Référence/type[1]	LOO C++ AP5		☐ TD ☒ AP
Thématique principale	Application multithread en C++ (moderne)		
Thématique(s) secondaire(s)	✓ Réalisation d'interface ✓ Implémentation diagramme de classes UML ✓ Exceptions spécifiques (dérivées de std::except) ✓ Programmation réseau		
Durée	2h	***Niveau[2]***	A
Prérequis	✓ Notions de multithreading		
Compétences opérationnelles et niveau cible[3]	✓ Définir et réaliser une interface (***A***) ✓ Créer et démarrer des threads (std::jthread) (***M***) ✓ Sécuriser l'accès à des ressources partagées (via smart mutex) (***M***) ✓ Créer des classes dérivées de std::except afin de créer des exceptions spécifiques (***M***)		
Modalités et critères d'évaluation	Evaluation en cours de séance		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]	Netcat (ou équivalent) pour version avec capteur « réseau ».		
Ressources			

Avant la séance…

Cette Activité Pratique a plusieurs objectifs qui s'inscrivent tous dans un même but global : préparer le passage vers le développement en C++ sur cible embarquée. En effet, la plupart des applications embarquées doivent exécuter, de manière plus ou moins synchronisée et de manière sûre plusieurs tâches. Cette mécanique peut être mise en place en bare-metal via l'utilisation intensive des interruptions. Une autre stratégie consiste en l'utilisation d'un RTOS, qui propose des outils (standard ou propriétaires) permettant d'implémenter « facilement » une application multithread. Une maîtrise a minima de ces outils (création et exécution de threads, protection de ressources par mutexes, synchronisation…) est donc nécessaire. L'un des objectifs de cette activité pratique est de mettre en œuvre une partie de ces outils sur cible PC « standard » (la bibliothèque standard C++ gère le multithread).

Nous consoliderons aussi la notion d'interface et de réalisation d'interface, concepts régulièrement mis en œuvre dans l'embarqué.

Enfin, nous évoquerons le concept de classe applicative et tenterons d'en identifier les principaux avantages et inconvénients.

[1] TD : ½ groupe, apports de cours, mise en pratique guidée – TP : ½ groupe, travail autonome majoritaire.

[2] Le niveau se rapporte à la thématique principale. Il peut ici être entendu comme un indicateur de la difficulté du travail (N-Facile, A-Sans grande difficulté, M-Quelques points complexes, MA-Difficile)

[3] Les niveaux cible correspondent pour chaque compétence opérationnelle au niveau d'acquisition de la celle-ci à l'issue du travail dans son intégralité (en incluant les phases préparatoires et de synthèse).

[4] En plus d'un environnement de développement C++

Travail encadré en séance

Apports de cours

Activités

Nous allons travailler sur une application qui pourrait trouver sa place dans un système embarqué : un ***thermostat d'ambiance*** basique : mesure de la température ambiante et pilotage d'une chaudière en fonction de l'écart entre la température mesurée et une température de consigne.

On part du principe que l'application doit être analysée pour un environnement d'exécution nativement multithread.

Les contraintes / exigences suivantes pour la version à développer aujourd'hui doivent être prises en compte :

- ✓ Concernant le captage de la température, différents types de capteurs doivent pouvoir être utilisés sans que cela n'impacte la partie « haut niveau » de l'application.
- ✓ Pour la version de l'application à développer, le capteur sera simulé par un des deux objets suivants (au choix) :
  - Objet « DummyCapteurTemp », retournant une température « au hasard » comprise entre un min et un max. Cependant, afin de rester assez proche de la réalité, la différence entre deux températures successives ne devra pas dépasser un « delta » maximum. Ces trois informations (a minima) seront nécessaires à la construction du « capteur ».
  - Objet « NetworkCapteurTemp », mettant en place un serveur UDP sur un port spécifique ou par défaut. L'envoi d'une donnée brute à ce serveur, représentant la température mesurée, en 1/10°C doit permettre de mettre à jour l'image de la température.
- ✓ La mise en chauffe se fait via une chaudière unique, éventuellement partagée avec d'autres processus. Plusieurs types de chaudières doivent pouvoir être envisagés.
- ✓ La chaudière concrète liée à cet exemple matérialise la mise en marche et l'arrêt par des messages console ou log.
- ✓ Les fréquences de mesure de la température et de « processing » (comparaison à la consigne et action éventuelle sur la chaudière) doivent être décorrélées.
- ✓ La température ambiante à prendre en compte est la moyenne des 5 dernières mesures
- ✓ Les températures de consigne ainsi que la valeur de l'hystérésis ne sont pas modifiables une fois que l'application a été compilée.

