Airplane Sensor System v1.4

Rapport AP4A - Julien Audoux

Introduction:

Le projet qu'il nous a été demandé de réaliser en C++, consiste en un system de capteurs de température, d'humidité, de lumière et de son. Les données doivent être récupérées par un serveur qui s'occupera de les enregistrer dans un fichier de log et ou en console. Le serveur devra être régit par un "scheduler", un objet qui permettra de gérer la fréquence d'acquisition des données capteur et ordonnera au serveur de les afficher, il s'agit du squelette du simulateur. Les données capteurs devront être des valeurs aléatoires cohérentes, pour les besoins du simulateur, en effet n'ayant pas de capteurs physique on réalisera uniquement une simulation avec des capteurs factices, même si on peut imaginer porter ce projet vers un réel system au sein d'un avion de ligne.

Le coding standard:

Tout d'abord avant la réalisation de ce projet il convient de prendre connaissance du coding standard — toutes sortes de règles d'appellation et de syntaxe — qu'il convient d'adopter pour une plus grande compréhension du code et de portabilités de celui-ci au sein d'une entreprise, il permet une compréhension accrue non négligeable.

Ma stratégie face à celui-ci, ne fut pas la bonne au premier abord, en effet le travail se faisant seul j'ai d'abord réaliser mon code sans prendre en compte ce dernier pour un plus grand confort de développement. Il est à présent évident que cela ne fut pas une bonne idée: à la fin j'ai dû réaliser un travail de "search and replace" à l'aide de mon IDE mais cela fut long et fastidieux. J'ai donc appris qu'il valait mieux prendre compte, avant même de commencer à développer, du coding standard et l'appliquer immédiatement, cela évite une correction totale du code à posteriori.

Documentation du code:

Il va de soit que pour la réalisation d'un tel projet, il convient de commenter son code efficacement. En effet cela étant déjà demandé dans le coding standard, j'ai donc commenté mon code au format doxygen. J'ai donc documenté chaque ligne de chaque fichier .h de mon projet en expliquant brièvement mais efficacement toutes les classes, méthodes et attributs de celles-ci. Cette documentation permet la compréhension globale du projet, ainsi que son fonctionnement précis.

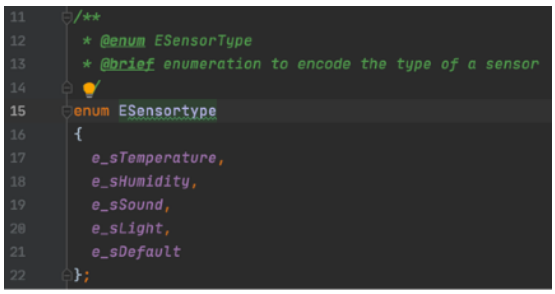
exemples:

```
1   /**
2    * @author Julien_Audoux
3    * @file Server.h
4    * @date 19/10/2020
5    * @brief definition of the Server type, to be able to gather the sensors data and showing them in logs
6    */
```

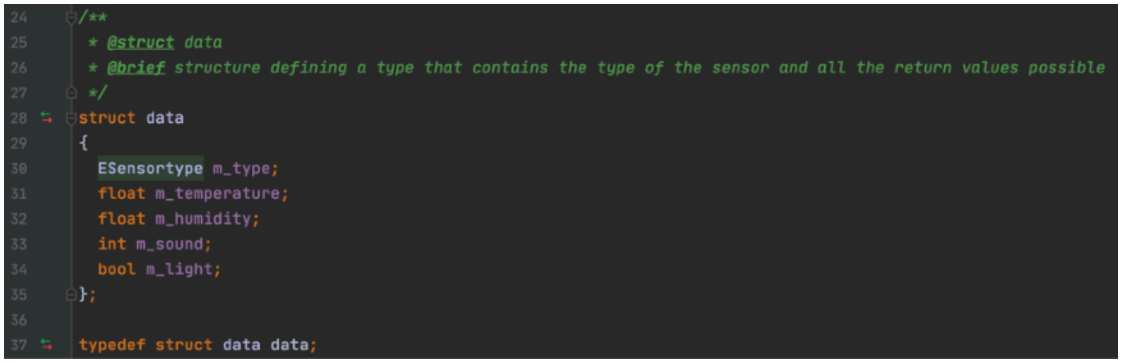
```
73    /**
74     * @brief receiving data from a sensor and paste it in the attribute dataRcved of the Server
75     * @param Sensor to receive data from
76     * @param int : the current lap of gathering data
77     * @return void
78     */
79    void dataRcv(Sensor, int);
```

