

TP Chaîne d'acquisition – 4 – IR

Programmation Orientée Objet C++

Objectif :

Dans le TP précédent, un nano-ordinateur sous Raspberry OS (Linux) doit recevoir des informations issues d'un système externe (Arduino) via sa liaison série, traiter ses informations pour les afficher en HTML ou les enregistrer dans une base de données.

Le code créé ne respect pas complètement la philosophie OBJET. Ce TP va nous aider à nous améliorer en Programmation Orientée Objet (POO)

On vous fournit :

- Une platine Arduino avec un logiciel embarqué qui génère des trames GPS, ou qui est capable de recevoir des commandes/
- Un nano-ordinateur Raspberry et sa carte SD préchargée.
- Un câble de liaison USB

Pré-requis : Le TP Chaîne d'acquisition 1

Sommaire

1	Analyse (A ETUDIER)	2
2	Développer et tester une classe	3
2.1	Finaliser la classe « cl_nmea »	3
2.2	Module de test	5
2.3	Compilation : MAKE + MAKEFILE	5
3	Réutilisez vos connaissances !!	6
3.1	La classe « cl_gpgga » : Héritage	6
3.2	La classe « cl_gpgga » : méthode « isGPGGA »	7
3.3	Classe « cl_gpsOnSerial »	8
3.3.1	Utilisation d'une classe extérieure	8
3.3.2	Le constructeur	8
3.3.3	La lecture des caractères sur le port série	8
3.3.4	Module de test	9
3.3.5	TRAVAIL	9
4	Test final	9
4.1	TRAVAIL :	9
4.2	Comportement de la classe sRs232	9

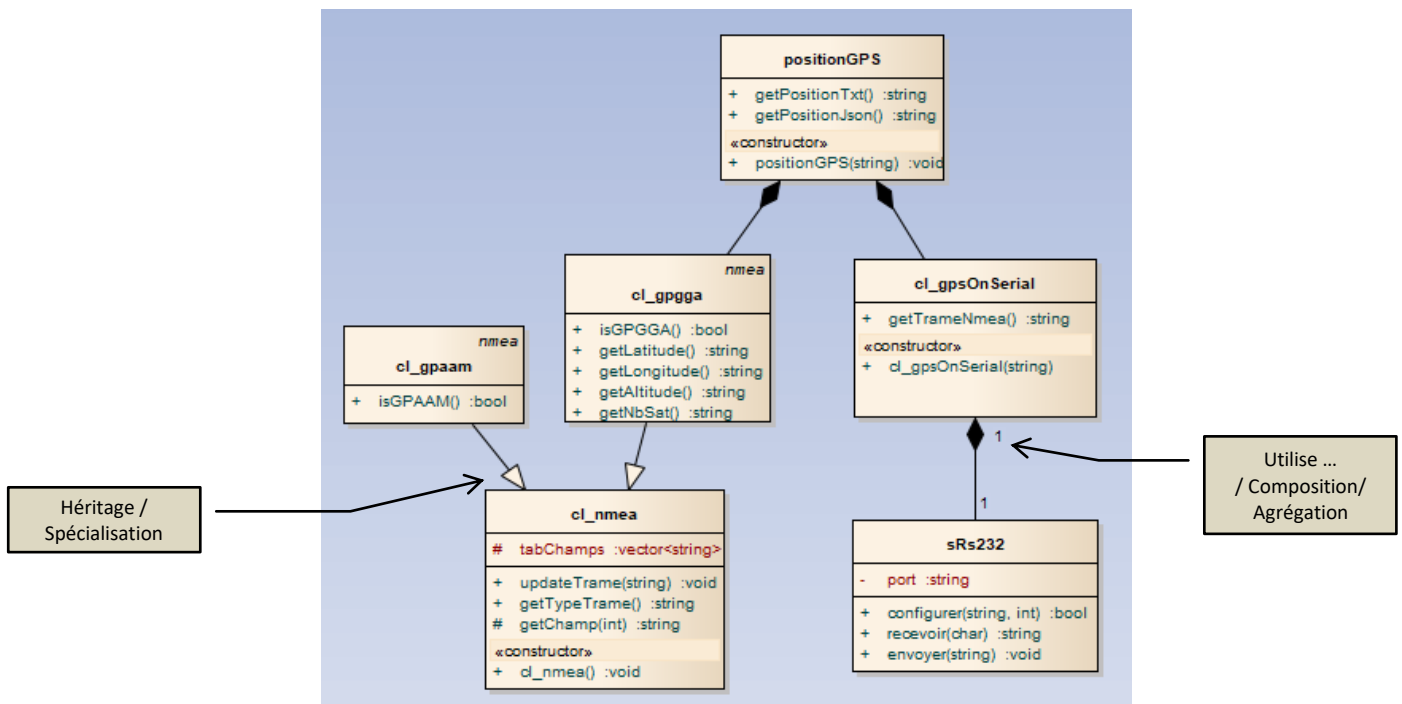
1 ANALYSE (A ETUDIER)

