



## Advanced C++

*Robustesse - Encapsulation*



## Lookup rule : namespace et overloading

```
namespace NS {  
    class A {};  
    void doIt(A a) {cout << "1\n";}  
}  
void doIt(NS::A a) {cout << "2\n";}  
  
int main() {  
    NS::A a;  
  
    doIt(a);  
}
```

?

## Lookup rule : autre exemple

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

template <typename T>
T max(T a, T b) {
    return a + a;
}

int main() {
    string s1{ "bonjour" };
    string s2{ "hello" };
    cout << max(s1, s2) << endl;
}
```

std::max

concaténation

?

© Artworks Tous droits d'utilisation et de reproduction réservés

3

## ADL (Argument Dependant Lookup)

- un opérateur est souvent une fonction non membre (friendship requis)
  - sa syntaxe d'appel doit être concise
  - il faut donc briser l'encapsulation stricte des namespaces
    - Les fonctions non membre dans le même namespace qu'un type T et prenant T en paramètre sont accessibles si T l'est

```
namespace NS {
    class A {
        int value;
    public:
        A(int value) : value{value}{}
        friend ostream& operator<< (ostream& o, const A& a) {
            return o << a.value; }
    }
    void doThis (){}
    void doThat (int, double, float, char*, int, A a){}
}

using NS::A;
cout << A{1};
doThat (2, 2.5, 4.1f, "hello", 9, A{1});
NS::doThis();
```

non-membre

!

↔ NS::operator<< (cout, A{1});

accessibles via l'ADL

non accessible, préfixe obligatoire

© Artworks Tous droits d'utilisation et de reproduction réservés

4

## ADL et STL

- Des fonctions telles que `std::swap` s'appliquent à tous types de données
  - il est possible d'en redéfinir une version spécifique à un type

```
namespace NS {
    class A {};
    void swap(A&, A&);
}

namespace std{
    template <typename T>
    void swap(T&, T&)
}

using inutile

int main(){
    NS::A a1, a2;
    swap(a1, a2); // implicite
    std::swap(a1, a2); // explicite
}
```

@ Artworks Tous droits d'utilisation et de reproduction réservés

5

## Constance

```
using Rayon = double;
using Volume = double;
const long double PI {
    3.141'592'653'589'793'238'4621};

class Sphère {
    Rayon rayon;
public:
    explicit Sphère (Rayon rayon) : rayon{rayon}{}
    Volume getVolume() const {
        return rayon = 0 ? 0 :
            4.0 / 3 * PI * pow (rayon, 3);
    }
    void setRayon (Rayon rayon){
        this->rayon = rayon;
    }
    Rayon getRayon() const {return rayon;}
};

const Sphère Terre {1000};
Terre.setRayon(10);
```

Sphère

rayon  
/volume

propriété dérivée  
(algorithmique)

ne compile pas: OK

constance bitwise

ne compile pas: OK

@ Artworks Tous droits d'utilisation et de reproduction réservés

6

## Le pattern Lazy Computation

C++ 17

```
class Sphère {
    Rayon rayon;
    mutable optional<Volume> volumeCalculé;
public:
    explicit Sphère(Rayon rayon) : rayon{ rayon }{}
    Rayon getRayon() const { return rayon; }
    void setRayon(Rayon rayon){
        if (this->rayon != rayon){
            this->rayon = rayon;
            volumeCalculé.reset();
        }
    }
    Volume getVolume() const {
        if (!volumeCalculé)
            volumeCalculé = 4.0 / 3 * PI * pow(rayon, 3);
        return volumeCalculé.value();
    }
};
```

C++ 17

Sphère

rayon  
/volume

constance logique

© Artworks Tous droits d'utilisation et de reproduction réservés

7

## Mutable et multithreading

- Un mutex doit être verrouillé dans chaque méthode
  - y compris les méthodes constantes

```
class Sphère {
    mutable optional<Volume> volume;
    mutable mutex volumeMutex;
public:
    Volume getVolume() const {
        lock_guard<mutex> loc{volumeMutex};
        if (!volumeCalculé)
            volume = 4.0 / 3 * PI * pow (rayon, 3);
        return volume.value();
    }
    void setRayon (Rayon rayon){
        lock_guard<mutex> loc{volumeMutex};
        this->rayon = rayon;
        volumeCalculé.reset();
    }
    // ...
};
```

la mutex est  
mutable ...

... car la méthode  
est const

© Artworks Tous droits d'utilisation et de reproduction réservés

8

## Constance et signature

