

Cours 2

Ownership & Héritage

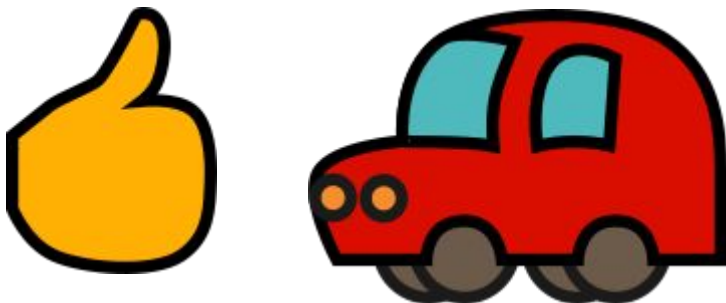
C++ - Master 1

1. Durée de vie
2. Ownership
3. Pointeurs intelligents
4. Héritage
5. Classes polymorphes

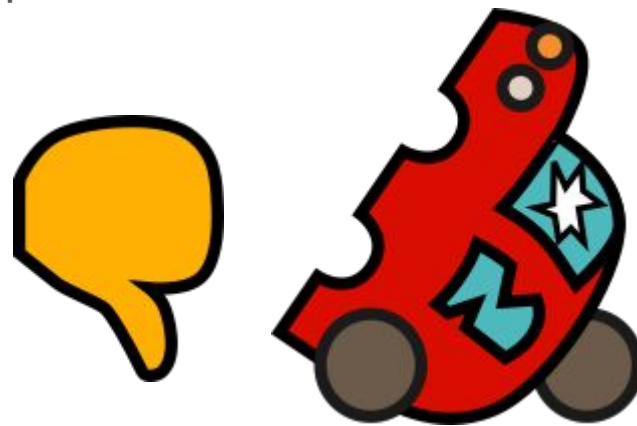
La durée de vie, c'est quoi ?

C'est la période pendant laquelle il est valide d'accéder à un élément ou de manipuler.

Si la durée de vie de la voiture n'est pas écoulée, on peut rentrer dedans et la conduire sans problème.



Si la durée de vie de la voiture est terminée, il est dangereux de s'en approcher.



Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
        ...  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
        ...  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

durée de vie de a

Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
        ...  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

durée de vie de b

Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

durée de vie de c

Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
        ...  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

durée de vie de i

Lorsqu'une variable **locale** est instanciée sur la **pile**, sa durée de vie correspond à sa **portée**.

```
void fcn()  
{  
    auto a = 1;  
    if (...)  
    {  
        auto b = 2;  
        ...  
        auto c = 3;  
        ...  
    }  
    else  
    {  
        for (auto i = 0; i < 4; ++i)  
        {  
            ...  
        }  
        auto d = 5;  
    }  
}
```

durée de vie de d

La durée de vie de la valeur de retour d'une fonction dépend du **type de la valeur de retour**.

Cas n°1 : retour par **valeur**

⇒ la durée de vie de la valeur retournée s'achève à la **fin de l'instruction contenant l'appel**

```
std::string pouet()  
{  
    return "Pouet";  
}
```

```
int main()  
{  
    std::cout << pouet() << std::endl;  
    return 0;  
}
```

durée de vie de la valeur de
retour de pouet()

Cas n°1 : retour par **valeur**

⇒ la durée de vie de la valeur retournée s'achève à la **fin de l'instruction contenant l'appel**

MAIS on peut stocker la valeur dans une nouvelle variable locale !

```
std::string pouet()  
{  
    return "Pouet";  
}
```

```
int main()  
{  
    const auto p = pouet();  
    std::cout << p << std::endl;  
    return 0;  
}
```

durée de vie de la valeur de
retour de pouet()

Cas n°1 : retour par **valeur**

⇒ la durée de vie de la valeur retournée s'achève à la **fin de l'instruction contenant l'appel**

