

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
```

cmake

```
cmake_minimum_required(VERSION 3.17)

project(hello)

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

target_compile_features(hello-world PRIVATE cxx_std_17)

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

cmake

```
cmake_minimum_required(VERSION 3.17)
```

```
project(hello)
```

Permet de générer un **exécutable**.

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

```
target_compile_features(hello-world PRIVATE cxx_std_17)
```

```
target_compile_options(hello-world PRIVATE
```

```
    -Wall
```

```
    -Wextra
```

```
    -Werror
```

```
)
```

cmake

```
cmake_minimum_required(VERSION 3.17)
```

```
project(hello)
```

Permet de générer un **exécutable**.

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

Nom de l'**exécutable**



Source de l'**exécutable**



```
target_compile_features(hello-world PRIVATE cxx_std_17)
```

```
target_compile_options(hello-world PRIVATE
```

```
-Wall
```

```
-Wextra
```

```
-Werror
```

```
)
```

Cmake

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

définition du contexte pour un
langage donné



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


C++ 17



Cmake

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

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
)
```

Ajouter un CMakeLists.txt

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

Active un premier set de **warnings**.

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

Active un second set de **warnings**.

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

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

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

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



Lire du texte depuis la console

```
#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

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

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

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

```
int main()
{
```

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

```
    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

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

```
int main()      Entrée standard.
```

```
{
```

```
    std::cout << "What's your name? " << std::endl;
```

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

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

```
    std::cout << "Hello " << name << std::endl;
```

```
    return 0;
```

```
}
```

Utiliser les arguments du programme

```
#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

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

```
#include <iostream>

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

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

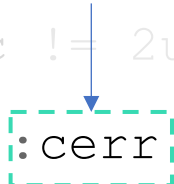
Nombre d'**arguments**
(+ 1 pour le chemin de l'exécutable)

Utiliser les arguments du programme

```
#include <iostream>

int main (sortie d'erreurs r** 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;
}
```



Sommaire

1. Présentation du module.
2. Hello, World!
- 3. Types.**
 - a. Types fondamentaux.
 - b. Définition de variables avec `auto`.
 - c. Chaînes de caractères.
 - d. Tableaux dynamiques.
 - e. Références.
 - f. Variables et références constantes.
4. Fonctions libres.
5. Classes.

Types fondamentaux

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

```
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();
```

Définition de variables avec auto

Avantages :

- Les variables de types **fondamentaux** sont nécessairement **initialisées**.
- **Pas de duplication** d'information dans le code (donc *refactoring* plus rapide)

Inconvénients :

- 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.

Chaînes de caractères

```
#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;
}
```

Tableaux dynamiques

```
#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(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]` (3 fois l'élément 2)

Références

```
#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.

Variables et références constantes

```
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.

Sommaire

1. Présentation du module.
2. Hello, World!
3. Types.
- 4. Fonctions libres**
 - a. Définir une fonction.
 - b. Surcharger une fonction.
 - c. Passage de paramètres.
5. Classes.

Définir une fonction - rappels de C

```
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

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

```
{  
    std::cout << e1 + e2 << std::endl;  
}
```

type de retour.

```
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

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

```
{  
    std::cout << e1 + e2 << std::endl;  
}
```

Identifiant de la fonction.

```
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

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

Paramètres de la fonction

```
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

```
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



Surcharger une fonction

Une fonction est entièrement définie par:
un identifiant, le type des arguments et le type de retour

Vocabulaire

- **Signature** — Identifiant + types des arguments et retour.
- **Surcharge** (ou *overloading*) — Définir une fonction avec le **même identifiant** qu'une autre, mais une **signature différente**.

Surcharge 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

```
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 de paramètres

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 de paramètres

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 de paramètres

Passage par **référence constante**.

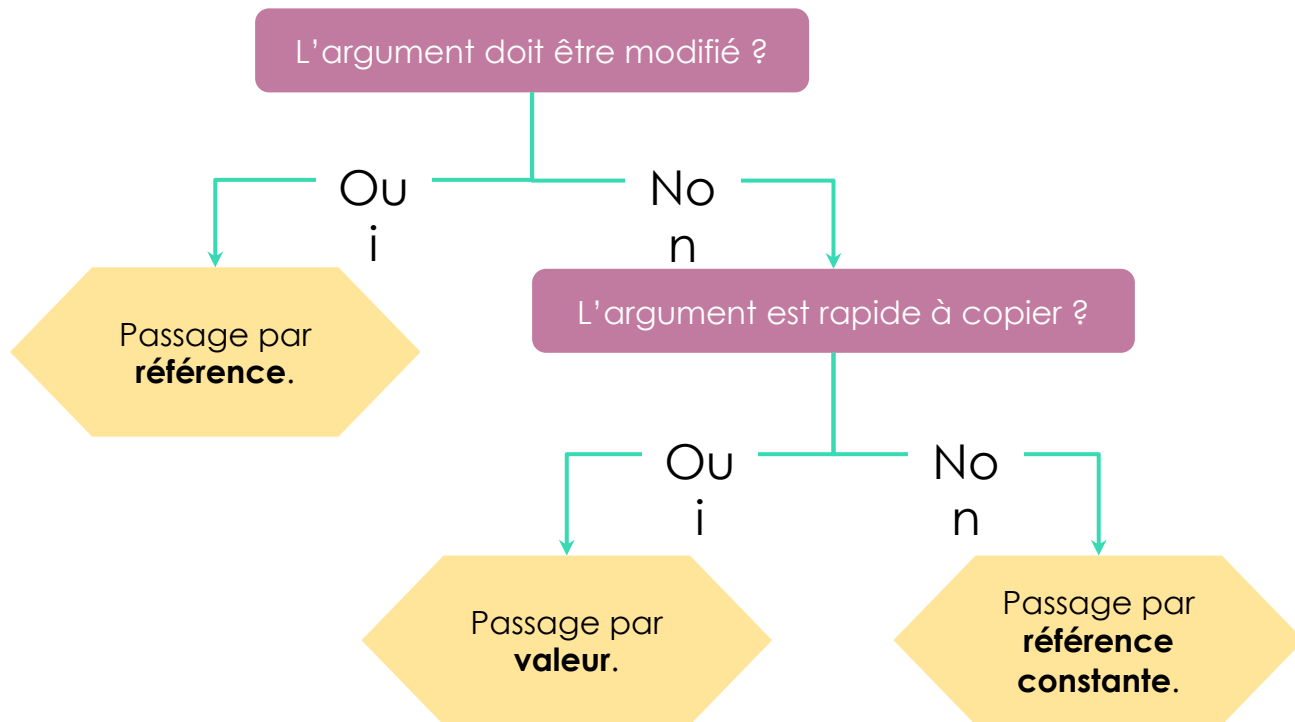
→ On crée un **alias non-mutable** sur l'argument au moment de l'appel.

```
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;
}
```

Passage de paramètres



Sommaire

1. Présentation du module.
2. Hello, World!
3. Types.
4. Fonctions libres.
- 5. Classes.**
 - a. Définir une classe.
 - b. Définir une fonction-membre.
 - c. Définir un constructeur.
 - d. Implémentation par défaut du constructeur par défaut.
 - e. Définir un opérateur de flux ami.

