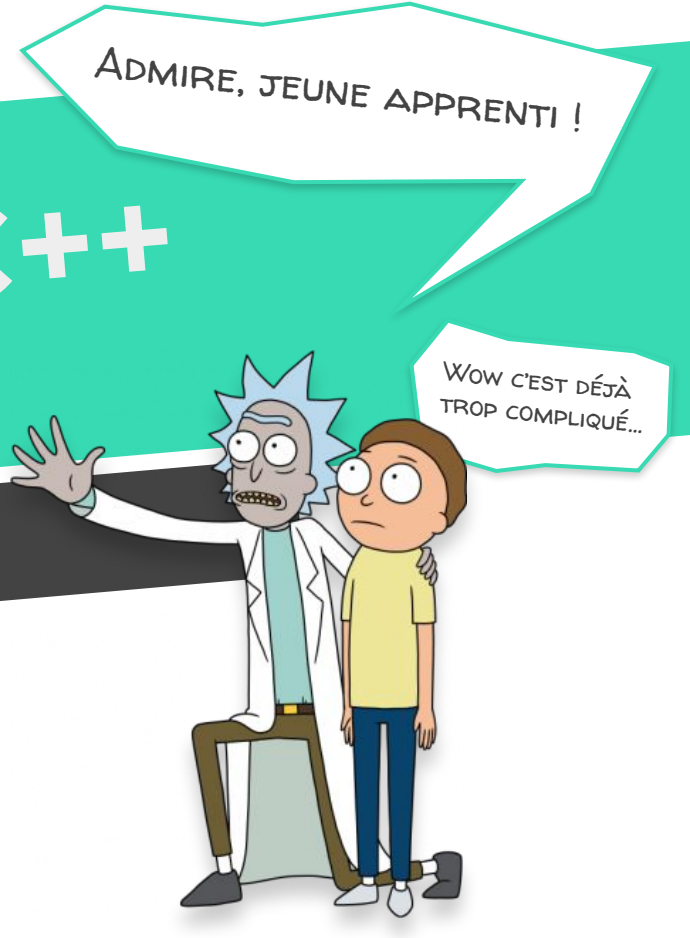
Cours de C++

Segment 1

2025-2026



Sommaire

Université Gustave Eiffel

Université Gustave Eiffel

Les enseignants de C++ sont:

- **Henri Derycke** (TP Apprentis GR3) — henri.derycke@univ-eiffel.fr
- **Anthony Labarre** (TP Initiaux GR1) — anthony.labarre@univ-eiffel.fr
- **Victor Marsault** (CM & TP Initiaux GR2) — victor.marsault@univ-eiffel.fr

Vous pouvez nous contacter par mail mais aussi sur ***Discord***.

- Les supports du cours ont principalement été écrit par **Céline Noël**

Contenu du module

Université Gustave Eiffel

Séance supervisées:

- **3** cours magistraux pour présenter les segments
- **11** séances de TPs

Travail en autonomie:

- **9** chapitres sur le site web du cours **:**
  - ➢ Questionnaire à la fin de chaque chapitre pour vérifier que vous avez compris
  - ➢ Possibilité de l'envoyer à votre encadrant de TP qui vous fera un retour
- **1** TP de révision à réaliser en autonomie
- **2 à 3 heures** par semaine

Modalités d'évaluation

Contrôle continu:

- TP noté de 2h à ⅓ du semestre **(coef 2)**
- TP noté de 2h à ⅔ du semestre **(coef 2)**
- TP noté final de 3h **(coef 3)**

Rattrapage: TP noté de 3h qui remplace **la plus faible** des trois notes !

Modalités:

- Utilisation du mode exam donc **pas d'accès à internet**
- Accès au site du cours et à la documentation du standard C++
- Tests automatiques

Déroulement du module

S1	CM
	Chapitre à lire
	TP
S2	**Chapitre à lire**
	TP
S3	**Chapitre à lire**
	TP
S4	TP de révision
	TP Noté

S5	CM
S6	**Chapitre à lire**
	TP
S7	**Chapitre à lire**
	TP
S8	**Chapitre à lire**
	TP
S9	TP de révision
	TP Noté

S10	CM
S11	**Chapitre à lire**
	TP
S12	**Chapitre à lire**
	TP
S13	**Chapitre à lire**
	TP
S14	**Révisions**
S15	TP Noté final

— En autonomie

Université Gustave Eiffel

Ressources du cours:

- Site web du cours avec des exercices d'entraînement
- Miroir local de la documentation du standard C++
- Dépôt git des Tps

Pour tester des *snippets* de code :

➔ Compiler Explorer sur godbolt

Pour développer des projets :

- ~~Visual Studio Code~~ Visual Studio Codium
- CMake
- Git

Outils & Ressources

Ressources du cours:

- Site web du cours avec des exercices d'entraînement
- Miroir local de la documentation du standard C++
- Dépôt git des Tps

Pour tester des *snippets* de code :

➔ Compiler Explorer sur godbolt

Pour développer des projets :

- ~~Visual Studio Code~~ Visual Studio Codium
- CMake
- Git

Certaines parties du site et des TPs pourraient ne pas être à jour

Quelques avertissements

- Les notes ne sont pas entièrement compensables
  ➢ Le TP noté de rattrapage ne remplace qu'une seule note
- Le système de test automatique est assez punitifs
  ➢ Un test ne passe passe pas ne vaut généralement pas de points
- La difficulté augmente avec le temps
  ➢ Le TPN1 est plus facile
- La référence du cours est le site web et pas le CM
  ➢ Si vous ne lisez pas les chapitres, il vous manquera des notions

Quelques avertissements

- Les notes ne sont pas entièrement compensables
  ➢ Le TP noté de rattrapage ne remplace qu'une seule note
- Le système de test automatique est assez punitifs
  ➢ Un test ne passe passe pas ne vaut généralement pas de points
- La difficulté augmente avec le temps
  ➢ Le TPN1 est plus facile
- La référence du cours est le site web et pas le CM
  ➢ Si vous ne lisez pas les chapitres, il vous manquera des notions

Travaillez tout au long du semestre

Avant de continuer…

Université Gustave Eiffel

🤨 des questions sur le déroulement du module ?