Organisation du travail (proposition)

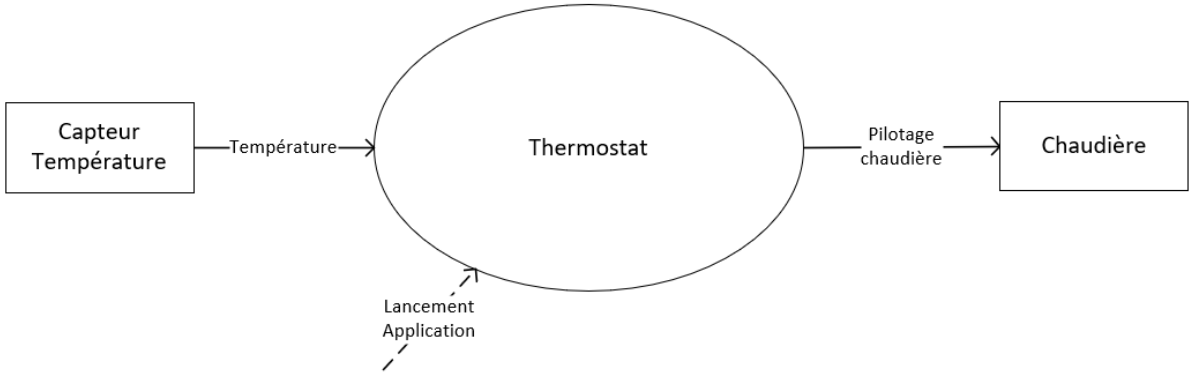
1. Récupérer les fichiers de départ (Teams), mettre en place le nouveau projet et vérifier que la construction (production d'un exécutable) se passe correctement.
2. Réaliser l'interface « Chaudiere » par un objet concret « DummyChaudiere ». Valider unitairement cette classe.
3. Réaliser l'interface « CapteurTemp » par un objet concret « DummyCapteurTemp » ou « NetworkCapteurTemp » (au choix). Valider unitairement cette classe.
4. S'occuper ensuite de la partie applicative, de manière incrémentale, en réalisant et en enrichissant au fur et à mesure l'interface « Application ». Valider étape par étape.

Travail de synthèse (après la séance, dans les 24h idéalement)

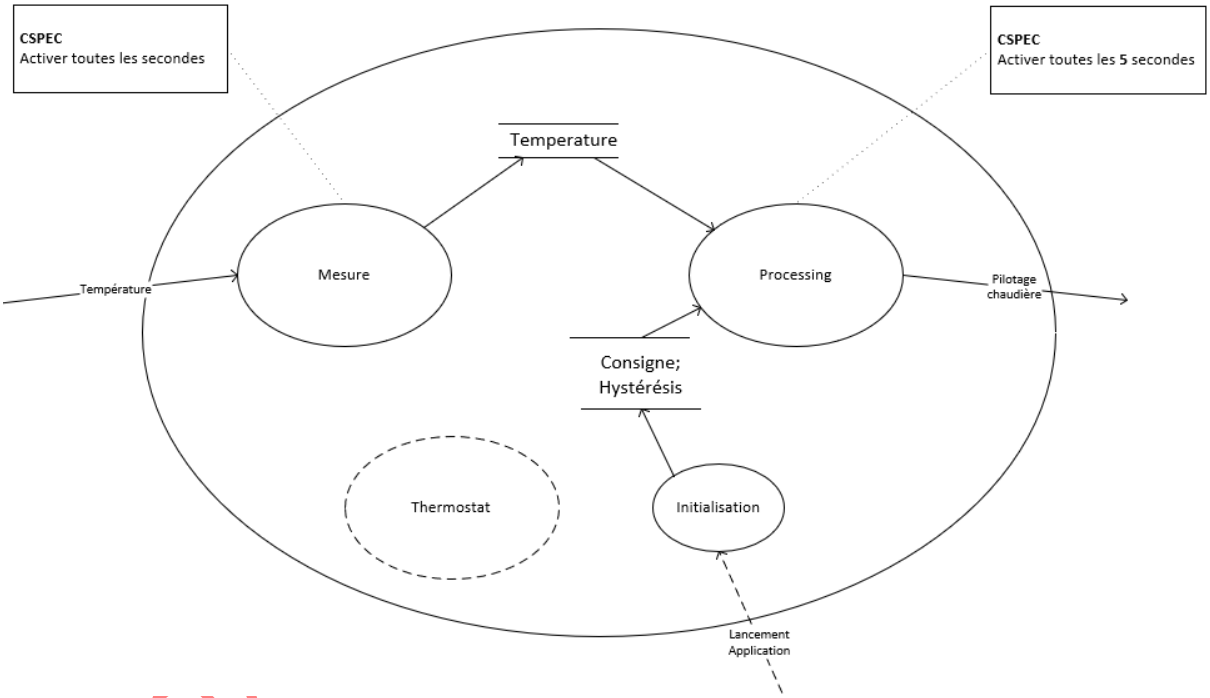
Pourquoi ne pas essayer d'implémenter le second capteur de température ?

