Polymorphisme en C++
Dynamique vs Statique

Quel impact sur votre code au quotidien ?
Comprendre les compromis entre flexibilité, performance, complexité et maintenabilité.

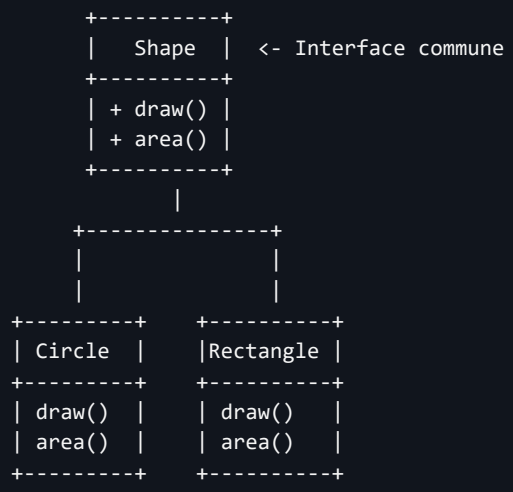
🎯 C++Frug #63 — new std::chrono::year{2025+1}

Florian Charrieau
eXalt Paris • 27/01/2026

Polymorphisme = « plusieurs formes »

Une interface unique pour manipuler des types différents

Hiérarchie



Application

```cpp
// Collection hétérogène
std::vector<Shape*> shapes = {
    new Circle(5.0),
    new Rectangle(3.0, 4.0)
};

// Traitement uniforme
for (auto* shape : shapes) {
    shape->draw();
    std::cout << shape->area();
}
```

Polymorphisme dynamique

Principe : Résolution à l'exécution via virtual + héritage

Mots-clés : `virtual` , `override` , `vtable` , `vptr`

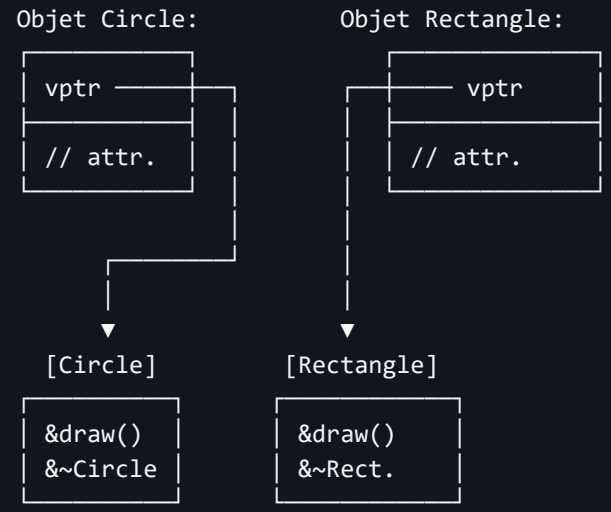
```cpp
class Shape {
public:
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Cercle";
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Rectangle";
    }
};

std::vector<std::unique_ptr<Shape>> shapes;
shapes.push_back(std::make_unique<Circle>());
shapes.push_back(std::make_unique<Rectangle>());
for (auto& s : shapes) {
    s->draw(); // Runtime
}
```

Mécanisme

1. Appel `shape->draw()`
2. Lecture du vptr
3. Accès vtable
4. Appel indirect via l'adresse

⚠️ Appel indirect → Difficulté d'optimisation par le compilateur et le CPU

Mémoire



Polymorphisme dynamique - Pièges

Destructeur non virtuel

Memory leak si destruction via pointeur de base

```
Shape* s = new Circle();
delete s;  // ~Shape() appelé
           // ~Circle() jamais appelé
```

❓ Sans `virtual`, undefined behavior.

✅ Solution :

```
struct Shape {
    virtual ~Shape() = default;
    virtual void draw() = 0;
};
```

Object slicing

Données perdues si passage par valeur

```
void process(Shape s) { }
Circle c(5.0);
process(c);  // radius perdu
```

❓ Une copie de `Shape` est créée. Les champs de `Circle` en trop sont tronqués. Pas d'erreur de compilation, mais données silencieusement perdues.

