Intro au C++

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

Le langage C++ est un successeur du Langage C auquel il apporte de nombreuses nouveautés qui vont permettre d'aller plus loin dans la complexité des programmes et simplifier le développement, notamment avec la **programmation objet** et la **librairie standard**.

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 et permettront le developpement de programme plus poussés.

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/

Autre ressource de cours en français:

https://zestedesavoir.com/tutoriels/822/la-programmation-en-c-moderne

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

Pour l'environnement de travail, vous pouvez suivre la même méthode d'installation que pour le C

Utilisation du C++

Systèmes d'exploitation et noyaux (OS & Kernels) :

- Microsoft Windows : Noyaux et composants critiques
- MacOS : Noyau Darwin
- Linux : Pilotes et composants user-space

Compilateurs et outils de développement :

- GCC
- Clang/LLVM

Environnements de développement intégrés (IDE) :

- Microsoft Visual Studio (pour le backend)

Bases de données :

- MySQL
- MongoDB

Applications de bureau complexes :

- Microsoft Office (Word, Excel, PowerPoint)
- Adobe Creative Suite : Photoshop, Illustrator, Premiere Pro, After Effects

Intelligence Artificielle et Machine Learning :

- TensorFlow, PyTorch : Interfaces Python sur coeurs en C++

- OpenCV

Systèmes embarqués et IoT (Internet des Objets) :

Voitures connectées, drones, appareils médicaux, microcontrôleurs

Finance quantitative et trading haute fréquence :

Algorithmes en temps réél

Graphismes 3D, réalité virtuelle/augmentée :

- Bibliothèques graphiques (OpenGL, Vulkan, DirectX)
- Moteurs de jeu (Unreal Engine, CryEngine)

Librairies C++ populaires:

- Boost: Calculs mathématiques avancés, gestion des fichiers, programmation réseau, etc.
- OpenGL/DirectX pour les graphismes 3D
- Qt: Développement d'interfaces graphiques

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 le standard 17 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 Mac/Linux, ne pas rajouter `.exe` à l'executable

Pour executer:

`./executable.exe`

Versions

Il existe differentes versions de C++, et par défaut **gcc** utilise le standard **17**

Les versions les plus recentes incluent toutes les modifications précedentes (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` `c++20` `c++23`)

Affichage dans la console

Afin de faciliter la compréhension du début du cours, nous allons voir basiquement comment afficher du texte dans la console.

Avant, en C, nous utilisitons `printf`:

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    int var = 5;
    printf("var: %d\n", var);
}
```

Maintenant, en C++, nous allons utiliser `std::cout`.

L'opérateur `<<` permet de concatener des elements à afficher et `std::endl` permet d'aller à la ligne.

```cpp
#include <iostream> // inclure la librairie d'entrée/sortie

int main() {
    std::cout << "Hello World!" << std::endl;
    int var = 4;
    std::cout << "var: " << var << std::endl;
    return 0;
}
```

Le fonctionnement de `std::cout` sera expliqué en détail dans la suite du cours.

Projet de départ

Un exemple de projet C++ de base est telechargeable ici

Vous devez avoir un compilateur et l'outil Makefile installé.

Pour compiler:

```bash
make
```

Pour lancer le programme:

```bash
make run
```

Nouveaux types de variables

Le C++ apporte de nouveaux types de variables qui faciliteront le travail pour les développeurs.

bool

Permet de représenter des 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
```

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 au moins la version C++11 : `g++ -std=c++11`

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

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

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` .

Ces conteneurs ont pour rôle de stocker des ensemble d'elements, de manière differentes, mais en offrant des outils similaires pour tous.

Types

- Array
- String
- Vector
- Map
- Set

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;
}
```

Vous pouvez répérer une nouvelle syntaxe dans la déclaration de ce nouveau type de variable: `<int, 5>`. Cela veut dire que le type `std::array` est un **generique**. Nous verrons ce que cela veut dire précisement plus tard, mais dans ce cas précis, nous indiquons que ce tableau contiendra des variables de type `int` et en contiendra exactement `5`, de manière similaire à la syntaxe `int xxx[5]`

- 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)
```

Nous ne pouvions pas connaitre la taille d'un tableau en C !