MAIS on peut stocker la valeur dans une nouvelle variable locale !

```
std::string pouet()  
{  
    return "Pouet";  
}
```

```
int main()  
{  
    const auto p = pouet();  
    std::cout << p << std::endl;  
    return 0;  
}
```

durée de vie de p

Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

```
std::string& pouet(std::string& s)
{
    s += "Pouet";
    return s;
}

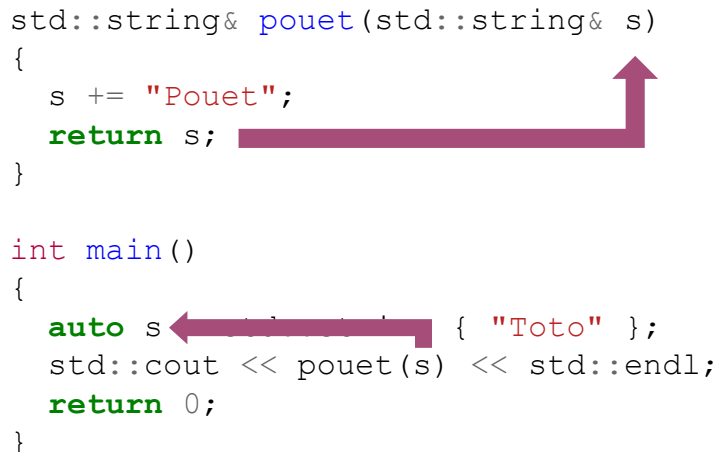
int main()
{
    auto s = std::string { "Toto" };
    std::cout << pouet(s) << std::endl;
    return 0;
}
```

Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

```
std::string& pouet(std::string& s)
{
    s += "Pouet";
    return s;
}

int main()
{
    auto s ← { "Toto" };
    std::cout << pouet(s) << std::endl;
    return 0;
}
```



Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

```
std::string& pouet(std::string& s)
{
    s += "Pouet";
    return s;
}

int main()
{
    auto s = std::string { "Toto" };
    std::cout << pouet(s) << std::endl;
    return 0;
}
```

durée de vie de s
= durée de vie de la valeur
retournée par pouet

Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

Attention aux dangling-references !!

```
std::string& pouet()  
{  
    auto s = std::string { "Pouet" };  
    return s;  
}  
  
int main()  
{  
    std::cout << pouet() << std::endl;  
    return 0;  
}
```

Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

Attention aux dangling-references !!

```
std::string& pouet()  
{  
    auto s = std::string { "Pouet" };  
    return s;  
}
```

durée de vie de la valeur de
retour de pouet

```
int main()  
{  
    std::cout << pouet() << std::endl;  
    return 0;  
}
```



accès invalide !

Cas n°2 : retour par **référence**

⇒ la durée de vie de la valeur retournée est **identique** à la durée de vie de l'**élément référencé**

Attention aux dangling-references !!

```
std::string& pouet()  
{  
    auto s = std::string { "Pouet" };  
    return s;  
}
```

```
int main()  
{  
    auto p = pouet();  
    return 0;  
}
```

tenter de stocker le contenu d'une dangling-reference constitue aussi un accès invalide !

L'ownership, c'est une notion assez...



L'ownership, c'est quoi ?

L'ownership, c'est une notion assez...

Hummm...



L'ownership, c'est quoi ?

L'ownership, c'est une notion assez...

Hummm...

ABSTRAITE !



L'ownership, c'est une notion assez...

Hummm...

ABSTRAITE !

C'est pas vraiment bien défini où que ce soit en fait...



Inventons un truc du coup...

On va dire qu'une *entité* A est **owner** (ou propriétaire) d'une *entité* B si A est chargée de **définir la durée de vie** de B.

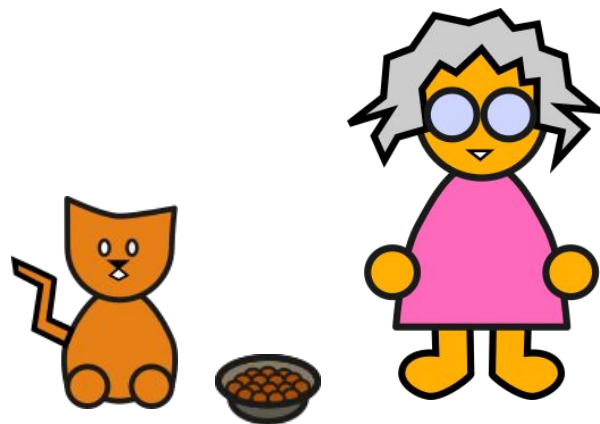


Exemple 1 :

Mamie nourrit Toto le chat.

Si Mamie arrête de nourrir Toto, bah Toto, il... voilà quoi.