Annexes

Analyse : DCD



Analyse : DFDo (proposition)



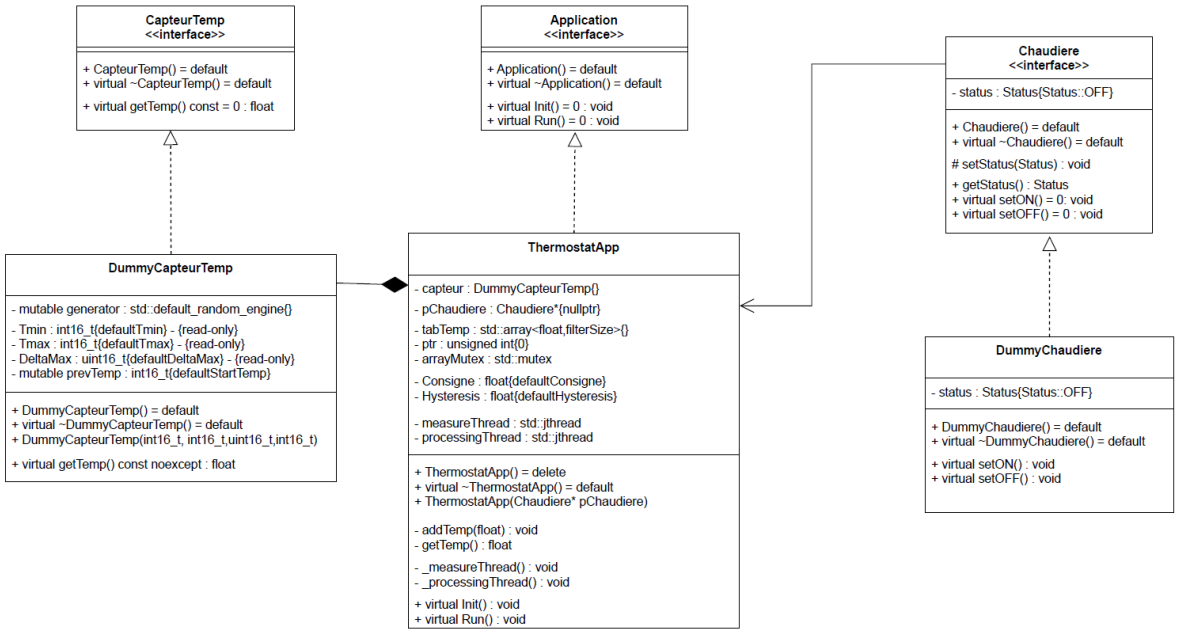
Analyse : diagrammes de classes (propositions)

Nb : Les éventuelles compositions de classes issues de la STL telles que std::jthread, std::mutex… n'apparaissent pas dans les diagrammes des classes proposés. De même d'éventuelles structures ou énumérations sous forme de classes imbriquées n'apparaissent pas non plus.

