

# Cours 1

# Introduction

C++ - Master 1

1. Présentation du module
2. Hello World
3. Types
4. Fonctions libres
5. Classes

## Le C++ est un langage...

- **compilé**
  - donc c'est rapide à l'exécution
- **orienté-objet**
  - donc on peut architecturer des grosses bases de code sans avoir trop envie de mourir
- **générique**
  - donc on peut facilement limiter le copier-coller d'algorithmes pour supporter différents types
- **verbeux**
  - donc ça pique un peu les yeux quand on lit du code les premières fois
- **populaire**
  - donc bien documenté et qui vous permettra de trouver un travail cool

Le module sera composé de :

- 4 séances de cours magistraux
- 12 séances de TPs
- 3 séances de TPs notés

Les ressources seront toutes accessibles en ligne :

- [Site Web](#) contenant les chapitres à lire avant chaque TP
- [Dépôt GitHub](#) contenant les énoncés de TP
- Slides (à venir)

Les séances de TP's seront encadrées par :

- Victor Marsault (Initiaux 1) - [victor.marsault@univ-eiffel.fr](mailto:victor.marsault@univ-eiffel.fr)
- Youssef Bergeron (Initiaux 2) - [youssefbergeron@gmail.com](mailto:youssefbergeron@gmail.com)
- Christophe Calle (Apprentis 1) - [christophe.calle.upem@gmail.com](mailto:christophe.calle.upem@gmail.com)
- Clément Chomicki (Apprentis 2) - [clement.chomicki@univ-eiffel.com](mailto:clement.chomicki@univ-eiffel.com)

Les cours magistraux seront animés par :

- Céline Noël - [celine.noel.7294@gmail.com](mailto:celine.noel.7294@gmail.com)

Vous pouvez nous contacter par mail mais aussi poser vos questions sur les channels Discord dédiés.

Pour tester des snippets de code :

- [Compiler Explorer](#)

Pour développer des projets :

- VSCode
- CMake
- Git

Identifiant : `main`

Arguments : `() / (int, int**)`

Type de retour : `int`

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

```
#include <iostream>

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



```
#include <iostream>
```

```
int main()  
{
```

```
    std::cout << "Hello World!" << std::endl;
```

```
    return 0;
```

cible les symboles  
du namespace std

donne accès aux symboles déclarées dans la  
section I/O (entrées/sorties) de la librairie standard

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    std::cout << "Hello World!" << std::endl;
```

```
    return;
```

```
}
```

saut de ligne + flush

sortie standard

Dans un terminal :

```
g++ hello-world.cpp -o hello-world
```

Pour exécuter :

```
./hello-world
```

```
cmake_minimum_required(VERSION 3.17)
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
)
```

```
cma
pro

    permet de générer
    un exécutable

    red (VERSION 3.17)
    nom de l'exécutable

add_executable(hello-world
    hello-world.cpp
    liste des sources
)

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

```
cmake_minimum_required(VERSION 3.17)
project(cours-1)
```

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

C++ 17

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

```

permet de sélectionner  
un set de features pour  
le langage

```
cmake_minimum_required(VERSION 3.17)
project(cours-1)
```

```
add_executable(hello-world)
)
```

permet de passer des options au  
compilateur lors de la phase de  
compilation

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

active un 1er set de warnings

active un 2nd set de warnings

considère les warnings  
comme des erreurs

```
#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;
}
```



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

type déduit de ce qu'il  
y a à droite du =

construit une instance  
de type std::string

```
int main() {
    std::string name;
    std::cin >> name;
    std::cout << "Nom: " << name << std::endl;
    return 0;
}
```

```
auto name = std::string {};
```

```
std::cin >> name;
```

entrée standard

```
std::cout << "Nom: " << name << std::endl;
return 0;
```

```
}
```

```
#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;
}
```

```
#include
```

nombre d'arguments + 1 pour  
le chemin de l'exécutable

```
int main(int argc, char** argv)  
{
```

```
    if (argc != 2u)  
    {
```

chemin de l'exécutable  
+ arguments

sortie d'erreurs

```
        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

```
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();
```

## Avantages :

- Variables primitives nécessairement initialisées
- Pas de duplication d'information dans le code (donc refactoring plus rapide)

## Inconvénient :

- Si pas d'IDE, nécessaire d'aller chercher le type de retour des fonctions pour connaître celui des variables

```
#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;
}
```

```
#include <vector>

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

    auto v2 = std::vector { 3, 2 };
    auto size = v2.size();
    auto sum = 0;
    for (auto e: v2)
    {
        sum += e;
    }

    auto v3 = std::vector<int>(3, 2);
    auto elt = v3[2];

    return 0;
}
```

Attention, les syntaxes d'initialisation pour v2 et v3 ne sont pas équivalentes :

- v2 vaut [3, 2]
- v3 vaut [2, 2, 2] (3x l'élément 2)



```
#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;
}
```

Pour définir une référence, on place & après le type.

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

```
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.