Le C++ est un langage de programmation…

- **Compilé**
  ➔ Donc **rapide** à l'exécution.
- **Orienté-objet**
  ➔ Donc on peut architecturer des **gros projets** sans avoir trop envie de mourir.
- **Générique**
  ➔ Donc on peut facilement limiter le copier-coller d'algorithmes pour supporter **différents types**.
- **Bien documenté**
  ➔ **Standard** actif et grande base d'utilisateurs !

Université Gustave Eiffel

Le C++ est un langage de programmation…

- **Compilé**
  ➔ Donc **rapide** à l'exécution.
- **Orienté-objet**
  ➔ Donc on peut architecturer des **gros projets** sans avoir trop envie de mourir.
- **Générique**
  ➔ Donc on peut facilement limiter le copier-coller d'algorithmes pour supporter **différents types**.
- **Bien documenté**
  ➔ **Standard** actif et grande base d'utilisateurs !
- **Quasi-rétro-compatible avec le C**
  ➔ Coexistence de types C et C++
  ➔ La syntaxe pour les concepts modernes **pique un peu les yeux** au début

Fonction `main`

Identifiant — `main`
Arguments — `()` ou `(int argc, char** argv)`
Type de retour — `int`

```c
int main()
{
    return 0;
}
```

Afficher du texte dans la console

Université Gustave Eiffel

```cpp
#include <iostream>

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

Afficher du texte dans la console

Université Gustave Eiffel

```
#include <iostream>

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

Donne accès aux **symboles** déclarés dans la section *I/O* (**I**nput/**O**utput, *i.e.* Entrées/Sorties) de la **librairie standard**.

Afficher du texte dans la console


```cpp
#include <iostream>

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

Cible les **symboles** du ***namespace*** `std` (librairie standard)

Afficher du texte dans la console


```cpp
#include <iostream>

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

Sortie standard.

Saut de ligne + *flush*.

Compiler en ligne de commande

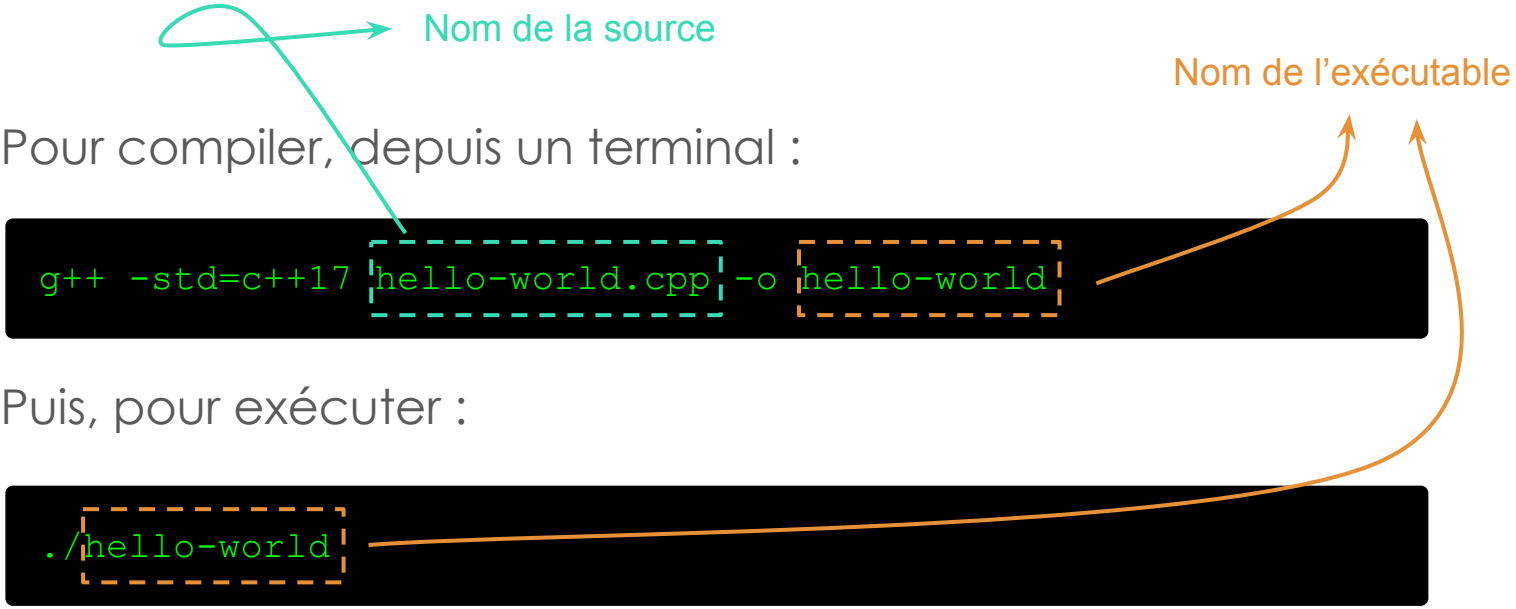
Pour compiler, depuis un terminal :

```
g++ -std=c++17 hello-world.cpp -o hello-world
```

Puis, pour exécuter :

```
./hello-world
```

Compiler en ligne de commande



Ajouter un `CMakeLists.txt`

Université Gustave Eiffel

```
cmake_minimum_required(VERSION 3.16)
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Ajouter un `CMakeLists.txt`


```
cmake_minimum_required(VERSION
project(cours-1)
```

```
add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Permet de générer un **exécutable**.

Ajouter un `CMakeLists.txt`

Université Gustave Eiffel

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Nom de l'exécutable.

