# Construction d’objet

*Durée : Deux séances de 2h minimum*

Tous les programmes sont à réaliser sous QtCreator. Vous créerez un projet de type « Non Qt project/Plain C++ Application »

Vous pouvez créer automatiquement le squelette du code (classe sample dans sample.h et code des méthodes dans sample.cpp) : il suffit de cliquer avec le bouton droit de la souris sur le nom du projet précédemment créé (en gras sur la gauche) ; faire « ajouter nouveau » (ou Add new) et choisir « C++ class »

## La classe Sample

On désire créer une classe permettant de sauvegarder des échantillons (sample) et effectuer des traitements dessus. A terme, on aimerait pouvoir avoir ceci :

V

Sample

4.95

4.95

4.8

3.7

3.2

3.1

2.9

2.3

2.4

m\_samples

4.8

3.7

3.2

Moyenne() : float

Min() : float

Max() : float

3.1

2.9

2.4

2.3

t

Un objet de type « Sample » possèdera un tableau de réels conservant les niveaux de tension d’un signal V(t) analogique. On ne se soucie pas, pour le moment, de la manière d’acquérir ces valeurs.

Cet objet sera capable de renvoyer la valeur moyenne des échantillons, la valeur min et max des échantillons.

### 1.1 Création de la classe

Définition de la classe (sample.h)

class Sample

{

float \*m\_samples;

int nbsamples;

public:

Sample();

};

Implémentations des méthodes (sample.cpp)

Sample::**Sample**(): m\_samples(new float[10]),nbsamples(10)

{

}

Rappels :

* La méthode Sample() est le constructeur par défaut.
* L’écriture « m\_samples(new float[10]),nbsamples(10) » est une liste d’initialisation.

***☞ Quel est le résultat de l’exécution de cette ligne de code ?***

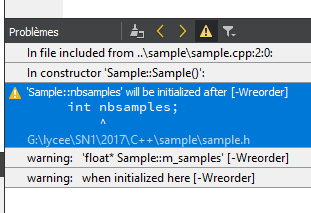
* Modifier votre code afin d’inverser la liste d’initialisation :

Sample::**Sample**(): nbsamples(10), m\_samples(new float[10])

{

}

En recompilant, vous obtenez un « Warning » :



***☞ Expliquer pourquoi a-t-on ce warning bien que l’exécutable soit quand même produit ?***

***☞ Que peut-on en conclure sur l’ordre des attributs dans la liste d’initialisation ?***

### 1.2 Création d’objets

Dans la fonction *main*, créer un objet de la classe « Sample » :

int **main**()

{

cout << "Echantillonage" << endl;

Sample sample1;

return 0;

}

On aimerait bien, tout de même, que l’objet créé possède les valeurs des échantillons du signal. Pour se faire, on part du principe qu’un autre module logiciel aura effectué l’échantillonage et aura sauvegardé les valeurs dans un fichier. Pour ne pas surcharger le travail, on considère que les valeurs ont été enregistrées dans un fichier texte : « data.txt »

Mettre ce fichier dans le répertoire « debug » de votre application. Chez moi, il s’appelle : *build-sample-Desktop\_Qt\_5\_9\_1\_MinGW\_32bit-Debug*

Ajouter la méthode « enregistrerSample() » à votre classe :

void Sample::**enregistrerSample**()

{

int i = 0;

string realch;

//ouverture du fichier

ifstream fileSample ("data.txt");

//Lecture des valeurs

while(getline(fileSample, realch))

{

//On enregistre les valeurs dans le tableau

m\_samples[i++] = atof(realch.c\_str());

}

fileSample.close();

}

Cette méthode ouvre et lit le contenu du fichier texte « data.txt ». Chaque ligne lue est une valeur réelle à enregistrer dans le tableau « m\_samples »

Le risque est que le tableau « m\_samples » n’ait pas été créé correctement ou que l’espace réservé ne soit pas suffisant.

***☞ Modifier la fonction précédente afin de s’assurer que le tableau « m\_samples » ait bien l’espace nécessaire réservé.***

Appeler cette méthode dans la fonction *main()* :

sample1.enregistrerSample();

### 1.3 Destructeur

Logiquement, à partir du moment où on crée un tableau lors de la création d’un objet, il est indispensable de « dé-réserver » le tableau « m\_samples ». Pour cela, il suffit d’effectuer un «delete » dans le destructeur de la classe :

***☞ Modifier et donner le code du destructeur***

Ajouter le destructeur à la classe.

