Classes, Opérateurs et Gestion des ressources

Classes et objets

```
// Déclaration (.hpp)
class Date {
private:
    int day, month, year;
public:
    Date(int d, int m, int y);  // Constructeur
    ~Date();                    // Destructeur
};

// Implémentation (.cpp)
Date::Date(int d, int m, int y) : day(d), month(m), year(y) {}
Date::~Date() {}  // Libération des ressources
```

Manipulation d'objets

```
// Directe
Date d{1, 1, 2023};
d.getMonth();

// Par pointeur
Date* ptr = &d;                 // Pointeur vers objet existant
Date* ptr2 = new Date{1,1,2023}; // Allocation dynamique
ptr2->getMonth();
delete ptr2;                    // Désallocation obligatoire

// Par référence
Date& ref = d;
ref.getMonth();
```

Passage de paramètres

```
// Sans modification des données originales
void f1(const string& s);  // Par référence constante (préférable)
void f2(const string* ptr); // Par pointeur constant (peut être nullptr)
void f3(string s);          // Par valeur (copie)

// Avec modification autorisée
void f4(string& s);         // Par référence (préférable)
void f5(string* ptr);       // Par pointeur (peut être nullptr)
```

Types de retour

```
// Valeur (copie)
X f1() { X x; return x; }

// Pointeur (attention à ne pas retourner d'adresse locale)
X* f2() {
    X* x = new X();
    return x; // OK, allocation dynamique
    // return &localVar; // ERREUR: variable locale détruite en sortie
}

// Référence (attention à ne pas retourner de référence locale)
X& f3() {
    static X x; // Variable statique
    return x;   // OK, perdure après la fonction
    // return localVar; // ERREUR: variable locale détruite en sortie
}
```

Constructeurs et destructeurs

```
class String {
    char* data;
public:
    // Constructeur avec paramètres
    String(const char* s) {
        data = new char[strlen(s) + 1];
        strcpy(data, s);
    }

    // Constructeur de copie - copie profonde
    String(const String& other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data);
    }

    // Constructeur avec liste d'initialisation (préférable)
    String(const char* s) : data(new char[strlen(s) + 1]) {
        strcpy(data, s);
    }

    // Constructeur explicit (empêche les conversions implicites)
    explicit String(int size) : data(new char[size + 1]) {
        data[0] = '\0';
    }

    // Destructeur
    ~String() {
        delete[] data;
    }
};
```

Surcharge d'opérateurs

```
class Vector {
    double x, y;
public:
    Vector(double x, double y) : x(x), y(y) {}

    // Opérateur unaire (négation)
    Vector operator-() const { return Vector{-x, -y}; }

    // Opérateur binaire (addition)
    Vector operator+(const Vector& v) const { return Vector{x + v.x, y + v.y}; }

    // Opérateur d'affectation
    Vector& operator=(const Vector& v) {
        if (this != &v) { x = v.x; y = v.y; }
        return *this;
    }

    // Opérateur d'indexation
    double& operator[](int i) {
        if (i == 0) return x;
        if (i == 1) return y;
        throw std::out_of_range("Index invalide");
    }

    // Opérateur d'appel
    double operator()(int i, int j) const {
        return (i == 0) ? x : y;
    }
};
```

Fonction friend

```
class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}

    // Fonction amie - accès aux membres privés
    friend std::ostream& operator<<(std::ostream& os, const Point& p);

    // Classe amie - accès à tous les membres privés
    friend class PointModifier;
};

std::ostream& operator<<(std::ostream& os, const Point& p) {
    return os << "(" << p.x << ", " << p.y << ")";
}

class PointModifier {
public:
    static void mirror(Point& p) {
        p.x = -p.x;  // Accès direct aux membres privés
        p.y = -p.y;
    }
};
```

Sémantique de déplacement et gestion des ressources

La règle des trois/cinq/zéro

```
class String {
    char* data;
public:
    // Constructeur
    String(const char* s);

    // 1. Destructeur
    ~String();

    // 2. Constructeur de copie
    String(const String& other);

    // 3. Opérateur d'affectation
    String& operator=(const String& other);

    // 4. Constructeur de déplacement (C++11)
    String(String&& other) noexcept;

    // 5. Opérateur d'affectation par déplacement (C++11)
    String& operator=(String&& other) noexcept;
};
```

Implémentation des fonctions de déplacement

```
// Constructeur de déplacement
String::String(String&& other) noexcept
    : data(std::exchange(other.data, nullptr))
{}

// Opérateur d'affectation par déplacement
String& String::operator=(String&& other) noexcept {
    if (this != &other) {
        delete[] data;
        data = std::exchange(other.data, nullptr);
    }
    return *this;
}
```