Ajouter un `CMakeLists.txt`

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Liste des **sources.**

Ajouter un `CMakeLists.txt`


```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Permet de sélectionner un set de **fonctionnalités** pour le langage.

Ajouter un `CMakeLists.txt`

Université Gustave Eiffel

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

C++ **17**

Ajouter un `CMakeLists.txt`

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Permet de passer des **options** au **compilateur** lors de la phase de **compilation**.

Ajouter un `CMakeLists.txt`

Université Gustave Eiffel

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Active un premier *set* de ***warnings***.

Université Gustave Eiffel

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Active un second *set* de ***warnings***.

Ajouter un `CMakeLists.txt`

Université Gustave Eiffel

```
cmake_minimum_required(VERSION
project(cours-1)

add_executable(hello-world
    hello-world.cpp
)

target_compile_features(hello-world PRIVATE cxx_std_17)
target_compile_options(hello-world PRIVATE
    -Wall
    -Wextra
    -Werror
)
```

Légende

Considère les ***warnings*** comme des **erreurs**.

Lire du texte depuis la console


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

int main()
{
    std::cout << "What's your name? " << std::endl;

    auto name = std::string {};
    std::cin >> name;

    std::cout << "Hello " << name << std::endl;
    return 0;
}
```

Lire du texte depuis la console


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

int main()
{
    std::cout << "What's your name? " << std::endl;

    auto name = std::string {};
    std::cin >> name;

    std::cout << "Hello " << name << std::endl;
    return 0;
}
```

Construit une **instance** de type `std::string`

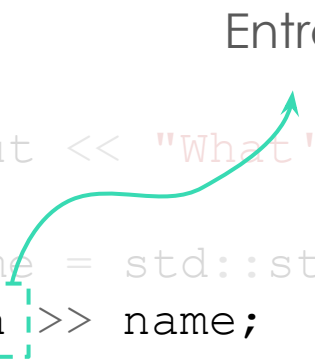
Lire du texte depuis la console


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

int main()
{
    std::cout << "What's your name? " << std::endl;

    auto name = std::string {};
    std::cin >> name;

    std::cout << "Hello " << name << std::endl;
    return 0;
}
```

Type **déduit** de ce qu'il y a à droite du symbole =

Lire du texte depuis la console


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

int main()
{
    std::cout << "What's your name? " << std::endl;

    auto name = std::string {};
    std::cin >> name;

    std::cout << "Hello " << name << std::endl;
    return 0;
}
```



Université Gustave Eiffel

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

int main()
{
    std::cout << "What's your name? " << std::endl;

    auto name = std::string {};
    std::cin >> name;

    std::cout << "Hello " << name << std::endl;
    return 0;
}
```

🤨 Des questions?

```cpp
#include <iostream>

int main(int argc, char** argv)
{
    if (argc != 2u)
    {
        std::cerr << "Program expects one argument: "
                  << (argc - 1)
                  << " were given." << std::endl;
        return -1;
    }

    std::cout << "Hello " << argv[1] << std::endl;
    return 0;
}
```

Utiliser les arguments du programme

Université Gustave Eiffel

```cpp
#include <iostream>

int main(int argc, char** argv)
{
    if (argc != 2u)
    {
        std::cerr << "Program expects one argument: "
                  << (argc - 1)
                  << " were given." << std::endl;
        return -1;
    }

    std::cout << "Hello " << argv[1] << std::endl;
    return 0;
}
```

Chemin de l'exécutable, puis **arguments**.

Nombre d'**arguments** (+ 1 pour le chemin de l'exécutable)

Université Gustave Eiffel

Sortie d'erreurs.

```cpp
#include <iostream>

int main(int argc, char** argv)
{
    if (argc != 2u)
    {
        std::cerr << "Program expects one argument: "
                  << (argc - 1)
                  << " were given." << std::endl;
        return -1;
    }

    std::cout << "Hello " << argv[1] << std::endl;
    return 0;
}
```

Université Gustave Eiffel

🤨 Des questions?

```cpp
#include <iostream>

int main(int argc, char** argv)
{
    if (argc != 2u)
    {
        std::cerr << "Program expects one argument: "
                  << (argc - 1)
                  << " were given." << std::endl;
        return -1;
    }

    std::cout << "Hello " << argv[1] << std::endl;
    return 0;
}
```

Les types hérités du C :

- Types **entiers** : `int`, `short`, `long`, `unsigned int`, ...
- Types **flottants** : `float`, `double`.
- Types **character** : `char`, `unsigned char`.

Mais aussi :

- Type **booléen** : `bool`.
- Types **entiers** de **taille fixe** : `int8_t`, `uint32_t`, ...
- Type **taille** : `size_t`.

Définition de variables avec `auto`

Université Gustave Eiffel

```
auto int_value = 3;
auto unsigned_value = 3u;
auto float_value = 3.f;
auto double_value = 3.0;
auto size_value = size_t { 3 };
auto return_value = fcn();
auto mavar = MaClasse{};
```

Avantages :

- Variables de types **fondamentaux** sont nécessairement **initialisées**.

```
int a; // Ici la valeur de a est n'importe quoi
```

- **Pas de duplication** dans le code (***refactoring*** plus rapide)

```
MaClasse a = mafonction();
```

- Meilleure **lisibilité** quand les types sont complexes (templates)

Inconvénient :

- Si on n'a pas d'**IDE**, il est nécessaire de **fouiller un peu** et d'aller chercher le **type de retour** des fonctions pour connaître celui des variables.

```cpp
#include <string>

int main()
{
    auto empty_str = std::string { "" };

    auto pouet = std::string { "pouet" };
    auto size = pouet.length();
    auto c0 = pouet.front();
    auto c3 = pouet[3];

    auto big_pouet = std::string {};
    for (auto c: pouet)
    {
        big_pouet += std::toupper(c);
    }

    auto half_pouet = pouet.substr(0, pouet.length() / 2);

    return 0;
}
```

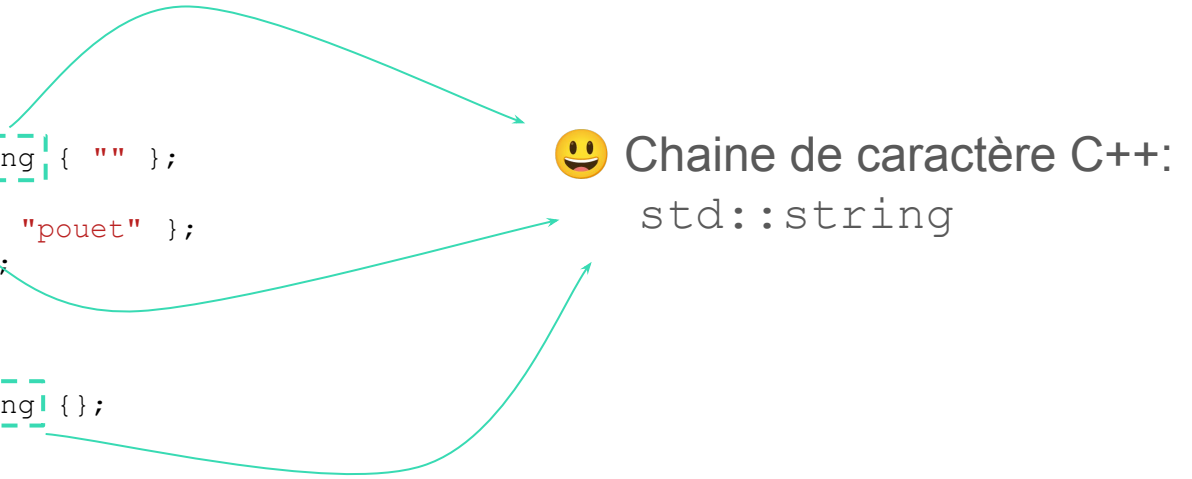
Université Gustave Eiffel

```cpp
#include <string>

int main()
{
    auto empty_str = std::string { "" };

    auto pouet = std::string { "pouet" };
    auto size = pouet.length();
    auto c0 = pouet.front();
    auto c3 = pouet[3];

    auto big_pouet = std::string {};
    for (auto c: pouet)
    {
        big_pouet += std::toupper(c);
    }

    auto half_pouet = pouet.substr(0, pouet.length() / 2);

    return 0;
}
```