Donc **Mamie est owner de Toto.**



Exemple 2 :

Ma Tesla-Starlink-Collector-Edition a quatre roues.

Si ma Tesla décide de s'auto-incendier (grâce à sa super IA), ses roues périront sans aucun doute dans les flammes de l'enfer également.

Donc ma **Tesla est owner de ses roues**.



Cas n°1 : attribut d'une classe

- `tesla_1` est owner du contenu de `tesla_1._ai`
- `tesla_2` est owner du contenu de `tesla_2._ai`

```
class Tesla
{
public:
    ...

private:
    AI _ai;
    ...
};

auto tesla_1 = Tesla {};
auto tesla_2 = Tesla {};
```

Cas n°2 : mémoire allouée dans un objet

- `vec` est owner de l'espace mémoire qu'il a alloué pour stocker les valeurs 1, 2 et 3
- `vec` est owner du contenu de `v1`
- `str` est owner de l'espace mémoire qu'il a alloué pour la chaîne "toto"
- `str` est owner du contenu de `s1`

```
auto vec = std::vector { 1, 2, 3 };  
auto& v1 = vec[0];  
  
auto str = std::string { "toto" };  
auto& s1 = str[0]
```

Cas n°3 : variables locales à une fonction

- `main` est owner du contenu de `car`
- `main` est owner du contenu de `driver`
- `main` est owner du contenu de `driver._car`
- `driver` **n'est pas owner** du contenu de `driver._car`

```
class Driver
{
public:
    Driver(Car& car) : _car { car }
    {}

private:
    Car& _car;
};

int main()
{
    auto car      = Car {};
    auto driver = Driver { car };
    return 0;
}
```

Si A est owner de B et que B est owner de C, alors A est indirectement owner de C.

- `main` est owner du contenu de `mamie`
- `mamie` est owner du contenu de `mamie._cat`
- `main` est donc aussi owner du contenu de `mamie._cat`

- `mamie._cat` est owner du contenu de `mamie._cat._name`
- `mamie` est donc aussi owner du contenu de `mamie._cat._name`
- `main` est donc aussi owner du contenu de `mamie._cat._name`

```
class Cat
{
private:
    std::string _name = "Toto";
};

class Mamie
{
private:
    Cat _cat;
};

int main()
{
    auto mamie = Mamie {};
    return 0;
}
```

Sous certaines conditions, il est possible de transférer l'ownership d'une *ressource* (mémoire allouée, descripteur de fichier, etc), en utilisant la fonction `std::move`.

```
auto v1 = std::vector { 1, 2, 3 };
```

```
auto v2 = std::move(v1);
```

```
auto s1 = std::string { "toto" };
```

```
auto s2 = std::move(s1);
```

```
auto f1 = std::fstream { "file.txt" };
```

```
auto f2 = std::move(f1);
```

Maîtriser la notion d'ownership dans son code a deux intérêts :

1. Prévenir l'accès à des éléments dont la durée de vie est écoulée.



2. Eviter les fuites mémoires.



C'est un pointeur qui, lorsque sa durée de vie arrive à échéance, libère automatiquement la mémoire qu'il own. Un peu comme `std::string` ou `std::vector`.

`smart_ptr` libère le segment mémoire
⇒ pas de fuite



`raw_ptr` ne libère le segment mémoire
⇒ fuite mémoire



```
void memory(bool smart)
{
    if (smart)
    {
        auto smart_ptr = std::make_unique<int>(3);
        std::cout << *smart_ptr << std::endl;
    }
    else
    {
        auto raw_ptr = new int { 3 };
        std::cout << *raw_ptr << std::endl;
    }
}
```

C'est le pointeur intelligent le plus courant.

- `std::make_unique<type>` pour allouer la mémoire et créer le `unique_ptr`
- la copie **n'est pas possible** : “unique” signifie qu'on autorise qu'un seul owner
- `std::move` pour déplacer le `unique_ptr` si besoin est
- disponible dans `<memory>`

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

tmp_car est maintenant vide, sa valeur
a été transférée dans many_cars

C'est un pointeur qui permet de gérer un ownership partagé.

