Référence AP	LOO_CPP_AP2
Thématique principale	Découverte avancée C++
Thématique(s) secondaire(s)	Le type std::vector Ecriture dans un fichier texte Méthodes utilisables par objets constants et attributs mutables Levée d'exceptions de type std::
Durée encadrée	2h
Prérequis[1]	✓ Structurer un projet C++, le compiler, générer un exécutable avec l'environnement de développement choisi, sur cible locale ✓ Déclarer et définir une classe ✓ Instancier un objet
Compétences opérationnelles et niveau cible[2]	✓ Mettre en place les accesseurs pour les attributs le nécessitant. Qu'il s'agisse d'accesseurs standard, ou avec contraintes (unidirectionnalité, validation de valeur…) (***M***) ✓ Mettre en œuvre des objets de type *vector* (***A***) ✓ Manipuler les fichiers (***N***) ✓ Lever des exceptions std (***M***) ✓ Mettre en place une politique de création d'objet conforme à la règle du zéro (***N***).
Modalités et critères d'évaluation	Evaluation en cours de TP : avancement et qualité finale du code (incluant les commentaires).
Matériel(s), équipement(s), composant(s) nécessaires[3]	
Logiciel(s) nécessaire(s)[4]	gnuplot (http://www.gnuplot.info/)
Ressources[5]	https://www.cplusplus.com/ https://www.cplusplus.com/reference/vector/vector/ https://cplusplus.com/reference/exception/exception/ https://www.cplusplus.com/reference/fstream/ofstream/ https://en.cppreference.com/w/cpp/filesystem https://en.cppreference.com/w/cpp/string/basic_string_view https://cplusplus.com/reference/string/string/

Travail préparatoire (la veille de la séance idéalement)

Il est nécessaire, en amont de la séance, de bien maîtriser les prérequis. La réalisation de l'AP1 suffit à entièrement couvrir ceux-ci.

Un rapide retour sur la notion de constructeur délégué peut être utile. De même, la notion d'accesseurs (getter/setter) doit être connue (sinon maîtrisée).

Enfin, la réalisation de cette AP nécessite la mise en œuvre d'un logiciel supplémentaire : ***gnuplot***.

Si l'environnement de développement est un OS GNU/Linux basé sur la distribution Debian (Ubuntu notamment), l'installation est très simple : *sudo apt install gnuplot* .

Pour toute autre situation, se référer aux instructions d'installation disponibles sur le site de ce logiciel.

Remarque : aucune maîtrise de gnuplot n'est nécessaire. Son utilisation se limitera en l'invocation simple d'une unique fonction (plot).

[1] Il convient, en amont de la séance, que ce soit ou non explicitement demandé dans le travail préparatoire, de s'assurer que les prérequis sont bien remplis.

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

[3] Matériel(s) nécessaire(s) ***en plus*** d'un ordinateur utilisable pour réaliser du développement C++ (carte MCU, modules matériels…)

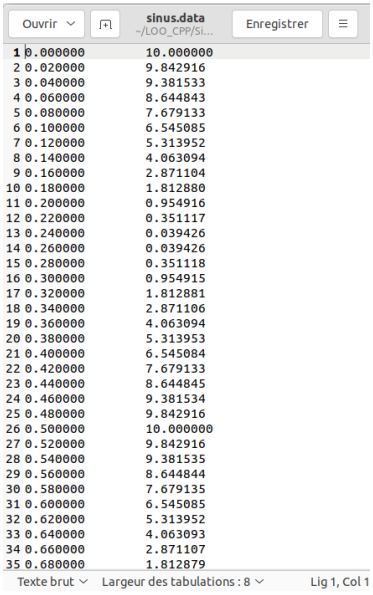
[4] Logiciel(s) nécessaire(s) en plus d'un environnement de développement C++ configuré (et maîtrisé).