😠 Chaine de caractère C : `char*`

Chaînes de caractères


```cpp
#include <string>

int main()
{
    auto empty_str = std::string{ "" };

    auto pouet = std::string{ "pouet" };
    auto size = pouet.length();
    auto c0 = pouet.front();
    auto c3 = pouet[3];

    auto big_pouet = std::string{};
    for (auto c: pouet)
    {
        big_pouet += std::toupper(c);
    }

    auto half_pouet = pouet.substr(0, pouet.length() / 2);

    return 0;
}
```



😃 Chaine de caractère C++:
`std::string`

```cpp
#include <string>

int main()
{
    auto empty_str = std::string { "" };

    auto pouet = std::string { "pouet" };
    auto size = pouet.length();
    auto c0 = pouet.front();
    auto c3 = pouet[3];

    auto big_pouet = std::string {};
    for (auto c: pouet)
    {
        big_pouet += std::toupper(c);
    }

    auto half_pouet = pouet.substr(0, pouet.length() / 2);

    return 0;
}
```

Les `std::string` sont des objets

```cpp
#include <string>

int main()
{
    auto empty_str = std::string { "" };

    auto pouet = std::string { "pouet" };
    auto size = pouet.length();
    auto c0 = pouet.front();
    auto c3 = pouet[3];

    auto big_pouet = std::string {};
    for (auto c: pouet)
    {
        big_pouet += std::toupper(c);
    }

    auto half_pouet = pouet.substr(0, pouet.length() / 2);

    return 0;
}
```

🤨 Des questions?

Tableaux dynamiques

Université Gustave Eiffel

```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }


    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```

```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }


    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```

On crée un vecteur avec 3 éléments

```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }


    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```

On ajoute 2 éléments en plus à la fin du tableaux `v1`

L'emplacement mémoire de v1 peut avoir changé

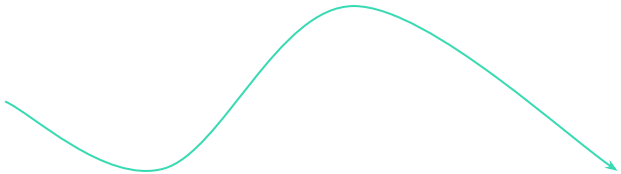
```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }

    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```

On parcourt `v1` “à la main” et on affiche ses éléments

Tableaux dynamiques

Université Gustave Eiffel

```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }

    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```



On parcourt `v1` avec un “for each” et on calcule la somme de ses éléments

```cpp
#include <vector>

int main()
{
    auto v1 = std::vector<int> {0,1,2};

    v1.emplace_back(4);
    v1.emplace_back(5);

    auto size = v1.size();
    for (unsigned i = 0; i<size; ++i)
    {
        std::cout << v1[i]
    }


    auto sum = 0;
    for (auto e: v1)
    {
        sum += e;
    }

    return 0;
}
```

🤨 Des questions?

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

int main()
{
    auto a = 1;
    std::cout << a << std::endl; // 1

    auto& b = a;
    b = 3;
    std::cout << a << std::endl; // 3

    auto  vec = std::vector { 1, 2, 3 };
    auto& last = vec.back();
    last = 5;
    std::cout << vec[2] << std::endl; // 5

    return 0;
}
```

Une **référence** est un *alias* d'une variable, elle partage donc le **même espace mémoire** qu'elle.

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

int main()
{
    auto a = 1;
    std::cout << a << std::endl; // 1

    auto& b = a;
    b = 3;
    std::cout << a << std::endl; // 3

    auto vec = std::vector { 1, 2, 3 };
    auto& last = vec.back();
    last = 5;
    std::cout << vec[2] << std::endl; // 5

    return 0;
}
```

Une **référence** est un *alias* d'une variable, elle partage donc le **même espace mémoire** qu'elle.

Pour définir une **référence**, on place une ***esperluette*** (&) après le type.

Université Gustave Eiffel

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

int main()
{
    auto a = 1;
    std::cout << a << std::endl; // 1

    auto& b = a;
    b = 3;
    std::cout << a << std::endl; // 3

    auto  vec = std::vector { 1, 2, 3 };
    auto& last = vec.back();
    last = 5;
    std::cout << vec[2] << std::endl; // 5

    return 0;
}
```

Une **référence** est un *alias* d'une variable, elle partage donc le **même espace mémoire** qu'elle.

Quand on modifie l'une l'autre est modifiée.

Université Gustave Eiffel

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

int main()
{
    auto a = 1;
    std::cout << a << std::endl; // 1

    auto& b = a;
    b = 3;
    std::cout << a << std::endl; // 3

    auto  vec = std::vector { 1, 2, 3 };
    auto& last = vec.back();
    last = 5;
    std::cout << vec[2] << std::endl; // 5

    return 0;
}
```

Une **référence** est un *alias* d'une variable, elle partage donc le **même espace mémoire** qu'elle.

Quand on modifie l'une l'autre est modifiée.

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

int main()
{
    auto a = 1;
    std::cout << a << std::endl; // 1

    auto& b = a;
    b = 3;
    std::cout << a << std::endl; // 3

    auto  vec = std::vector { 1, 2, 3 };
    auto& last = vec.back();
    last = 5;
    std::cout << vec[2] << std::endl; // 5

    return 0;
}
```

Une **référence** est un *alias* d'une variable, elle partage donc le **même espace mémoire** qu'elle.

