Classes

Regrouper des donnees : struct

Une `struct` permet de regrouper plusieurs variables dans une meme entite.

```
struct vec3 {
    float x;
    float y;
    float z;
};
```

```
vec3 v;
v.x = 1.0f;
v.y = 2.0f;
v.z = 3.0f;

// Initialisation directe
vec3 w{1.0f, 2.0f, 3.0f};
```

`struct` regroupe des donnees liees dans un meme type.

Les membres sont **publics par defaut** : accessibles directement via l'operateur `.`

Adapte pour des **agregats de donnees simples** (vecteurs, couleurs, sommets, ...).

Methodes (fonctions membres)

Une `struct` (ou `class`) peut contenir des **fonctions** qui operent sur ses donnees.

```cpp
#include <cmath>

struct vec3 {
    float x, y, z;

    float norm() const {
        return std::sqrt(x*x + y*y + z*z);
    }

    void normalize() {
        float n = norm();
        x /= n; y /= n; z /= n;
    }
};
```

```cpp
vec3 v{1.0f, 2.0f, 2.0f};

float n = v.norm(); // n = 3.0

v.normalize();
// v = (1/3, 2/3, 2/3)
```

`const` apres la signature : la methode **ne modifie pas** l'objet.

`norm()` est `const` (lecture), `normalize()` ne l'est pas (modification).

Implementation hors classe possible avec `vec3`

`norm`.

Le pointeur implicite this

Dans chaque methode, le compilateur fournit un pointeur `this` vers l'objet courant.

Disambiguation de noms

```
struct S {
    int x;

    void set(int x) {
        this->x = x;  // this->x = attribut
    }                  // x = parametre

    int get() const {
        return this->x;
    }
};
```

Chainage d'appels avec `*this`

```
struct vec3 {
    float x, y, z;

    vec3& set_x(float v) {
        x = v; return *this;
    }
    vec3& set_y(float v) {
        y = v; return *this;
    }
};

vec3 v;
v.set_x(1).set_y(2); // chainage
```

`this` est un **pointeur** : type `S*` (ou `S const*` dans une methode `const`).

Retourner `*this` permet de **chainer** les appels de methodes.

Encapsulation : struct vs class

`struct` : membres **publics** par defaut. `class` : membres **privés** par defaut.

struct (public par defaut)

```
struct vec3 {
    float x, y, z; // public
};

vec3 v;
v.x = 1.0f; // OK
```

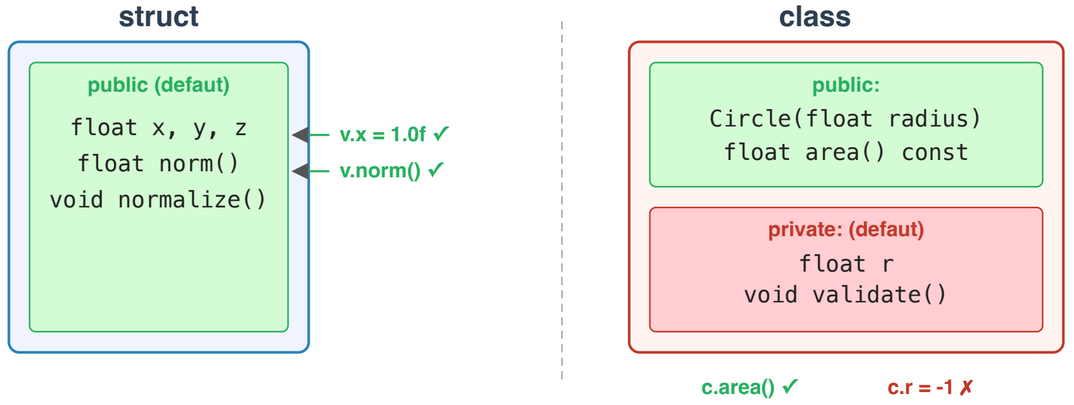
class (private par defaut)

```
class vec3 {
    float x, y, z; // prive
};

vec3 v;
v.x = 1.0f; // ERREUR !
```

Encapsulation : controler l'acces aux donnees

```
class Circle {
  public:
    Circle(float radius) { set_radius(radius); }
    float area() const { return 3.14159f * r * r; }
    void set_radius(float radius) {
        if (radius > 0.0f) r = radius;  // validation
    }
  private:
    float r;  // acces interdit de l'exterieur
};
```



`public:` / `private:` controlent la **visibilite** des membres.

