ℹ️

Avant d'entamer ce cours, il est nécessaire de bien connaitre les bases du langage C, qui ne seront pas revues ici.

Vous retrouverez les pré-requis ici:

Langage C: Cours bases

Langage C: Cours avancé

Du C au C++

Le langage C++ est un successeur du Langage C auquel il apporte de nombreuses nouvelles choses qui vont permettre d'aller plus loin dans la complexité des programmes, notamment la programmation objet.

Il n'est pas exactement un remplaçant, car le C, plus léger et bas niveau, reste à privilégier dans certains cas d'usages tels que les systèmes embarqués.

Son compilateur principal reste **gcc**, et sa syntaxe reste compatible avec celle du C, mais de nouvelles façons d'écrire du code viennent apparaitre, qui faciliteront souvent la vie du développeur.

Nous retrouverons notamment sa bibliothèque standard `std` qui nous fournira beaucoup d'outils pour aider à la gestion de la memoire, des tableaux, pointeurs etc...

Enfin son caractère **objet** sera très utile pour construire des architectures logicielles propres, modulaires et maintenables.

La documentation officielle:

https://en.cppreference.com/w/

Cheat sheet: https://cheatsheets.zip/cpp

Compilation

Pour compiler nos programmes en C++ nous utiliserons la commande suivante:

```
g++ -Wall -std=c++17 code.cpp -o executable.exe
```

Décomposons:

- `g++` la version C++ de **gcc**
- `-Wall` afficher les “warnings”, avertissements sur problèmes dans le code
- `-std=c++17` pour utiliser la dernière version de C++
- `code.cpp` notre fichier texte de code que l’on souhaite compiler
- `-o executable.exe` la sortie (output, o) de la compilation (donc l’éxécutable) s’appellera `executable.exe`

Sous Max/Linux, ne pas rajouter `.exe` *à l’executable*

Pour executer:

```
./executable.exe
```

Versions

Il existe differentes versions de C++, et par défaut **gcc** utilise la version **97**

Les versions les plus recentes incluent les modifications (17 inclut 11, qui inclut 97)

Parfois en utilisant des exemples trouvé sur internet, vous verrez que votre code ne compile plus car ces exemples utiliseront probablement du code de nouvelles versions, pour laquelle la syntaxe a changé.

Pour changer la version de C++ utilisée, rajouter `-std=c++17` pour la version `2017` par exemple (il existe `c++97` `c++11` `c++17`)

Entrées/sorties

En C, pour afficher et demander des informations dans la console, nous utilisions `printf` et `scanf`.

Désormais nous utiliserons les entrées et sorties de la bibliothèque standard.

Il est nécessaire d'inclure `<iostream>`

Sorties

Voici comment fonctionne la sortie avec `std::cout`

```cpp
#include <iostream>

int main()
{
    // Un caractère peut être une lettre.
    std::cout << 'A' << std::endl;
    // printf("A\n");

    // Ou bien un chiffre.
    std::cout << 7 << std::endl;

    // Ou bien une chaine de caractère.
    std::cout << "Bonjour" << std::endl;

    // Ou bien une variable
        int temperature = 25;
    std::cout << temperature << std::endl;

    // On peut aussi les combiner
    std::cout << "Il fait " << temperature << " degrées" << std::end
    // printf("Il fait  %d degrées\n", temperature);

    return 0;
}
```

Pour sauter des lignes, nous utilisons `std::endl` (équivalent de `\n`)

Les doubles crochets `<<` sont des opérations de `stream` et permettent de gerer un flux de données entre des entités.

Et plus besoin de se rappeler les codes des types de variables avec `printf` !

Entrées

Pour les entrées nous utiliserons `std::cin` en changeant le sens des crochets `>>`

cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Entre ton age : " << std::endl;
    int age = 0;
    std::cin >> age;
    std::cout << "Tu as " << age << " ans.\n";

    return 0;
}
```

Pour les chaines de caractères, utiliser `std::getline`

cpp

```cpp
#include <iostream>

int main()
{
    std::string phrase;
    std::getline(std::cin, phrase);
    std::cout << phrase;
```

```
    return 0;
}
```

Gestions des erreurs des entrées

Avec le code suivant, si on tape une chaine de caractère au lieu d'un entier, on aura un problème pour taper son nom:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "Entre ton age : ";
    unsigned int age = 0;

    std::cin >> age;
    std::cout << "Tu as " << age << " ans.\n";
    std::cout << "Entre ton nom : ";
    std::string nom = "";
    std::cin >> nom;
    std::cout << "Tu t'appelles " << nom << ".\n";

    return 0;
}
```

La solution réside dans l'exemple suivant, qui va tester si l'entrée s'est bien passée, et remettre a zero `cin` autrement

```cpp
#include <iostream>
#include <limits>
#include <string>

template <typename T>
void input(T &var)
{
```

```
    while (!(std::cin >> var))
    {
        std::cout << "Il y a eu une erreur, recommencer:" << std::en
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
    }
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n
}

int main()
{
    unsigned int age = 0;

    std::cout << "Entre ton age : ";
    input(age);
    std::cout << "Ton age " << age << std::endl;

    return 0;
}
```

Types de variables

bool

Enfin nous avons officiellement un type de variable pour les booléens `true` et `false`.

```cpp
bool isReady = true;
```

Modifier const

Si une variable ne doit jamais changer, ont dit que c'est une constante. Pour forcer une variable à être constante, nous rajouterons le mot clé `const` avant *ou* après son type lors de sa definition.

```cpp
int const PI = 3.14;
const int PI = 3.14; // equivalent

PI = 6.2; // Plus possible
```

Les pointeurs

Les references

En C nous avions les pointeurs, en C++ est introduite la notion de *reference*.

Le concept est similaire, a la difference près qu'une reference s'utilise de la même manière qu'une variable normale, et quelle est liée à une autre variable.

```cpp
#include <iostream>
#include <string>

void fonctionParPointeur(int *pointeur)
{
    *pointeur = 0;
}
void fonctionParRef(int &reference)
{
    reference = 0;
}

int main()
{
    int variable = 24;
    // reference, sa valeur sera toujours identique à variable
    int & reference_variable = variable;

    fonctionParPointeur(&variable);
    std::cout << variable << std::endl;
```

```
    fonctionParRef(variable);
    std::cout << variable << std::endl;
    std::cout << reference_variable << std::endl; // même valeur que

    return 0;
}
```

On va souvent passer des arguments par reference aux fonctions, car cela evitera de devoir les copier (particulièrement pour les classes).

cpp
```cpp
#include <vector>
#include <iostream>


void print_container(const std::vector<int>& c)
{
    for (int i : c)
        std::cout << i << ' ';
    std::cout << '\n';
}

int main()
{
    std::vector<int> c = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    print_container(c);
}
```

On utilisera souvent le mot clé `const` avec les references.