La classe évolution de de la notion de structure

```
#include <iostream>

class point_c {
public:
    double x_=0,y_=0;
};

struct point_s {
    double x_=0,y_=0;
};

int main(void){
    point_c A;
    point_s B;
    A.x_ = 1.0 ;
    B.x_ = 1.0 ;
}
```

```
#include <iostream>

class point_c {
    //public:
    double x_=0,y_=0;
};

struct point_s {
    double x_=0,y_=0;
};

int main(void){
    point_c A;
    point_s B;
    A.x_ = 1.0 ; // ne compile pas!
    B.x_ = 1.0 ;
}
```

Une classe permet de généraliser la notion d'une structure vue en cours de C.

Le C++ introduit la notion d'attributs publics/privés pour donner ou non accès à ses données.

Une classe minimale

Une classe comprend :

- des membres/attributs
- des méthodes

```
Etudiant etu;
```

etu est une instance de la classe Etudiant

Les membres peuvent être :

- public
- private

```
class Etudiant {  
    public:  
    std::string nom_ { "default_name" };  
    int age_ = 99 ;  
};  
  
int main(void){  
    Etudiant etu;  
    etu.nom_ = "new_name";  
    etu.age_ = 21;  
}
```

Erreurs fréquentes

```
#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

Erreurs fréquentes

```
#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

Erreurs fréquentes

```
#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 des fonctions-membres

```
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;
}
```

Une classe intègre des méthodes.
Dans le corps de la méthode on a accès aux autres membres (attributs et méthodes) de la classe.

Définir des fonctions-membres méthodes

```
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;
}
```

On indique à la méthode que l'on ne modifiera pas les attributs de la classe par le mot-clé : const

Une méthode particulière : le constructeur

```
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;
}
```

Instanciation d'une classe se fait à l'aide de la méthode constructeur. Cette méthode ne possède pas d'argument de retour et porte le même que sa classe

Implémentation par défaut du constructeur par défaut

```
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;
}
```

Le mécanisme de surcharge des opérateurs est universel ! On peut l'exploiter pour définir plusieurs syntaxes de d'instanciation.

Inclure une fonction globale

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

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

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

Définir un opérateur de flux ami

```
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;
};
```

Spécifie que la fonction est **amie**.

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


Définir un opérateur de flux ami

```
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)

Fonctions, méthodes et classes amies

En C++, l'encapsulation impose une protection des membres (protected/public). On peut toutefois avoir accès aux membres d'une classe en faisant appel à la notion d'amitié à l'aide du mot clé friend.

Fonctions amies

```
class poly{  
...  
friend void dbg(const std::string& , const poly& );  
...  
};
```

```
void dbg(const std::string& nom, const poly& P){  
    std::cout << nom << " " ;  
    for(auto i=0; i < P.monomes.size() ; ++i){  
        std::cout << P.monomes[i] << " " ;  
    }  
    std::cout << std::endl;  
};
```

```
poly P(10);  
std::vector<double> w {1,2,3} ;  
poly Q( w );  
dbg("toto",Q);
```

Fonctions, méthodes et classes amies

En C++, l'encapsulation impose une protection des membres (protected/public). On peut toutefois avoir accès aux membres d'une classe en faisant appel à la notion d'amitié à l'aide du mot clé friend.

Classes amies

```
class poly{  
...  
friend void dbg(const std::string& , const poly& );  
...  
};
```

```
void dbg(const std::string& nom, const poly& P){  
    std::cout << nom << " " ;  
    for(auto i=0; i < P.monomes.size() ; ++i){  
        std::cout << P.monomes[i] << " " ;  
    }  
    std::cout << std::endl;  
};
```

```
poly P(10);  
std::vector<double> w {1,2,3} ;  
poly Q( w );  
dbg("toto",Q);
```

Opérateurs

En C++, l'encapsulation impose une protection des membres (protected/public). On peut toutefois avoir accès aux membres d'une classe en faisant appel à la notion d'amitié à l'aide du mot clé friend.

Classes amies

```
class poly{
...
friend void dbg(const std::string& , const poly& );
...
};

void dbg(const std::string& nom, const poly& P){
    std::cout << nom << " " ;
    for(auto i=0; i < P.monomes.size() ; ++i){
        std::cout << P.monomes[i] << " " ;
    }
    std::cout << std::endl;
};
```

```
poly P(10);
std::vector<double> w {1,2,3} ;
poly Q( w );
dbg("toto",Q);
```

Accesseurs

Qu'est ce qu'un accesseur ?

Accesseur et mutateur signifient des méthodes pour consulter ou modifier des données d'un objet. Pourquoi introduire cette notion? Rappelez-vous qu'un attribut est déclaré en *private*.... Il nous faut donc une méthode *public* pour définir l'accès.

Mutateur simple (setter)

Cette méthode permet de modifier un attribut .

```
class Point {  
private:  
    double x_ {0}, y_ {0};  
public:  
    Point& SetX(const double x ) {  
        this->x_=x;  
        return *this;}  
    Point& SetY(const double x ) {  
        this->y_=x;  
        return *this; }  
};
```

```
Point A;  
A.SetX(1.0).SetY(2) ;
```

Le setter return une référence sur la propre instance !

Accesseur simple (getter)

Le getter a pour vocation à renvoyer un attribut. ici c'est un renvoie par référence.

```
class Point {  
private:  
    double x_ {0}, y_ {0};  
public:  
    double GetX() {return this->x_;}  
    double GetY() {return this->y_;}  
};
```

```
Point A;  
  
std::cout << A.GetX()+ A.GetY();
```

Accesseur simple amélioré (getter)

Le getter a pour vocation de fournir l'accès/modification de l'instance.

```
class Point {  
private:  
    double x_ {0}, y_ {0};  
public:  
    double& GetX() {return this->x_;}  
    double& GetY() {return this->y_;}  
};
```

```
Point A;  
A.GetX()=3;  
std::cout << A.GetX() << std::endl;  
  
A.GetY()=2;
```

Commentaires

Les getter setter sont importants dans la construction d'une classe. Ils répondent aux besoins d'un accès aux attributs que l'on protège. Leurs corps peuvent aussi agir sur l'instance en redimensionnant de la donnée allouée dynamiquement.