L'encapsulation garantit la **coherence interne** de l'objet

ex: rayon toujours positif.

Constructeurs et initialisation

Un **constructeur** est appele automatiquement a la creation d'un objet. Il garantit un **etat valide**.

Probleme : valeurs non initialisees

```
struct vec3 {
    float x, y, z;
};

int main() {
    vec3 v;
    std::cout << v.x; // indefini (UB)
}
```

Solution : constructeur

```
struct vec3 {
    float x, y, z;

    vec3() : x(0), y(0), z(0) {}
};

int main() {
    vec3 v;  // x=y=z=0 (garanti)
}
```

Liste d'initialisation et syntaxes d'appel

```
struct vec3 {
    float x, y, z;

    vec3(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}
};

vec3 a(1.0f, 2.0f, 3.0f);    // appel direct
vec3 b{1.0f, 2.0f, 3.0f};    // initialisation uniforme (recommande)
```

`: x(...), y(...)` = **liste d'initialisation** : initialise les attributs **avant** le corps du constructeur.

Preferable a l'affectation dans le corps (obligatoire pour `const` et references).

Surcharge de constructeurs et explicit

On peut definir **plusieurs constructeurs** avec des signatures differentes.

Surcharge de constructeurs

```
struct vec3 {
    float x, y, z;

    vec3() : x(0), y(0), z(0) {}
    vec3(float v) : x(v), y(v), z(v) {}
    vec3(float x_, float y_, float z_)
        : x(x_), y(y_), z(z_) {}
};

vec3 a;                     // (0,0,0)
vec3 b(1.0f);               // (1,1,1)
vec3 c(1.0f, 2.0f, 3.0f); // (1,2,3)
```

`explicit` : empecher les conversions implicites

```
struct vec3 {
    float x, y, z;

    explicit vec3(float v)
        : x(v), y(v), z(v) {}
};

vec3 a(1.0f);    // OK
// vec3 b = 1.0f; // ERREUR (explicit)
```

Membres `const` et references : init list obligatoire

```
struct sample {
    int const id;      // doit etre initialise a la construction
    float& ref;        // reference : idem

    sample(int id_, float& ref_) : id(id_), ref(ref_) {}  // OK
};
```

Destructeur

Le **destructeur** `~Class()` est appele automatiquement quand l'objet est detruit (fin de scope, `delete`).

Ordre de construction / destruction

```cpp
#include <iostream>

struct tracer {
    std::string name;

    tracer(std::string n) : name(n) {
        std::cout << "Construct " << name << "\n";
    }
    ~tracer() {
        std::cout << "Destroy " << name << "\n";
    }
};

int main() {
    tracer a("A");
    tracer b("B");
} // destruction dans l'ordre inverse
```

Sortie :

```
Construct A
Construct B
Destroy B
Destroy A
```

Surcharge d'operateurs : principe

En C++, on peut **redefinir** le comportement des operateurs (+, *, [], ...) pour ses propres types.

Le compilateur traduit les operateurs en appels de fonctions

```
a + b    // est equivalent a :
operator+(a, b);    // fonction non-membre
a.operator+(b);     // ou methode membre
```

Regle de conception

```
Operateurs qui MODIFIENT l'objet (+=, *=, [])   →  methode membre
Operateurs SYMETRIQUES (+, -, *, ==)             →  fonction non-membre
```

La surcharge ne cree **pas** de nouvel operateur : elle redefinit un operateur existant.

Un operateur surcharge doit rester **intuitif** et **coherent** avec son sens usuel.

Operateurs arithmetiques

`operator+=` : methode membre

```
struct vec3 {
    float x, y, z;

    vec3& operator+=(vec3 const& v) {
        x += v.x;
        y += v.y;
        z += v.z;
        return *this;
    }
};
```

`operator+` : non-membre, reutilise `+=`

```
vec3 operator+(vec3 a, vec3 const& b) {
    a += b;
    return a;
}

// Utilisation
vec3 a{1,2,3}, b{4,5,6};
vec3 c = a + b;  // (5,7,9)
a += b;          // a = (5,7,9)
```

Multiplication par un scalaire

```
vec3 operator*(float s, vec3 const& v) {
    return vec3{s*v.x, s*v.y, s*v.z};
}
vec3 operator*(vec3 const& v, float s) { return s * v; }

vec3 w = 2.0f * vec3{1,2,3};  // (2,4,6)
```

`+=` retourne `*this` par reference pour permettre le chainage.