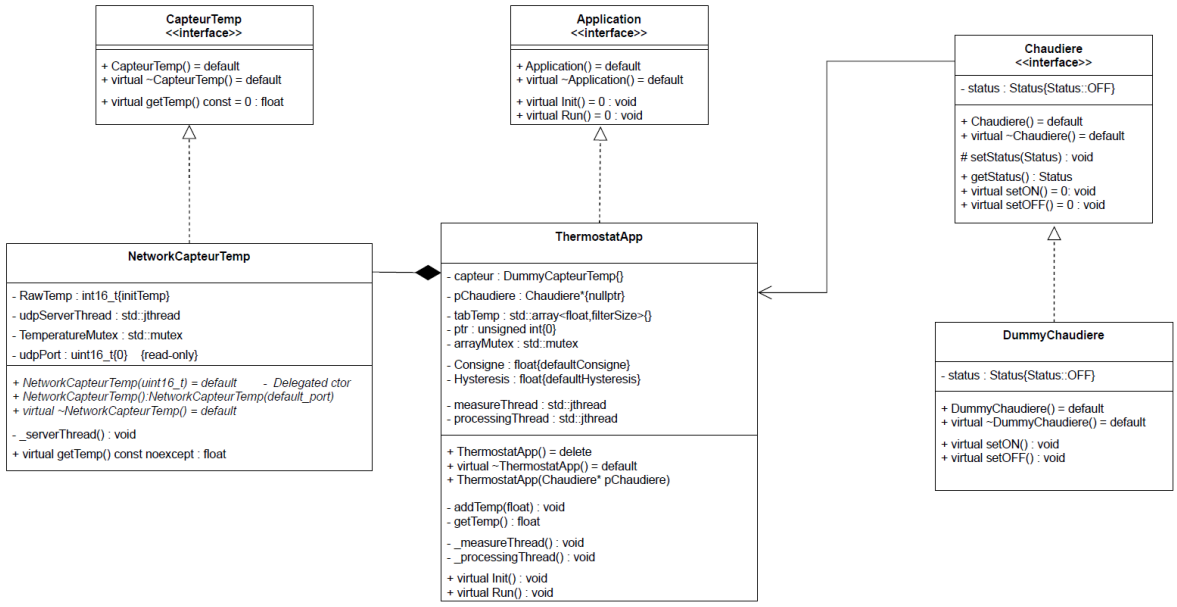
Ces diagrammes des classes sont à prendre comme:

- ✓ des pistes de réflexion
- ✓ potentiellement incomplets
- ✓ potentiellement incompatibles avec une autre façon (correcte) de voir la solution

Version DummyCapteurTemp



Version NetworkCapteurTemp



Implémentation : encapsulation de thread dans un objet

Encapsuler un thread (ou un jthread) dans un objet n'est pas compliqué en soi. Seule la syntaxe permettant de démarrer le thread est un peu pointue…

1. Déclarer le thread comme un champ (a priori) privé de la classe. Il est à noter que même lorsque l'objet englobant sera construit, cet objet « thread » ne sera pas encore réellement un thread. Il faudra pour cela le démarrer.
   - *std::jthread monObjetThread ;*
2. Déclarer et définir la fonction membre (méthode), privée elle aussi, à associer au thread
   - *void maFonctionThread() ; // dans le .hpp*
   - *void maClasse::maFonctionThread() { …. } // dans le .cpp*
3. Démarrer le thread, depuis une fonction init, un constructeur délégué, ou toute autre méthode dont ce serait le rôle :
   - *monObjetThread = std::jthread(&maClasse::maFonctionThread, this) ;*

Références :

- ✓ https://en.cppreference.com/w/cpp/thread/jthread
- ✓ https://stackoverflow.com/questions/17472827/create-thread-inside-class-with-function-from-same-class

Découverte STL : le type std::array

Les ***std::array*** sont des conteneurs de la STL implémentant des tableaux à taille fixe. Ils sont à préférer aux std::vector lorsque la taille du tableau est fixe car plus performants face à ces objets.

Référence : https://en.cppreference.com/w/cpp/container/array

Exemples :

- ✓ Création d'un tableau de 5 entiers, initialisés à 0 :
  - std::array<int,5> tab{} ;
- ✓ Avec initialisation des valeurs :
  - std::array<int,5> tab = {1,2,3,4,5} ;
  - std::array<int,5> tab {{1,2,3,4,5}} ;

Implémentation : Création de classes d'exception (dérivées de std::except)

Dans le cas où les exceptions standard ne suffisent pas, il est possible de créer ses propres objets « exception ». Toute expression valide C++ peut devenir une exception (il est par exemple possible de lever une exception de type int : throw -1 ou autre).

Il est cependant conseillé de s'appuyer sur les classes d'exception existantes.