- 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.";

        phrase = "Voici une autre phrase.";

    return 0;
}
```

Leur fonctionnement de base est similaire aux Array, avec des possibilités en plus.

Dans l'exemple precedent, un amateur du C remarquera quelque chose d'étrange: en effet, en C, lorsqu'une chaine de caractère a été définit, sa taille

est fixe et ne peut plus changer. Ici, on voit que la deuxieme chaine de caractère est plus longue que la première, et cela fonctionne quand même.

C'est un des gros avantage des conteneurs: il effectuent automatiquement des opérations essentielle, comme ici redimensionner la taille du tableau de caractères sous-jacent.

- **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"
```

```cpp
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`.

Ici, on voit qu'il n'est plus necessaire d'indiquer la taille du tableau dans sa déclaration, uniquement son type avec `<int>`. A leurs création, les vecteurs

seront vides, (de taille nulle), et on pourra rajouter ou supprimer des elements quand on le souhaite.

- **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;
}
```

```cpp
// 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`. Cependant, certains types de conteneurs (que nous verrons bientôt) n'auront pas d'index composé d'entiers numériques successifs.

Pour parcourir les éléments de ces conteneurs, il a donc été inventé un nouveau type qu'on appelle **itérateur**.

Utilisation

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 pointant sur le premier élément d'un conteneur, on utilise `std::begin()` ou juste `iterateur.begin()`

```cpp
std::vector<int> tableau = { 11, 22, 33, 44 };
std::vector<int>::iterator tableau_iterateur = std::begin(tableau);
std::vector<int>::iterator tableau_iterateur_bis = tableau.begin();

// pour les conteneurs constants
```

```cpp
std::vector<float> const tableau_constant = { 50, 51, 52, 53 } ;
std::vector<float>::const_iterator tableau_contant_iterateur = std::
```

Un itérateur permet de parcourir tous les élements d'un conteneur, et pour cela il va prendre la forme d'un pointeur vers un de ces éléments. Nous ferons avancer ce pointeur vers les éléménts suivants au fur et à mesure que nous avançons dans le parcours.

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

```cpp
std::cout << *tableau_iterateur << std::endl; // 11
```

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

```cpp
std::cout << *tableau_iterateur << std::endl; // 11
tableau_iterateur++;
std::cout << *tableau_iterateur << std::endl; // 22
tableau_iterateur--;
std::cout << *tableau_iterateur << std::endl; // 11
tableau_iterateur += 2;
std::cout << *tableau_iterateur << std::endl; // 33
```

Fin d'un itérateur

`std::end()` est une valeur spéciale que peut prendre un itérateur, qui indique que le nous avons parcouru tous les éléments du conteneur.

Celui-ci est également similaire à un pointeur, mais il ne pointera vers aucun élément du conteneur, plutot sur un element virtuel inexistant, **après** le dernier element.

Attention, il ne faut surtout pas déréferencer un tel itérateur, sinon le programme lancera une runtime exception et s'arretera.

```cpp
std::vector<int> tableau = { 11, 22, 33, 44 };
std::vector<int>::iterator tableau_iterateur = std::end(tableau);
```

```cpp
std::vector<int>::iterator tableau_iterateur_bis = tableau.end();

tableau_iterateur--;
std::cout << *tableau_iterateur << std::endl; // 53
```

Exemple de parcours

```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;
}
```

Exemple raccourci

Il existe une manière encore plus simple d'itérer sur les conteneurs, mais sur laquelle on a moins le contrôle.

```cpp
void print_container(const std::vector<int>& container)
{
    for (int value : container) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

Si la syntaxe est encore floue, continuez le cours et revenez ici plus tard.

Utilisation des itérateurs dans les containers

Certaines fonctions proposées par les conteneurs pour les manipuler ne peuvent pas prendre d'index numériques comme parametres, mais doivent recevoir des itérateurs vers des posiitons.