`+` prend `a` **par copie** et reutilise `+=` : evite la duplication de code.

Operateurs ==, [], <<

Comparaison == et !=

```
bool operator==(vec3 const& a,
                vec3 const& b) {
    return a.x==b.x && a.y==b.y
           && a.z==b.z;
}

bool operator!=(vec3 const& a,
                vec3 const& b) {
    return !(a == b);
}
```

Acces indexe [ ] (const et non-const)

```
struct vec3 {
    float x, y, z;

    float& operator[](int i) {
        return (&x)[i];
    }
    float const& operator[](int i) const {
        return (&x)[i];
    }
};

vec3 v{1,2,3};
v[0] = 4.0f;    // ecriture
float y = v[1];  // lecture
```

Flux de sortie <<

```
#include <iostream>

std::ostream& operator<<(std::ostream& out, vec3 const& v) {
    out << "(" << v.x << ", " << v.y << ", " << v.z << ")";
    return out;
}

vec3 v{1,2,3};
std::cout << v << std::endl;  // affiche (1, 2, 3)
```

Heritage : principe

L'**heritage** permet de definir une classe a partir d'une classe existante : `class Derived : public Base`.

Classe de base

```
class Shape {
  public:
    float x, y;

    Shape(float x_, float y_)
        : x(x_), y(y_) {}

    void translate(float dx, float dy) {
        x += dx;
        y += dy;
    }
};
```

Classe derivee

```
class Circle : public Shape {
  public:
    float radius;

    Circle(float x_, float y_, float r_)
        : Shape(x_, y_), radius(r_) {}
};

// Utilisation
Circle c(0.0f, 0.0f, 1.0f);
c.translate(1.0f, 2.0f); // herite
```

`Circle` **herite** de `Shape` : elle recoit `x`, `y` et `translate()`.

Le constructeur de la classe derivee **appelle le constructeur de base** dans la liste d'initialisation.

Heritage exprime une relation **"est-un"** (*is-a*) : un `Circle` est un `Shape`.

Le probleme : conteneur heterogene

On souhaite stocker des objets de **types differents** dans un meme conteneur.

```cpp
struct Circle {
    float r;
    float area() const { return 3.14159f * r * r; }
};

struct Rectangle {
    float w, h;
    float area() const { return w * h; }
};
```

Probleme : pas de type commun

```cpp
std::vector<Circle> shapes;     // que des cercles
std::vector<Rectangle> shapes;  // que des rectangles

// std::vector<???> shapes;   // les deux ensemble ? Impossible !
```

`Circle` et `Rectangle` ont une methode `area()`, mais **aucun lien de type**.
Sans heritage ni polymorphisme, il est **impossible** de les stocker ensemble.
Le **polymorphisme** résout ce probleme.

Classes abstraites et fonctions virtuelles

`virtual` permet le **dispatch dynamique** : l'appel est resolu selon le type reel de l'objet.

Classe abstraite (interface)

```
class Shape {
  public:
    // = 0 : methode virtuelle pure
    virtual float area() const = 0;

    // destructeur virtuel obligatoire
    virtual ~Shape() = default;
};

// Shape s; // ERREUR : abstraite
```

Classes concretes avec `override`

```
class Circle : public Shape {
  public:
    float r;
    explicit Circle(float r_) : r(r_) {}
    float area() const override {
        return 3.14159f * r * r;
    }
};

class Rectangle : public Shape {
  public:
    float w, h;
    Rectangle(float w_, float h_)
        : w(w_), h(h_) {}
    float area() const override {
        return w * h;
    }
};
```

Utilisation

```
void print_area(const Shape& s) {
    // Appel polymorphique :
    //  Circle::area() ou Rectangle::area()
    // selon le type reel de l'objet
    std::cout << "Area = " << s.area();
}

int main() {
    Circle c(2.0f);
    Rectangle r(3.0f, 4.0f);

    print_area(c);   // Area = 12.5664
    print_area(r);   // Area = 12
}
```

`virtual ... = 0` : methode **virtuelle pure** → classe **abstraite** (non instanciable).