Si il n'est n'est pas nécessaire de modifier un argument d'une fonction, on l'indiquera en tant que *constante*.

cpp
```cpp
// s est un pointeur modifiable vers une variable non modififable
const std::string & s;
// Equivalent
std::string const & s;

// s est un pointeur non modifiable vers une variable modifiable
```

```cpp
std::string * const s;
// s est un pointeur non modifiable vers une variable non modifiable
const std::string * const s;
```

```cpp
void fn(std::string& s)
{
        s = "hello"; // possible
}
void fn(const std::string& s)
{
        s = "hello"; // pas possible
}
```

Allocation de la mémoire

En nous C utilisions `malloc` pour allouer de la mémoire, et `free` pour la libérer. Désormais nous utiliseront `new` et `delete`

```cpp
int * entier = new int; // alloue un entier

delete entier;          // libère l'entier

int * tableau = new int[10]; // alloue un tableau de 10 entiers

delete [] tableau;    // Libère un tableau d'entier, ne pas oublier
```

Les conteneurs

Une notion appelée conteneurs fait son apparition, et servira a stocker des ensembles de variables.

En C, notre seule option était d'utiliser des tableaux, qui servaient à stocker des suites de variables du même type. On les écrivait comme cela:

```cpp
int tableau[5] = { 1, 2, 3, 4, 5 };
```

Désormais, nous pourrons utiliser les conteneurs de la librairie standard, dont particulièrement `std::array` , `std::vector` et `std::string`

Array

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

`std::array` est l'équivalent des tableaux que nous avions en C, c'est à dire **statiques**.

Nous ne pourrons plus modifier sa taille, ajouter ou supprimer des éléments une fois qui aura été créé.

Il est nécessaire d'inclure `<array>`

- Déclarer un tableau statique

```cpp
#include <array>

int main()
{
        // valeurs modifiables
        std::array<int, 5> tableau = { 1, 2, 3, 4, 5 };

        // similaire à
        int tableau[5]

        // valeurs non modifiables
        std::array<int, 5> const tableau_constant = { 1, 2, 3, 4, 5

    return 0;
}
```

- Acceder aux elements

```cpp
// Comme en C
tableau[3] // lit le 4 element
```

- Modifier un element

```cpp
// Comme en C
tableau[3] = 12;
```

- Remplir le tableau

```cpp
tableau.fill(10);
```

- Connaitre la taille

```cpp
std::size(tableau)
```

- Copier un tableau

```cpp
std::array<int, 5> copie_du_tableau = tableau;
// On obtient deux tableaux distincts
// si on modifie l'un, l'autre ne sera pas modifié

// Different de
int t1[5];
int t2[5];
t2 = t1;  // ici on a deux pointeurs sur un seul tableau
// si on modifie l'un, l'autre sera modifié aussi
```

- Itérer sur le tableau

```cpp
// For classique
for (int i = 0; i < std::size(tableau); i++)
{
    std::cout << tableau[i] << std::endl;
```

```
}
// For Each
for (int const element : tableau)
{
    std::cout << element << std::endl;
}
```

String

https://en.cppreference.com/w/cpp/string/basic_string

Pour rappel, les chaines de caractères sont en fait des suites de caractères simples. En C, nous utilisions les tableaux de caractères `char []` ou `char *`, en C++ nous utiliserons `std::string`

Il est nécessaire d'inclure `<string>`

```cpp
#include <string>

int main()
{
        std::string phrase = "Voici une phrase normale.";

        phrase = "Voici une autre phrase normale.";

    return 0;
}
```

Leur fonctionnement de base est le meme que pour les Array, avec des possibilités en plus

- **Premier et dernier caractère**

```cpp
std::cout << "Première lettre : " << phrase.front() << std::endl;
std::cout << "Dernière lettre : " << phrase.back() << std::endl;
```

- **Vérifier qu'une chaîne est vide**

```cpp
std::cout << "Est ce vide ? " << std::empty(phrase) << std::endl;
```

- **Ajouter ou supprimer un caractère à la fin**

```cpp
phrase.pop_back();     // supprimer un caractère a la fin
phrase.push_back('.'); // ajouter un caractère a la fin
```

- **Supprimer tous les caractères**

```cpp
phrase.clear();
```

- **Comparer deux chaines**

```cpp
phrase.compare("test");
std::string p2 = "test";
phrase.compare(p2);
```

- **Découper une chaine**

```cpp
std::string phrase = "Hello world";
std::string s1 = phrase.substr(0, 5);  // = "Hello"
std::string s2 = phrase.substr(6, 11); // = "world"
```

Ajouter un `s` apres une chaine de caractère le force à être un `std::string`

Vector

https://en.cppreference.com/w/cpp/container/vector

Enfin, le nouveau type de tableau le plus interessant que nous allons retrouver en C++ est `std::vector`

Ce dernier nous permettra de créer des **tableaux dynamiques**, dont la taille pourra être modifiée. On pourra ajouter et supprimer des éléments à ce

tableau.

Notons ici que les valeurs stockées en mémoire sont **contiguës**.

Il est nécessaire d'inclure `<vector>`

```cpp
#include <string>
// N'oubliez pas cette ligne.
#include <vector>

int main()
{
    std::vector<int> tableau_de_int;
    std::vector<double> tableau_de_double;
    // Même avec des chaînes de caractères c'est possible.
    std::vector<std::string> tableau_de_string;

    return 0;
}
```

Le fonctionnement est encore similaire à Array, avec des choses en plus. Ce qui est faisable avec `array` est aussi faisable avec `vector`

- **Premier et dernier element**

```cpp
std::cout << "Premier : " << tableau_de_int.front() << std::endl;
std::cout << "Dernier : " << tableau_de_int.back() << std::endl;
```

- **Vérifier si un tableau est vide**

```cpp
std::cout << "Est-ce vide ? " << std::empty(tableau_de_int) << std::
```

- **Ajouter un element à la fin**

```cpp
tableau_de_int.push_back(36);
```

- **Supprimer le dernier élément**

```cpp
tableau_de_int.pop_back();
```

- **Assigner des valeurs**

```cpp
tableau_de_int.assign(10, 42); // on affecte 10 fois la valeur 42
```

Exemple d'utilisation

```cpp
std::vector<std::string> tableau_de_string;

tableau_de_string.push_back("Phrase 1"); // Ajout à la fin
tableau_de_string.push_back("Phrase 2"); // Ajout à la fin

// For normal
for(int i=0; i< std::size(tableau_de_string); i++) {
    std::cout << tableau_de_string[i] << std::endl;
}

// For Each
for(std::string s : tableau_de_string) {
    std::cout << s << std::endl;
}

tableau_de_string.pop_back(); // Suppression du dernier
tableau_de_string.clear(); // Suppression de tous les elements
```

Itérateurs

Les types de conteneurs que nous venons de voir sont particuliers car nous pouvons acceder à leurs élements par leur index numérique avec la notation `container[index] = element`

Certains types de conteneurs n'auront pas d'index composé d'entiers numériques successifs. Pour parcourir les éléments de ces conteneurs, il a

donc été inventé ce qu'on appelle les itérateurs.