Principale difficulté - la structure data et énumération ESensorType:

La principale difficulté que j'ai rapidement rencontré fut le type de renvoie de chaque capteur : int pour le son, bool pour la lumière et float pour la température et l'humidité. Tous ces capteurs dérivent de la classe Sensor et toutes ces données sont récupérées par la classe Server. Il me fallait donc une base solide me permettant d'échanger des données entre toutes ces classes peu importe le type de revoie pour éviter un grand nombre de surcharge. J'ai donc opté pour la solution de la structure, j'avais d'abord pensé à utiliser les "templates", cependant je ne suis pas arrivé à utilisé efficacement cet outils.



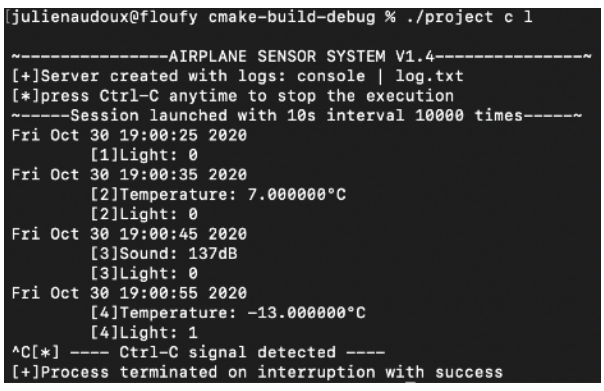
La structure me permettait de créer un nouveau type contenant chaque type primitifs de renvoie demandé. J'ai également rajouté dans la structure un élément de type ESensorType, il s'agit d'une énumération rendant compte du type de capteur en question. L'idée étant que chaque capteur ait un attribut data avec sa valeur dans le champ en concerné et le type du sensor qui sera utile pour reconnaitre quel champ est présentement utile ou non.



Les différentes classes du projets:

Le projet se compose de 7 classes différentes, pour chacune il existe un fichier .cpp et un fichier .h ou se situe la documentation. Chacune des classes est réalisée dans la forme canonique de Coplin : avec les constructeurs adéquats (par défaut, de recopie, avec arguments), un destructeur et une surcharge de l'opérateur =. Pour la réalisation de toutes ces classes je me suis inspiré de l'UML fournis dans le sujet tout en ajoutant de nouvelles méthodes ou attributs lorsque cela était nécessaire.

Fonctionnement global:



Le fonctionnement global du projet ce place dans un contexte très précis qui mérite précisions. En effet il s'agit d'un system de capteurs pour avion de ligne. On peut donc facilement imaginer que ce programme — en condition réelles — ne sera jamais exécuté ou arrêté par une personne physique.

Exécution:

C'est pourquoi le choix des logs (console ou fichier log) se fait en paramètres d'exécution, car on peut imaginer que le programme soit exécuté

par un autre processus de l'avion comme un programme gérant un tableau de bord et que les choix de logs viennent de boutons physiques ou autre par exemple. Ainsi la commande pour lancer le projet avec log et console ressemblera à : "./project c l". Ainsi un "c" permet d'activer les logs consoles et "l" pour activer les logs fichiers, cela fonctionne dans n'importe quel ordre.

Relevés de capteurs:

Par défaut le temps maximum d'exécution sans interruption est de 10 secondes entre chaque relevé et 10 000 relevés, ce qui correspond à environ 27.7 heures soit largement assez pour un vol en avion. Cela est inscrit dans des "#define" et est donc modifiable. A chaque relevé la date précise est affichée dans les logs. Tous les capteurs ne sont pas relevés en même temps, en effet certains apparaissent à tous les tours, certains tous les 2, 3 ou 5 tours.