```
class Voiture{
    class Moteur{ classe imbriquée
        unsigned cylindrée;
    public:
        explicit Moteur(unsigned cylindrée) : cylindrée{cylindrée}{}
        unsigned getCylindrée() const { return cylindrée; }
        void démarrer(){cout << "vroum\n"; }
        void régler(){cout << "reglage\n"; }
    };
    private:
        Moteur moteur;
    public:
        explicit Voiture(unsigned cylindrée) : moteur{cylindrée}{}
        void démarrer(){ moteur.démarrer(); }
        const Moteur& getMoteur() const { return moteur; } deux accesseurs différents
        Moteur& getMoteur() { return moteur; }
};
```

```
Voiture v1 {1000};
v1.getMoteur().régler();
const Voiture v2 {2000};
cout << v2.getMoteur().getCylindrée()
    << " cm3" << endl;
```

reglage  
2000 cm3

© Artworks Tous droits d'utilisation et de reproduction réservés

9


## Friendship

```
class Voiture{
    class Moteur{
        unsigned cylindrée;
    public:
        explicit Moteur(unsigned cylindrée) : cylindrée{cylindrée}{}
        unsigned getCylindrée() const { return cylindrée; }
        void démarrer(){cout << "vroum\n"; }
        void régler(){cout << "reglage\n"; }
    };
    private:
        Moteur moteur;
    public:
        explicit Voiture(unsigned cylindrée) : moteur{cylindrée}{}
        void démarrer(){ moteur.démarrer(); }
        const Moteur& getMoteur() const { return moteur; }
};
```

accès public en lecture seule

```
friend void Garagiste::réparerVoiture (Voiture&);
```

accès à la partie privée de Voiture



© Artworks Tous droits d'utilisation et de reproduction réservés

10

## Friendship et héritage

```
class A {
    static void f(){}
    friend class AdminA;
};

class B : public A {
    static void g(){}
    friend class AdminB;
};
```

```
class AdminA {
    void doIt(){
        A::f();
        B::g();
    }
};

class AdminB {
    void doIt(){
        A::f();
        B::g();
    }
};
```

compile ?

compile ?

© Artworks Tous droits d'utilisation et de reproduction réservés

11

## Encapsulation et C++

```
namespace mySpace {
    // ...
    namespace mySubSpace {
        struct A {
            // ...
            struct B {
                int tab[]{3, 67, -12};
                void f(){
                    struct functor {
                        void operator()(int n) { /* ... */ }
                        // ...
                    };
                    for_each(begin(v), end(v), functor{});
                }
            };
        };
    };
};
```

namespace embarqué

classe dans une classe

classe dans une fonction

objet dans un appel



© Artworks Tous droits d'utilisation et de reproduction réservés

12

### Imbrication et généricité

une classe générique ne peut être embarquée dans un bloc de pile ...

```
int main() {  
    template <typename T>  
    class A {  
        // ...  
    };  
  
    A<int> a;  
}
```

```
template <typename T1>  
class A {  
public:  
    template <typename T2>  
    class B {  
        // ...  
    };  
};  
  
int main(){  
    A<double>::B<int> ab;  
}
```

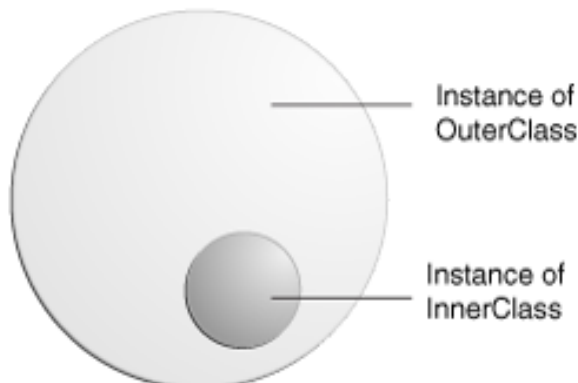
... mais peut être embarquée dans une classe

@ Artworks Tous droits d'utilisation et de reproduction réservés

13

### Classe imbriquée

- On peut déclarer une classe dans le contexte d'une autre
  - un objet imbriqué à accès à la partie privée de l'objet englobant
  - l'objet englobant n'a pas accès à la partie privée de l'objet englobé



@ Artworks Tous droits d'utilisation et de reproduction réservés

14

## Exemple

```
class Voiture {
public:
    void démarrer();
    void stopper();
    Voiture() : moteur{ *this }{}

private:
    void afficherRPM(int rpm){cout << rpm << " tours/minute";}


    class Moteur {
    public:
        Moteur(Voiture& voiture) : voiture{voiture}{}
        void start(){ rpm += 2000; voiture.afficherRPM(rpm); }
        void stop(){ rpm = 0; voiture.afficherRPM(rpm); }
    private:
        int rpm {};
        Voiture& voiture;
    };