✅ Solution :

```
void process(const Shape& s) { }
void process(const Shape* s) { }
```

Appels virtuels dans le ctor

```
struct Base {
    Base() { init(); }
    virtual void init() { /* setup base */ }
};
struct Derived : Base {
    Derived() { /* vtable pas prête */}
    void init() override { /* JAMAIS appelé */ }
};
```

❓ Pendant le ctor, la vtable pointe sur la classe en cours de construction, pas sur la classe dérivée. Appeler une méthode virtuelle appelle la mauvaise version.

Polymorphisme statique

Principe : Résolution à la compilation via templates

Mots-clés : `template`, instanciation, duck typing, inlining

```cpp
class Circle {
public:
    Circle() {}
    void draw() const {
        std::cout << "Cercle";
    }
};

class Rectangle {
public:
    Rectangle() {}
    void draw() const {
        std::cout << "Rectangle";
    }
};

template <typename T>
void render(const T& shape) {
    shape.draw();
}

render(Circle{});
render(Rectangle{});
```

Mécanisme

1. Appel `render(circle)`
2. Type connu à la compilation
3. Adresse directe calculée
4. Appel direct (inliné)

✅ Le compilateur peut inliner et optimiser aggressivement

Code généré

```cpp
void render_Circle(const Circle& shape) {
    shape.draw();
}
// → inliné en :
std::cout << "Cercle";

void render_Rectangle(const Rectangle& shape) {
    shape.draw();
}
// → inliné en :
std::cout << "Rectangle";

// Appels directs
render_Circle(circle);
render_Rectangle(rectangle);
```

Polymorphisme statique - Pièges

Code bloat

Instanciation pour chaque type → exécutable énorme

```
template<typename T> void render(T s);
render(Circle{});     // render<Circle>
render(Rectangle{}); // render<Rectangle>
render(Triangle{});  // render<Triangle>
// ⚠️ 3 copies du code généré
```

❓ Chaque instantiation crée une version complète du code. Sur 100 types = 100 copies identiques en mémoire. Exe gonflé, cache pollué, linking lent.

✅ Solution :

Utiliser une base commune non-template (CRTP) ou std::variant pour partager le code

Sur-généricité

Ajouter templates sans besoin réel → complexité gratuite et maintenance difficile

❓ Templater "au cas où" complique le code et la compilation. Bénéfice performant annulé par la dette technique. Mieux vaut du code simple et lisible.

✅ Solution :

Commencer par dynamique, mesurer bottlenecks, puis templater seulement si nécessaire

Contrats implicites

Les exigences ne sont pas documentées au compilateur

```
template<typename Shape>
void process(Shape s) {
    s.draw();
    s.area();  // requises
}

struct BadType { };
process(BadType{});
// "no member named 'draw'"
```

❓ Le compilateur n'impose rien au template. `process(BadType{})` compile, puis échoue en instantiation. Pas de contrat clair entre template et appelant.

✅ Solution (C++20) :

```
template <typename T>
concept Drawable = requires(T t) {
    t.draw(); t.area();
};
template <Drawable T>
void process(T s) { }
```

Notre cas de test : Stress benchmark

🎯 Objectif

Comparer polymorphisme dynamique vs statique sur un cas identique : 500 appels de fonction

📦 Dynamique

- 500 objets `Shape*`
- Tous partagent la même vtable
- Appels via `virtual` dispatch

Statique

- 500 instanciations de templates
- Chaque instantiation = fonction unique
- Appels résolus à la compilation

📊 Métriques mesurées

Temps de compilation (O0 et O3) + Temps d'exécution (1000 iterations × 500 appels)

Syntaxe & Maintenabilité

Dynamique

Hiérarchie avec virtual

```
struct ShapeDyn {
    virtual ~ShapeDyn() = default;
    virtual double area(double r)
        const = 0;
};

struct CircleDyn : ShapeDyn {
    double area(double r)
        const override {
        return PI * r * r;
    }
};
```

Création des objets

```
std::vector<std::unique_ptr<ShapeDyn>>
    shapes;
shapes.reserve(500);

// 500 objets (même vtable)
for (int i = 0; i < 500; ++i) {
    shapes.push_back(
        std::make_unique<CircleDyn>());
}
```

Exécution avec dispatch dynamique

```
void run_dynamic(int iterations) {
    volatile double sum = 0.0;
    for (int iter = 0; iter < iterations; ++iter) {
        for (const auto& shape : shapes) {
            sum += shape->area(1.0 + iter * 0.001); // via vtable
        }
    }
}
```