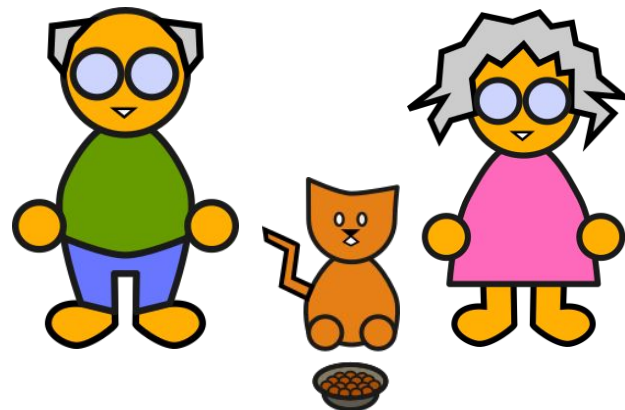
- `std::make_shared<type>` pour allouer la mémoire et créer le `shared_ptr`
- la copie permet d'étendre l'ownership à une nouvelle entité
- disponible dans `<memory>`

```
int main()
{
    auto mamie = std::make_unique<Mamie>();
    mamie->cat = std::make_shared<Cat> {};

    auto papi = std::make_unique<Papi>();
    papi->cat = mamie->cat;

    mamie = nullptr; // the cat is still alive
    papi = nullptr; // the cat's memory is freed

    return 0;
}
```



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

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

```
protected:
    int _x = 0;

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

```
class B : public A
{
public:
    B(int l, int m, int n) : A { l+m, l*m }, _z { n },
    {
        // _x = 1; --> private dans A, donc inaccessible depuis B
        _y = 3;
    }

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

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

    int get_y() const { return _y; }

protected:
    int _x = 0;

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

chaque instance de B peut être
considérée comme un A

```
class B : public A
{
public:
    B(int l, int m, int n) : A { l+m, l*m }, _z { n },
    {
        // _x = 1; --> private dans A, donc inaccessible depuis B
        _y = 3;
    }

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



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

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

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

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

```
class B : public A
{
```

```
public:
```

```
    B(int l, int m, int n) : A { l+m, l*m }, _z { n },
```

```
    {
        // _x = 1; --> private dans A, donc inaccessible depuis B
        _y = 3;
    }
```

```
private:
```

```
    int _z = 0;
};
```

pour appeler le constructeur de
A depuis le constructeur de B

```
class A
```

```
{
```

```
public:
```

```
  A(int
```

```
  {}
```

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

pour définir des membres accessibles
depuis les classes-filles

```
protected:
```

```
  int _x = 0;
```

```
private:
```

```
  int _y = 0;
```

```
};
```

```
class B : public A
```

```
{
```

```
public:
```

```
  B(int l, int m, int n) : A { l+m, l*m }, _z { n },
```

```
  {
```

```
    // _x = 1; --> private dans A, donc inaccessible depuis B
```

```
    _y = 3;
```

```
  }
```

```
private:
```

```
  int _z = 0;
```

```
};
```

On peut ensuite référencer les instances de type B avec le type A.

```
void do_with_a(const A& a)
{
    ...
}

int main()
{
    auto b1      = B {};
    A&  b1_as_a = b1;

    auto b2 = B {};
    do_with_a(b2);

    return 0;
}
```

On peut appeler les fonctions publiques de A sur une instance de B.

```
int main()
{
    auto b      = B {};
    A& b_as_a = b;

    std::cout << b.get_y() << std::endl;
    std::cout << b_as_a.get_y() << std::endl;

    return 0;
}
```

On peut également stocker des pointeurs de B dans des pointeurs de A.

```
int main()
{
    auto many_a = std::vector<std::unique_ptr<A>> {};;

    many_a.push_back(std::make_unique<A>());
    many_a.push_back(std::make_unique<B>());

    auto tmp_b = std::make_unique<B>();
    many_a.push_back(std::move(tmp_b));

    return 0;
}
```

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

- éviter la duplication de code
- redéfinir des comportements

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

On parle de **classes polymorphes**.