Travaux Pratiques 2

Développer une classe de polynômes avec redimensionnement automatique lorsque l'on accède à un monôme non-alloué...

Surcharge des opérateurs

Qu'est ce que la surcharge des opérateurs ?

Le C++ nous permet d'associer à une classe des opérateurs qui permettront de manipuler des instances d'une classe. Il y a deux intérêts :

1. Cela permet d'écrire du code plus lisible,
2. On se rapproche des opérations définies sur les types primitifs.

Périmètre de la surcharge

On peut surcharger les opérateurs existants, à l'exception de : `::` `*` `.` `?:` `sizeof`

```
int main(void){  
    Complex z {1,1};  
    Complex u ;  
    u = z ; // copy opea  
    v = z*(~z);  
    std :: cout << u << std::endl ;
```

*Dans cet exemple on surcharge les opérateurs, =
, * , ~*

2 techniques de surcharges des opérateurs

Implantation d'opérateurs par une méthode

- opérateurs à seule opérande
- opérateurs à deux opérandes

Implantation d'opérateurs par une fonction globale

- opérateurs à seule opérande
- opérateurs à deux opérandes

```
class T{
    //...
public:
    T operator@() const;
//...
public:
    T operator@() const;
    T operator@( const T&) const;
}

int main(){
    T t;
    T t1,t2; // équivalent à
    t1@t2;    t.operator@();
    // équivalent à
    t1.operator@(t2);
}
```

Illustration sur une classe des nombres complexes

Implantation d'opérateurs par une méthode- 1 opérande

```
class Complex {
private :
    double re_,im_;
public:
    Complex ( ) : re_ {0.}, im_ {0.} {};
    Complex (const double re, const double im ) : re_ {re}, im_ {im} {};
    Complex (const Complex& z):re_{z.re()},im_{z.im()} {};

    double re() {return this->re_};
    const double& re() const {return this->re_};

    double& im() {return this->im_};
    const double& im() const {return this->im_};

    // opérateur unaire
    Complex operator~() const {
        Complex conjugate { re() , -im() };
        return conjugate ;
    };
};
```

```
Complex z {1,1},v;
v = ~z ;
```

Question: `Complex& operator~() const` que se passe t il ?

Implantation d'opérateurs par une méthode- 2 opérandes

```
Complex operator*(const Complex u) const {  
    Complex z;  
    z.re()= this->re_*u.re() - this->im_*u.im();  
    z.im()= this->im_*u.re() + this->re_*u.im() ;  
    return z ;  
};
```

L'appel à l'opérateur est alors simple et devient très lisible !

```
Complex v = z*(~z);
```

Opérateurs définis par fonction globale : 1 opérande

```
class T{  
    //...  
    public:  
        T operator@() const;  
}  
  
int main(){  
    T t;  
    @t;  
    // équivalent à  
    t.operator@();  
}
```

Opérateurs définis par fonction globale : 2 opérandes

```
class T{  
    //...  
public:  
    T operator@() const;  
    T operator@( const T&) const;  
}  
  
int main(){  
    T t1,t2;  
    t1@t2;  
    // équivalent à  
    t1.operator@(t2);  
}
```

Commentaires sur le passage d'arguments

Le choix du mécanisme de passage des arguments d'une méthode ou d'une fonction qui implante un opérateur est important. Les commentaires sur le passage par référence ou par valeur sont aussi valables pour les opérateurs.

Les choix possibles sont: *valeur*, *valeur constante*, *référence*, *référence constante*.

Valeur: ce choix implique une duplication de l'argument d'entrée. Cela isole du reste du code car on travaille sur une copie. Mais le coût de la copie peut être importante.

Valeur constante: ce choix implique une duplication de l'argument d'entrée mais le compilateur vérifie qu'il n'y a pas de modification dans l'argument.

Référence : par référence on transmet à l'opérateur un pointeur. On gagne en temps car moins de données à transférer. Mais attention on peut modifier l'argument dans l'opérateur.

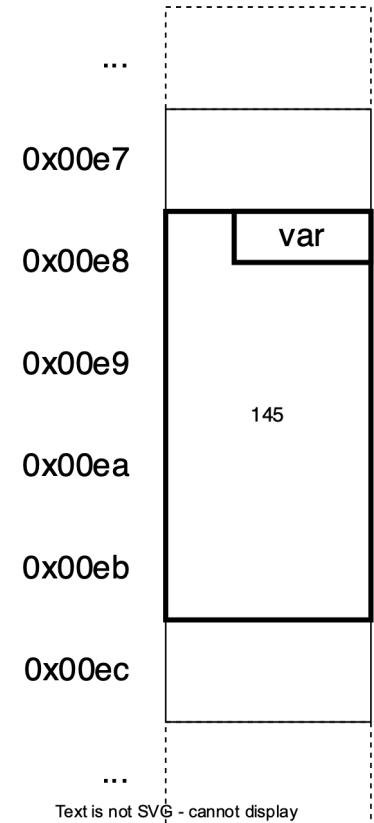
Référence constante: c'est sûrement le mode le plus utilisé, on contrôle à la fois le coût du passage des args et l'accessibilité de la donnée dans le corps de l'opérateur.

Représentation des variables

Une variable est un identifiant qui accède à une donnée de taille fixe en mémoire, située à une adresse précise, encodée sur 64 bits. La mémoire peut être vue comme un tableau d'octets, où une variable est représentée par une série de cases contiguës. Exemple : le code suivant alloue `var` à l'adresse `0x00e8` (en hexadécimal).

```
int var = 145;
```

Comme `var` est de type entier, la zone mémoire s'étend sur 4 octets, c'est-à-dire entre `0x00e8` et `0x00eb`.