- Exemple: **Supprimer** un élément dans un `vector` 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 troisiem
nombres.erase(std::begin(nombres), std::begin(nombres) + 2);
```

Algorithms

La librairie standard offre un ensemble de fonctions qui se trouvent très utile pour utiliser les conteneurs.

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

- **compter** avec `count`

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

- **trouver** avec `find`

```cpp
std::string const phrase { "Exemple avec plusieurs mots." };

// On obtient un itérateur pointant sur le premier espace trouvé.
```

```cpp
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 et les caractères. Cette fonction modifiera le container donné 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, qui eux ne possèdent pas forcement d'index numérique incrémental.

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

Set

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 };
```

Map

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

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;
```

```cpp
}

// 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

Map et Set sont des types appellés “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.

Resumé des differents conteneurs

	Type d'elements	Taille variable	Elements uniques
array	*	non	non
vector	*	oui	non
string	char	oui	non
set	*	oui	oui

	Type d'elements	Taille variable	Elements uniques
map	* > *	oui	non

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
```

Polymorphisme

Le polymorphisme est un concept de la programmation orientée objet qui permet à utiliser des outils similaires sur différents types de données.

On peut distinguer le polymorphisme statique, qui est résolu à la compilation, et le polymorphisme dynamique, qui est résolu à l'exécution.

Templates

Les templates sont des fonctions ou structure dont des variables ont des types qui peuvent changer.

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

template <typename T>
T addition(T a, T b)
```

```
{
    return a + b;
}

template<class T>
struct Coordinates
{
    T x;
    T y;
};

int main()
{
    float c = addition<float>(1.2, 2.3);
    int d = addition<float>(2, 3);
    std::cout << c << std::endl;
    std::cout << d << std::endl;

    struct Coordinates<int> c1;
    c1.x = 10;
    struct Coordinates<float> c2;
    c2.x = 10.5;
}
```

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, laquelle appeler.

Les paramètres des prototypes des differentes fonction surchargées doivent être differentes pour qu'il puisse choisir. Le choix pour le compilateur **ne doit pas être ambigü**

L'exemple suivant ne fonctionne pas:

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

fonction(2); // Laquelle des deux choisir ?
```

Surcharge d'opérateurs

Prochainement...

Les flux

De nouveaux opérateurs font leur entrée, appelés flux (stream), représentés par les symboles `<<` et `>>`.

Ils permettent de gérer un flux de données entre des entités, et la donnée se dirige dans le sens des flèches.

`a << b;` : `b` va vers `a`

`a >> b;` : `a` va vers `b`

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

Pour afficher des messages dans la console, on utilisera la **sortie standard**. Voici comment fonctionne la sortie avec `std::cout`

```cpp
#include <iostream>

int main()
{

    // On peut afficher une chaine de caractère
    std::cout << "Bonjour" << std::endl;
    // printf("Bonjour\n");
```

```
    // Ou bien un caractère seul.
    std::cout << 'A' << std::endl;

    // Ou bien un chiffre.
    std::cout << 7 << 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`)

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

Entrées

Pour les entrées nous utiliserons l'**entrée standard** avec `std::cin` en changeant le sens des crochets `>>`

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
#include <iostream>

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

    return 0;
}
```

Pour savoir dans quel sens placer les flèches, il suffit de se demander dans quel sens va l'information:

- `std::cout << "hello";` : "hello" va vers `cout` (sortie)
- `std::cin >> variable;` : `cin` (entrée) va vers `variable`

Fichiers

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

Lecture

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

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

```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;
}
```

Les pointeurs

Rappel sur les pointeurs

Les pointeurs sont des outils qui nous permettent de savoir où est stockée une donnée dans la mémoire, et de pouvoir modifier des variables en dehors du scope dans lequel elles sont définies.

Les references

En C++ est introduite la notion de **reference**. Le concept est similaire à celui des pointeurs, à la difference près qu'une reference s'utilise de la même manière qu'une variable normale, tout en restant liée à une autre variable.