Nous avons créé un code utilisant une ou plusieurs **fonctions**, au fur et à mesure des besoins.

Une réflexion davantage orientée « objet » pourrait nous amener à ce genre de décision :

1. Un GPS pourrait être un objet logiciel qui utilise un objet liaison série.
Cet objet fournirait les trames NMEA présentes sur la liaison série.
2. Une trame NMEA pourrait être un objet logiciel.
On lui confie une chaîne de caractères et il l'analyse.
Si c'est une trame NMEA, il pourrait nous fournir le type de trame (\$GPGGA, \$GPAAM, ...)
3. Une trame \$GPGGA qui est une trame particulière NMEA pourrait être une spécialisation de la classe NMEA.

Dans ce cas on aurait le diagramme de classes suivant :



Classe ou Objet ? Une « classe » est un modèle. L'« objet » est une « matérialisation » de la classe. On utilise l'expression « instance de classe ».

Pour obtenir un « objet logiciel » il faut obligatoirement créer d'abord la classe correspondante.

Comparaison : Le plan d'une maison correspond à une « classe ».

La maison construite selon ce plan correspond à l'« objet »

La maison est réelle, le plan est juste l'idée de la maison

Le même plan peut servir à construire plusieurs maisons identiques.

Une fois construites, les maisons ont leur propre vie indépendante.

La classe **sRs232** existe déjà, nous l'avons déjà utilisée.

La classe **cl_gpsOnSerial** représente un GPS qui reçoit ses trames (repérées par des délimiteurs) par une liaison série.

La classe **cl_nmea** représente une trame nmea quelconque. Une trame commence par \$ et est séparée par des virgules. Le premier champ indique le type de trame.

La classe **cl_gpgga** est une spécialisation de la classe `cl_nmea`. Elle permet d'obtenir les informations précises de la trame nmea (latitude, longitude, ...)

La classe **positionGPS** est la classe principale de l'application. Grâce aux classes `cl_gpgga` et `gpsOnSerial`, elle est capable de fournir la position GPS. Elle ne sera pas développée. On la remplacera par un module de test.

La classe **cl_gpaam** ne sera pas développée. Elle représente une trame GPAAM

L'ensemble va générer plus de code que dans la version non-objet.
Mais l'avantage de la solution est sa modularité et ses possibilités d'évolution.

Ce diagramme de classe n'est pas la seule façon de représenter notre besoin. Chaque programmeur aura certainement sa propre analyse ...

TRAVAIL :

A partir du logiciel **Enterprise Architect**, on peut créer un diagramme de classe enrichi de commentaires, et ensuite générer le squelette de l'application.

Le fichier EA est fourni, vous pouvez générer les fichiers qui serviront à réaliser l'application :

Menu : *Tools* → *Source Code Engineering* → *Generate Package Source Code*

Cochez la case « *Auto Generate files* » pour choisir le dossier où les fichiers seront générés.

Ensuite, transférez ces fichiers sur le Raspberry et travaillez dessus en mode ssh/sftp avec BitVise.

2 DEVELOPPER ET TESTER UNE CLASSE

2.1 Finaliser la classe « cl_nmea »

Dans les propriétés de la classe (fichier « cl_nmea.h »), vous notez mon choix d'utiliser une variable de type VECTOR :

```
protected:
    /**
     * La trame complète
     */
    std::vector<std::string> tabChamps;
```

Un VECTOR correspond à un tableau de valeurs d'un même type.