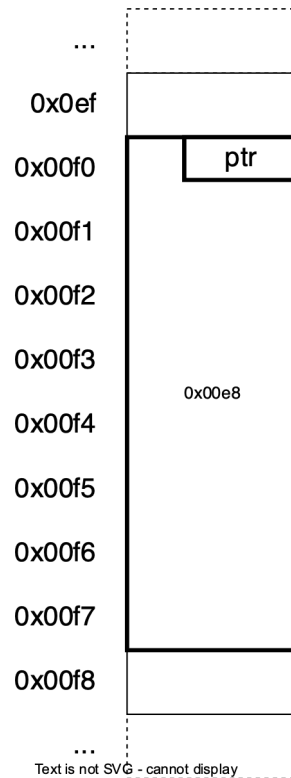


Représentation des pointeurs

Un pointeur est une variable dont le rôle est de stocker l'adresse d'une autre variable. Si les adresses sont encodées sur 64 bits, la taille d'un pointeur est de 8 octets.

```
int* ptr = &var;
```

Comme `var` est de type entier, la zone mémoire s'étend sur 4 octets, c'est-à-dire entre `0x00e8` et `0x00eb`.



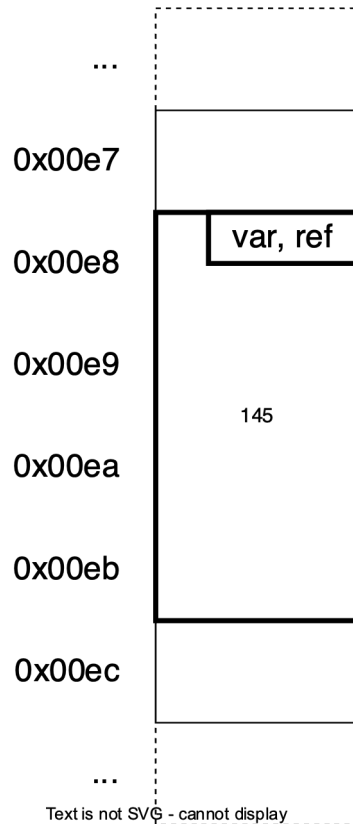
Représentation des références

Une référence est un alias de variable. Elle identifie le même emplacement que la variable d'origine.

```
int& ref = var;
```

ref correspond donc au même bloc que *var*.

Il est représenté en italique pour indiquer que la durée de vie de *var* de la donnée n'est pas couplée à l'identifiant *ref*.

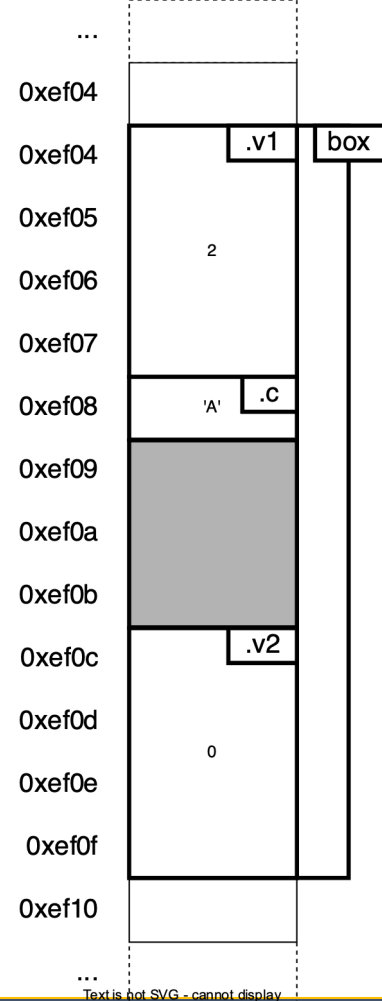


Représentation des structures/classes

Pour les types structurés, les attributs d'un objet sont alloués dans l'espace mémoire de l'objet lui-même.

```
class Box{  
    private:  
    int v1 = 2;  
    char c = 'A';  
    int v2 = 0;  
};
```

Les placement mémoire est géré par le compilateur et dépend de l'architecture cible.



La mémoire statique

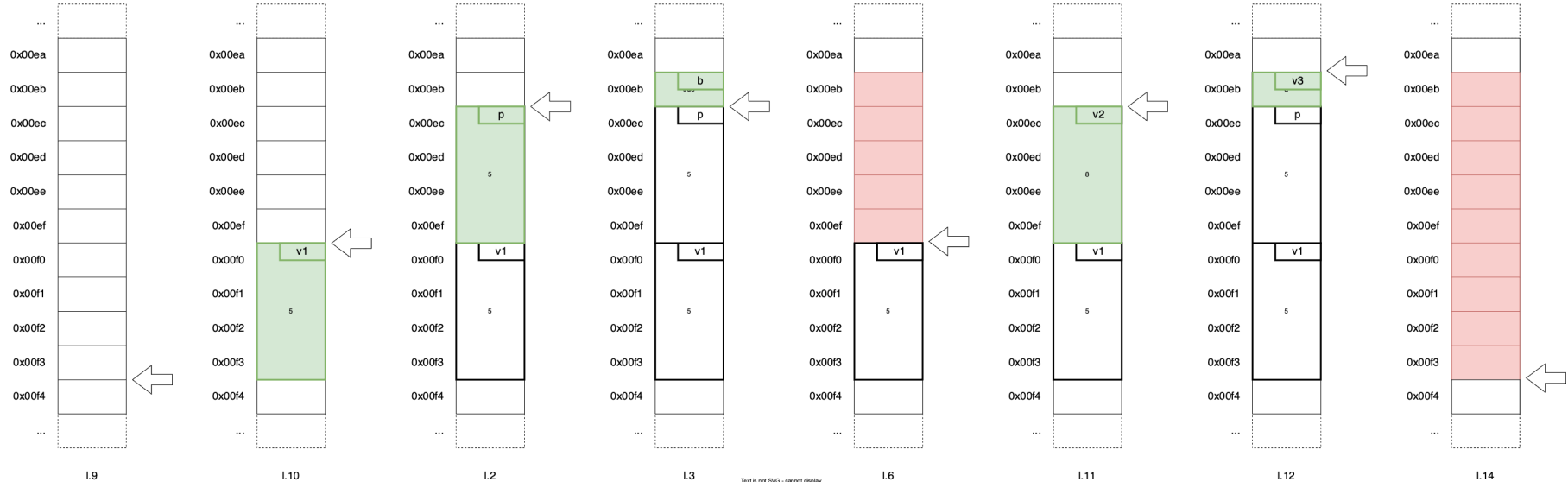
- La mémoire statique stocke les données des variables globales du programme.
- La taille de cette mémoire est déterminée lors de la compilation, en fonction du type des variables.
- L'espace est alloué au lancement du programme et libéré à la fin de ``main``.
- Cet espace ne peut pas être modifié durant l'exécution, d'où le terme "statique".