🤨 Des questions?

Références constantes

```cpp
int main()
{
    const auto const_var = 1;
    const_var = 3; // invalide

    auto        mutable_var = 1;
    const auto& const_ref   = mutable_var;
    const_ref = 3; // invalide

    return 0;
}
```

Pour définir une variable ou une référence **constante**, on place `const` sur le type.

Avantages :

- Facilite le *debug* (si c'est constant, c'est que ça ne changera pas)
- Facilite la **compréhension** du code.

Inconvénient :

- Verbeux, donc il faut s'habituer à la lecture.

Référence pendante (dangling reference)

Université Gustave Eiffel

```cpp
int main()
{
    auto v1 = std::vector<std::string> {"Hello","World"};
    auto& ref = v1[1];

    v1[1] = "Universe";
    std::cout << ref << std::endl; // Universe

    v1.emplace_back("My");
    v1.emplace_back("Name");
    v1.emplace_back("is");
    v1.emplace_back("World");

    std::cout << ref << std::endl; // ???
}
```

Référence pendante (dangling reference)

Université Gustave Eiffel

```cpp
int main()
{
    auto v1 = std::vector<std::string> {"Hello","World"};
    auto& ref = v1[1];

    v1[1] = "Universe";
    std::cout << ref << std::endl; // Universe

    v1.emplace_back("My");
    v1.emplace_back("Name");
    v1.emplace_back("is");
    v1.emplace_back("World");

    std::cout << ref << std::endl; // ???
}
```

`ref` est un alias vers la case 1 de `v`

Référence pendante (dangling reference)

Université Gustave Eiffel

```cpp
int main()
{
    auto v1 = std::vector<std::string> {"Hello","World"};
    auto& ref = v1[1];

    v1[1] = "Universe";
    std::cout << ref << std::endl; // Universe

    v1.emplace_back("My");
    v1.emplace_back("Name");
    v1.emplace_back("is");
    v1.emplace_back("World");

    std::cout << ref << std::endl; // ???
}
```

On modifie la case référencée, et on l'affiche

Aucun problème : c'est bien la nouvelle valeur qui s'affiche

Référence pendante (dangling reference)

Université Gustave Eiffel

```cpp
int main()
{
    auto v1 = std::vector<std::string> {"Hello","World"};
    auto& ref = v1[1];

    v1[1] = "Universe";
    std::cout << ref << std::endl; // Universe

    v1.emplace_back("My");
    v1.emplace_back("Name");
    v1.emplace_back("is");
    v1.emplace_back("World");

    std::cout << ref << std::endl; // ???
}
```

On ajoute des élément à `v`. Il a peut-être été déplacé.

Question: vers quoi pointe `ref`?

Référence pendante (dangling reference)


```cpp
int main()
{
    auto v1 = std::vector<std::string> {"Hello","World"};
    auto& ref = v1[1];

    v1[1] = "Universe";
    std::cout << ref << std::endl; // Universe

    v1.emplace_back("My");
    v1.emplace_back("Name");
    v1.emplace_back("is");
    v1.emplace_back("World");

    std::cout << ref << std::endl; // ???
}
```

🤨 Des questions?

Définir une fonction


```cpp
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}


size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Définir une fonction


```cpp
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}

size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Type de retour.

Définir une fonction


```cpp
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}

size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Identifiant de la fonction.

Définir une fonction


```cpp
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}


size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Paramètres de la fonction.

Définir une fonction


```
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}


size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Passage de paramètre **par référence constante**

Passage de paramètre **par copie**

Définir une fonction


```
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}

size_t count_letter(const std::string& words, char letter)
{
    auto count = size_t { 0 };
    for (auto l: words)
    {
        if (l == letter)
        {
            ++count;
        }
    }
    return count;
}
```

Corps de la fonction.

Vocabulaire :

- **Signature** — Identifiant + Types des paramètres.
- **Surcharge** (ou *overloading*) — Définir une fonction avec le **même identifiant** qu'une autre, mais une **signature différente**.
- **Prototype** — Signature + type de retour

La surcharge est possible si au moins l'une de ces conditions est vérifiée :

- Le **nombre de paramètres** est différent.
- La **succession des types** de paramètres est différente.

Surcharger une fonction

Vocabulaire :

- **Signature** — Identifiant + Types des paramètres.
- **Surcharge** (ou *overloading*) — Définir une fonction avec le **même identifiant** qu'une autre, mais une **signature différente**.
- **Prototype** — Signature + type de retour

La surcharge est possible si au moins l'une de ces conditions est vérifiée :

- Le **nombre de paramètres** est différent.
- La **succession des types** de paramètres est différente.

🤨 Des questions?

Surcharger une fonction


```cpp
void print_sum(int e1, int e2)
{
    std::cout << e1 + e2 << std::endl;
}

void print_sum(int e1, int e2, int e3)
{
    std::cout << e1 + e2 + e3 << std::endl;
}

void print_sum(const std::string& e1, const std::string& e2)
{
    std::cout << e1 + e2 << std::endl;
}
```

Université Gustave Eiffel

Passage par **valeur** (ou par **copie**)

➔ L'argument est **copié** au moment de l'appel.

```
int sum(int v1, int v2)
{
    v1 += v2;
    return v1;
}

int main()
{
    auto v1 = 3;
    auto v2 = 5;
    std::cout << sum(v1, v2) << std::endl; // 8
    std::cout << v1 << std::endl;          // 3