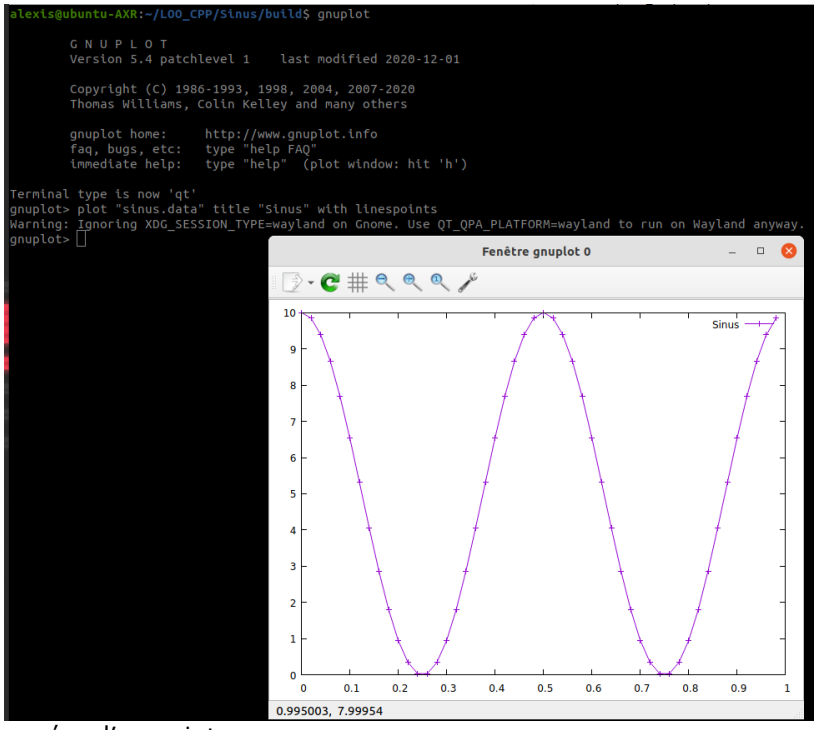
[5] Ressources spécifiques à l'activité pratique, en plus des ressources générales et transversales déjà mises à disposition.

Travail à effectuer en séance

Il s'agit lors de cette séance de développer (et valider) une classe « ***Sinus*** » permettant de construire des objets destinés à générer des fichiers de données (qui seront exploités par gnuplot) représentant des tracés de signaux sinusoïdaux.

Ces fichiers sont de simples fichiers texte, composés de n lignes et deux colonnes :





Chaque ligne comporte les coordonnées d'un point :

- ✓ Colonne 1 : coordonnée x, qui sera ici du temps (l'unité importe peu)
- ✓ Colonne 2 : coordonnée y, la valeur du signal au temps t

Le fichier de donnée généré sera exploité par gnuplot :

- ✓ plot "sinus.data" title "Sinus" with linespoints
  - sinus.data est le fichier contenant les points
  - Sinus est le titre donné au graphique
  - with linespoints indique à gnuplot que l'on souhaite que les points affichés soient reliés par une ligne

Modèle et Cahier des charges de la classe Sinus

Les principales propriétés d'un objet de type Sinus sont celles permettant de paramétrer le signal ainsi que l'échantillonnage : *Ao*, *Amplitude*, *Omega*, *Phio (phi **zéro**)*, *tStart*, *tStop* et *NbPoints*. L'annexe 1 rappelle le rôle de chacun de ces éléments.

Interface d'un objet Sinus

Outre la dimension codage, validation… on va s'attacher à travailler sur la qualité de l'interface (API) de l'objet. La seule règle à avoir à l'esprit lors de la spécification d'une API est : « *Make Your API Hard To Use Wrong* » (you may take a look at https://www.youtube.com/watch?v=zL-vn_pGGgY). Les points importants sur ce sujet seront abordés en cours de séance.

Accesseurs

Chacune des sept propriétés doit être associée à une paire d'accesseurs publics. Si des contraintes sont identifiées pour une propriété, elles doivent être vérifiées au niveau du mutateur (setter). Face à une contrainte non vérifiée, une action doit être entreprise, selon le cas il s'agira de :

- ✓ Lever une exception standard
- ✓ Corriger la valeur du paramètre fourni

Le tableau suivant recense les contraintes identifiées menant à la levée d'une exception en cas de non respect. Le type de l'exception ainsi que la valeur de retour de la méthode what().