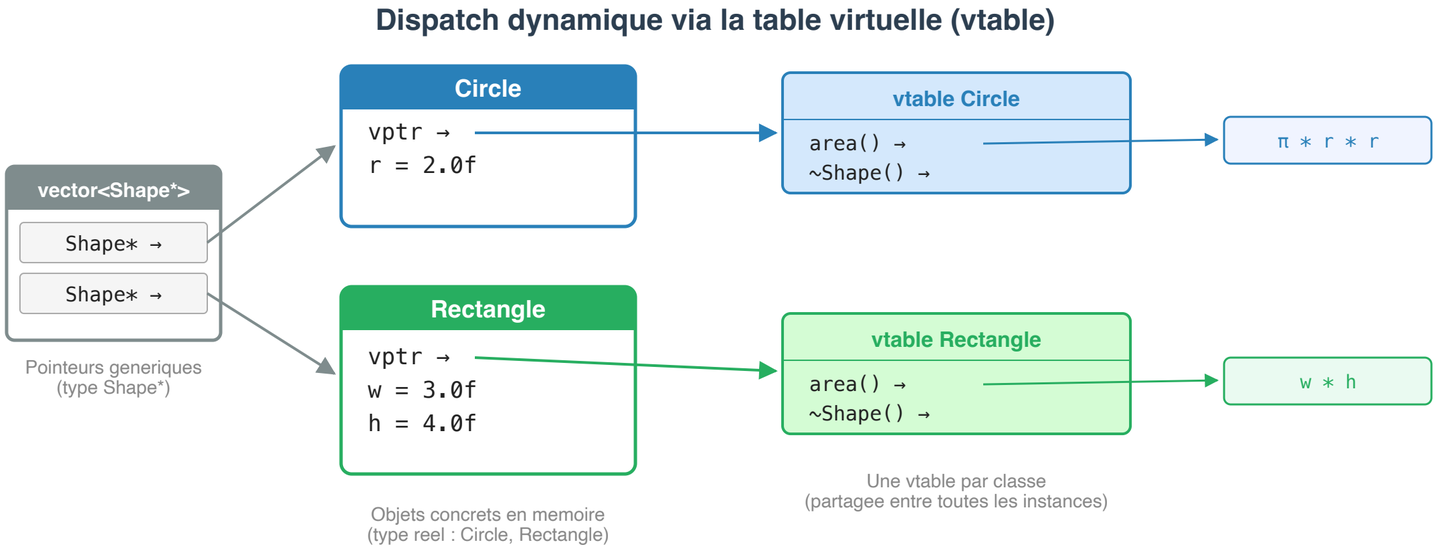
`override` (C++11) : verifie a la compilation la redefinition d'une methode virtuelle.

Destructeur virtuel : obligatoire en hierarchie polymorphique (sinon fuite memoire).

Equivalent Python : `abc.ABC` + `@abstractmethod` → en C++ : `virtual ... = 0`.

Fonctions virtuelles : organisation interne (vtable)

Comment le compilateur sait-il quelle version de `area()` appeler ?



Appel polymorphique

```
void print_area(const Shape& s) {
    s.area();
    // 1. Lire s.vptr
    // 2. Chercher area() dans la vtable
    // 3. Appeler le pointeur de fonction
}
```

Mecanisme : la vtable (virtual table)

Chaque classe avec des methodes `virtual` possede une **table de fonctions virtuelles** (vtable).
La vtable est un tableau de **pointeurs de fonctions** : une entree par methode virtuelle.
Chaque objet contient un **pointeur caché** vers la vtable de sa classe (le `vptr`).

Cout : un pointeur supplementaire par objet (`vptr`, 8 octets) + une indirection a chaque appel.
En pratique, le cout est negligeable pour la plupart des cas d'utilisation.
Pas de vtable si aucune methode `virtual` → pas de surcout pour les classes non polymorphiques.

Exemple concret : pattern Visitor

Ajout de nouvelles operations sur une hierarchie de classes **sans modifier** les classes existantes.