Pour déclarer un itérateur, nous prendrons le type de son conteneur et ajouterons `::iterator`

Par exemple, `std::array<float, 5>::iterator`

Pour des conteneurs constants, utiliser `const_iterator`

Pour créer un itérateur au début du tableau, on utilise `std::begin()`

`std::end()` lui pointera non pas sur le dernier element, mais sur un element virtuel inexistant, **après** le dernier element

```c
std::vector<int> tableau = { -1, 28, 346, 1000 };
std::vector<int>::iterator debut_tableau = std::begin(tableau);

std::vector<float> const tableau_constant = { -1, 28, 346 } ;
std::vector<float>::const_iterator fin_tableau_constant = std::end(t
```

Pour acceder à un element pointé par un itérateur, il faut le **déreferencer**:

```cpp
std::cout << *debut_tableau << std::endl; // -1
```

Pour se déplacer, on incrémentera ou décrementera l'itérateur:

```cpp
std::cout << *debut_tableau << std::endl; // 28
debut_tableau++;
std::cout << *debut_tableau << std::endl; // 346
debut_tableau--;
std::cout << *debut_tableau << std::endl; // 28
debut_tableau += 2;
std::cout << *debut_tableau << std::endl; // 1000
```

Exemple dans une boucle:

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau = { -1, 28, 346, 84 };

    for (std::vector<int>::iterator it = std::begin(tableau); it !=
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Utilisation des itérateurs dans les containers

Certains conteneurs tels que `std::vector` necessite en paramètres des itérateurs pour certaines de leurs fonction

- **Supprimer** un élément dans un tableau avec `erase`

```cpp
std::vector<int> nombres = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// suppression du quatrième element
nombres.erase(std::begin(nombres) + 3);
// suppression des elements entre le premier (inclus) et le dernier
nombres.erase(std::begin(nombres), std::begin(nombres) + 2);
```

Algorithms

```cpp
#include <algorithm>
```

- **compter** avec `count`

```cpp
std::string const phrase = "Exemple de phrase.";
// compter les 'e' dans toute la phrase
int const total_phrase { std::count(std::begin(phrase), std::end(phr
```

- **trouver** avec `find`

```cpp
std::string const phrase { "Exemple illustrant le tutoriel C++ de Ze

// On obtient un itérateur pointant sur le premier espace trouvé.
std::string::iterator iterateur_mot = std::find(std::begin(phrase),
// Si l'on n'avait rien trouvé, on aurait obtenu std::end(phrase) co
```

- **trier** avec `sort`

Fonctionne avec les nombres, caractères... Va modifier le container de départ pour le réordonner

```cpp
std::vector<double> constantes_mathematiques = { 2.71828, 3.1415, 1.
std::sort(std::begin(constantes_mathematiques), std::end(constantes_
```

- **inverser** l'ordre avec `reverse`

Va également modifier le container de départ

```cpp
std::list<int> nombres { 2, 3, 1, 7, -4 };
std::reverse(std::begin(nombres), std::end(nombres));
```

- **Etc** ... Il y en a beaucoup, allez lire les documentations ! Il existe certainement déjà un outil pour réaliser l'opération que vous souhaitez realiser...

Plus de containers

Maintenant que nous avons vu les bases des containeurs et les itérateurs, nous pouvons utiliser des types plus complexes.

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

Sets

Les sets sont des conteneurs qui contient un ensemble de valeurs uniques. A voir comme un `vector` mais où l'on ne pourra pas mettre deux fois la même valeur.

```cpp
std::set<int> set = {1, 5, 3};

std::array<int, 7> arr = {1, 2, 2, 3, 3, 3, 4};
std::set<int> s2 = std::set(arr.begin(), arr.end());
// s2 == { 1, 2, 3, 4 };
```

Maps

Les maps sont des listes indexées par des Clés→Valeurs. Les clés sont uniques.

Imaginer un tableau à deux colonnes:

Clé	Valeur
CPU	10
GPU	15
RAM	20

```cpp
std::map<int, int> redirections;
redirections[80] = 8000; // ajouter/modifier un element
redirections[21] = 1021;
f.erase(21); // Supprimer un element
```

```cpp
std::map<std::string, int> computer;
computer["CPU"] = 10;
computer["GPU"] = 15;
computer["RAM"] = 20;
std::cout << computer["GPU"];  // Recuperation de valeur

// Initialisation
std::map<std::string, int> details{{"CPU", 10}, {"GPU", 15}, {"RAM",

// Iterer avec iterateurs
for (std::map<std::string, int>::const_iterator it = m.begin(); it !
    std::cout << it->first << " = " << it->second;
}

// Iterer avec For Each
// Chaque ligne du tableau est une paire
for (std::pair<std::string, int> paire : m)
{
    std::cout << "Clé : " << paire.first << std::endl;
    std::cout << "Valeur : " << paire.second << std::endl << std::en
}
```

Unordered

Ces deux types sont des versions appellées “ordonnées”, leur clés/valeurs sont triés par ordre croissant. Elles possèdent aussi une version “unordered”, qui est plus performante, mais dont l’ordre de lecture sera aléatoire.

```cpp
std::unordered_map<std::string, int> m;
std::unordered_set<int> s;
```

- Utilisez `std::map` lorsque vous avez besoin de maintenir les éléments triés ou lorsque vous avez besoin d'itérer sur les éléments dans un ordre spécifique.
- Utilisez `std::unordered_map` lorsque la performance est critique et que l'ordre des éléments n'a pas d'importance.

Polymorphisme

La surcharge de fonction

La surcharge de fonctions (ou surchage de méthodes) est un concept en C++ qui permet de définir plusieurs fonctions ayant le même nom mais des signatures différentes au sein d'une même portée. La signature d'une fonction inclut son nom, le nombre et le type de ses paramètres. La surcharge permet d'implémenter des fonctions qui accomplissent des tâches similaires mais avec des types ou nombres de paramètres différents.

Par exemple:

```cpp
int addition(int a, int b) {
    return a + b;
}

// Fonction pour ajouter deux nombres à virgule flottante
double addition(double a, double b) {
    return a + b;
}

// Fonction pour ajouter trois entiers
int addition(int a, int b, int c) {
    return a + b + c;
}

std::cout << addition(3, 4) << std::endl; // Appelle addition(int, i
std::cout << addition(3.5, 2.5) << std::endl; // Appelle addition(do
std::cout << addition(1, 2, 3) << std::endl; // Appelle addition(int
```

Au moment de l'appel à la fonction, le programme reconnaitra automatiquement, en fonction des paramètres donnés, la quelle appeler.

Les paramètres des prototypes des differentes fonction surchargées doivent être differentes pour qu'il puisse choisir.

L'exemple suivant ne fonctionne pas:

```cpp
int fonction(int a);
double fonction(int a); // Erreur : le type de retour seul ne peut p
```

La programmation Objet

Avant de plonger dans les concepts de base de la programmation orientée objet (POO) en C++, rappelons-nous que la POO est un paradigme de programmation qui utilise des "objets" pour modéliser des éléments du monde réel. Les objets sont des instances de classes, qui définissent leurs propriétés et comportements.

Classes et Objets

Les classes représente des élément de notre programme qui existeront en plusieurs exemplaires et qui auront chacune des valeurs et leur propre logique.