Propriété	***Doit vérifier***	***Exception à lever***	***What()***
Omega	Supérieur ou égal à 0	std::domain_error	« Omega can't be negative. »
tStart	Supérieur ou égal à 0	std::domain_error	« tStart can't be negative. »
	Inférieur à tStop	std::overflow_error	« tStart can't be greater or equal than tStop. »
tStop	Supérieur à 0	std::domain_error	« tStop can't be negative or null. »
	Supérieur à tStart	std::underflow_error	« tStop can't be lesser than tStart. »
nbPoints	Supérieur à 0	std::invalid_argument	« nbPoints must be greater than 0. »

Concernant *Phio*, le mutateur doit contraindre cette valeur dans la gamme [-2π ;+2π].

Génération du fichier de points

Une méthode ***generate*** permet de lancer la fabrication du fichier de données représentant le sinus actuellement défini. Si aucun nom de fichier n'est fourni à cette méthode, le fichier créé prend la valeur par défaut « sinus.data ». D'un point de vue comportemental, cette méthode :

- ✓ Ne doit lancer un calcul du sinus que si un ou des paramètres ont été modifiés depuis la dernière génération de fichier.
- ✓ Faire remonter d'éventuelles exceptions liées à des erreurs fichiers
- ✓ Retourne la taille en octets du fichier généré

Création d'objet

Dans la mesure du possible, on tentera de respecter la règle du zéro quant aux ***Special Member Functions***. On n'hésitera pas à le faire apparaître explicitement en affichant dans le fichier d'entête l'utilisation des SMF par défaut.

Objet sans paramètre

Sans paramètre, l'objet crée est un Sinus dont les caractéristiques sont les suivantes :

- ✓ Ao = 0, Amplitude = 1, Omega = 2π, Phio = 0

Les paramètres de simulation associés sont :

- ✓ tStart = 0, tStop = 1, nbPoints = 100

Objet paramétré

La construction d'un objet, constant ou non, avec des paramètres est possible. Le constructeur prend alors comme paramètres :

- ✓ Les paramètres du sinus en lui-même (Ao, Amplitude, Omega et Phio)
- ✓ Les paramètres de simulation (tStart, tStop, nbPoints)

Ce constructeur a pour responsabilité de s'assurer que les paramètres passés sont cohérents. Dans le cas contraire, une exception est levée et la création de l'objet est abandonnée.
Les mêmes interruptions que celles des accesseurs sont à utiliser. A celles-ci va s'en ajouter une, qui sera levée si Phio est en dehors de la gamme]-2π ;+2π[:

- ✓ std::domain_error("Initial phio can't be outside bounds.")

Organisation du travail

Pour aller plus loin…

S'il reste du temps, on pourra réfléchir à ajouter à cette classe (liste non ordonnée):

- ➢ Des accesseurs directement pour les champs SimulParams et Parameters
- ➢ De nouveaux constructeurs prenant comme paramètres :
  - o Seulement les paramètres d'un sinus
  - o Seulement les paramètres de simulation
  - o Dans ces cas, les autres champs restent initialisés avec les valeurs par défaut. La mise en place d'un ***constructeur délégué*** peut-être ici envisagée.
- ➢ Une série de tests permettant de valider le comportement des constructeurs et opérateurs par défaut (constructeur par copie, opérateur de copie, constructeur « move », opérateur de « move »).

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

Nous avons actuellement à notre disposition une classe « Sinus » nous permettant de créer des objets destinés à « tracer » des signaux sinusoïdaux.
En partant de cette expérience, et sachant qu'il existe d'autres types de signaux (en fait non, mais oui…) proposer des idées, stratégies, permettant, en conservant tout ou partie des travaux de gérer ces autres types de signaux. Présenter, sans trop entrer dans les détails, l'architecture que prendrait alors le diagramme des classes.
Proposer, en les justifiant, mais sans juger de la complexité/faisabilité, de nouvelles fonctionnalités pouvant être intéressantes dans ce nouveau cadre.