Il s'agit d'un objet logiciel du c++, donc il intègre un ensemble d'outils (méthodes) pour manipuler les valeurs.

Ici, c'est un vecteur d'objets « string ». Or chaque « string » est aussi un objet du C++.

On a donc un vecteur d'objets « string ».

Le constructeur initialise le vecteur avec une chaîne bidon pour que la méthode « isGPGGA » de la classe fille soit en mesure de répondre même si la trame n'a pas encore été mise à jour :

```
cl_nmea::cl_nmea() {
    tabChamps.push_back("vide");
}
```

Notre méthode **updateTrame** sert à intégrer une trame venant du programme principal. On remplit le vecteur avec les différentes zones de la trame fournie en argument, en prenant la virgule comme séparateur (on reproduit ici la fonction **explode** du PHP) :

```

/**
 * Mettre à jour la trame dans l'objet
 */
void cl_nmea::updateTrame(std::string trame) {
    string champ;
    int tag = 1;
    tabChamps.erase(tabChamps.begin(), tabChamps.end());

    while(tag != trame.npos) {
        tag = trame.find(",");
        champ = trame.substr(0, tag);
        trame = trame.substr(tag + 1);
        tabChamps.push_back(champ);
    }
}

```

Notre méthode **getTypeTrame** retourne la première *string* du vecteur, qui correspond au type de trame NMEA :

```

/**
 * Obtenir le premier champ de la trame
 */
std::string cl_nmea::getTypeTrame () {
    return tabChamps[0];
}

```

TRAVAIL :

Saisir les codes des 3 méthodes dans le fichier « cl_nmea.cpp » et réfléchir à leur fonctionnement en utilisant les modes d'emploi de **npos**, **find**, **substr** des objets *string*, et **erase**, **begin**, **end**, **push_back** des objets *vector*. (site internet c++ : <https://www.cplusplus.com/>).

ATTENTION !!!

Ce que n'a pas fait « Entreprise Architect » : il manque :

```

#include <iostream>
using namespace std ;

```

A ajouter dans TOUS les fichiers.h générés par E.A.

2.2 Module de test

Une classe logicielle ne possède pas de « main ». Donc il faut un programme de test spécifique pour vérifier son fonctionnement.

Voici une proposition :

```
// Module de test classe cl_nmea
// Fichier testNmea.cpp

#include <iostream>
#include "cl_nmea.h"
using namespace std;

int main()
{
    // Objet de la classe cl_nmea:
    cl_nmea c;
    // Trame bidon pour le test :
    string trame = "$GPGGA,DATE,43.5,N,5.5,E,1,2,3,100,M,5,7,8,9,10";
    // On transmet la trame à l'objet :
    c.updateTrame(trame);
    // Test d'affichage :
    cout << "Type de trame : " << c.getTypeTrame() << endl;
}
```

TRAVAIL : Créez le fichier `testNmea.cpp` avec le code ci-dessus.

2.3 Compilation : MAKE + MAKEFILE

Sur un projet à plusieurs fichiers, on fait des compilations séparées de fichier, pour ensuite tout réunir dans un exécutable.

Principe :

- Compilation d'une classe seule : `g++ -c nom_classe.cpp`
Si cela fonctionne, un fichier `nom_classe.o` est généré.
- Idem pour le programme principal (C'est le programme avec la fonction MAIN mais qui fait `in #include` de la classe externe)
`g++ -c principal.cpp`
Si cela fonctionne, un fichier `principal.o` est généré.
- Ensuite, on produit l'exécutable qui s'appellera **principal**:
`g++ nom_classe.o principal.o -o principal`
☺ Ne confondez pas les « .o » et les « -o » !!!!!

AU SECOURS ??

Les outils comme Qt ou Visual Studio s'occupent de ce travail ... mais sur notre Raspberry, on va faire autrement !!! On va utiliser l'outil **make**.