Interruption:

De la même manière pour l'interruption du programme, j'ai opté pour un signal d'interruption: le processus parent pourra terminer le system à n'importe quel moment en envoyant simplement un signal d'interruption "SIGINT" qui sera reconnu par ce dernier et affichera un message d'interruption tout en finissant correctement avec un code d'exit 0. Cela pourra également être réalisé par une personne physique avec un simple Ctrl-C.

Le découpage par classe:

La classe "Sensor":

Cette classe définit tous les attributs et méthodes nécessaires à chaque capteurs. On y retrouve l'attribut data et la méthode getData() qui lance une génération aléatoire de la valeur du capteur à l'aide de la méthode aleaGenVal(), puis getData() renvoie l'attribut data du capteur en question avec une valeur aléatoire pour le bien de la simulation.

Les classes "Temperature", "Humidity", "Light" et "Sound":

Ces quatre classes correspondent aux différents type de capteurs, elles dérivent toutes de la classe "Sensor". Les seuls méthodes surchargées dans ces classes sont celles de la forme canonique et en particulier les constructeurs, qui écrivent dans leur attribut data en renseignant les champs correspondant notamment au type de capteurs.

La classe "Server":

Cette classe gère la récolte de donnée des capteurs à l'aide de la méthode dataRcv(Sensor) qu'elle stock dans son attribut dataRcved de type std::string pour ensuite être affiché à tout moment à l'aide de sa méthode display(). Cette dernières va également utiliser deux attributs booléens de la classe Server, qui ne sont autres que l'activation des logs fichier et des logs consoles.

La classe “Scheduler”:

La classe Scheduler possède trois attributs, un objet Server dédié, un int pour l’interval entre deux relevés et un int pour le nombre total de relevés. En dehors de sa forme canonique elle possède une seul méthode représentative de son fonctionnement. La méthode :

```
void Scheduler::launchScheduler(std::vector<Sensor> tab_p)
```

Cette méthode prend en paramètre un tableau dynamique d’objet Sensor peu importe leur type, on peut donc en mettre autant que l’on souhaite, même plusieurs objets Sensor de même type. Ainsi à l’aide d’un boucle for() va relever les données des capteurs avec l’attribut Server selon l’interval (toutes les 10 secondes par défaut) jusqu'au nombre total d'exécutions (10 000 par défaut). Cependant tous les capteurs ne sont pas relevés en même temps : certains sont relevés tous les temps, tous les 2, 3 ou 5 temps. Une fois une données relevés dans le Server cette méthode va également ordonner au Server de les afficher en appelant sa méthode display() qui se gère d’orienter les logs vers la console ou dans le fichier de log suivant les choix précédents dans l’exécution.

Améliorations possibles:

Ce projet n’est, bien sûr, pas parfait et non applicable à un environnement réel en l’état, car il s’agit d’un exercice dans le cadre d’une UV. Cependant on peut envisager des améliorations possibles pour le rendre utilisable dans un avion de ligne bien réel.

Par exemple on peut imaginer des arguments d'exécution plus poussés en prenant en compte l’interval entre deux mesures, et le nombres total de mesures ainsi que le chemin pour le fichier de log ou une console particulière liée à l’avion par exemple. Cela rendrait le programme plus flexible et adaptable à son environnement.

On peut également imaginer un système de signaux plus complexe pour correctement mettre le processus en pause, le relancer, ou désactiver/activer les logs console ou fichier pendant une execution.

Bien évidemment si on porte ce programme en environnement réel il faudrait remplacer l’appel de aleaGenVal() (la méthode générant les valeurs aléatoires dans la classe Sensor) par un appel de méthode, de Classe ou de processus externe relevant des valeurs de capteurs réels.

Matériel utilisé:

- Clion de JetBrains : IDE C/C++ utilisé pour la réalisation de ce projet

- Multiples recherches internet telles que StackOverflow, OpenClassroom, etc…

- Terminal