Annexes

Annexe 1 : Un signal sinusoïdal, sa représentation temporelle et son calcul

Un signal sinusoïdal est défini par l'équation suivante :

$$Sin_{tn} = S(t_n) = A_0 + A.\sin(\omega.t_n + \varphi_0)$$

Où :

- ✓ t_n est l'échantillon de temps
- ✓ Sin_{tn} est la valeur du signal au temps t_n
- ✓ ***Ao*** est la valeur moyenne, ou composante continue du signal
- ✓ ***A*** est l'amplitude du signal (Valeur max = Ao+A, Valeur min = Ao-A)
- ✓ ω est la pulsation du signal en rad.s^{-1} .On rappelle :
  - ω=2πF, avec F Fréquence en Hz
  - ω = 2π/T, avec T période en secondes
  - ω ne peut être négative
- ✓ φ_o est la phase à l'origine. Il s'agit d'un angle compris entre -2π et +2π.

Une fois que les paramètres du signal (*Ao, A, ω et φo*) sont déterminés, il suffit de calculer pour chaque échantillon de temps la valeur de l'échantillon en sortie.
Le vecteur (tableau) des échantillons de temps en entrée est construit à partir de 3 paramètres :

- ✓ ***tStart*** : le temps « 0 »
- ✓ ***tStop*** : le temps « final »
- ✓ ***NbPoints*** : Le nombre d'échantillons entre ces deux temps
  - ***dt***: est l'intervalle de temps entre deux échantillons
    - $dt = \frac{tStop - tStart}{NbPoints}$
- ✓ La seule contrainte est ici d'avoir un dt > 0 c'est-à-dire que ***tStop doit être strictement supérieur à tStar***t.
  - On considère, bien entendu, que ces temps ne peuvent être négatifs.

Le vecteur (tableau) des entrées peut donc être facilement rempli :

Pour (i de 0 à NbPoints(exclu) par pas de 1) tab_input[i] <- tStart + i * dt

Le tracé est dépendant des paramètres du signal mais aussi des paramètres d'échantillonnage.
Ainsi, pour réaliser le tracé d'une période d'un signal à 1Hz, à partir du temps « 0 », il faut :

- ✓ tStart = 0
- ✓ tStop = 1 (en considérant que l'unité est ici la seconde)
- ✓ ω = 2π rad.s^{-1}

Annexe 2 : API Design

How to « Make your API hard to use wrong »…
Ref : CPPCON 2022 – Jason TURNER « API Design »

- ✓ https://www.youtube.com/watch?v=zL-vn_pGGgY
- ✓ https://github.com/CppCon/CppCon2022/blob/main/Presentations/CppCon-2022-Jason-Turner-API-Design-Back-to-Basics.pdf

Better naming

Naming is hard…

Intéressé par le sujet :Kate Gregory - CppCon 2019 – « Naming is hard, let's do better »

- ✓ https://www.youtube.com/watch?v=MBRoCdtZOYg
- ✓ https://github.com/CppCon/CppCon2019/blob/master/Presentations/naming_is_hard_lets_do_better/naming_is_hard_lets_do_better__kate_gregory__cppcon_2019.pdf

[[nodiscard]]

Instructs the compiler to generate a warning if a return value is dropped. ***Can be applied to types or function declarations (including ctors)***.

```
[[nodiscard]] int get_value();

int main() {
  get_value(); // warning issued from any reasonable compiler
}                                                https://godbolt.org/z/xvrjhjGdK
```

```
struct [[nodiscard]] ErrorType{};
ErrorType get_value();

int main() {
  get_value(); // warning issued from any reasonable compiler
}                                                https://godbolt.org/z/Gdv8YsMcG
```

```
struct FDHolder {
  [[nodiscard]] FDHolder(int FD);
  FDHolder();
};

int main() {
  FDHolder{42};   // warning
  FDHolder h{42}; // constructed object not discarded, no warning
  FDHolder{};     // default constructed, no warning
}                                                https://godbolt.org/z/x5e87r7Ka
```

- ✓ Used to indicate when it is an error to ignore a return value from a function
- ✓ Can be applied to constructors as of C++20
- ✓ Can have a message to explain the error
  - [[nodiscard("Lock objects should never be discarded")]]
- ✓ Should be used extensively
- ✓ Any non-mutating (getter/accessor/ const) function should be [[nodiscard]]

noexcept

noexcept notifies the user (and compiler) that a function may not throw an exception. If an exception is thrown from that function, terminate MUST be called.

```
void myfunc() noexcept {
  // required to terminate the program
  throw 42;
}

int main() {
  try {
    myfunc();
  } catch(...) {
    // catch is irrelevant, `terminate` is called
  }
}
```

Never return a Raw Pointer

- ✓ It simply raises too many questions. Who owns it? Who deletes it? Is it a singleton global?
- ✓ Consider owning_ptr , non_owning_ptr or some kind of wrapper to document intent, if you must.