### 1.4 Constructeur par recopie

On apprécie particulièrement de pouvoir créer un objet à partir d’un autre. Pour cela, on doit ajouter le constructeur par recopie. Voici son prototype, à ajouter dans la définition de la classe :

Sample(const Sample&);

Son implémentation :

Sample::**Sample**(const Sample &dessamples): m\_samples(dessamples.m\_samples),nbsamples(dessamples.nbsamples)

{

}

Rappels :

* Le constructeur par recopie est avant tout un constructeur
* Il prend en paramètre une référence (alias) sur un objet du même type
* C’est un des rares cas où on peut accéder à un contenu privé : rendu possible parce qu’on a précisé que la référence passée est une référence constante donc on s’interdit de pouvoir modifier l’objet passé et de plus c’est forcément un objet du même type

### 1.5 Méthode d’affichage

A des fins de test, on réalise une méthode d’affichage. On pourra ainsi demander à l’objet d’afficher les valeurs de son tableau :

void Sample::**afficherSample**()

{

for (int i = 0; i<nbsamples ; i++)

{

cout << m\_samples[i] << endl;

}

}

Ajouter cette méthode à la classe.

## Test de la classe

### 2.1 Un objet « sample1 »

Modifier la fonction *main()* de la sorte :

int **main**()

{

cout << "Echantillonage" << endl;

Sample sample1;

sample1.enregistrerSample();

sample1.afficherSample();

return 0;

}

### Un second objet « sample2 »

int **main**()

{

cout << "Echantillonage" << endl;

Sample sample1;

sample1.enregistrerSample();

sample1.afficherSample();

Sample sample2(sample1);

sample2.enregistrerSample();

sample2.afficherSample();

return 0;

}

On crée « sample2 » à partir de « sample1 »

Vous pouvez remarquer que l’affichage est exactement le même. Vous vous dîtes sans doute que c’est normal puisqu’on lit le même fichier avec les mêmes valeurs. Vous avez sans doute raison mais il y a une autre possibilité : peut-être est-ce le même tableau ?

### Y a-t-il un second tableau ?

Modifier la fonction *main()* de la sorte :

int **main**()

{

cout << "Echantillonage" << endl;

Sample sample1;

sample1.enregistrerSample();

sample1.afficherSample();

{

Sample sample2(sample1);

}

sample1.enregistrerSample();

return 0;

}

Normalement, votre programme devrait planter !

Il affiche les valeurs puis plante.

***☞ Expliquer pourquoi le programme plante juste après l’affichage du tableau de « sample1 »***

Le problème vient du constructeur par recopie. Il faut, en fait, créer un tableau et lui affecter les valeurs du tableau de l’objet passé en référence :

Sample::**Sample**(const Sample &dessamples): nbsamples(dessamples.nbsamples)

{

m\_samples = new float[nbsamples];

for (int i = 0 ; i < nbsamples; i++)

{

m\_samples[i] = dessamples.m\_samples[i];

}

}

On recopie élément par élément du tableau de l’objet « dessamples » vers le tableau « m\_samples » de l’objet qu’on est en train de créer.

Remarque : on pourrait faire toutes ces lignes de code dans la liste d’initialisation mais c’est un peu la limite de l’exercice. Ça peut devenir très vite illisible alors autant le mettre dans la zone de code.

### Méthode surchargée

On aimerait bien pouvoir lire le contenu d’un fichier dont on passe le nom en paramètre. La grande force de la programmation objet est de pouvoir créer des méthodes portant le même nom : on appelle cela la surcharge (overloading)

Modifier votre classe en conséquence :

class Sample

{

float \*m\_samples;

int nbsamples;

public:

void **enregistrerSample**();

void **enregistrerSample**(string);

void **afficherSample**();

Sample();

Sample(const Sample&);

~Sample();

};

Réaliser l’implémentation de la méthode void **enregistrerSample**(string);

Elle sera très proche de la méthode sans paramètre.

### Test de la méthode

Créer un second fichier texte contenant des valeurs aléatoires et modifier la fonction *main()* de façon à ce que l’objet « sample2 » possède les valeurs contenues dans ce second fichier.

***☞ Procéder à l’affichage.***

### Méthodes *Moyenne(), Min(), Max()*

***☞ Proposer une implémentation de ces trois méthodes.***