettre en œuvre cela. En C++, les entrées / sorties sont réalisées à l'aide de mécanismes nommés **flux**. Un flux ou *stream* en anglais est une manière abstraite de représenter un flot de données entre un producteur d'information et un consommateur.



Le flux est toujours considéré du point de vue du processeur. C'est le processus qui reçoit l'information par le flux entrant et qui l'envoie à travers le flux sortant.

Les entrées et sorties utilisant les flux sont définies par deux classes déclarées dans le fichier d'en-tête ***<iostream>*** :

- ***ostream*** pour *Output stream*, définit le flux sortant. Cette classe surcharge l'opérateur d'insertion **<<**.
- ***istream*** pour *Input stream*, définit le flux entrant. Cette classe surcharge l'opérateur d'extraction **>>**.

10.2. Flux standards

Tout comme le langage C utilise l'entrée standard ***stdin*** et la sortie standard ***stdout***, le C++ associe un flux sortant vers cette sortie et un flux entrant depuis cette entrée. On trouve également un flux vers la sortie d'erreur et un dernier vers la sortie technique ou sortie d'information.

- ***cout*** : écrit vers la sortie standard, l'écran,
- ***cin*** : lit à partir de l'entrée standard, le clavier,
- ***cerr*** : écrit sur la sortie d'erreur, cette sortie est non tamponnée,
- ***clog*** : écrit sur la sortie technique, cette sortie est tamponnée.

Ces quatre instances représentent les flux standards en C++. Elles sont définies dans l'espace de nommage ***std***. Les deux dernières instances sont utilisées pour afficher ou mémoriser les messages d'erreur ou d'information. L'instance ***std::cerr*** ne possède pas de tampon cela implique que le message d'erreur s'affiche directement sans attendre que le tampon de sortie soit plein ou l'utilisation des manipulateurs ***std::flush*** ou ***std::endl***.

Le manipulateur ***std::endl*** écrit une fin de ligne **'\n'** sur le flux sortant puis vide le tampon. Le manipulateur ***std::flush*** vide uniquement le tampon d'écriture, le flux est physiquement envoyé sur l'unité de sortie.

Écriture sur la sortie standard ***bienvenue.cpp***

```
#include <iostream>          // pour cout
using namespace std;  // évite d'écrire std::cout

int main()
{
   cout << "Bienvenue en C++" ; // idem printf("Bienvenue en C++"); du langage C
   return 0;
}
```

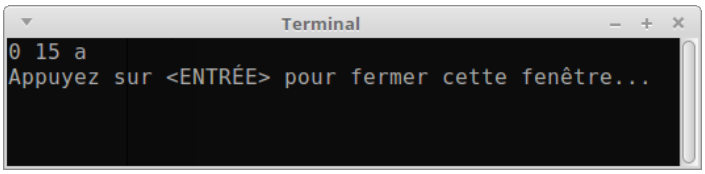
Écriture :

La classe ***ostream*** permet de réaliser un affichage formaté à la manière du ***printf*** de la librairie ***stdio*** en langage C. Ce formatage est beaucoup plus simple puisqu'il suffit d'enchaîner les opérateurs **<<,** comme le montre l'exemple ci-dessous. La surcharge de l'opérateur a été réalisée pour tous les types simples du C++ et les chaînes de caractères.

Écriture formatée ***formatage.cpp***

```
#include <iostream> // pour cout

using namespace std;        // évite d'écrire std::cout

int main()

{
      bool sortie = false;
      int  entier = 15;
      char car = 'a';
      cout << sortie << " " << entier << " " << car  << endl;
      return 0;
}
```

Le résultat obtenu est présenté à la figure suivante :



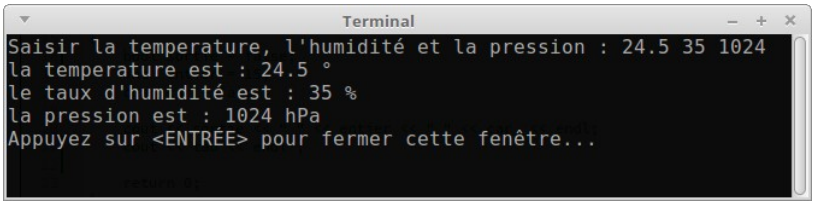
Lecture :

La classe ***istream*** permet la saisie de données à la manière du ***scanf*** de la librairie ***stdio*** du langage C. Pour cette classe, l'opérateur **>>** a été surchargé pour l'ensemble des types simples et les chaînes de caractères.

Lecture ***lecture.cpp***

```
#include <iostream>     // pour cin et cout
using namespace std;

int main()
{
    float temperature;
    float humidite;
    int pression;

    cout << "Saisir la température, l'humidité et la pression : " ;
    cin >> temperature >> humidite >> pression ;

    cout  << "la température est : " << temperature << " °"  << endl;
    cout  << "le taux d'humidité est : " << humidite << " %" << endl;
    cout  << "la pression est : " << pression <<  " hPa" << endl ;

    return 0;
}
```

La saisie des différentes données doit être séparée par un espace, une tabulation, ou un retour chariot. La dernière saisie est prise en compte après un retour chariot [**Entrée**].



10.3. Les manipulateurs de flux d'entrée-sortie

Ces manipulateurs sont des objets dont l'insertion dans un flux en modifie le fonctionnement, voici quelques manipulateurs les plus courament utilisés

Rôle	Manipulateur	Remarque
Affichage des caractères en majuscule	uppercase	
	nouppercase	Valeur par défaut
Affichage des nombres dans une base des bases : décimale, hexadécimale et octale (voir setbase)	dec	Valeur Par défaut
	hex	
	oct	
Mise en forme de l'affichage : aligné à gauche ou à droite	left	
	right	
Notation des nombres flottants	fixed	
	scientific	
Écriture de la fin de ligne '\n' sur le flux de sortie et vidage du tampon	endl	applicable avec cout
Spécifier la largeur d'une zone	setw(int n)	**n** le nombre de caractères
Spécification du caractère de remplissage	setfill(char c)	**c** le caractères de remplissage, par défaut espace
Spécification de la base	setbase(int base)	La base est 8, 10, 16 équivalent à oct, dec, hex.
Spécification du nombre de chiffres significatifs	setprecision(int n)	Nombre de chiffres après la virgule

Exemple

Exemple de manipulateurs — ***manipulateurs2.cpp***

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int val = 192;
    float val2 = 3.141592654;

    cout  << "Affichage par défaut     : " << val << endl;
    cout  << "Affichage en hexadécimal : " << hex << val << endl;
    cout  << "Affichage en décimal     : " << dec << val << endl;
    cout << hex << val << " " << uppercase << val << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << "|" << setw(20) << left << "abc" << "|" << endl;
    cout << "|" << setw(20) << right << "abc" << "|" << endl;
    cout << "+" << setfill('-') << setw(21) << "+" << setfill(' ') << endl;
    cout << val2 << " " << fixed << val2 << " " << scientific << val2 << " ";
    cout << fixed << setprecision(2) << val2 << endl;
    cout << "Entrez un nombre en octal : " ;
    cin >> oct >> val ;
    cout << "vous avez saisie " << dec << val << " en décimal" << endl;
    return 0;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Affichage par défaut     : 192
Affichage en hexadécimal : c0
Affichage en décimal     : 192
c0 C0
+-------------------+
|abc                |
|                abc|
+-------------------+
3.14159 3.141593 3.141593E+00 3.14
Entrez un nombre en octal : 770
vous avez saisie 504 en décimal
Appuyez sur <ENTREE> pour fermer cette fenêtre...
```