Les classes sont composées d'attributs (variables membres) ainsi que de méthodes (fonctions membre)

Exemples de classes

1. **Utilisateur**
   - **Attributs** : identifiant, nom, email, mot de passe (haché), rôle (admin, utilisateur, etc.)
   - **Méthodes** : authentifier(), changerMotDePasse(), afficherProfil()
2. **Session**
   - **Attributs** : idSession, utilisateur, heureDebut, heureFin, adresseIP
   - **Méthodes** : démarrer(), terminer(), estActive()
3. **Pare-feu**
   - **Attributs** : règles, état (activé/désactivé)
   - **Méthodes** : ajouterRègle(), supprimerRègle(), vérifierPaquet()

4. **Cryptographie**
   - **Attributs** : algorithme, clé
   - **Méthodes** : chiffrer(données), déchiffrer(données), générerClé()
5. **Réseau**
   - **Attributs** : adressesIP, sous-réseaux
   - **Méthodes** : scannerPorts(), analyserTrafic(), configurerRoute()

Définition d'une Classe

- **Syntaxe de base** :

```cpp
class NomDeClasse {
    // portée
  public:
      // Attributs (variables membres)
      int attribut;

      // Méthodes (fonctions membres)
      void methode();
};
```

- **Exemple** :

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

      void afficherDetails() {
          std::cout << "Marque: " << marque << ", Année: " << annee
      }
};
```

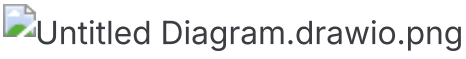
Création d'un Objet

Pour utiliser les classes, nous allons les **instancier** pour creer des **objets**.

Une classe n'est pas utilisable en soit. Nous allons creer des variables du type de la classe, et ces variables seront les objets.

- **Instance d'une classe** :

```cpp
Voiture maVoiture;
maVoiture.marque = "Toyota";
maVoiture.annee = 2022;
maVoiture.afficherDetails();
```



Difference entre structures et classes

- Avec structures:

```cpp
#include <iostream>
#include <string>

struct Personne {
    std::string nom;
    int age;
};

// Méthode pour définir le nom
void setNom(struct Personne& personne,  const std::string& nom) {
    personne.nom = nom;
}

// Méthode pour obtenir le nom
std::string getNom(struct Personne& personne) {
    return personne.nom;
}

// Méthode pour afficher les informations de la personne
void afficherInfos(struct Personne& personne) const {
    std::cout << "Nom: " << nom << ", Age: " << age << std::endl;
```

```
}

int main() {
    Personne personne1;
    setNom(personne1, "Alice");
    personne1.age = 12;
    afficherInfos(personne1);

    Personne personne2;
    setNom(personne1, "Bob");
    personne2.age = 21;
    afficherInfos(personne2);

    return 0;
}
```

- Avec les classes:

cpp

```cpp
#include <iostream>
#include <string>

class Personne
{
public:
    std::string nom;
    int age;

    // Méthode pour définir le nom
    void setNom(const std::string &n)
    {
        this->nom = n;
    }

    // Méthode pour obtenir le nom
    std::string getNom()
    {
        return this->nom;
    }
```

```
    // Méthode pour afficher les informations de la personne
    void afficherInfos()
    {
        std::cout << "Nom: " << this->nom << ", Age: " << this->age
    }
};

int main()
{
    Personne personne1;
    personne1.setNom("Alice");
    personne1.age = 32;
    personne1.afficherInfos();

    Personne personne2;
    personne1.setNom("Bob");
    personne1.age = 12;
    personne1.afficherInfos();

    return 0;
}
```

cpp

```
class Personne
{
public:
    std::string nom;
    int age;

    // Méthode pour définir le nom
    void setNom(const std::string &n);
};

void Personne::setNom(const std::string &n)
{
    this->nom = n;
}
```

Classes et pointeurs

Il est possible d'avoir des pointeurs vers des instances de classes (=objets), et dans ce cas nous accederons à ses membres avec le symbole `->` plutot que le point `.` de la même manière que pour les structures.

```cpp
Classe objet;
Classe *pointeurObjet = &objet;

std::cout << objet.membre << std::endl;
std::cout << pointeurObjet->membre << std::endl;
```

Le mot-clé this

Afin d'acceder au membres d'une classe à l'intérieur même d'une classe, on utilisera le pointeur spécial `this` qui pointe vers l'instance actuelle de l'objet.

```cpp
class Classe {
public:
    int var;

    int get() {
        return this->var;
    }
    int set(int var) {
        this->var = var;
    }
};
```

Constructeurs et Destructeurs

Constructeurs

Définition :

- Un constructeur est une méthode spéciale d'une classe qui est automatiquement appelée lorsqu'un objet de cette classe est créé. Il initialise l'objet nouvellement créé.
- On s'en servira principalement pour initialiser les variables membre de la classe, allouer la mémoire nécessaire et initialiser le contexte

Caractéristiques :

- **Nom** : Le constructeur porte le même nom que la classe.
- **Pas de type de retour** : Contrairement aux autres méthodes, les constructeurs n'ont pas de type de retour, même pas `void`.
- **Automatique** : Il est appelé automatiquement lors de l'instanciation de la classe.

Surcharge :

- Les constructeurs peuvent être surchargés, c'est-à-dire que plusieurs constructeurs peuvent être définis dans une même classe, chacun ayant une signature différente (différents paramètres).

Types de Constructeurs :

- **Constructeur par défaut** : Un constructeur sans paramètres. Si aucun constructeur n'est défini, le compilateur en génère un par défaut.
- **Constructeur paramétré** : Un constructeur qui prend des arguments pour initialiser les attributs de la classe avec des valeurs spécifiques.
- **Constructeur de copie** : Un constructeur qui initialise un objet en le copiant à partir d'un autre objet de la même classe. Sa signature prend un argument qui est une référence constante à un objet de la même classe.

```cpp
class MaClasse {
  public:
      MaClasse() {} // Constructeur par defaut
      MaClasse(int a, float b) {} // Constructeur parametré
      MaClasse(const MaClasse & objet) {} // Constructeur de copie
};
```

Initialisation des Membres :

- Les constructeurs peuvent utiliser une liste d'initialisation des membres pour initialiser les attributs avant que le corps du constructeur ne soit exécuté. Cela est souvent plus efficace et nécessaire pour les membres constants ou les références.

```cpp
class MaClasse {
  public:
      int nombre;

    MaClasse(): nombre(0) {}
};
```

- exemple avec Voiture

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

            // Constructeur par defaut
      Voiture() {
          marque = "";
          annee = 0;
      }
      // Constructeur parametré
      Voiture(std::string m, int a) {
          marque = m;
          annee = a;
      }
      // Equivalent à
      Voiture(std::string m, int a): marque(m), annee(a) {}

      // Constructeur de copie
      Voiture(const Voiture& v): marque(v.marque), annee(v.annee) {}
};

int main() {
```

```
    Voiture v1 = Voiture("Toyota", 2001);
}
```

Destructeurs

Définition :

- Un destructeur est une méthode spéciale d'une classe qui est automatiquement appelée lorsqu'un objet de cette classe est détruit. Il nettoie les ressources allouées par l'objet avant que celui-ci ne soit retiré de la mémoire.