    return 0;
}
```

Cet appel **ne modifie pas** le `v1` dans `main`

Université Gustave Eiffel

Passage par **référence**.

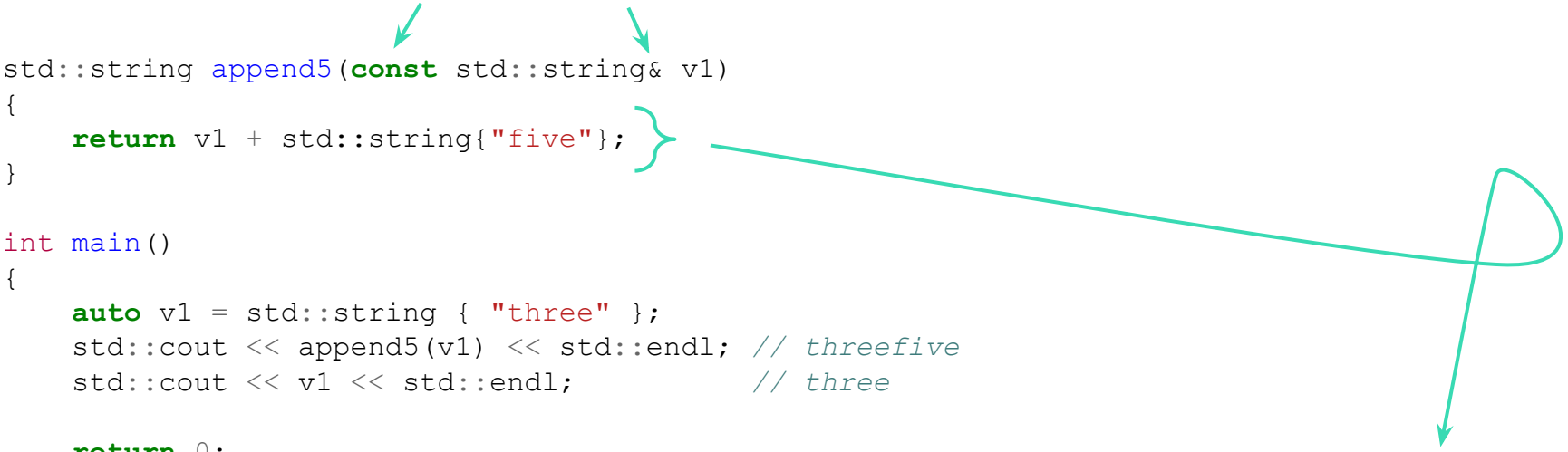
➔ On crée un ***alias*** sur l'argument au moment de l'appel.

```cpp
int sum(int& v1, int v2)
{
    v1 += v2;
    return v1;
}

int main()
{
    auto v1 = 3;
    auto v2 = 5;
    std::cout << sum(v1, v2) << std::endl; // 8
    std::cout << v1 << std::endl;          // 8

    return 0;
}
```

Université Gustave Eiffel

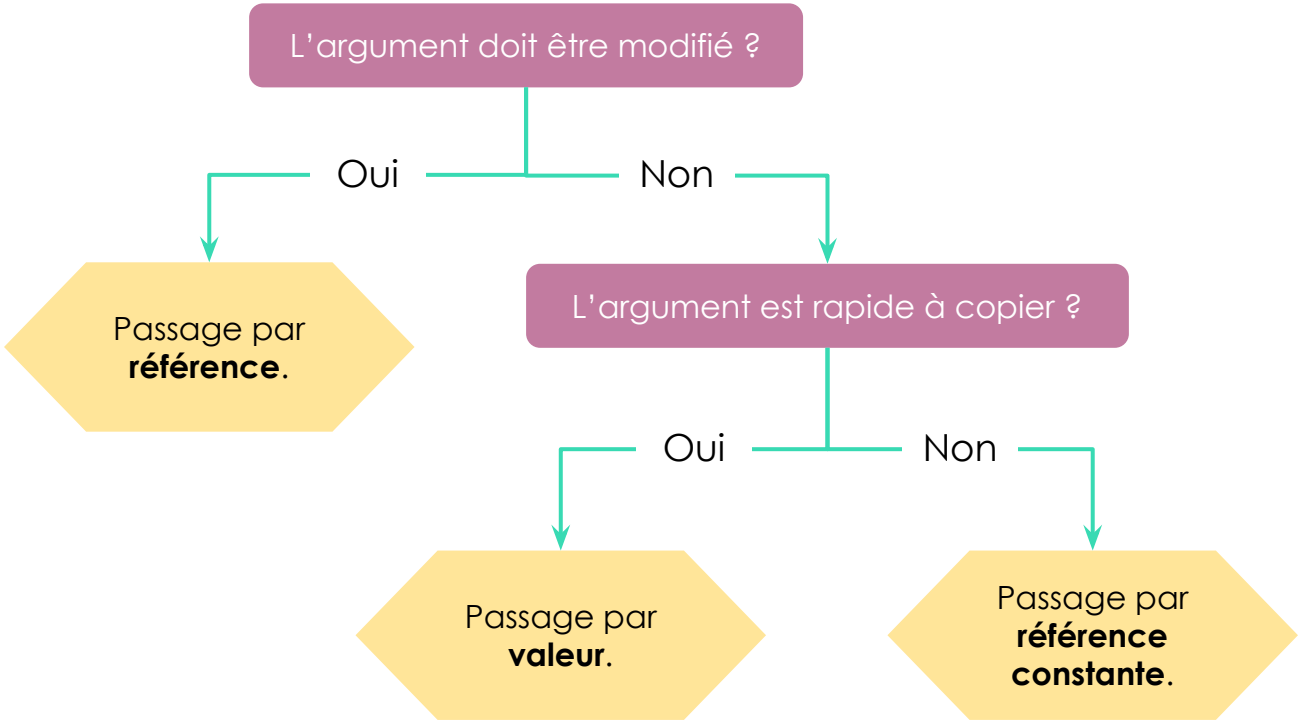
Passage par **référence constante**.

➔ On crée un ***alias* non-mutable** sur l'argument au moment de l'appel.



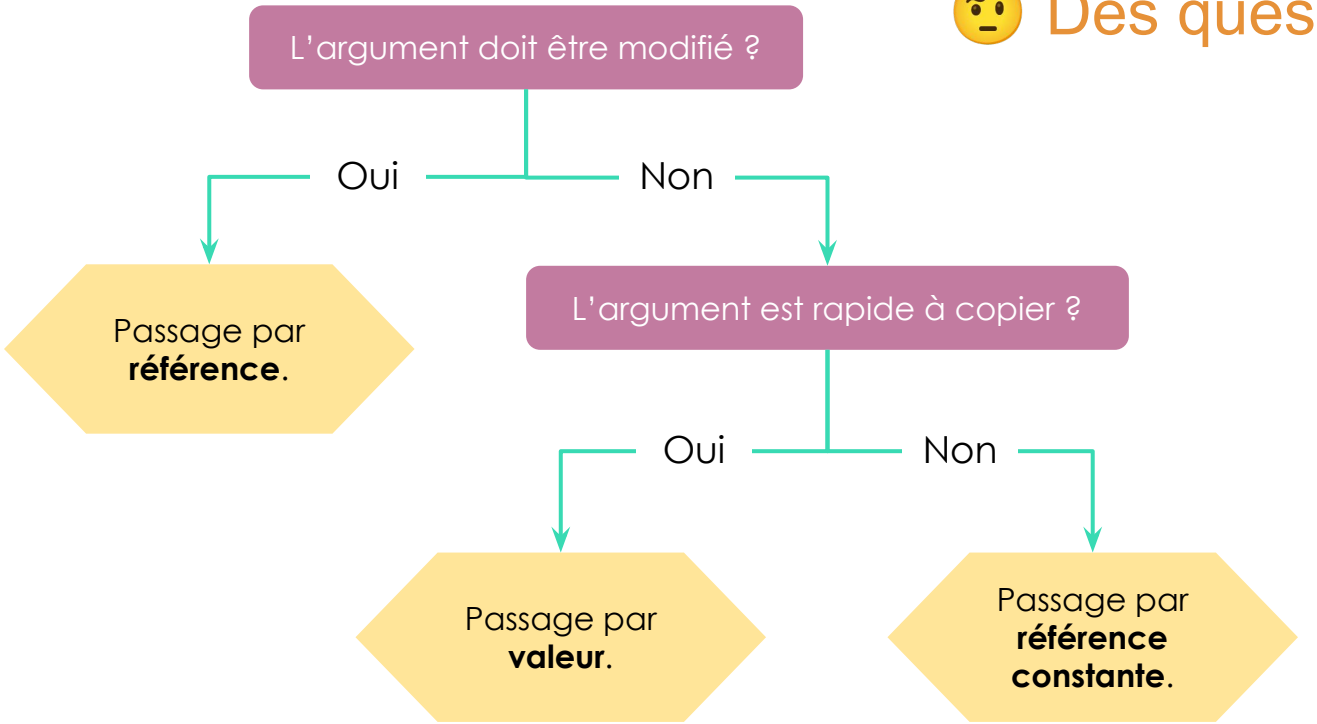
```cpp
std::string append5(const std::string& v1)
{
    return v1 + std::string{"five"};
}