La pile - heap -

- La pile est l'espace mémoire dédié à la plupart des variables locales.
- Sa taille est limitée (quelques méga-octets, selon le système), mais l'accès et la modification des données y sont très rapides.
- L'allocation est immédiate, car cet espace est réservé dès le démarrage.

```
int f2(int p){  
    auto b = true;  
    return p + 3;}
```

```
void f1(){  
    auto v1 = 5;  
    auto v2 = f2(v1);  
    auto v3 = 'a';}
```



Le tas - stack -

- Les données dynamiques (new) sont allouées sur le tas, ce qui est plus lent que la pile (appel système nécessaire).
- Accès moins rapide : données non regroupées, plus de cache-miss.

• Avantages :

- Allocation flexible : grande quantité de données possible.
- Données persistantes jusqu'à désinstanciation explicite.

```
int* make_int(int value)
{
    int* ptr = new int { value };
    return ptr;
}

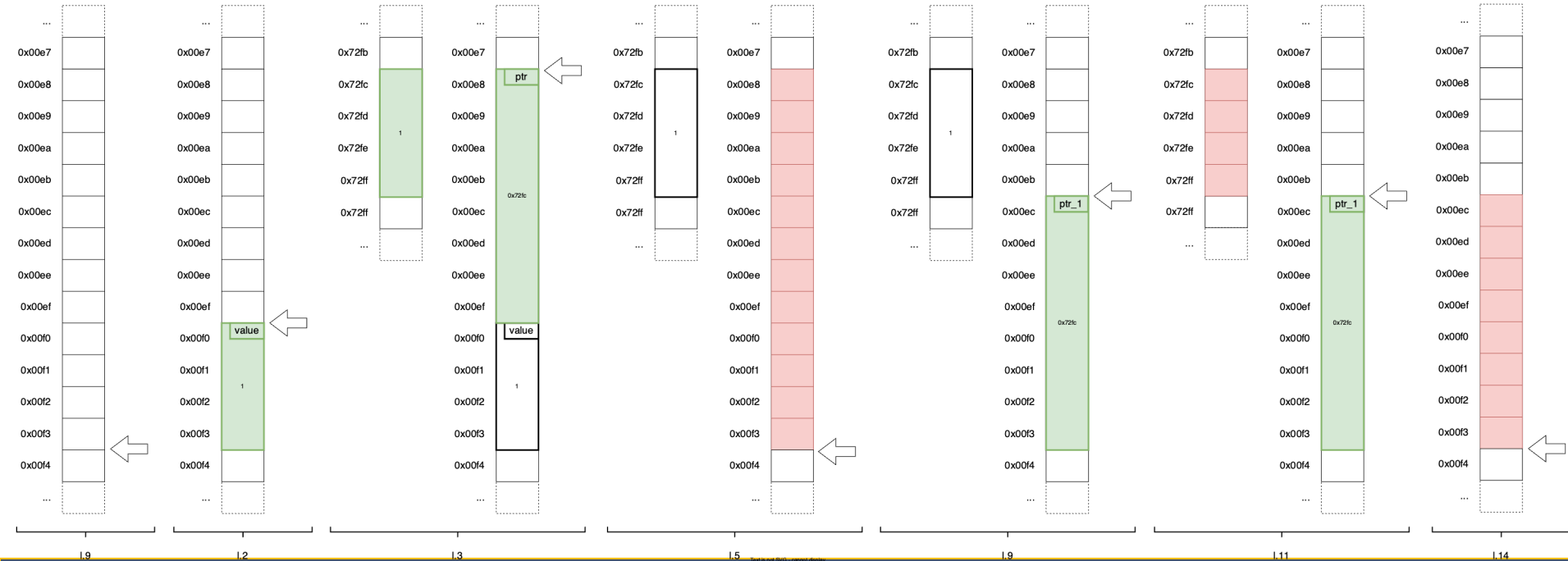
int main()
{
    auto* ptr_1 = make_int(1);
    std::cout << *ptr_1 << std::endl;
    delete ptr_1;

    return 0;
}
```

Le tas - heap -

```
int* make_int(int value)
{
    int* ptr = new int { value };
    return ptr;
}
```

```
int main(){
    auto* ptr_1 = make_int(1);
    std::cout << *ptr_1 << std::endl;
    delete ptr_1;
    return 0;
}
```



Les références... pour aller un petit plus loin

Modifier un argument

Le passage des arguments C/C++ se fait par copie. Les deux seules approches pour modifier un argument sont les pointeurs et les références.

```
void inc(int* a){++(*a);};  
void inc(int& a){++( a);};  
int main() {  
    int var = 10;  
    inc(&var);  
    std::cout << var << std::endl;  
    inc(var);  
    std::cout << var << std::endl;  
    Return 0;  
}
```


Eviter la copie de variables

```
void fonction(int* tableau, int taille) {  
    for (int i = 0; i < taille; ++i) {  
        tableau[i] = tableau[i] * 2;  
    }  
}  
int main() {  
    int monTableau[5] = {1, 2, 3, 4, 5};  
    fonction(monTableau, 5);  
    return 0;}
```

```
void fonction(int (&tableau)[5]) {  
    for (int i = 0; i < 5; ++i) {  
        tableau[i] = tableau[i] * 2;  
    }  
}  
int main() {  
    int monTableau[5] = {1, 2, 3, 4, 5};  
    fonction(monTableau);  
    return 0; }
```

Les tableaux pouvant être de grandes tailles on en évite les copies lors du passage dans le corps des fonctions.

Eviter la copie des valeurs de retour

```
class Person{
public:
    Person(const std::string& name): _name { name }{}
    const std::string& get_name() const { return _name; }
private:
    std::string _name;
};

auto bob = Person {" bob" };
const auto& name = donatien.get_name();
std::cout << name << " is my favorite person in the world" <<
std::endl;
```

On retrouve ici nos getter/setter rencontrés précédemment.

Agrégation vs Composition en C++

En C++, l'agrégation et la composition sont deux types de relations entre classes qui indiquent différentes formes de "has-a" (possession) entre objets.

Agrégation

L'agrégation est une forme de relation "has-a" où un objet contient ou utilise un autre objet, mais les deux objets peuvent exister indépendamment l'un de l'autre. En d'autres termes, l'objet contenu peut vivre même si l'objet conteneur est détruit. L'agrégation est souvent représentée par un membre de type **pointeur** ou **référence** dans une classe.