Une réference est définie avec le symbole `&` lors de la déclaration, contrairement aux pointeurs qui sont définis avec `*`.

Lors de l'utilisation, pas besoin d'utiliser `*` pour accéder à la valeur pointée, ni à `&` pour obtenir l'adresse.

```cpp
int variable = 24;
int &variable_ref = variable;
// variable et variable_ref représentent éxactement la même chose et

variable_ref = 42;
std::cout << variable << std::endl; // 42
std::cout << variable_ref << std::endl; // 42
```

Passage par référence

Dans les fonctions, on peut passer des arguments par référence, en utilisant le symbole `&` dans la déclaration de la fonction. Cela signifie que la fonction

pourra modifier la variable passée en argument, et aussi que celle-ci ne sera pas copiée (gain de performance).

En effet, quand on passe une variable par valeur à une fonction (pas de * ou &), celle-ci est copiée dans une nouvelle variable locale à la fonction. Si la variable est grosse (comme un tableau ou une classe), cela peut être couteux en performance. Le fait de passer un pointeur ou une reference évite cette copie et améliore la vitesse d'éxecution et la consommation mémoire.

```cpp
#include <iostream>

void fonctionParRef(int &reference)
{
    reference = 10;
}

int main()
{
    int variable = 24;

    fonctionParRef(variable);
    std::cout << variable << std::endl; // 10

    return 0;
}
```

Cette syntaxe est un peu plus simple à écrire et utiliser que les pointeurs, mais elle ne convient pas dans tous les cas.

Petit rappel de l'équivalent avec des pointeurs:

```cpp
#include <iostream>

void fonctionParPointeur(int *pointeur)
{
    *pointeur = 10;
}
```

```
int main()
{
    int variable = 24;

    fonctionParPointeur(&variable);
    std::cout << variable << std::endl; // 10

    return 0;
}
```

Lorsqu'on veut bénéficier du gain de performances, mais qu'on ne souhaite pas qu'une fonction puisse modifier les variables passées en arguments, on peut les passer en **réference constante**.

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

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);
}
```

Types de pointeurs et références constantes

```cpp
std::string a = "hello";

// s est une reference non modifiable vers une variable modififable
const std::string &s1 = a;

s1 = "hello"; // Pas possible de le modifier

// s est un pointeur non modifiable vers une variable modifiable
std::string *const s3 = &a;

s3 = &b;      // Pas possible de modifier le pointeur
*s3 = "hehe"; // Ok de changer la valeur

// s est un pointeur non modifiable vers une variable non modifiable
const std::string *const s4 = &a;

s4 = &b;      // Pas possible de modifier le pointeur
*s4 = "hehe"; // Pas possible de modifier la valeur
```

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
```

La Programmation Orientée Objet

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 représentent 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**).

On peut voir les classes comme des structures améliorées, qui contiennent, en plus des données, de la logique sous forme de méthodes.