int main()
{
    auto v1 = std::string { "three" };
    std::cout << append5(v1) << std::endl; // threefive
    std::cout << v1 << std::endl;          // three

    return 0;
}
```

Dans le corps de `append5`, tout ce qui modifie `v1` est interdit !


L'argument doit être modifié ?
Oui
Non
Passage par **référence**.
L'argument est rapide à copier ?
Oui
Non
Passage par **valeur**.
Passage par **référence constante**.

Université Gustave Eiffel

🤨 Des questions?



Définir une classe

Université Gustave Eiffel

```cpp
#include <string>

class Student
{
public:
    std::string   name;
    int           age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

Définir une classe


```cpp
#include <string>

class Student
{
public:
    std::string   name;
    int           age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

Nom de la classe.

Attributs de la classe.

Le **constructeur par défaut** est appelé

```cpp
#include <string>

class Student
{
public:
    std::string    name;
    int            age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

Attention aux oublis !

Oubli du **modificateur** `public`

```
error: '<attribute>' is private within this context
```

```cpp
#include <string>

class Student
{
public:
    std::string   name;
    int           age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

Attention aux oublis !

Oubli du **point-virgule** (`;`)

```
error: expected ';' after class definition
```

```cpp
#include <string>

class Student
{
public:
    std::string    name;
    int            age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

Attention aux oublis !

Non **initialisation** des **attributs de types fondamentaux**.

`Undefined behavior` (à l'exécution)

Définir une classe

Université Gustave Eiffel

```cpp
#include <string>

class Student
{
public:
    std::string   name;
    int           age = 0;
};

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;

    return 0;
}
```

🤨 Des questions?

Définir des fonctions-membres

Université Gustave Eiffel

```cpp
class Student
{
public:
    void set_attributes(const std::string& name,
                        int                age)
    {
        m_name = name;
        m_age  = age;
    }

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {};
    student.set_attributes("David",22);
    student.print();
    return 0;
}
```

Définir des fonctions-membres


```cpp
class Student
{
public:
    void set_attributes(const std::string& name,
                        int                age)
    {
        m_name = name;
        m_age  = age;
    }

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {};
    student.set_attributes("David",22);
    student.print();
    return 0;
}
```

Indique que la fonction **ne modifie pas** les **attributs** de l'instance.

Définir un constructeur


```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

Définir un constructeur


```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

Un constructeur a des paramètres comme une fonction

Définir un constructeur

Université Gustave Eiffel

```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

Et s'appelle comme ceci

Un constructeur a des paramètres comme une fonction

Définir un constructeur


```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

La **liste d'initialisation**

Définir un constructeur


```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

La **liste d'initialisation** permet d'initialiser les attributs

Définir un constructeur

Université Gustave Eiffel

```cpp
class Student
{
public:
    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    auto student = Student {"David", 22};
    student.print();
    return 0;
}
```

🤨 Des questions?

Implémentation par défaut du constructeur par défaut

Université Gustave Eiffel

```cpp
class Student
{
public:
    Student() = default;

    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    const auto david = Student { "David", 22 };
    david.print();

    const auto default_student = Student {};
    default_student.print();

    return 0;
}
```

Implémentation par défaut du constructeur par défaut

Université Gustave Eiffel

```cpp
class Student
{
public:
    Student() = default;

    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    const auto david = Student { "David", 22 };
    david.print();

    const auto default_student = Student {};
    default_student.print();

    return 0;
}
```

Si on écrit un constructeur, le compilateur **ne** génère **plus** le constructeur par défaut

Implémentation par défaut du constructeur par défaut

Université Gustave Eiffel

```cpp
class Student
{
public:
    Student() = default;

    Student(const std::string& name, int age)
        : m_name { name }
        , m_age { age }
    {}

    void print() const
    {
        std::cout << "Student called " << m_name
                  << " is " << m_age << " years old"
                  << std::endl;
    }

private:
    std::string m_name;
    int         m_age = 0;
};
```

```cpp
int main()
{
    const auto david = Student { "David", 22 };
    david.print();

    const auto default_student = Student {};
    default_student.print();

    return 0;
}
```

Rétablit l'**implémentation par défaut** du constructeur par défaut.

Définir un opérateur de flux ami