    Moteur moteur;
};
```

classe interne privée

accès aux membres privés de la classe imbriquante

objet embarqué



15

© Artworks Tous droits d'utilisation et de reproduction réservés

## Autre exemple: la classe `std::bitset<N>`

**C++ 11**

- la classe `bitset<N>` embarque publiquement la class reference
  - `constexpr bool operator[](size_t pos) const;`

```
template<size_t _Bits>
class bitset {
public:
    constexpr bool operator[](size_t pos) const;
    reference operator[](size_t pos);
    // ...

    class reference {
    friend bitset<_Bits>;
    // ...
    };
    // ...
};
```

joue le rôle de proxy pour l'écriture

classe interne publique

- depuis **C++11**, la classe `bitset` propose de nouvelles méthodes
  - `any`, `all`, `none`

© Artworks Tous droits d'utilisation et de reproduction réservés

16



### Objets globaux et création / destruction

- Les objets globaux et les membres données statiques sont construits avant l'exécution de la fonction main
  - dans une même unité de compilation, l'ordre est de haut en bas
  - dans plusieurs unités de compilation, l'ordre est indéfini
- Les objets définis dans un namespace sont toujours construits avant l'accès à une fonction ou une variable du namespace.
- La destruction a lieu en sens contraire
- Pour plus de déterminisme, il est utile de recourir à des **objets statiques locaux**.

© Artworks Tous droits d'utilisation et de reproduction réservés

17

### Objets locaux statiques

**C++ 11**

- le constructeur d'un objet statique local n'est invoqué que lors du premier appel de la fonction

```
A globalA {"global"};
```

objet global

```
void f(){  
    static A localA {"local"};  
}
```

objet statique local  
C++11 thread safety  
tdouble-checked locking pattern (DCLP).

```
int main(){  
    cout << "debut" << endl;  
  
    f();  
    f();  
    cout << "fin" << endl;  
    f();  
}
```

invocations multiples

```
A::ctor: global  
debut  
A::ctor: local  
fin  
A::dtor local  
A::dtor global
```

© Artworks Tous droits d'utilisation et de reproduction réservés

18

## L'idiome Static Constructor (C#)

```
using Valeur = unsigned;
class Dé {
Valeur valeurFace;
    static const Valeur VALEUR_MIN {1};
    static const Valeur VALEUR_MAX {6};
public:
    Dé() {
        struct StaticDé {
            StaticDé() {::srand(unsigned(::time(nullptr)));};
        };
        static const StaticDé instance;
        this->lancer();
    }
    void lancer() {valeurFace = VALEUR_MIN + rand() %
        (VALEUR_MAX - VALEUR_MIN + 1);}
    Valeur getValeur() const { return valeurFace; }
};

array<Dé, 10> dés;
for (const Dé& dé : dés)
    cout << dé.getValeur() << " ";
```



1 4 1 3 3 4 3 2 5 4

© Artworks Tous droits d'utilisation et de reproduction réservés

19

## Composition de namespaces - Versioning

```
namespace A {
    struct T {};
    void foo(){}
}
namespace B {
    struct U {};
    void bar(){}
}
namespace C {
    using A::T;
    using B::bar;
}
```

```
int main(){
    using namespace C;
    T t;
    bar();
}
```

```
int main(){
    {
        mylib::T t;
        mylib::foo();
    }
    {
        namespace thelib = mylib::v1;
        thelib::T t;
        thelib::foo();
    }
}
```

```
namespace mylib {
    namespace v1 {
        class T{};
        void foo(){}
    }
    namespace v2 {
        class T{};
        void foo(){}
    }
    using namespace v2;
}
```

"exportation" de la version par défaut

choix d'une version spécifique

© Artworks Tous droits d'utilisation et de reproduction réservés

20

## Espace de nom inline

C++ 11

- les membres d'un espace de nom inline sont accessible comme s'ils étaient directement membre de l'espace de noms englobant
  - et ce, de façon récursive ...

```
namespace mylib {
    namespace v1 { // #include v1.h
        class T{};
        void foo(){}
    }
    inline namespace v2 { // #include v2.h
        class T{};
        void foo(){}
    }
}
```

version par défaut

nom optionnel

```
using namespace mylib;
foo();           // défaut
v2::foo();       // le plus récent
v1::foo();       // ancien
```

«header»  
v1.h

«header»  
v2.h

«include»

«include»

«header»  
mylib.h

exemple  
d'organisation

@ Artworks Tous droits d'utilisation et de reproduction réservés

21

## Imbrication de namespaces

C++ 17

```
namespace A {
    namespace B {
        namespace C {
            /* ... */
        }
    }
}
```

avant

```
namespace A::B::C {
    /* ... */
}
```

C++17

@ Artworks Tous droits d'utilisation et de reproduction réservés

22