TRAVAIL :

1. Vérifiez que la commande **make** est installée sur votre Raspberry.
2. On va expliquer à « make » ce qu'il doit faire. Pour cela, créez un fichier nommé « **makefile** » dans le dossier où se trouvent vos fichiers. Et entrez les instructions suivantes en suivant scrupuleusement les indentations :

```
testNmea: cl_nmea.o testNmea.o
        g++ cl_nmea.o testNmea.o -o testNmea

testNmea.o : testNmea.cpp
        g++ -c testNmea.cpp

cl_nmea.o: cl_nmea.h cl_nmea.cpp
        g++ -c cl_nmea.cpp
```

Explications :

- testNmea: La cible finale. En invite de commande il faudra taper : `make testNmea`
- testNmea: cl_nmea.o testNmea.o On précise les 2 fichiers « .o » nécessaires à la compilation.
- *Make* va chercher les cibles intermédiaires « testNmea.o: » et « cl_nmea.o: ».
- Si les fichiers mentionnés après les cibles ont été modifiés, il fait les compilations intermédiaires et fait ensuite la compilation de la cible finale.
- Grâce à **make**, on ne compile que ce qui est nécessaire : gain de temps, surtout sur un Raspberry qui compile plus lentement qu'un PC survitaminé.
- NB : Dans un **makefile**, on peut créer plusieurs cibles finales, par exemple pour chaque module de test que nous allons réaliser.

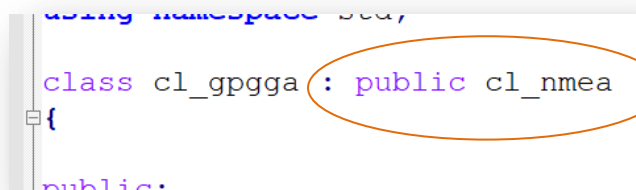
Test :

1. Si vous tapez `make testNmea` vous devriez voir un exécutable **testNmea**.
2. Pour le tester, tapez : `./testNmea`
3. Trouver le code pour la méthode **getChamp** de la classe « cl_nmea.cpp »
4. Complétez le programme `testNmea.cpp` pour tester la méthode **getChamp** que vous avez écrite.
5. FAITES VALIDER VOTRE TRAVAIL

3 REUTILISEZ VOS CONNAISSANCES !!

3.1 La classe « cl_gpgga » : Héritage

On remarque dans le fichier « cl_gpgga.h » comment on déclare qu'une classe hérite d'une autre :



```
using namespace std;

class cl_gpgga : public cl_nmea
{
public:
```

La classe « cl_gpgga » hérite de « cl_nmea »
 Cela signifie que « cl_gpgga » peut utiliser directement les méthodes et les propriétés (publiques et protected) de la classe « cl_nmea »

3.2 La classe « cl_gpgga » : méthode « isGPGGA »

Voici le code de la méthode :

```

/**
 * Retourne VRAI si la trame est $GPGGA
 */
bool cl_gpgga::isGPGGA() {
    if ( getTypeTrame() == "$GPGGA" )
        return true;
    return false;
}

```

TRAVAIL :

1. Tapez le code de la méthode « isGPGGA » dans le fichier « cl_gpgga.cpp »
2. Créez le programme de test nommé « testGGA.cpp » :

```

#include <iostream>
#include "cl_gpgga.h"
using namespace std;

int main()
{
    cl_gpgga c;
    string trame = "$GPGGA,DATE,43.5,N,5.5,E,1,2,3,100,M,5,7,8,9,545454";
    c.updateTrame(trame);
    cout << c.getTypeTrame() << endl;
    if ( c.isGPGGA() ) {
        cout << "Lat:" << c.getLatitude() << endl;
        cout << "Lon:" << c.getLongitude() << endl;
        cout << "Alt:" << c.getAltitude() << endl;
    }
}

```

3. Ajoutez au **makefile** une nouvelle cible **testGPGGA**
4. Compilez et testez !
5. Complétez « cl_gpgga.cpp » pour obtenir l'affichage des latitudes, longitude, et altitude. Vous devriez avoir le résultat suivant :

```

pi@pi3os:~/TPChaine $ ./testGGA
$GPGGA
Lat:43.5N
Lon:5.5E
Alt:100M
pi@pi3os:~/TPChaine $

```

3.3 Classe « cl_gpsOnSerial »

Cette classe est assez simple vu que la classe sRs232 fait tout le travail !