✅ Simple : interface explicite, tous les objets partagent la même vtable

Statique

Templates paramétrés

```
template <int Id>
struct CircleStatic {
    double area(double r) const {
        return PI * r * r
            * (1.0 + Id*0.0001);
    }
};

template <int Id>
double callStatic(double r) {
    return CircleStatic<Id>{}
        .area(r);
}
```

Récursion template

```
template <int N>
struct RunStatic {
    static double call(double r) {
        return callStatic<N>(r)
            + RunStatic<N-1>::call(r);
    }
};

template <>
struct RunStatic<0> {
    static double call(double r) {
        return callStatic<0>(r);
    }
};
```

500 fonctions générées à la compilation

```
void run_static(int iterations) {
    volatile double sum = 0.0;
    for (int iter = 0; iter < iterations; ++iter) {
        sum += RunStatic<499>::call(1.0 + iter * 0.001);
    }
}
```

⚠️ Complexe : templates, récursion, spécialisation, 500 fonctions distinctes

Compilation vs Exécution

🔨 Temps de compilation

Sans optimisation (-O0)

→ Statique 2.0× plus lent

Avec optimisation (-O3)

→ Statique 1.2× plus lent

🚀 Temps d'exécution

Sans optimisation (-O0)

→ Dynamique 1.5× plus lent

Avec optimisation (-O3)

→ Quasiment égal (1.05×)

💡 Le statique transforme du temps d'exécution en temps de compilation.

Guide de décision : quel polymorphisme ?

exalt IT

❓ Question 1

Collection hétérogène ?

Différents types dans un même conteneur

OUI →

DYNAMIQUE

virtual/vtable
- Seule option viable
- std::vector<Base*>
- Résolution runtime

↓ NON

❓ Question 2

Performance critique ?

Hot path, boucle serrée

OUI →

STATIQUE

templates/CRTP
- Performance maximale
- Résolution compile-time
- Code verbeux

NON →

DYNAMIQUE

virtual/vtable
- Simplicité prioritaire
- Maintenance facile
- Overhead acceptable

💡 En pratique : Dynamique par défaut → Mesurer → Optimiser vers statique si besoin

À retenir

Les deux formes de polymorphisme se complètent,
elles ne s'opposent pas.

Complémentarité
Choisir selon le contexte

Mesurer
Benchmark avant optimisation

Balance
Maintenabilité / Performance

"Premature optimization is the root of all evil."
Donald Knuth

Techniques modernes de polymorphisme

📦 std::variant (C++17)

Polymorphisme statique discriminé sans vtable.

```
using Shape = std::variant<Circle, Rectangle>;

std::visit([](auto&& shape) {
    shape.draw();
}, myShape);
```

✅ Pas de vtable, résolution statique

⚠️ Types connus à la compilation

🎯 Concepts (C++20)

Contrats explicites pour les templates.

```
template <typename T>
concept Drawable = requires(const T& t) {
    { t.draw() } -> std::same_as<void>;
};
```

✅ Erreurs claires, contrats documentés

⚠️ Nécessite C++20

Techniques avancées

🔄 CRTP

Polymorphisme statique avec hiérarchie.

```cpp
template <typename Derived>
class ShapeBase {
    void draw() const {
        static_cast<const Derived*>(this)->drawImpl();
    }
};

class Circle : public ShapeBase<Circle> {
    void drawImpl() const { /* ... */ }
};
```

✅ Appels inlinés, pas de vtable

⚠️ Couplage fort, erreurs complexes

🎭 Type Erasure

Polymorphisme dynamique sans héritage.

```cpp
class Drawable {
    template <typename T>
    Drawable(T obj)
      : self_(std::make_shared<Model<T>>(obj)) {}

    void draw() const { self_->draw(); }
private:
    struct Concept {
        virtual void draw() const = 0;
    };
    template <typename T>
    struct Model : Concept { /*...*/ };
    std::shared_ptr<const Concept> self_;
};
```

✅ Interface uniforme

⚠️ Coût similaire aux virtual

Questions ?

Scannez pour me contacter

Personnel
foukinhell@proton.me

Professionnel
florian.charrieau@exalt-it.com

Merci ! 🙏