Hierarchie existante

```
class Circle;
class Rectangle;

class ShapeVisitor {
  public:
    virtual void visit(const Circle& c) = 0;
    virtual void visit(const Rectangle& r) = 0;
    virtual ~ShapeVisitor() = default;
};

class Shape {
  public:
    virtual void accept(ShapeVisitor& v) const = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape {
  public:
    float r;
    explicit Circle(float r_) : r(r_) {}
    void accept(ShapeVisitor& v) const override {
        v.visit(*this);
    }
};

class Rectangle : public Shape {
  public:
    float w, h;
    Rectangle(float w_, float h_) : w(w_), h(h_) {}
    void accept(ShapeVisitor& v) const override {
        v.visit(*this);
    }
};
```

Nouvelles operations sans toucher aux classes

```
// Operation 1 : calcul d'aire
class AreaCalculator : public ShapeVisitor {
  public:
    float result = 0;
    void visit(const Circle& c) override {
        result = 3.14159f * c.r * c.r;
    }
    void visit(const Rectangle& r) override {
        result = r.w * r.h;
    }
};

// Operation 2 : export SVG
class SvgExporter : public ShapeVisitor {
  public:
    void visit(const Circle& c) override {
        std::cout << "<circle r=\"" << c.r << "\"/>";
    }
    void visit(const Rectangle& r) override {
        std::cout << "<rect w=\"" << r.w
                  << "\" h=\"" << r.h << "\"/>";
    }
};
```

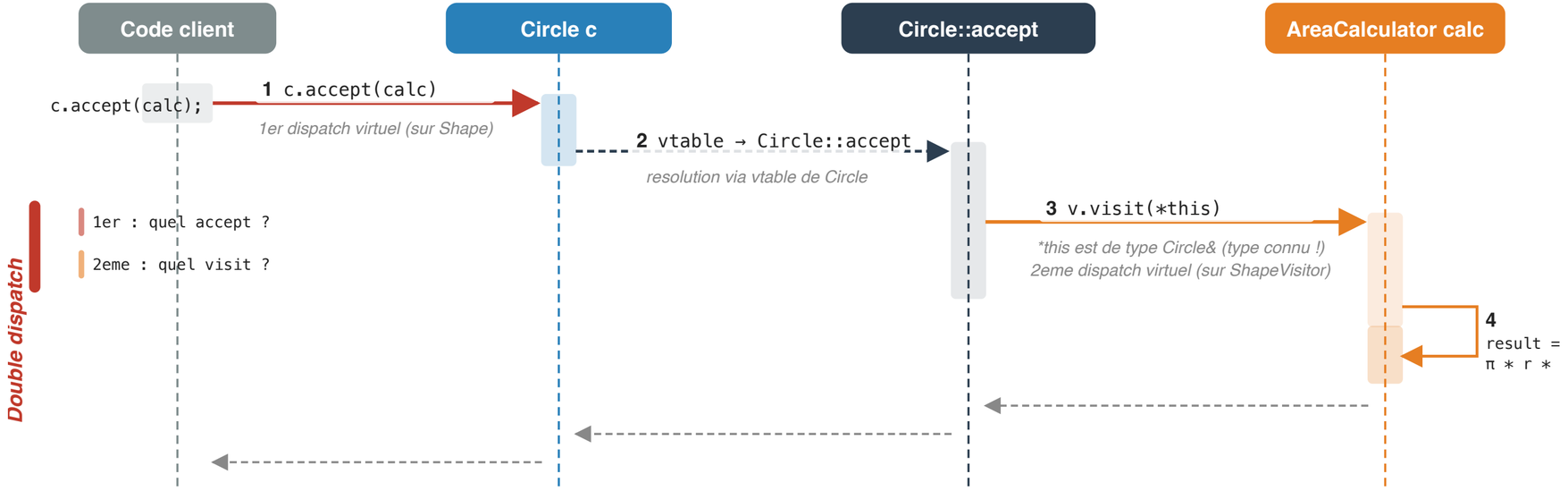
Utilisation

```
Circle c(2.0f);
AreaCalculator calc;
c.accept(calc);
std::cout << calc.result; // 12.5664

SvgExporter svg;
c.accept(svg); // <circle r="2"/>
```

Visitor : mecanisme du double dispatch

Exemple : `c.accept(calc)` avec `Circle c` et `AreaCalculator calc`



Classe	Role
`Shape` (abstraite)	`virtual accept(ShapeVisitor&) = 0`
`Circle` `Rectangle`	Implementent `accept` : appellent `v.visit(*this)`
`ShapeVisitor` (abstraite)	Declare `virtual visit(Circle&)` et `virtual visit(Rectangle&)`
`AreaCalculator` `SvgExporter`	Implementent `visit` pour chaque forme (nouvelles operations)

Stockage polymorphique et smart pointers

Pour stocker des objets heterogenes : `std::vector<std::unique_ptr<Base>>`.