Il suffit de demander une lecture du port série jusqu'à réception de retour à la ligne (« \n »), et de vérifier que le premier caractère de la chaîne est bien un « \$ »

3.3.1 Utilisation d'une classe extérieure

Le fichier « cl_gpsOnSerial.h » nous renseigne sur la structure de la classe :

On y remarque le lien « utilise » sous la forme d'un pointeur (voir diagramme de classes)

```
class cl_gpsOnSerial
{
public:
    cl_gpsOnSerial();           // Constructeur par défaut (non-utilisé)
    cl_gpsOnSerial(string port); // Constructeur utilisé
    virtual ~cl_gpsOnSerial(); // Destructeur (non utilisé)
    sRs232 *m_sRs232;          // Pointeur vers un objet de classe sRs232

    string getTrameNmea();
};
```

3.3.2 Le constructeur

On y effectue l'ouverture et la configuration du port série. Le nom du port série est transmis en argument.

```
/**
 * Argument : le nom du port serie
 */
cl_gpsOnSerial::cl_gpsOnSerial(string port)
{
    m_sRs232 = new sRs232(port);
    m_sRs232->Configurer(BPS4800, BIT8, PAS_DE_PARITE);
}
```

3.3.3 La lecture des caractères sur le port série

Comme annoncé, la classe sRs232 fait tout ... Donc il reste juste à contrôler que le premier caractère de la chaîne est un « \$ » :

```
/**
 * Retourne une trame complète NMEA (du $ au retour à la ligne)
 */
string cl_gpsOnSerial::getTrameNmea() {
    string chaineRecue = "rien";
    while ( chaineRecue[0] != '$' ) {
        chaineRecue = m_sRs232->Recevoir('\n', SANS_FINAL);
    }
    return chaineRecue;
}
```


3.3.4 Module de test

Programme très simple : juste l'affichage d'1 trame :

```
#include <iostream>
#include "cl_gpsOnSerial.h"
using namespace std;

int main()
{
    cl_gpsOnSerial com("/dev/ttyUSB0");
    cout << com.getTrameNmea() << endl;
}
```

3.3.5 TRAVAIL

1. Intégrez les codes ci-dessus aux fichiers « cl_gpsOnSerial.cpp et au programme de test « testGpsOnSerial ».
2. Complétez le « *makefile* » pour y intégrer la nouvelle cible **testGpsOnSerial**.
3. Testez ! Si tout est bien branché, une trame apparaîtra.

4 TEST FINAL

Nous allons sauter l'étape de la création de la classe « **positionGPS** » du diagramme. Elle est prévue pour s'intégrer dans un logiciel complet.

Dans notre cas, nous allons valider l'ensemble de nos classes en un module de test.

4.1 TRAVAIL :

Réalisez le programme « testPositionGPS » qui affiche la latitude, la longitude, et l'altitude des trames venant du simulateur GPS.

4.2 Comportement de la classe sRs232

La position GPS renvoyée par le simulateur est toujours la même.

Cela vient de la configuration du port série : Le signal RTS-CTS provoque le rechargement du programme Arduino à chaque ouverture de port. Il faut bloquer le RTS-CTS à la fermeture du port.

Fichier à modifier : sRs232.cpp, méthode Configurer, vers la ligne 26 :

```
// Ajout SeA //////////////////////////////////////
Config.c_cflag |= (CLOCAL | CREAD); // Enable the receiver and set local mode
Config.c_cflag &= ~CSTOPB; // 1 stop bit
Config.c_cflag |= CRTSCTS; // SeA : enable hardware flow control (Arduino)
Config.c_cflag &= ~HUPCL; // SeA : disable HUPCL pour que l'arduino ne reboote pas
Config.c_lflag &= ~ICANON; // Enlever le mode canonique
// Mode canonique activé : on attend le RETURN pour sortir du read
// Mode canonique désactivé : le read fonctionne normalement
```