Agrégation version pointeur ou référence

```
#include <iostream>
class A {
public:
void f(){cout<<"A::f()"<<endl;}
};
class B {
private:
    A* a; // Agrégation
public:
B(A* ptr) : a(ptr) {}
void g() { a->f(); }
};
```

```
A objA;
B objB(&objA); // B utilise un objet A
objB.g();
```

```
#include <iostream>
class A {
public:
void f(){cout<<"A::f()"<<endl;}
};
class B {
private:
    A& a; // Agrégation
public:
B(A* ptr) : a(ptr) {}
void g() { a.f(); }
};
```

```
A objA;
B objB(objA); // B utilise un objet A
objB.g();
```

Composition de classes

```
class C {  
public:  
    void f(){cout << "C::f()" << endl;}  
};
```

```
class D {  
private:  
    C c;  
public:  
    void g() { c.f(); }  
};
```

```
D objD; // C est automatiquement créé avec D  
objD.g();
```

Composition

La composition est une forme **plus forte de** relation "has-a", où l'objet contenu ne peut pas exister sans l'objet conteneur. Si l'objet conteneur est détruit, l'objet contenu est également détruit. En composition, l'objet contenu est souvent créé et détruit par l'objet conteneur, ce qui signifie qu'il n'a pas de sens en dehors de ce conteneur.

Agrégation vs Composition de classes

➤ Durée de vie :

- **Agrégation** : L'objet contenu peut survivre à l'objet conteneur.
- **Composition** : L'objet contenu est détruit lorsque l'objet conteneur est détruit.

➤ Gestion de la mémoire :

- **Agrégation** : L'objet contenu est souvent passé par pointeur ou référence. Il est créé à l'extérieur de l'objet conteneur et peut être partagé entre plusieurs objets.
- **Composition** : L'objet contenu est généralement une instance directe. Il est géré entièrement par l'objet conteneur.

➤ Relation d'indépendance :

- **Agrégation** : Les objets peuvent exister indépendamment.
- **Composition** : L'existence de l'objet contenu dépend de l'objet conteneur.

Sommaire

1. Copie.
2. Déplacement.
3. L-Value et R-Value.
4. Pointeurs ownants.
- 5. Héritage.**
 - a. Syntaxe
 - b. Instance d'une classe dérivée
6. Classes polymorphes.

Héritage

L'héritage en C++ est un mécanisme de programmation orientée objet qui permet à une classe (appelée classe dérivée ou classe-fille) d'hériter des propriétés et des comportements (c'est-à-dire des membres de données et des méthodes) d'une autre classe (appelée classe de base ou classe mère). L'héritage permet de réutiliser du code existant, de créer des hiérarchies de classes, et d'établir des relations "est-un" (is-a) entre les classes.

On introduit ici notre motivation à introduire la notion d'héritage. Pour cela on considère une classe de vecteur 3d.

```
class Vecteur3{
private:
double x_,y_,z_ ;
public:
Vecteur3(double x=0, double y=0, double z=0) ;
// accesseurs
double x() const ;
double y() const ;
double z() const ;
// Opérateurs
friend Vecteur3& operator+=(Vecteur3& v1, const Vecteur3& v2) ;
friend Vecteur3 operator+(const Vecteur3& v1, const Vecteur3& v2) ;
friend Vecteur3 operator-(const Vecteur3& v1, const Vecteur3& v2) ;
double norme() const ;
// Output
ostream& print(ostream& os) const ;};
ostream& operator<<(ostream&, const Vecteur3& v) ;
```

Les méthodes de cette classe sont implémentées par le source suivant:

```

Vecteur3::Vecteur3(double x , double y , double z ){x_ = x ; y_ = y ; z_ = z ;}

double Vecteur3::x() const { return x_ ; }
double Vecteur3::y() const { return y_ ; }
double Vecteur3::z() const { return z_ ; }

Vecteur3& operator+=( Vecteur3& v1, const Vecteur3& v2){
    v1.x_ += v2.x_ ; v1.y_ += v2.y_ ; v1.z_ += v2.z_ ;
    return v1 ;
}
Vecteur3 operator+(const Vecteur3& v1, const Vecteur3& v2){

Vecteur3 v(v1) ;
    return v += v2 ;
}

Vecteur3 operator-(const Vecteur3& v1, const Vecteur3& v2){
    return Vecteur3(v1.x_-v2.x_, v1.y_-v2.y_, v1.z_-v2.z_) ; }

double Vecteur3::norme() const{ return sqrt(x_*x_ + y_*y_ + z_*z_) ;}

ostream& Vecteur3::print(ostream& os) const {return os<<x_<<" "<<y_<<" "<<z_ ;}

ostream& operator<<(ostream& os, const Vecteur3& v){return v.print(os) ;}

```

Question: Comment réutiliser la classe vecteur ?
Par exemple on souhaite ajouter un attribut *nom_*

Réponse 1: on recopie le code

```
class Vecteur3N{  
private:  
    double x_,y_,z_;  
    char nom_[20];  
public:  
    //....  
}
```

Réponse 2: composition

```
class Vecteur3N{  
private:  
    char nom_[20];  
public:  
    Vecteur3 v_;  
}
```

R1: On duplique le code. C'est à proscrire à cause de la maintenabilité.

R2: On perd la **relation sémantique** entre les classes *Vecteur3N* et *Vecteur3*.

```
Vecteur3 u;  
Vecteur3N v;  
cout << u+v << endl;  
v.v_.norm();
```

u+v n'a pas de sens
v.v_.norm(); est un peu lourd à écrire.

Héritage public

La solution offerte par le C++ est l'héritage. On dit que `Vecteur3N` hérite publiquement de `Vecteur3`.

```
class Vecteur3N : public Vecteur3 {  
private:  
    char nom_[20];  
public:  
    Vecteur3N(const char *nom, double x=0 , double y=0, double z=0);  
  
};
```

Hérite ainsi des composantes publiques.