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
      }
};
```

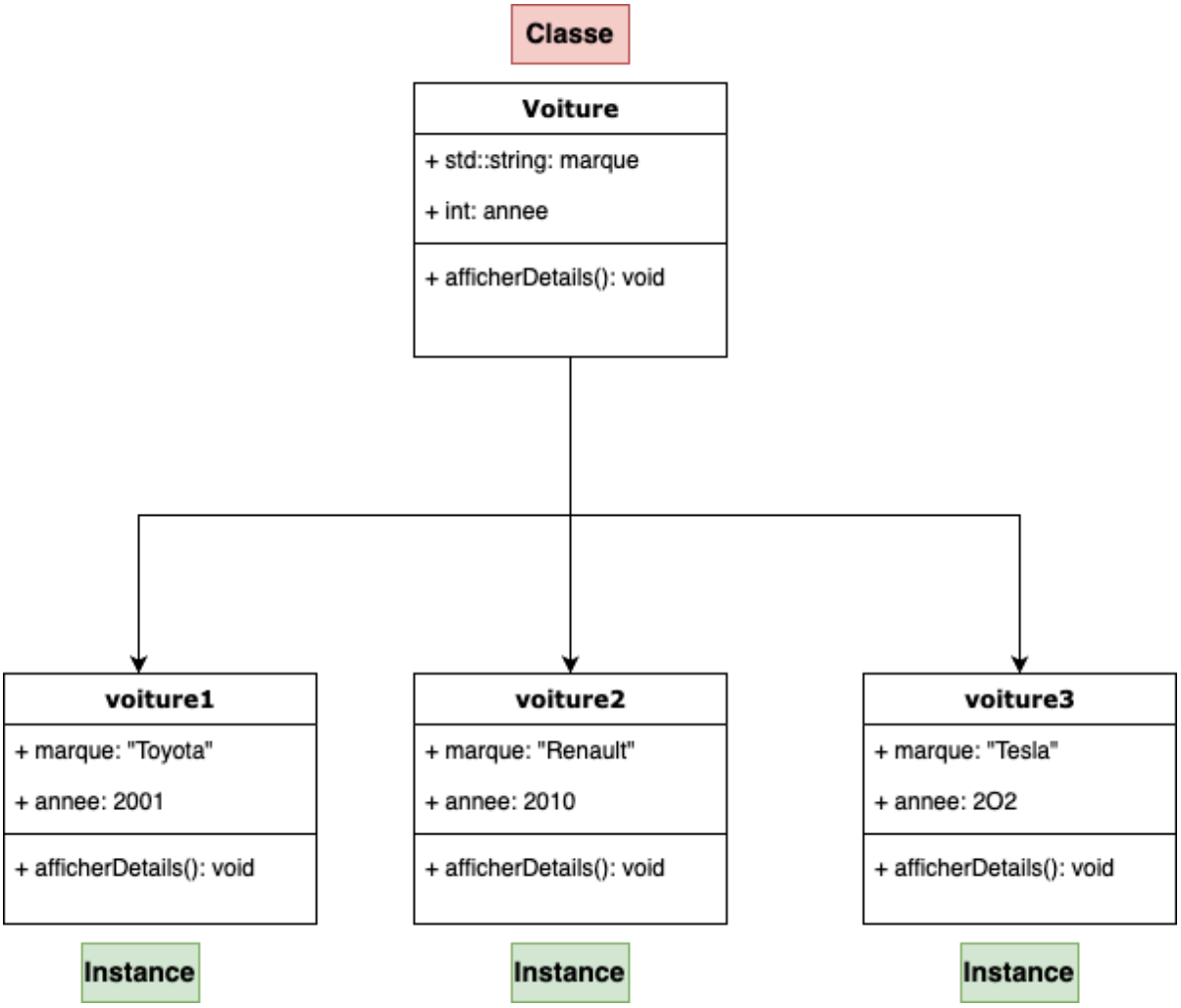
Création d'un Objet

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;              // Instantiation d'une classe
maVoiture.marque = "Toyota";    // Modification d'un attribut
maVoiture.annee = 2022;         // Modification d'un attribut
maVoiture.afficherDetails();    // Appel d'une méthode
```



Difference entre structures et classes

- Avec structures:

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

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

```cpp
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 aux membres d'une classe à l'intérieur d'elle même, 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).
- Cela correspond au concept de polymorphisme statique vu précedemment, où une fonction peut avoir plusieur forme, et le compilateur choisit la bonne en fonction des arguments passés.

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) {}

    // ou

    MaClasse(int n): nombre(n) {}

};
```

- 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
```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
```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 ne seront accessibles que dans la classes et classes heritées

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

Cela peut permettre d'éviter des incohérences dans les données à cause d'une mauvaise utilisation de la classe, ou empecher un comportement anormal.

cpp
```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;
              }
        }

        // Possible de récupérer la marque
        int getMarquee() { // getter
            return annee;
        }

        // Impossible de modifier la marque, pas de setter
};

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 d'attributs supplémentaires
    int nombreDePortes;

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

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