Consistent Error Handling

- ✓ Use one consistent method of reporting errors in your library
- ✓ Strongly avoid out-of-band error reporting (get_last_error() or errno)
- ✓ Make errors impossible to ignore (no returning an error code!)
- ✓ Never use std::optional<> to indicate an error condition. (it does not convey a reason, and the
- ✓ reason becomes out of bound).
- ✓ Consider std::expected<> (C++23) or similar

Avoid Easily Swappable Parameters

```
FILE *fopen(const char *pathname, const char *mode);
```

Two (or more) parameters beside each other of the same type are easy to swap.

Avoid Implicit Conversions / Use Strong Types

- ✓ std::filesystem::path and std::string_view appear to be strongly typed but are not
- ✓ Implicit conversions between ***const char ****, ***string***, ***string_view*** , and ***path*** break type safety
- ✓ ***Conversion operators and single parameter constructors*** (including variadic and ones with default parameters) ***should be explicit***

=delete Problematic Overloads

- ✓ Any function can be =delete d.
- ✓ If you =delete a template, it will become the match for any non-exact parameters, and prevent implicit conversions

```
1  using FilePtr = std::unique_ptr<FILE, decltype([](FILE *f) { fclose(f); })>;
2
3  [[nodiscard]] FilePtr fopen(const std::filesystem::path &path,
4                              std::string_view mode);     https://godbolt.org/z/rb7TvhGc9
```

Assuming `std::filesystem::path` and `std::string_view` are the most correct types for this use case, can we make this better?

```
1  void fopen(const auto &, const auto &) = delete;
```

Only Pass Raw Pointers for Single Optional Objects / If you pass a pointer, you must check it for nullptr

Prefer & Parameters For Non-Small, Non-TrivialObjects

```
#include <string>

void use_string(std::string * const * str) {
  if (str) { // is str optional?
    // do things
  } else {
    // do other things
  }
}
```

```
#include <string>

void use_string(std::string const * const str) {
  puts(str->c_str()); // do not do, unsafe
}
```

Fuzz Your Interfaces

- ✓ fuzzer - a tool that tests your API against a set of "random" inputs.
- ✓ Should be run with something like address/undefined sanitizers enabled
- ✓ Uses your API in ways that you never would
- ✓ Can be used with any API with creativity
- ✓ Helps discover patterns of misuse internal to your API

Summary…

- ✓ Try to use your API incorrectly
- ✓ Use better naming
- ✓ Use [[nodiscard]] (with reasons) liberally
- ✓ Never return a raw pointer
- ✓ Use noexcept to help indicate the type of error handling
- ✓ Provide consistent, impossible to ignore, in-band error handling
- ✓ Use stronger types and avoid default conversions
- ✓ (Sparingly) delete problematic overloads / prevent conversions
- ✓ Avoid passing pointers (only to be used for single/optional objects)
- ✓ Avoid passing smart pointers Limit your API as much as possible
- ✓ Fuzz your API

Annexe 3 : Diagramme des classes (partiel)

Une première analyse a permis de produire le diagramme des classes ci-dessous.
Ce diagramme doit être observé avec quelques précautions :

- ✓ Il n'est pas forcément complet
- ✓ Il doit être, notamment au niveau des accesseurs, interprété : les accesseurs doivent être liés aux champs des structures ComputeParameters et SinusParam et non pas aux objets de ces types (on a des accesseurs pour Amplitude, Ao… et non pas pour Parameters)
- ✓ Il peut, si les arguments pour sont valides, être modifié (l'intérêt de la classe « helper » peut notamment être remis en question)
- ✓ Il doit être complété si des champs et/ou méthodes apparaissent a posteriori
- ✓ Il doit être commenté en lien avec le code pour bien faire apparaître les liens entre analyse et code.
- ✓ …