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

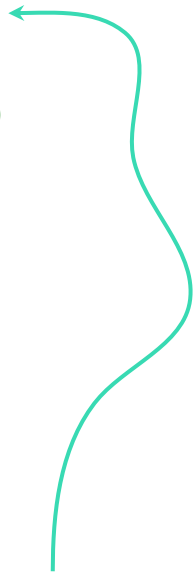
    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```



toute instance de Derived peut être
considérée comme une instance de Base

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

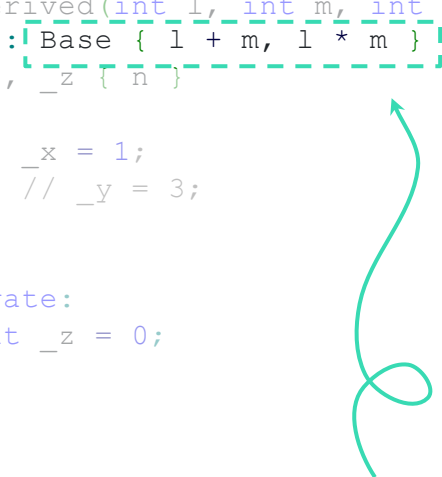
    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```



permet d'appeler le
constructeur de la classe-parente

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

permet l'accès aux attributs
depuis les instances-filles

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }
}
```

```
protected:
    int _x = 0;
```

```
private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

accès
valide

Syntaxe

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

accès
invalide

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

Instance d'une classe dérivée

On peut ensuite référencer les instances du type-enfant par le type-parent.

```
int main()
{
    auto derived = Derived { ... };
    Base& ref_base = derived;

    return 0;
}
```

```
void fcn(const Base& base)
{
    ...
}

int main()
{
    auto derived = Derived { ... };
    fcn(derived);

    return 0;
}
```

Instance d'une classe dérivée

On peut ensuite référencer les instances du type-enfant par le type-parent.

```
int main()
{
    auto derived = Derived { ... };
    Base& ref_base = derived;
    -----
    return 0;
}
```

derived peut être référencé par
son type parent Base

```
void fcn(const Base& base)
{
    -----
    ...
}
```

```
int main()
{
    auto derived = Derived { ... };
    fcn(derived);

    return 0;
}
```

Instance d'une classe dérivée

Cela fonctionne aussi avec des pointeurs-**observants**.

```
int main()
{
    auto derived = Derived { ... };
    Base* ref_base = &derived;

    return 0;
}
```

```
void fcn(const Base* base)
{
    ...
}

int main()
{
    auto derived = Derived { ... };
    fcn(&derived);

    return 0;
}
```

Instance d'une classe dérivée

Cela fonctionne aussi avec des pointeurs-**observants**.

```
int main()
{
    auto derived = Derived { ... };
    Base* ref_base = &derived;
    -----
    return 0;
}
```

```
void fcn(const Base* base)
{
    -----
    ...
}
```

```
int main()
{
    auto derived = Derived { ... };
    fcn(&derived);

    return 0;
}
```

Derived* est convertible en Base*

Instance d'une classe dérivée

On peut appeler les fonctions publiques du type-parent sur les instances-filles.

```
int main()
{
    auto derived = Derived { ... };
    std::cout << derived.get_y() << std::endl;


    return 0;
}
```

Instance d'une classe dérivée

On peut appeler les fonctions publiques du type-parent sur les instances-filles.

```
int main()
{
    auto derived = Derived { ... };
    std::cout << derived.get_y() << std::endl;

    return 0;
}
```



`get_y()` est définie dans la partie publique de `Base`,
donc on peut l'appeler sur une instance de `Derived`

Sommaire

1. Copie.
2. Déplacement.
3. L-Value et R-Value.
4. Pointeurs ownants.
5. Héritage.
- 6. Classes polymorphes.**
 - a. Définition
 - b. Redéfinir le comportement d'une classe
 - c. Résolution d'appels
 - d. Fonctions virtuelles pures

Définition

En C++, l'héritage permet de répondre à 2 besoins orthogonaux :

- éviter la duplication de code
- spécialiser un comportement

Définition

En C++, l'héritage permet de répondre à 2 besoins orthogonaux :

- éviter la duplication de code
- spécialiser un comportement

Une classe dont on a pu **redéfinir le comportement** via héritage est une classe dont les instances peuvent se comporter différemment selon le **type dynamique** de l'objet.

On parle de **classes polymorphes**.

Redéfinir le comportement d'une classe

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};
```

Redéfinir le comportement d'une classe

```
class Instrument
{
public:
    [virtual] std::string get_name() const
    {
        return "???";
    }

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};
```

indique que la fonction peut-être redéfinie par les classes-filles



Redéfinir le comportement d'une classe

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
class Guitar: public Instrument
{
public:
    std::string get_name() const override
    {
        return "guitar";
    }
};
```

Redéfinir le comportement d'une classe

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
class Guitar: public Instrument
{
public:
    std::string get_name() const override
    {
        return "guitar";
    }
};
```



demande au compilateur de vérifier
que la fonction est bien virtuelle

Redéfinir le comportement d'une classe

```
int main()
{
    Piano piano;
    Guitar guitar;

    std::vector<Instrument*> instruments {
        &piano, &guitar };

    for (const auto* instrument: instruments)
    {
        std::cout << instrument->get_name() <<
std::endl;
    }
}
```


Redéfinir le comportement d'une classe

```
int main()
{
    Piano piano;
    Guitar guitar;

    std::vector<Instrument*> instruments {
        &piano, &guitar };

    for (const auto* instrument: instruments)
    {
        std::cout << instrument->get_name() <<
        std::endl;
    }
```

comme describe est virtuelle, on appelle la redéfinition
contenue dans le **type dynamique** de chaque instance

Redéfinir le comportement d'une classe

```
int main()
{
    Piano piano;
    Guitar guitar;

    std::vector<Instrument*> instruments {
        &piano, &guitar };

    for (const auto* instrument: instruments)
    {
        std::cout << instrument->get_name() <<
        std::endl;
    }
```

piano

guitar

comme describe est virtuelle, on appelle la redéfinition
contenue dans le **type dynamique** de chaque instance

Résolution d'appels

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

Résolution d'appels

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
get_name() const
    {
        return "???" ;
    }
};
```

```
class Piano: public
Instrument
{
public:
    std::string get_name()
const
    {
        return "piano";
    }
};
```

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
get_name() const
    {
        return "???" ;
    }
};
```

fonction
virtuelle

```
class Piano: public
Instrument
{
public:
    std::string get_name()
const
    {
        return "piano";
    }
};
```

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
get_name() const
    {
        return "???" ;
    }
};
```

fonction
virtuelle

```
class Piano: public Instrument
{
public:
    std::string get_name()
const
    {
        return "piano";
    }
};
```

donc virtuelle
aussi

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
    get_name() const
    {
        return "???" ;
    }
};
```

fonction
virtuelle

! Attention !
aux signatures

```
class Piano: public Instrument
{
public:
    std::string get_name()
    {
        return "piano";
    }
};
```


Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
    get_name() const
    {
        return "???" ;
    }
};
```

fonction
virtuelle



! Attention !
aux signatures

```
class Piano: public Instrument
{
public:
    std::string get_name()
    {
        return "piano" ;
    }
};
```

fonction non
virtuelle !