```
    }
};
```

Exemples 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()
  - **Classe Dérivée : SystèmeDeFichierDistant**
    - **Attributs supplémentaires** : urlServeur, tokenAccès
    - **Méthodes supplémentaires** : connexion(), testerDebit()

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.

Cette technique est souvent utilisée pour manipuler differents objets dont l'implementation peut etre differente, mais dont l'utilisation est la même (on peut accelerer et freiner avec une voiture, sans avoir besoin de savoir si le moteur est diesel ou essence)

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;
    }
};
```

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

int main()
{
    Animal chienA = Chien();        // Chien a été tronqué en Animal
    Animal *chienB = new Chien();
    Animal *chat = new Chat();

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


    // References similaires aux pointeurs
    Animal &refChien = chienA;
    refChien.parler();  // chien

    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. Elle sert à définir une interface commune pour les classes dérivées.

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;
}
```

Organisation 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`

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);
```

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"
```

Remise à niveau en C

C

Pour les élèves souhaitant se remettre à niveau en C, vous pouvez faire les exercices de C

- Exercices C
- Exercices C avancé

Puissance

- Créer une fonction Puissance, qui prend en entrée 2 entiers et retourne le premier nombre puissance le deuxième. Exemple : `2^3 = 2**2**2 = 8`
- 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()` proposée ici retourne un entier aléatoire.

Partir de ce code de base:

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

int main(void)
{
    srand(time(0));                          // initialisation de l'a
    int randomNumber = rand() % 100;         // Generation d'un nombr
    printf("nombre: %d\n", randomNumber); // A supprimer plus tard p
}
```

Avancé

Pour les curieux, avant d'utiliser les nouveaux types de la librairie standard, vous pouvez vous entrainer à implémenter les équivalents en C.

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

Implémenter un système de listes chainées

Reflexion

Comparer les avantages et inconvénients des tableaux dynamiques et des listes chainées. Etudier les differences de performances et de coûts en calculs/mémoire entre ces deux méthodes et expliquer pourquoi privilegier l'une ou l'autre dans certains cas.

Reflechir à des alternatives possibles qui amélioreraient les performances.