```cpp
class Student
{
public:
  Student(const std::string& name, int age)
    : m_name { name }
    , m_age { age }
  {}

  friend std::ostream& operator<<(std::ostream& stream,
                                  const Student& student)
  {
    stream << "Student called " << student.m_name
           << " is " << student.m_age << " years old.";
    return stream;
  }

private:
  std::string m_name;
  int         m_age = 0;
};
```

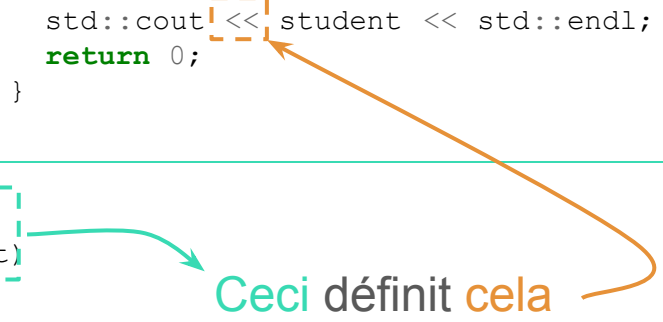
```cpp
int main()
{
  const auto student = Student { "David",22 };
  std::cout << student << std::endl;
  return 0;
}
```

Définir un opérateur de flux ami

Université Gustave Eiffel

```cpp
class Student
{
public:
  Student(const std::string& name, int age)
    : m_name { name }
    , m_age { age }
  {}

  friend std::ostream& operator<<(std::ostream& stream,
                                  const Student& student)
  {
    stream << "Student called " << student.m_name
           << " is " << student.m_age << " years old.";
    return stream;
  }

private:
  std::string m_name;
  int         m_age = 0;
};
```

```cpp
int main()
{
  const auto student = Student { "David",22 };
  std::cout << student << std::endl;
  return 0;
}
```



Définir un opérateur de flux ami


```cpp
class Student
{
public:
  Student(const std::string& name, int age)
    : m_name { name }
    , m_age { age }
  {}

  friend std::ostream& operator<<(std::ostream& stream,
                                  const Student& student)
  {
    stream << "Student called " << student.m_name
           << " is " << student.m_age << " years old.";
    return stream;
  }

private:
  std::string m_name;
  int         m_age = 0;
};
```

```cpp
int main()
{
  const auto student = Student { "David",22 };
  std::cout << student << std::endl;
  return 0;
}
```

Spécifie que la fonction est **amie**, c'est-à-dire peut accéder aux champs privés

Définir un opérateur de flux ami


```cpp
class Student
{
public:
  Student(const std::string& name, int age)
    : m_name { name }
    , m_age { age }
  {}

  friend std::ostream& operator<<(std::ostream& stream,
                                  const Student& student)
  {
    stream << "Student called " << student.m_name
           << " is " << student.m_age << " years old.";
    return stream;
  }

private:
  std::string m_name;
  int         m_age = 0;
};
```

⚠️ **Attention !**

Une fonction amie est une **fonction libre**. Il faut donc lui passer une **instance** en **paramètre** pour accéder à ses membres (attributs et méthodes)

Définir un opérateur de flux ami


```cpp
class Student
{
public:
  Student(const std::string& name, int age)
    : m_name { name }
    , m_age { age }
  {}

  friend std::ostream& operator<<(std::ostream& stream,
                                  const Student& student)
  {
    stream << "Student called " << student.m_name
           << " is " << student.m_age << " years old.";
    return stream;
  }

private:
  std::string m_name;
  int         m_age = 0;
};
```

```cpp
int main()
{
  const auto student = Student { "David",22 };
  std::cout << student << std::endl;
  return 0;
}
```

🤨 Des questions?

- **Statique** = au moment de la **compilation**
  ⚠ **Le mot-clef `static` n'a rien à voir**

  Ex: La taille d'un entier est connue statiquement (32 ou 64 bit)

  - Dès lors qu'on alloue quelque chose **sur la pile** (variable), on doit connaître sa taille statiquement

- **Dynamique** = au moment de l'**exécution**
  Ex: Le nombre d'éléments dans un tableau **dynamique** n'est pas connu à la compilation

  - Allocation dynamique = allocation **sur le tas** (~~`malloc`~~ ou `new`)

```cpp
int main()
{
    auto words = std::vector<std::string>{};

    while (true) {
        auto name = std::string {};
        std::cin >> name;
        words.emplace_back(name);
    }

}
```

Université Gustave Eiffel

Un objet est

- alloué ⟶ un segment de mémoire est attribué
  - ≈ `malloc`
- construit ⟶ un **constructeur** est appelé pour remplir ce segment

…

- détruit ⟶ un **destructeur** est appelée pour nettoyer
- désalloué ⟶ la mémoire est rendu disponible
  - ≈ `free`

Un objet est

- alloué ⟶ un segment de mémoire est attribué
  - ≈ `malloc`
- construit ⟶ un **constructeur** est appelé pour remplir ce segment

…

Il ne faut utiliser l'objet que pendant cette période

- détruit ⟶ un **destructeur** est appelée pour nettoyer
- désalloué ⟶ la mémoire est rendu disponible
  - ≈ `free`

Une et une seule fois

Problème: Gérer la mémoire est difficile !

- Fuite mémoire
- Double désallocation
- Dangling référence/pointeurs

Dans la plupart des autres langages: gestion automatique via un **garbage collector**

En C++: gestion semi-automatique via l'**ownership**

- On indique clairement qui a la charge de désallouer quoi
  - Usuellement indiqué dans le type
  - On sait si un objet **own** un autre objet ou s'il l'**observe**
- Destruction automatique des ressources owned
- On n'observe que des ressources qui ont une durée de vie plus longue

Ce qu'on va apprendre dans le segment 1

- Les types fondamentaux du C++ (int, bool,...)
- Quelques types de la bibliothèques standard (std::string, std::vector)
- Comprendre et corriger les messages d'erreurs les plus courants
- Bases des classes (membres, constructeurs, destructeurs)
- Copie vs Réference vs Référence constante vs Pointeurs
- Bases de la gestion de mémoire (Allocation, durée de vie, ownership)