Créer une simple classe dérivée

```
class maNouvelleException : public std::exception{} ;
```

Remarque : la nouvelle classe peut dériver d'une sous-classe de std::exception comme std::runtime_error, std::invalid_argument…

Ajouter le support du « what() »

Exemple avec comme exception de départ une runtim_error (cf remarque précédente).

```
class monExceptionVerbeuse : public std::runtime_error {
      explicit monExceptionVerbeuse(const std::string&
      what_arg):std::runtime_error(what_arg){};
      } ;
```

Levée de l'exception :

- ✓ throw monExceptionVerbeuse(''bla bla bla'') ;

Ressources : https://en.cppreference.com/w/cpp/error/exception

Découverte STL : unique_ptr

Référence : https://en.cppreference.com/w/cpp/memory/unique_ptr

Un std::unique_ptr est un « smart pointer », objet RAII permettant de gérer de manière sûre la vie d'un autre objet. Deux garanties sont notamment apportées :

- ✓ L'objet créé via un unique_ptr ne peut être pointé par d'autres pointeurs
- ✓ L'objet créé via un unique_ptr sera détruit lors de la destruction du pointeur

Exemple de création d'objet via unique_ptr :

- ➢ `std::unique_ptr<Class> pObjet = std::make_unique<Classe>(…);`

L'utilisation de ce « pointeur » se fait ensuite uniquement via les méthodes associées, notamment ***get()*** qui permet de récupérer le « vrai » pointeur sur l'objet géré.

Découverte STL : les « chrono literals »

Référence : https://en.cppreference.com/w/cpp/symbol_index/chrono_literals

Les « chrono literals » permettent de gérer les temps, durées… d'une manière simple claire. Cette simplification est apportée par l'utilisation de suffixes tels que s, ms, min…

Afin d'utiliser ces outils il faut passer par le namespace « std::chrono_literals » :

```
using namespace std::literals::chrono_literals;
```

La définition d'une variable de ce type est aisée en utilisant la déduction de type du compilateur (« type » auto) :

```
auto Delai = 1s ;   ou    auto Delai{1s} ;          // Delai vaut 1 seconde
```

Découverte STL : mutex et smart-mutex « scoped_lock »

Références :

- ✓ https://en.cppreference.com/w/cpp/thread/mutex
- ✓ https://en.cppreference.com/w/cpp/thread/scoped_lock

Un objet ***std::scoped_lock*** permet ***d'utiliser*** un ***std::mutex*** de manière sécurisée (RAII). Il ne faut donc pas confondre les deux objets :

- ✓ Un objet std::mutex doit tout d'abord avoir été instancié :
  - Par exemple : std::mutex monMutex;
- ✓ Le mutex est ensuite acquis, via un std::scoped_lock, avant d'accéder à la ressource protégée :
  - std::scoped_lock lock(monMutex);

Comme son nom l'indique, le verrouillage est limité au « scope » d'acquisition (méthode, fonction, bloc de code, etc…). La destruction automatique du scoped_lock en sortie de « scope » libère automatiquement le mutex, empêchant ainsi des interblocages.

Découverte STL : algorithme « accumulate »

Référence : https://en.cppreference.com/w/cpp/algorithm/accumulate

Cet algorithme permet de récupérer la somme d'éléments d'un conteneur entre deux itérateurs (début et fin de la zone à « sommer »).

Implémentation : serveur UDP basique (threadable)

```
void UDPserver (){

int sockfd;
sockaddr_in    servaddr, cliaddr;
char buffer[MAXLINE];

// Creating socket file descriptor
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) throw NetworkException("Error
                                                    while creating socket.");
// Filling server information
servaddr.sin_family    = AF_INET; // IPv4
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(this->getudpPort());

// Bind the socket with the server address
if ( bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0 ) throw
        NetworkException("Error while binding the socket.");

socklen_t len = sizeof(cliaddr);  //len is value/result;

int n;

while(true){
        n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, ( struct sockaddr *)
        &cliaddr, &len);
        buffer[n] = '\0';

        std::string stmp{buffer};
        std::cout << "Message received = " << stmp << std::endl;

        …

        }
    }
```

Ressources (tutos & exemples):

- ✓ https://www.geeksforgeeks.org/udp-server-client-implementation-c/
- ✓ https://bousk.developpez.com/cours/reseau-c++/UDP/01-introduction-premiers-pas/
- ✓ https://gist.github.com/sunmeat/02b60c8a3eaef3b8a0fb3c249d8686fd