Bonne pratique : `unique_ptr`

```cpp
#include <vector>
#include <memory>

int main() {
    std::vector<std::unique_ptr<Shape>> shapes;

    shapes.push_back(
        std::make_unique<Circle>(2.0f));
    shapes.push_back(
        std::make_unique<Rectangle>(3.0f, 4.0f));

    float total = 0.0f;
    for (auto const& s : shapes)
        total += s->area(); // dispatch dynamique
}
```

A eviter : pointeurs bruts

```cpp
int main() {
    std::vector<Shape*> shapes;

    shapes.push_back(new Circle(2.0f));
    shapes.push_back(new Rectangle(3.0f, 4.0f));

    for (Shape* s : shapes)
        total += s->area();

    // Oubli de delete → fuite memoire !
    for (Shape* s : shapes)
        delete s;
}
```

Methodes const

Une methode `const` garantit qu'elle **ne modifie pas** l'etat de l'objet.

Declaration d'une methode const

```
class vec3 {
  public:
    float x, y, z;

    float norm() const {  // ne modifie rien
        return std::sqrt(x*x + y*y + z*z);
    }

    void normalize() {    // modifie l'objet
        float n = norm();
        x /= n; y /= n; z /= n;
    }
};
```

Objet const → methodes const uniquement

```
vec3 v{1.0f, 2.0f, 2.0f};
v.norm();       // OK
v.normalize();  // OK

const vec3 w{1.0f, 2.0f, 2.0f};
w.norm();       // OK (const)
// w.normalize(); // ERREUR !
```

Erreur de compilation : modification dans une methode const

```
float norm() const {
    x = 0.0f;  // ERREUR : modification interdite
    return 0.0f;
}
```

Un objet `const` ne peut appeler **que** des methodes `const`.

Separation claire entre methodes de **lecture** et de **modification**.

Surcharge const / non-const

Une methode `const` et une methode non `const` de meme nom sont **deux fonctions distinctes**.

`operator[]` : deux versions

```
class vec3 {
  public:
    float x, y, z;

    // version non-const → ecriture
    float& operator[](int i) {
        return (&x)[i];
    }

    // version const → lecture seule
    float const& operator[](int i) const {
        return (&x)[i];
    }
};
```

Le compilateur choisit automatiquement

```
vec3 a{1,2,3};
a[0] = 5.0f;      // non-const
float y = a[1];   // non-const

const vec3 b{1,2,3};
float x = b[0];   // const
// b[0] = 5.0f;   // ERREUR !
```

Autre exemple : accesseur en lecture/ecriture

```
class Buffer {
  public:
    float& value()       { return data; }   // ecriture
    float  value() const { return data; }   // lecture
  private:
    float data;
};
```

Le compilateur selectionne la version selon la **constness** de l'objet.

Fournir les deux versions est une bonne pratique pour les accesseurs.

Mot-cle static dans les classes

Un membre `static` appartient a la **classe**, pas aux objets. Il est **partage** par toutes les instances.

Attribut et methode statiques

```
class Counter {
  public:
    Counter() { ++count; }

    static int get_count() {
        return count;
    }
  private:
    static int count; // declaration
};

// Definition (dans un .cpp)
int Counter::count = 0;

Counter a, b, c;
int n = Counter::get_count(); // 3
```

Contrainte : pas de `this`

```
class Example {
  public:
    static void f() {
     // x = 3; // ERREUR : pas de this
        y = 4; // OK : y est static
    }
  private:
    int x;
    static int y;
};
```

Acces via `NomClasse::membre` (pas besoin d'objet).

Methode `static` : **pas de `this`**, ne peut acceder qu'aux membres statiques.

Espaces de noms (namespace)

Un `namespace` regroupe des symboles sous un prefixe commun pour **eviter les conflits de noms**.

Declaration d'un namespace

```
namespace math {

    struct vec3 {
        float x, y, z;
    };

    float dot(vec3 const& a, vec3 const& b) {
        return a.x*b.x + a.y*b.y + a.z*b.z;
    }

} // namespace math
```

Utilisation avec l'opérateur ::

```
math::vec3 a{1,2,3};
math::vec3 b{4,5,6};
float p = math::dot(a, b);
```

Eviter les conflits entre modules

```
namespace io {
    int load(char const* f) { /*...*/ }
}
namespace gpu {
    int load(char const* f) { /*...*/ }
}

int a = io::load("mesh.obj");   // clair
int b = gpu::load("shader.vert");
```

`std::vector`, `std::cout` : la bibliotheque standard utilise le namespace `std`.