Caractéristiques :

- **Nom** : Le destructeur porte le même nom que la classe, précédé d'un tilde (`~`).
- **Pas de type de retour** : Comme le constructeur, le destructeur n'a pas de type de retour.
- **Automatique** : Il est appelé automatiquement lorsque l'objet sort de son scope ou est explicitement détruit.

Unique :

- Une classe ne peut avoir qu'un seul destructeur. Contrairement aux constructeurs, les destructeurs ne peuvent pas être surchargés.

Libération des Ressources :

- Le rôle principal du destructeur est de libérer les ressources que l'objet a acquises durant sa durée de vie, comme la mémoire dynamique, les descripteurs de fichiers, ou les connexions réseau.

Ordre d'Appel :

- Les destructeurs sont appelés dans l'ordre inverse de la création des objets. Pour les objets membres d'une classe, leurs destructeurs sont appelés après celui de la classe enveloppante.

Destructeur virtuel :

- Si une classe est destinée à être dérivée, il est souvent nécessaire de déclarer son destructeur comme `virtual` pour assurer que le destructeur de la classe dérivée est appelé lorsque l'objet est détruit via un pointeur de la classe de base.
- **Syntaxe** :

```cpp
class NomDeClasse {
  public:
      ~NomDeClasse() {
          // Corps du destructeur
      }
  };
```

- **Exemple** :

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

      ~Voiture() {
          std::cout << "Destruction de la voiture " << marque << std
      }
};

int main()
{
    Voiture *v1 = new Voiture();
    delete v1; // suppression de l'objet alloué, appel du destructeu

    if (true)
    { // Nouveau bloc
        Voiture v2;
    } // on quitte le bloc, v2 disparait, destructeur appelé

    Voiture v3;
    // Programme terminé, destructeur appelé
}
```

- Exemple avec allocation de mémoire:

```cpp
class AutoFree {
   public:
       int *tableau;

       AutoFree(int size) {
          tableau = new int [size];
       }
       ~AutoFree() {
          delete [] tableau;
       }
 };

 int main() {
     Autofree a = Autofree(100);
 } // la mémoire est automatiquement liberée à la fin du programme
```

Modificateurs de méthodes

Méthodes statiques

Les méthodes statiques sont des fonction membres d'une classe qui ne sont pas liées à une instance de celle ci. Elles permettent d'effectuer des opérations qui ne dépendent pas de l'état d'un objet

```cpp
class MaClasse {
   public:
       static int staticMethod() {
           return 0;
       }
 };

 int main() {
```

```
    int a = MaClasse::staticMethod();
}
```

Méthodes constantes

On peut rajouter le modifier const apres le prototype d'une methode pour indiquer que celle ci ne modifiera pas les variables membres de la classe.

```cpp
class MaClasse {
   public:
        int a;
      int getter() const {
        return a;
      }
      int setter(int a) {
        this->a = a;
      }
};
```

Encapsulation, héritage et polymorphisme

Encapsulation

L'encapsulation consiste à protèger certaines variables ou fonctions membres d'un classe et restreindre leur usage en dehors d'elle même.

Il existe differentes categories d'accès, qui autoriseront ou nous l'accès aux membres sur les objets:

- `public` : Les membres seront accessibles de partout
- `private` : Les membres ne seront accessible que depuis la classe même
- `protected` : Les membres seront accessibles que dans la classes et classes heritées

Afin de rendre disponible l'information ou être en mesure de modifier les membres protégées, nous créerons ce que l'on appele des **getters** et **setters**

```cpp
class Voiture {
  private:
      std::string marque;
      int annee;

  public:
      Voiture(std::string &m, int a): marque(m), annee(a) {}

      int getAnnee() { // getter
          return annee;
      }

      void setAnnee(int a) { // setter
           if(annee > 0) {
              annee = a;
          } else {
              std::cout << "Erreur annee < 0" << std::endl;
            }
      }

      int getMarquee() { // getter
          return annee;
      }

      void setMarque(std::string const &m) { // setter
          marque = m;
      }

};

int main() {
    Voiture v1 = Voiture("Toyota", 2001);
    v1.setAnnee(2010);
    std::cout << v1.getAnnee() << std::endl;
}
```

Heritage

L'héritage est un principe fondamental de la programmation orientée objet (POO) qui permet de créer de nouvelles classes à partir de classes existantes. Cette capacité de dériver une classe de base pour créer une classe dérivée permet de réutiliser du code, de simplifier la maintenance et de promouvoir la modularité.

- **Classe de Base (ou Superclasse)** : La classe dont les propriétés et méthodes sont héritées.
- **Classe Dérivée (ou Sous-classe)** : La classe qui hérite des propriétés et méthodes de la classe de base.

```cpp
class ClasseDeBase {
    public:
    // Membres de la classe de base
};

class ClasseDerivee : public ClasseDeBase {
    public:
    // Membres supplémentaires de la classe dérivée
};
```

Exemple

```cpp
// Classe de base
class Vehicule {
public:
    std::string marque;
    int annee;

    Vehicule(std::string const &m, int a) : marque(m), annee(a) {}

    void afficherDetails() {
        std::cout << "Marque: " << marque << ", Année: " << annee <<
    }
};
```

```
// Classe derivée
class Voiture : public Vehicule {
public:
        // rajout de nouvelles variables de classe
    int nombreDePortes;

    // on n'oublie pas le constructeur de base
        Voiture(std::string m, int a, int portes) : Vehicule(m, a),
        }