Intérêts :

- facilite le debug (si c'est const, c'est que ça ne changera pas)
- facilite la compréhension du code

Désavantage :

- verbeux, donc il faut s'habituer à la lecture

```
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;
}
```

```
void print_sum(int e1, int e2)
```

```
{  
    + e2 << std::endl;  
}
```

type de retour

paramètres

```
size_t count_letter(const std::string& words, char letter)
```

```
{  
    a size_t { 0 };  
    f words)
```

identifiant

corps

```
{  
    if (l == letter)  
    {  
        ++count;  
    }  
}  
return count;  
}
```

Vocabulaire :

- Signature = Identifiant + Types des paramètres
- Surcharge (ou overloading) = définir une fonction avec le même identifiant qu'une autre, mais une signature différente

La surcharge est possible si au moins l'une des conditions suivantes est vérifiée :

- le nombre de paramètres n'est pas le même
- la succession des types de paramètres n'est pas la même

```
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;
}
```

## 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;
}
```

## Passage par référence

⇒ on crée un alias sur l'argument au moment de l'appel

```
int add(int& res, int v)
{
    res += v;
    return res;
}

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

    return 0;
}
```



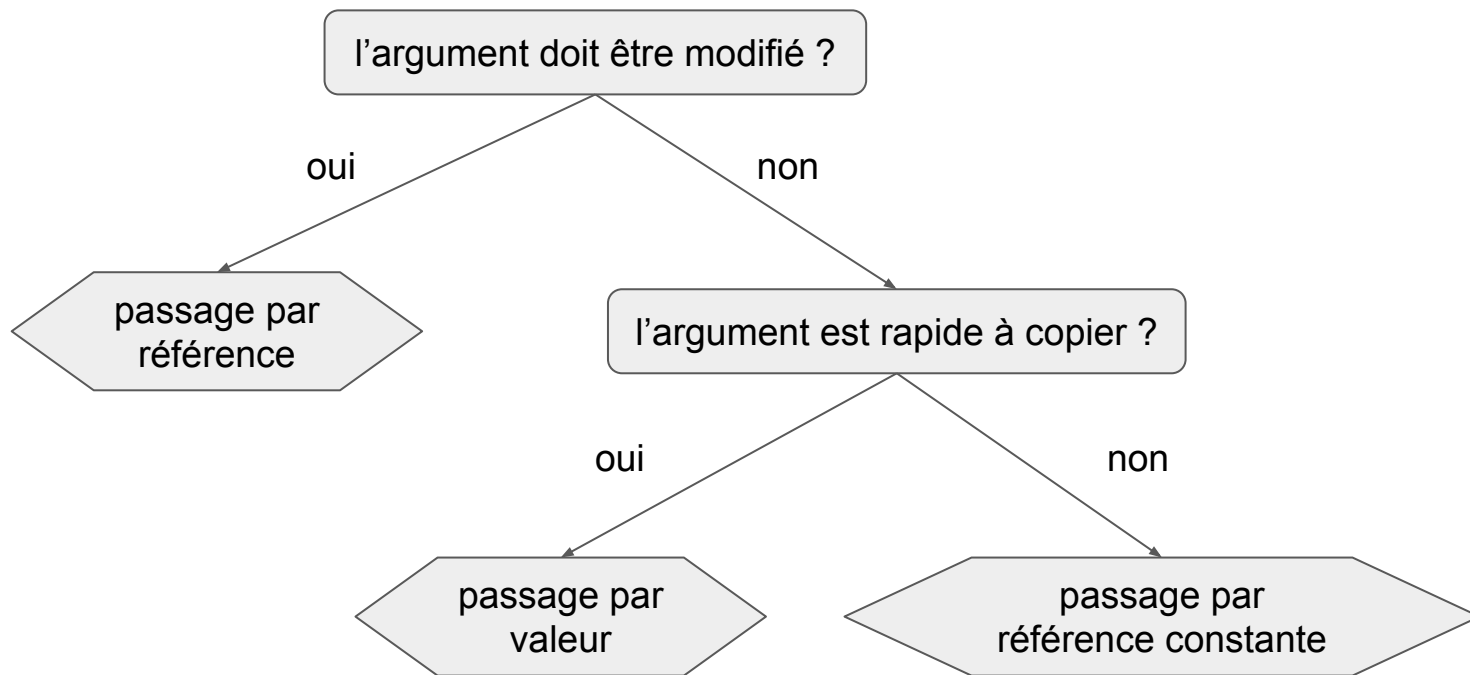
## Passage par référence constante

⇒ on crée un alias non-mutable sur l'argument au moment de l'appel

```
std::string sum(const std::string& v1, const std::string& v2)
{
    return v1 + v2;
}

int main()
{
    auto v1 = std::string { "three" };
    auto v2 = std::string { "five" };
    std::cout << sum(v1, v2) << std::endl; // threefive
    std::cout << v1 << std::endl;         // three

    return 0;
}
```



```
#include <string>

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

int main()
{
    auto student = Student {};
    student.name = "David";
    student.age = 22;
    return 0;
}
```

```
#include <string>
```

nom de la classe

```
class Student
```

```
{
```

```
public:
```

```
    std::string name;
```

```
    int age = 0;
```

```
};
```

attributs

```
int main()
```

```
{
```

```
    auto student = Student {};
```

```
    student.name = "David";
```

```
    student.age = 22;
```

```
    return 0;
```

```
}
```

```
#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 erreurs fréquentes :

- oubli du modificateur **public**  
error: '<attribute>' is private within this context
- oubli du ;  
error: expected ';' after class definition
- non initialisation des attributs primitifs  
⇒ undefined behavior à l'exécution

```
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;
};
```

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

```
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;
};
```


indique que la fonction ne modifie  
pas les attributs de l'instance

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

```
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;
};
```



```
int main()
{
    const auto student = Student { "David", 22 };


    student.print();
    return 0;
}
```



```
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;
};
```



liste d'initialisation

```
int main()
{
    const auto student = Student { "David", 22 };

    student.print();
    return 0;
}
```

```
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;
};
```

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

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

    return 0;
}
```

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

    Student(const string& name, int 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;
};
```

rétablit l'implémentation par défaut  
du constructeur par défaut

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

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

    return 0;
}
```

```
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;
};
```

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

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

spécifie que la fonction est amie

```
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;
};
```

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

```
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 passer une instance en paramètres pour accéder à ses champs.

```
error: invalid use of non-static data
member '<Class>::<attribute>'
```