Le qualificateur `::` desambiguise les symboles de meme nom.

Namespaces : imbriques, anonymes, alias

Namespaces imbriques (C++17)

```
// Syntaxe classique
namespace engine {
namespace math {
    struct vec2 { float x, y; };
}
}

// Syntaxe compacte C++17
namespace engine::math {
    struct vec2 { float x, y; };
}

engine::math::vec2 v{1,2};
```

Namespace anonyme (visibilite fichier)

```
// Equivalent de static pour fonctions
namespace {
    int helper(int x) { return 2*x; }
}
// helper() visible uniquement dans ce fichier
```

Alias de namespace

```
namespace em = engine::math;
em::vec2 v{1,2};
```

Bonne pratique

```
// using precis dans un .cpp
using math::vec3;
vec3 v{1,2,3}; // OK
```

A eviter dans un header

```
// Pollue tous les fichiers incluants
using namespace std;
// Risque de conflits de noms
```

Séance pratique 3

Classes et Polymorphisme

TP — Héritage et polymorphisme (b_hierarchy)

Contexte — Hiérarchie de classes simulant des entités d'un jeu vidéo.

```cpp
class Entity {
public:
    Entity(std::string const& n);
    virtual ~Entity() = default;

    virtual void update() = 0;
    virtual void interact(Entity& other);

    std::string name;
    int health = 100;
};
```

Identification du type — utiliser `dynamic_cast<Type*>(&other)`

```cpp
void Player::interact(Entity& other) {
    if (dynamic_cast<Enemy*>(&other)) {
        other.health -= 20;
        std::cout << "Player " << name << " strikes " << other.name << " (-20 HP)\n";
    } else if (dynamic_cast<NPC*>(&other)) {
        health += 5; // ...
    } else {
        std::cout << "Player " << name << " waves at " << other.name << "\n";
    }
}
```

Créer 3 classes dérivées : `Player`, `Enemy`, `NPC`

Player — `update()` : "explores the world" | `interact(Enemy)` : -20 HP | `interact(NPC)` : +5 HP
Enemy — `update()` : "is roaming" | `interact(Player)` : -10 HP | `interact(Enemy)` : +3 HP alliance
NPC — `update()` : "is waiting" | `interact(Player)` : quest greeting | sinon ignore

TP — Image dynamique : formes (c_shapes)

Contexte — Génération dynamique d'une image 32×32 pixels, affichée en temps réel

Implémenter les méthodes de dessin :

1. `set_rectangle(x1, y1, x2, y2, color)`

```
Pour x allant de min(x1,x2) à max(x1,x2) :
  Pour y allant de min(y1,y2) à max(y1,y2) :
    set_pixel(x, y, color)
```

2. `set_disc(x_center, y_center, radius, color)`

```
Pour x allant de (x_center - radius) à (x_center + radius) :
  Pour y allant de (y_center - radius) à (y_center + radius) :
    Si (x - x_center)² + (y - y_center)² < radius² :
      set_pixel(x, y, color)
```

TP — Image dynamique : triangle (c_shapes)

`set_triangle(x1,y1, x2,y2, x3,y3, color)`

Principe — Test d'appartenance d'un pixel P à un triangle ABC via les **coordonnées barycentriques**.

$P = \lambda_1 A + \lambda_2 B + \lambda_3 C$ avec $\lambda_1 + \lambda_2 + \lambda_3 = 1$.
P est dans le triangle si $\lambda_1 \geq 0$, $\lambda_2 \geq 0$, $\lambda_3 \geq 0$.

Algorithme :

```
denom = (y2 - y3)*(x1 - x3) + (x3 - x2)*(y1 - y3)

Pour x allant de min(x1, x2, x3) à max(x1, x2, x3) :
  Pour y allant de min(y1, y2, y3) à max(y1, y2, y3) :

    λ1 = ( (y2 - y3)*(x - x3) + (x3 - x2)*(y - y3) ) / denom
    λ2 = ( (y3 - y1)*(x - x3) + (x1 - x3)*(y - y3) ) / denom
    λ3 = 1 - λ1 - λ2

    Si λ1 ≥ 0 et λ2 ≥ 0 et λ3 ≥ 0 :
      set_pixel(x, y, color)
```

Note : `denom` ne dépend que des sommets du triangle, il se calcule une seule fois avant les boucles.