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
    get_name() const
    {
        return "???" ;
    }
};
```

fonction
virtuelle



! Attention !
aux signatures

```
class Piano: public Instrument
{
public:
    std::string
    get_name()
    {
        return "piano";
    }
};
```

fonction non
virtuelle !

! BUG OBSCUR !

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
    get_name() const
    {
        return "???" ;
    }
}
```

toujours mettre `override` pour que le compilateur nous prévienne si on se trompe dans la signature

! Attention !
aux signatures

```
class Piano: public Instrument
{
public:
    std::string get_name()
    override
    {
        return "piano" ;
    }
}
```

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string
    get_name() const
    {
        return "???" ;
    }
}
```

toujours mettre `override` pour que
le compilateur nous prévienne si on se
trompe dans la signature

! Attention !
aux signatures

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano" ;
    }
}
```

ERREUR DE COMPILATION

Résolution d'appels

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

Résolution d'appels

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

Résolution d'appels

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
int main()
{
    Pian type statique
    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on résout l'appel à `get_name()`



Résolution d'appels

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

fonction non virtuelle

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
int main()
{
    Pian type statique

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on résout l'appel à `get_name()`

Résolution d'appels

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

fonction non
virtuelle

```
int main()
{
    Pian type statique

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

on réalise un **appel statique**

Résolution d'appels

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

???

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

Résolution d'appels

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
int main()
{
    Pian type statique

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on résout l'appel à `get_name()`



Résolution d'appels

fonction
virtuelle

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on résout l'appel à `get_name()`

Résolution d'appels

fonction
virtuelle

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

int type dynamique

```
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on réalise un **appel dynamique**

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

fonction
virtuelle

```
int type dynamique
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

on réalise un **appel dynamique**

Résolution d'appels

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```



piano

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

Résolution d'appels

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

Résolution d'appels

```
class Instrument
{
public:
    ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

Résolution d'appels

```
class Instrument
{
public:
    ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};
```

```
class Piano: public Instrument
{
public:
    ~Piano() { std::cout << "piano destroyed" << std::endl; }
};
```

```
int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

type statique

on résout l'appel à ~Instrument()

Résolution d'appels

```
class Instrument
{
public:
    ~Instrument() { std::cout << "??? destroyed" << std::endl; }
```

fonction non
virtuelle

```
class Piano: public Instrument
{
public:
    ~Piano() { std::cout << "piano destroyed" << std::endl; }
```

```
int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

type statique

on résout l'appel à ~Instrument()

Résolution d'appels

```
class Instrument
{
public:
    ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};


class Piano: public Instrument
{
public:
    ~Piano() { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

fonction non virtuelle

type statique



on réalise un **appel statique**


Résolution d'appels

```
class Instrument
{
public:
    ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```



??? destroyed

Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

on utilise maintenant un destructeur virtuel



Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

type statique

on résout l'appel à ~Instrument()

Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

fonction virtuelle

type statique

on résout l'appel à ~Instrument()

Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

fonction
virtuelle

type dynamique

on réalise un **appel dynamique**

Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```

fonction virtuelle

type dynamique

on réalise un **appel dynamique**

Résolution d'appels

```
class Instrument
{
public:
    virtual ~Instrument() { std::cout << "??? destroyed" << std::endl; }
};

class Piano: public Instrument
{
public:
    ~Piano() override { std::cout << "piano destroyed" << std::endl; }
};

int main()
{
    std::unique_ptr<Instrument> piano_as_instrument = std::make_unique<Piano>();
    piano_as_instrument = nullptr;

    return 0;
}
```



piano destroyed

Résolution d'appels

Pour garantir qu'un objet **polymorphe** sera **correctement détruit**, en particulier dans le cas d'**allocations dynamiques**, il faut donc penser à définir un **destructeur virtuel** dans la classe-mère (même s'il ne fait "rien").

Fonctions virtuelles pures

Si une fonction n'a pas de sens à être définie dans la classe-mère, il n'est pas nécessaire de lui fournir une implémentation. On parle de **fonctions virtuelles pures**.

Si une classe contient des fonctions virtuelles pures, elle devient **abstraite** et n'est plus instanciable.

Les classes-filles doivent **redéfinir toutes les fonctions virtuelles pures** des types-parents pour **pouvoir être instanciées**.

Fonctions virtuelles pures

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};

int main()
{
    Instrument instrument;

    return 0;
}
```


Fonctions virtuelles pures

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};

int main()
{
    Instrument instrument;

    return 0;
}
```

définit une fonction **virtuelle pure**



Fonctions virtuelles pures

```
class Instrument
```

```
{
```

```
public:
```

```
    virtual std::string get_name() const [= 0;]
```

```
    void describe() const
```

```
{
```

```
    std::cout << "This is a " << get_name() << std::endl;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
    Instrument instrument;
```

```
    return 0;
```

```
}
```

Instrument est donc **abstraite**

Fonctions virtuelles pures

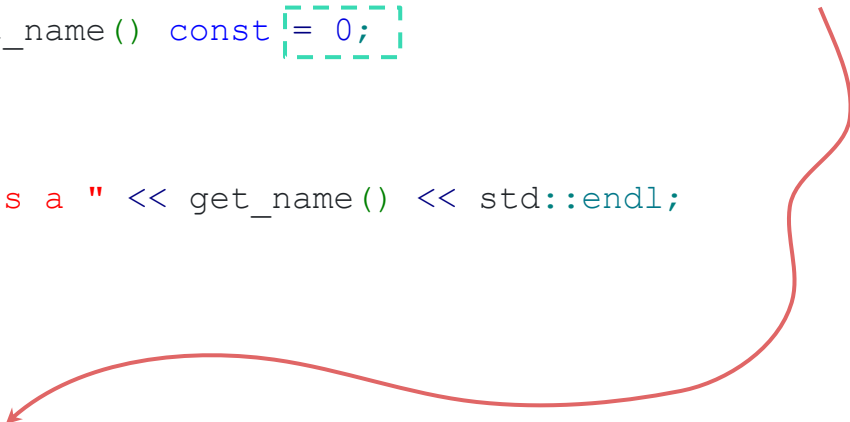
```
class Instrument
{
public:
    virtual std::string get_name() const [= 0];

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};

int main()
{
    Instrument instrument;

    return 0;
}
```

Instrument est donc **abstraite**
... et n'est plus instanciable



Fonctions virtuelles pures

```
class Instrument
{
public:
    virtual std::string get_name() const [= 0];

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};

int main()
{
    Instrument i;
    i.get_name();
}
```

Instrument est donc **abstraite**
... et n'est plus instanciable

! ERREUR DE COMPILATION !