```
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 dans une classe-fille

```
class  
{  
public:  
    virtual std::string get_name() const  
    {  
        return "???" ;  
    }  
  
    void describe() const  
    {  
        std::cout << "This is a " << get_name() << std::endl ;  
    }  
};
```



```
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 classe-mère a une fonction-membre redéfinissable avec la bonne signature

```
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";
    }
};
```

```
int main()
{
    auto instruments = std::vector<std::unique_ptr<Instrument>> {};;
    instruments.push_back(std::make_unique<Piano>());
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        instrument->describe();
    }

    return 0;
}
```

```
int main()
{
    auto instruments = std::vector<std::unique_ptr<Instrument>> {};;
    instruments.push_back(std::make_unique<Piano>());
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        instrument->describe();
    }

    return 0;
}
```

appelle les fonctions redéfinies
dans les classes-filles

```
int main()
{
    auto instrument = Instrument();
    instruments.push_back(instrument);
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        instrument->describe();
    }

    return 0;
}
```

attention à ne pas faire une copie
en oubliant le symbole & !

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 est virtuelle, on utilise le **type dynamique** de l'objet pour décider quelle version de la fonction sera appelée
3. Si une fonction n'est pas virtuelle, on utilise le **type statique** de l'objet pour décider quelle version de la fonction sera appelée
4. L'appel au destructeur répond aux mêmes règles que les autres fonctions

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la même signature)

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

    std::string get_name_stc() const
    {
        return "???";
    }
};
```

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

    std::string get_name_stc() const
    {
        return "piano";
    }
};
```

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la même sig

virtuelle dans la classe-mère, donc virtuelle dans la classe-fille (même sans le override)

```
class Instrument
```

```
{
```

```
public:
```

```
virtual std::string get_name_dyn() const
```

```
{
```

```
    return "???"
```

```
}
```

```
std::string get_name_stc() const
```

```
{
```

```
    return "???"
```

```
}
```

```
};
```

```
public:
```

```
std::string get_name_dyn() const
```

```
{
```

```
    return "piano"
```

```
}
```

```
std::string get_name_stc() const
```

```
{
```

```
    return "piano"
```

```
}
```

```
};
```

non virtuelle

2. Si une fonction est virtuelle, on utilise le **type dynamique** de l'objet pour décider quelle version de la fonction sera appelée

```
int main()
{
    auto instruments = std::vector<std::unique_ptr<Instrument>>> {};
    instruments.push_back(std::make_unique<Piano>());
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        std::cout << instrument->get_name_dyn() << std::endl;
    }

    return 0;
}
```

2. Si une fonction est virtuelle, on utilise le **type dynamique** de l'objet pour décider quelle version de la fonction sera appelée

```
int main()
{
    auto instruments = std::vector<std::unique_ptr<Instrument>> {};
    instruments.push_back(std::make_unique<Piano>());
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        std::cout << instrument->get_name_dyn() << std::endl;
    }

    return 0;
}
```

type dynamique = type utilisé à la création

appel virtuel : "piano", "guitar"

3. Si une fonction n'est pas virtuelle, on utilise le **type statique** de l'objet pour décider quelle version de la fonction sera appelée

```
int main()
{
    auto instruments = std::vector<std::unique_ptr<Instrument>>> {};
    instruments.push_back(std::make_unique<Piano>());
    instruments.push_back(std::make_unique<Guitar>());

    for (const auto& instrument: instruments)
    {
        std::cout << instrument->get_name_stc() << std::endl;
    }

    return 0;
}
```

3. Si une fonction n'est pas virtuelle, on utilise le **type statique** de l'objet pour décider quelle version de la fonction sera appelée

```
int main()
```

type statique =
type utilisé pour **définir l'identifiant**

```
    std::vector<std::unique_ptr<Instrument>> {};  
    std::make_unique<Piano>(),  
    std::make_unique<Guitar>());  
    instruments.push_back(std::make_unique<Guitar>());  
  
    for (const auto& instrument: instruments)  
    {  
        std::cout << instrument->get_name_stc() << std::endl;  
    }  
  
    return 0;  
}
```

appel statique : “???” , “???”

4. L'appel au destructeur répond aux mêmes règles que les autres fonctions

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

destructeur non virtuel

```
o: 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->reset(nullptr);

    return 0;
}
```

type statique

appel statique : "??? destroyed"

4. L'appel au destructeur répond aux mêmes règles que les autres fonctions

```
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;
}
```

appel dynamique :
"piano destroyed", "??? destroyed"

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.

Dans ce cas, la classe-mère devient **abstraite** et elle n'est plus instanciable directement.

Il est nécessaire de redéfinir les fonctions virtuelles pures dans les classes-filles si on veut pouvoir les instancier.

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

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