    void afficherDetails() {
        Vehicule::afficherDetails(); // Appel de la methode heritée
        std::cout << "Nombre de portes: " << nombreDePortes << std::
    }
};
```

Exemple d'héritage de classes

1. **Classe de Base : Compte**
   - **Attributs** : identifiant, nom, email
   - **Méthodes** : afficherProfil(), modifierEmail()
     - **Classe Dérivée : CompteAdministrateur**
       - **Attributs supplémentaires** : permissions
       - **Méthodes supplémentaires** : gérerUtilisateurs(), afficherLogs()
     - **Classe Dérivée : CompteUtilisateur**
       - **Attributs supplémentaires** : historiqueConnexions
       - **Méthodes supplémentaires** : afficherHistoriqueConnexions()
2. **Classe de Base : SystèmeDeFichier**
   - **Attributs** : cheminRacine, espaceLibre
   - **Méthodes** : créerFichier(), supprimerFichier()
     - **Classe Dérivée : SystèmeDeFichierSécurisé**
       - **Attributs supplémentaires** : niveauChiffrement

- **Méthodes supplémentaires** : chiffrerFichier(), déchiffrerFichier()

3. **Classe de Base : Transaction**

- **Attributs** : montant, date, idTransaction
- **Méthodes** : validerTransaction(), annulerTransaction()

Classe Dérivée : TransactionBancaire

- **Attributs supplémentaires** : numéroCompteBancaire
- **Méthodes supplémentaires** : vérifierSolde(), appliquerFrais()

Classe Dérivée : TransactionCryptographique

- **Attributs supplémentaires** : adresseWallet
- **Méthodes supplémentaires** : vérifierSignature(), confirmerTransaction()

4. **Classe de Base : BaseDeDonnees**

- **Attributs** : urlConnection, connection
- **Méthodes** : connecter(), deconnecter(), insert(), delete()
  - **Classe Dérivée : BaseDeDonneesSQL**
    - **Attributs supplémentaires** : tables
    - **Méthodes supplémentaires** : select(), outerJoin(), innerJoin()
  - **Classe Dérivée : BaseDeDonneesNoSQL**
    - **Attributs supplémentaires** : collections
    - **Méthodes supplémentaires** : filter()

5. **Classe de Base : ChiffrementSymetrique**

- **Attributs** : message
- **Méthodes** : chiffrer(), dechiffrer()

Classe Dérivée : ChiffrementSymetriqueAES

- **Attributs supplémentaires** : keyLength
- **Méthodes supplémentaires** : setKeyLength()

Classe Dérivée : ChiffrementSymetriqueBlowFish

- **Attributs supplémentaires** : pary, sbox
- **Méthodes supplémentaires** :

Polymorphisme

Le polymorphisme est l'un des piliers fondamentaux de la programmation orientée objet (POO). En C++, il permet aux objets de différentes classes dérivées d'être traités comme des objets de la classe de base, facilitant ainsi l'écriture de code plus flexible et extensible. Il existe principalement deux types de polymorphisme en C++ : le polymorphisme statique (ou de compilation) et le polymorphisme dynamique (ou d'exécution).

Statique

Le polymorphisme statique est celui que nous avons vu plus haut pour les surcharges de fonctions. Il est possible de declarer plusieurs méthodes dans une classe qui ont le même nom mais différents arguments.

Le polymorphisme statique est résolu au moment de la compilation. Le choix est fait en fonction des arguments utilisés.

```cpp
#include <iostream>

class Calculateur {
public:
    int ajouter(int a, int b) {
        return a + b;
    }

    double ajouter(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculateur calc;
    std::cout << "Addition de deux entiers : " << calc.ajouter(3, 4)
    std::cout << "Addition de deux doubles : " << calc.ajouter(3.5,
```

```
    return 0;
}
```

Dynamique

Le polymorphisme dynamique est résolu au moment de l'exécution et est généralement implémenté à l'aide de pointeurs ou de références à des classes de base. Il repose sur l'utilisation de fonctions virtuelles.

Une fonction virtuelle est une fonction membre qui peut être redéfinie dans une classe dérivée. Pour qu'une fonction soit virtuelle, il faut rajouter le mot clé `virtual` lors de sa definition.

Lorsqu'une fonction virtuelle est appelée sur un objet via un pointeur ou une référence à la classe de base, la version de la fonction qui est exécutée est déterminée par le type de l'objet réel, et non par le type du pointeur ou de la référence.

Attention, si on place un objet de classe herité dans une variable de type de la classe mère (sans pointeur), les methodes de la classe mère seront appelés

```cpp
#include <iostream>

class Animal
{
public:
    virtual void parler() const
    {
        std::cout << "L'animal fait un bruit." << std::endl;
    }
};

class Chien : public Animal
{
public:
    void parler() const
    {
        std::cout << "Le chien aboie." << std::endl;
```

```
    }
};

class Chat : public Animal
{
public:
    void parler() const
    {
        std::cout << "Le chat miaule." << std::endl;
    }
};

int main()
{
    Animal chienA = Chien();
    Animal *chienB = new Chien();
    Animal *chat = new Chat();

    chienA.parler();    // Animal
    chienB->parler();   // chien
    chat->parler();     // chat

    return 0;
}
```

Classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée et qui est destinée à être une classe de base pour d'autres classes. Elle contient au moins une méthode virtuelle pure. On ne pourra pas l'instancier

Méthode Virtuelle Pure : Une méthode virtuelle pure est déclarée en assignant `0` à la déclaration de la méthode virtuelle dans la classe de base.

cpp

```
#include <iostream>

class Forme {
public:
```

```
    virtual void dessiner() const = 0; // Méthode virtuelle pure
};

class Cercle : public Forme {
public:
    void dessiner() const {
        std::cout << "Dessiner un cercle." << std::endl;
    }
};

class Rectangle : public Forme {
public:
    void dessiner() const {
        std::cout << "Dessiner un rectangle." << std::endl;
    }
};

void afficherForme(const Forme &f) {
    f.dessiner();
}

int main() {
    Cercle cercle;
    Rectangle rectangle;

    afficherForme(cercle);     // Affiche : Dessiner un cercle.
    afficherForme(rectangle);  // Affiche : Dessiner un rectangle.

        Forme f; // Erreur de compilation

    return 0;
}
```

Separation du code

Afin d'éviter d'avoir des fichiers de code trop gros, il est habituel de séparer le code en plusieurs fichiers. Les classes seront en general écrite chacune dans un fichier à part.

Pour pouvoir utiliser des classes, fonctions ou autre déclarées dans d'autres fichiers, il faudra créer des fichiers de header et les inclure là où on les utilise

Contenus des fichiers

cpp

```cpp
// Voiture.hpp

#ifndef _VOITURE_H
#define _VOITURE_H

#include <string>

class Voiture
{
private:
    std::string marque;

public:
    Voiture(const Voiture &v);
    Voiture(std::string m, int a);
    ~Voiture();

    std::string getMarque();
    void setMarque(const std::string &m);
};

#endif
```

cpp

```cpp
// Voiture.cpp

#include <iostream>
#include <string>
#include "Voiture.hpp"

Voiture::Voiture(const Voiture &v) : marque(v.marque), annee(v.annee
{
}
Voiture::Voiture(std::string m, int a)
```

```
{
    marque = m;
    annee = a;
}

Voiture::~Voiture()
{
    std::cout << "Destruction de la voiture " << marque << std::endl
}

std::string Voiture::getMarque()
{
    return marque;
}

void Voiture::setMarque(const std::string &m)
{
    marque = m;
}
```

```c
// main.cpp
#include "Voiture.hpp"

int main()
{
    Voiture v1 = Voiture("Toyota", 2001);
    v2->setMarque("Renault");
}
```

Compilation

Pour compiler un programme qui contient plusieurs fichiers, il suffit d'ajouter tous les fichiers .cpp au compilateur:

`g++ -o executable main.cpp class.cpp`

Makefile

Cet outil permet de faciliter la compilation.

Voici un exemple pour le C++:

```makefile
OUT = main

SRC = *.cpp
CFLAGS = -O -Wall -std=c++17
CC = g++
OBJ = $(SRC:.cpp = .o)

$(OUT): $(OBJ)
    $(CC) $(CFLAGS) -o $(OUT) $(OBJ)

clean:
    rm -f $(OUT) *.o
```

Cette configuration compilera tous les fichiers .cpp dans le dossier courant et créera un executable qui s'appellera `main` Pour compiler, lancer la commande `make`

Exemple de projet tout prêt

makefiles.zip

Compiler et executer avec la commande `make run`

Les flux

Concatenation

Lorsqu'on a besoin de remplir des chaines de caractères avec des valeurs de variables, on peut utiliser `std::streamstring`