Intro au C++

Compilation

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

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

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

    return 0;
}
```

Entrées/Sorties

Remplacer l'utilisation de `printf` et `scanf` par `std::cout` et `std::cin`

- 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`

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.
- Créer une fonction qui prend en paramètre un tableau `std::array` et qui affiche ses valeurs. Cette fonction doit être optimisée et ne doit pas pouvoir modifier les valeurs du tableau.

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

- Créer une liste dynamique de chaines de caractères avec `std::vector`.
- Ajouter quatre chaines de caractères à celle-ci
- Afficher toutes les chaines de caractères et la taille de la liste
- Supprimer la dernière chaine de caractères
- Vider toute la liste

Itérateurs

Reprendre l'exercice precedent sur les vecteurs de chaines de caractères.

- Pour afficher la liste de chaines de caractères, utiliser des itérateurs pour parcourir la liste au lieu d'un simple `for` avec un index.

Avant de vider la liste:

- Inverser la liste

- Supprimer le deuxieme element de la liste
- Trier la liste par ordre alphabétique

Algorithmes

Search string

Ecrire une fonction qui prend deux chaines de caractères en entrée et retourne vrai si la première contient la deuxième (utiliser une fonction existante de la librairie standard)

Fait maison

Réimplementer certaines fonctions fournies par la librairie standard:

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

Vous pouvez vous référer à la documentation officielle pour connaitre le fonctionnement de ces fonctions:
https://en.cppreference.com/w/cpp/algorithm

Les pointeurs

References

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

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

---

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.

Templates

Créer une fonction générique `max` qui prendra deux variables et qui retournera la plus grande des deux. Le type de ces variables sera défini lors de l'appel de la fonction.

Programmation Orientée Objet

Exemples de classe

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 calculer la moyenne de l'élève

Classe

Créer une classe `Classe` qui contiendra une liste dynamique d'éléves.

- Créer une méthode `moyenne()` qui retournera la moyenne de la classe

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

*(**Indice**: utiliser la portée des attributs)*

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 héritage

- Créer deux classes héritées de `Joueur`
  - `Guerrier`
    - 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

---

Separation des fichiers

Pour l'exercice "Joueurs avec héritage", organiser le code de manière à ce que chaque classe soit dans un fichier séparé, avec les headers et l'implémentation dans deux fichiers distincts.

Ecrire la commande necessaire pour compiler le programme dans un fichier readme à coté du code.

Projet

Dans le but de construire un programme utile et fonctionnel, nous allons d'abord avancer étape par étape par mini-projets en rajoutant des fonctionnalités au fur et à mesure.

CLI interactive

Pour les éxercices suivants, vous allez commencer par créer une programme qui proposera une interface console interactive (CLI). Celle-ci proposera differentes actions et demandera à l'utilisateur laquelle effectuer.

Pour cela, commencer par afficher une liste de commandes disponibles avec une description pour chacune d'elle, et effectuer l'action demandée par l'utilisateur. Une fois effectuée, re-afficher la liste des commandes et continuer ainsi jusqu'à ce que l'utilisateur demande à quitter le programme.

Actions de départ:

- `help` : Afficher la liste des commandes
- `exit` : Quitter le programme
- `hello` : Afficher `Hello World!`

Liste de phrases

Ajouter au programme précedent liste de chaine de caractères dynamique.

Ajouter les commandes suivantes:

- `add` : Demander une phrase à l'utilisateur et l'ajouter à la liste
- `show` : Afficher toutes les phrases de la liste avec leur index
- `pop` : Supprimer la dernière phrase de la liste
- `clear` : Vider la liste

Itérateurs

A partir de l'exercice précedent:

- Pour la commande `show`, utiliser des itérateurs pour afficher toutes les phrases plutot que boucler traditionellement sur les index.

Ajouter les commandes suivantes:

- `remove` : supprimer une des chaines de caractère
  - Demander l'index de la phrase a supprimer (l'index des phrases doit être affiché quand on affiche la liste)
  - Supprimer cette phrase
- `reverse` : Inverser l'ordre phrase
- `sort` : Trier les phrases par ordre alphabetique
- `search` (bonus) : Chercher une chaine de caractère dans la liste

---

Annuaire télephonique

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

Partir de ce projet:

annuaire-template.zip

Compiler avec la commande suivante:

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

Ou, si vous avez Makefile installé, simplement `make` pour compiler et `make run` pour lancer.

Reprendre le travail précedent sur la CLI, Vecteurs et Itérateurs pour utiliser l'annuaire. Au lieu d'utiliser des `vector`, utiliser des `maps` pour stocker chaque entrée du repertoire par nom → numéro de telephone

Conserver le code de la CLI dans le ficher principale `main.cpp` (ou le mettre dans un fichier séparé si vous le souhaitez), mais écrire tout le code de gestion de l'annuaire dans une classe `Annuaire`.

Cette classe devra posseder:

- Une methode `add` pour ajouter une entrée dans l'annuaire
- Une methode `get` pour récupérer un numéro de téléphone par nom
- 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 (attribut privé).

Au fur et à mesure que les fonctionnalités sont implémentées, vous pourrez décommenter les `assert` pour verifier que votre programme fonctionne comme demandé et en rajouter de nouveaux si besoin.

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

Dupliquer le code du programme d'annuaire télephonique, 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 la complexité des mot de passes (nombre de caractères, caractères speciaux, ...)
- Comparer les mots de passes avec les plus utilisés
  - https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/
- Génerateur de mot de passe
- Créer une classe dédiée au stockage des données. Commencer par créer une classe abstraite qui définira le contrat de stockage quelque soit le

support, puis implementer une ou plusieurs classes de stockage (fichier, base de données, s3, google drive, ...)

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;
}
```