```cpp
#include <sstream>

int a = 5;
float b = 12.5;
std::string s = "coucou";

std::stringstream ss;
ss << "a: " << a << " b: " << b << " s: " << s;

std::string final = ss.str(); // -> "a: 5 b: 12.5 s: coucou"
```

Exemple de serialisation/deserialisation

```cpp
    std::stringstream ss;
    std::string a = "bonjour";
    std::string b = "aurevoir";
    ss << a << ':' << b << std::endl;

    std::string concat = ss.str();

    size_t pos = concat.find(':');
    std::string aa = concat.substr(0, pos);   // == bonjour
    std::string bb = concat.substr(pos + 1);  // == aurevoir
```

Fichiers

Pour lire et écrire dans des fichiers, nous utiliserons `std::ifstream` et `std::ofstream`

Lecture

```cpp
#include <iostream>
#include <fstream>
```

```cpp
#include <string>

int main() {
    std::ifstream file("example.txt"); // ouverture de fichier en le
    if (!file.is_open()) {             // verification qu'il s'est b
        std::cerr << "Erreur lors de l'ouverture du fichier." << std
        return 1;
    }

    std::string line;
    while (std::getline(file, line)) {  // lecture ligne par ligne
        std::cout << line << std::endl;
    }

    file.seekg(0, std::ios::beg);       // deplacement du curseur au

    char c;
    while (file.get(c)) {  // lecture lettre par lettre
        std::cout << c;
    }

    file.close();                       // Fermeture du fichier
    return 0;
}
```

Ecriture

```cpp
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("output.txt");  // ouverture de fichier en ec
    if (!file.is_open()) {
        std::cerr << "Erreur lors de l'ouverture du fichier." << std
        return 1;
    }

    file << "Ceci est une ligne de texte." << std::endl; // ecriture
    file << "Ceci est une autre ligne de texte." << std::endl;
```

```
    file.close();
    return 0;
}
```

Attention, par défaut ici on va écraser le contenu du fichier. Tout le contenu precedent sera perdu.

Si on veut rajouter du contenu à la fin, on écrira:

```cpp
std::ofstream file("output.txt", std::ios::app); // append
```

Divers

assert

Si l'on souhaite garantir une condition pendant l'execution de notre programme, on peut utiliser assert. Si la condition est fausse, le programme va s'arreter.

```cpp
#include <cassert>

int a = 5;
int b = 6;
assert(a < b);
```

auto

`auto` n'est pas vraiment un type de variable, mais un mot clé pour que le compilateur décide automatiquement du type de la variable, si on l'initialise au moment de sa déclaration et que son type n'est **pas ambigu**

Nécessaire d'utiliser > C++11 : `g++ -std=c++11`

```cpp
// ok
int varInt = 5;
```

```
auto varAuto = varInt;

// pas ok
auto varAuto2;
if(a < b) {
    varAuto2 = 5.5;
} else {
    varAuto2 = true;
}
```

Enum

Surcharge d’opérateurs

Consignes de rendu

Retourner les exercices dans une archive **.zip** contenant **un fichier par exercice**

Merci de nommer votre archive comme ceci :

`NOM_Prénom_groupe_seance.zip`

Exemple

`Boisson_Pacien_G1_S2.zip`

Remise à niveau en C

Bases

Pour les élèves souhaitant revoir les bases:

- Exercices C

Puissance

1- Créer une fonction Puissance, qui prend en entrée 2 entiers et retourne le premier nombre puissance le deuxième. Exemple : 2^3 = 222 = 8 2- Utiliser cette fonction et afficher le résultat dans la fonction main.

Guess my number

Vous allez creer un petit jeu dans lequel vous devrez deviner un nombre.

Au début, le jeu choisit un nombre aléatoire, et vous proposera de le deviner. Si vous trouvez le bon chiffre, vous gagnez. Si vous vous trompez, le jeu vous dit si son nombre est plus grand ou plus petit et vous redemande. On compte le

nombre de fois ou vous vous trompez, et le but est de deviner avec le moins d'essais possibles. On affichera le score a la fin.

La fonction rand() déjà utilisée ici retourne un entier aléatoire.

Partir de ce code de base, qui part d'un nombre aléatoire dejà generé

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    srand(time(0)); // initialisation de l'aleatoire
    int randomNumber = rand() % 100; // Generation d'un nombre aleat
      // ...
}
```

Avancé

Pour tout le monde, ces exercices serviront pour la suite:

Tableaux dynamiques

Créer un programme qui gère des tableaux de taille dynamique, c'est à dire dont la taille pourra changer au fil du temps, contrairement aux tableaux traditionnels.

Concevoir des structures et des fonctions qui seront utilisées dans le main.

Avec à ces tableaux dynamiques, nous pourrons:

- Créer un nouveau tableau dynamique vide
- Ajouter un element a la fin du tableau
- Afficher tous les elements d'un tableau
- Récuperer un element du tableau via son index

- Supprimer le dernier element d'un tableau
- Copier un tableau dynamique
- Supprimer un element d'un tableau via son index

Listes chainées

[Cours langage C sur les listes chainées](../c-avance/ langage-c-avance#listes-chainees)

Intro au C++

Compilation

S'assurer que son environnement de développement fonctionne, et compiler et executer le code suivant:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "Hello world !" << std::endl;

    return 0;
}
```

Entrées/Sorties

- Créer des variables de différents types et afficher leurs valeurs dans la console avec `std::cout`
  - `int`
  - `float`
  - `char`

- Demander à l'utilisateur de rentrer des valeurs pour les affecter à ces différents types de variables avec `std::cin`

Les pointeurs

References

Ecrire une fonction qui inverse les données de deux variables avec l'utilisation de variables passées en reference (`&`)

Allocations

Demander à l'utilisateur une taille et allouer un tableau d'entier de cette taille avec `new` puis liberer la mémoire pour ce tableau

Conteneurs

Array

- Créer un tableau statique d'entiers avec `std::array`, et demander a l'utilisateur de remplir les valeurs pour chacun de ses elements.
- Copier ce tableau, modifier les valeurs d'un des deux tableaux et s'assurer que les deux tableaux sont bien différents.

String

- Créer une chaine de caractère initialisée à `"ZDJSIJ2393D"`
- Supprimer les deux derniers caractères
- Ajouter le caractère `'Y'` à la fin
- Supprimer tous les caractères
- Ajouter le caractère `'B'`
- Afficher la chaine de caractère

Vector

- Creer un `vector` de `string`

- En boucle:
  - Demander à l'utilisateur d'entrer une phrase
  - Si la phrase rentrée est `exit`
    - Quitter la boucle
  - Sinon si la phrase rentrée est `cancel`
    - Supprimer la dernière phrase
  - Sinon
    - Ajouter cette phrase au `vector`
  - Afficher toutes les phrases

Itérateurs

- Reprendre l'exercice precedents sur les Vector
- Utiliser des itérateurs pour afficher toutes les phrases plutot qu'utiliser les index
- Si l'utilisateur rentre la phrase `remove`
  - Demander l'index de la phrase a supprimer
  - Supprimer cette phrase
- Si l'utilisateur rentre la phrase `reverse`
  - Inverser l'ordre phrase
- Si l'utilisateur rentre la phrase `sort`
  - Trier les phrases par ordre alphabetique

Bibliothèque standard

Réimplementer certaines fonctions fournies par la librairie standard:

- `find(iterator_debut, iterator_fin, search)`
- `count(iterator_debut, iterator_fin, search)`
- `compare(string1, string2)`

Polymorphisme

Surcharge

Créer deux fonctions `carre` qui retourneront le carré d'un nombre. L'une d'elle prendra et retournera des entiers, l'autre des nombres flottants.

Classes

Bases

Films

Créer une classe `Film` qui contiendra

- Le nom du film
- Le nom du réalisateur
- Une note / 10
- Le nombre d'entrées au cinema
- Une méthode `estBien` qui dira si le film est *bien* ou pas
  - Un film est *bien* si il a une note supérieure à 8 et plus de 1000 entrées au cinema

Eleve

Créer une classe `Eleve` qui contiendra

- Le nom de l'élève
- Le nom de la classe
- Un tableau dynamique de notes (utiliser `vector`)
- Une methode pour ajouter une note
- Une methode pour récuperer sa note moyenne

Joueur

Créer une classe `Joueur` qui représentera un personnage dans un jeu video.

La classe doit posséder:

- Un nom
- Un niveau
- Un nombre de points de vie
- Un nombre de points d'experience acquis

Il doit être possible de:

- Lire et Changer son nom
- Lire le niveau
- Lire les points de vie

Il ne doit pas être possible de:

- Changer directement son niveau
- Changer directement ses points d'experience
- Changer directement ses points de vie

Puis:

- Implémenter une methode `attaquer` , qui prendra un autre joueur en paramètre et qui:
  - réduira les points de vie de l'ennemi
  - Si le joueur ennemi meurt (vie < 0), ajouter des points d'experience. Si on dépasse un seuil d'xp, on monte de niveau
  -

Joueurs avec heritage

- Créer deux classes héritées de `Joueur`
  - `Soldat`
    - Possède des points d'endurance

- `Magicien`
  - Possède des points de magie
- Modifier la fonction `attaquer` dans chacune de ces classes pour qu'elle enlève les points d'energie correspondant à la classe du joueur

Pare-feu

Nous allons ébaucher un programme de pare feu, qui servira a analyser du traffic réseau et autoriser ou refuser des communications.

Pour commencer, on va créer un simple filtre sur des numéros de ports à autoriser ou non.

Il devra être possible de:

- Ajouter un port autorisé
- Supprimer un port autorisé
- Analyser un port
- Activer/Désactiver le pare-feu
  - Quand le pare feu est désactivé, il laisse passer tout le traffic
  - Par defaut, il est desactivé
  - Par défaut, tous les ports sont interdits

Utiliser `assert` pour tester la classe et verifier son fonctionnement, par exemple:

```cpp
#include <cassert>
#include "Firewall.hpp"

int main()
{
    Firewall firewall;

    assert(!firewall.check(21));
    firewall.allow(21);
    assert(firewall.check(21));
```

```
    return 0;
}
```

Projet

Annuaire

Créer un programme qui gère une annuaire de numéro de telephone sous forme nom / numéro de telephone.

S'inspirer de l'exercice vecteurs/itérateurs ou nous stockions des phrases, en gardant le système de commandes (ajouter, supprimer, lister, ...)

Partir de ce projet:

annuaire-template.zip

Compiler avec la commande suivante:

```
g++ -O -Wall -std=c++17 Annuaire.cpp main.cpp -o ./main
```

Au lieu d'utiliser des `vector`, utiliser des `maps` pour stocker chaque entrée du repertoire par nom → numéro de telephone

Créer une classe Annuaire, qui s'occupera de la gestion de celui ci.

Cette classe devra posseder:

- Une methode `add` pour ajouter une entrée dans l'annuaire
- Une methode `remove` pour supprimer une entrée par nom
- Une methode `exists` pour verifier si une entrée existe
- une methode `print` pour afficher tout l'annuaire
- Un constructeur de copie
- Tout autre methode qui vous semblera utile

Les variables contenant les données de l'annuaire ne doivent pas être modifiables en dehors de la classe.

Flux

Concatenation

- Demander à l'utilisateur de rentrer deux phrases
- Créer une chaine de caractère qui contient ces deux phrases bout à bout, séparées par un `:`

Utiliser `std::stringstream`

- Séparer cette chaine de caractère en deux

Fichiers

- Reprendre l'exercice Annuaire
- Ajouter une commande `save`, qui va enregistrer dans un fichier toutes les contacts qui ont été ajoutés
- Au démarrage du programme, charger les contacts depuis ce fichier texte

[Projet Final] Gestionnaire de mot de passes

Description

Créer un programme qui s'occupe de la gestion de mots de passes.

A la manière d'un Keepass, ce programme pourra stocker les mots de passe de l'utilisateur sur differentes plateformes

Chaque entrée sera composé de:

- Nom de la plateforme (google, discord, instagram,...)
  - Celui ci servira de clé dans l'annuaire
- Nom d'utilisateur (mail/username)

- Mot de passe
- Plus si vous le souhaitez (2FA, commentaires, ...)

Les mots de passes devront êtres sauvegardés dans un fichier et chargés à l'ouverture du programme

Implementation

Reprendre le code du programme d'annuaire telephonique, qui lui même aura été démarré sur le template proposé ici: Partir de ce projet:

Modifier celui ci pour les besoins de ce projet

- Ecrire le code du gestionnaire de mots de passe dans la classe `Annuaire` qui pourra être renommée si besoin
- Ecrire des tests dans la fonction `test()` pour verifier que l'annuaire de mot de passes fonctionne bien
- Ecrire une interface utilisateur dans la fonction `ui()` qui s'occupera de demander des actions à l'utilisateur (ajouter/supprimer/lister entrées, ...)

Chiffrement

Pour améliorer la sécurité, il serait preferable que le fichier ou sont stockés les mots de passe soit chiffré.

Lors du chargement à partir du fichier ou de l'enregistrement, utiliser une librairie de chiffrement symetrique. Quand le programme démarre, demander le mot de passe à l'utilisateur

Telecharger cette archive, ajouter les fichiers à votre projet et utiliser ces fonctions pour le chiffrement

AES.zip

```cpp
#include "AES.hpp"

EasyAES aes;
```

```
std::string message = "phrase secrete";
std::string key = "password";
std::string cipher = aes.encrypt(message, key);
std::string decipher = aes.decrypt(cipher, key);
assert(message == decipher);
```

Fonctionnalités supplementaires

- Verifier securité mot de passes (nb de caractère, caractères speciaux,...)
- Comparer mots de passes avec les plus utilisés
  - https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/
- Génerateur de mot de passe

Rendu

Le projet complet sera a rendre sous la forme d'une archive zip

Retourner le projet complet dans une archive **.zip**

Merci de nommer votre archive comme ceci :

`NOM_Prénom_groupe.zip`

Exemple

`Boisson_Pacien_G1.zip`