Lvalues et Rvalues

- **lvalue**: Expressions identifiables (objets avec nom)
- **rvalue**: Expressions temporaires (objets sans nom)

```
int x = 10;  // x est lvalue, 10 est rvalue
int&& r = 10;  // r est lvalue (bien que référence rvalue)
std::move(x);  // Convertit lvalue en rvalue
```

Références universelles et Perfect Forwarding

```
// Référence universelle
template<typename T>
void f(T&& param) {
    g(std::forward<T>(param));  // Perfect forwarding
}

// Utilisation
string s = "hello";
f(s);           // T est string&, param est string&
f(string("hi")); // T est string, param est string&&
```

Héritage et polymorphisme

Déclaration d'une classe dérivée

```
class Animal {
protected:
    std::string name;
public:
    Animal(const std::string& n) : name(n) {}
    virtual ~Animal() = default;  // Destructeur virtuel
    virtual void makeSound() const { std::cout << "..." << std::endl; }
    const std::string& getName() const { return name; }
};

class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}
    void makeSound() const override { std::cout << "Woof!" << std::endl; }
};
```

Types d'héritage

```
// Public: tous les membres conservent leur visibilité
class PublicDerived : public Base {};

// Protected: membres public deviennent protected
class ProtectedDerived : protected Base {};

// Private: tous les membres deviennent private
class PrivateDerived : private Base {};
```

Polymorphisme

```
void hearSound(const Animal& animal) {
    std::cout << animal.getName() << " says: ";
    animal.makeSound();  // Liaison dynamique
}

// Utilisation
Dog d("Rex");
Animal a("Unknown");
hearSound(d);  // "Rex says: Woof!"
hearSound(a);  // "Unknown says: ..."
```

Classes et méthodes abstraites

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;  // Méthode virtuelle pure (abstraite)
    virtual double perimeter() const = 0;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override { return M_PI * radius * radius; }
    double perimeter() const override { return 2 * M_PI * radius; }
};
```

Transtypage

```
// dynamic_cast - fonctionne uniquement avec des classes polymorphiques
Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);
if (dogPtr) {
    // C'est un chien
}

// static_cast - conversion entre types compatibles
float f = static_cast<float>(42);

// const_cast - supprime const
void legacy_API(char* data);
void process(const char* data) {
    legacy_API(const_cast<char*>(data));
}

// reinterpret_cast - conversion bas niveau dangereuse
int* p = reinterpret_cast<int*>(0x12345678);
```

Héritage multiple

```
class A { public: void foo(); };
class B { public: void bar(); };

class C : public A, public B {
public:
    // C hérite des méthodes foo() et bar()
};

// Résolution d'ambiguïté
class D {
public:
    void method();
};

class E {
public:
    void method();
};

class F : public D, public E {
public:
    void callMethod() {
        D::method();  // Spécifie quelle méthode appeler
    }
};

// Héritage en diamant (virtuel)
class Base { protected: int data; };
class Left : virtual public Base {};
class Right : virtual public Base {};
class Derived : public Left, public Right {
    // Un seul membre 'data' hérité
};
```

Attributs et méthodes statiques

```
class Counter {
private:
    inline static int count = 0;  // C++17
public:
    Counter() { ++count; }
    ~Counter() { --count; }

    static int getCount() { return count; }
};

// Utilisation
Counter c1, c2;
cout << Counter::getCount();  // 2
```

Pointeurs intelligents

unique_ptr

```
// Création
auto p = std::make_unique<Person>("John");

// Transfert
auto p2 = std::move(p);  // p devient nullptr

// Accès
p2->getName();
```

shared_ptr

```
// Création
auto sp1 = std::make_shared<Person>("John");

// Partage (incrémente le compteur)
auto sp2 = sp1;  // Maintenant 2 références

// Vérification du nombre de références
sp1.use_count();  // 2
```

weak_ptr

```
// Création
auto sp = std::make_shared<Person>("John");
std::weak_ptr<Person> wp = sp;

// Utilisation sécurisée
if (auto locked = wp.lock()) {
    std::cout << locked->getName();
}
```

enable_shared_from_this

```
class Person : public std::enable_shared_from_this<Person> {
public:
    std::shared_ptr<Person> getShared() {
        return shared_from_this();  // Retourne un shared_ptr sur this
    }
};

auto p = std::make_shared<Person>();
auto p2 = p->getShared();  // p et p2 partagent le même objet
```

Foncteurs et lambdas

Foncteurs

```
class Sum {
    int total = 0;
public:
    void operator()(int value) { total += value; }
    int getTotal() const { return total; }
};

// Utilisation
Sum summer;
summer(5);  // total = 5
summer(10); // total = 15
```

Lambdas

```
// Lambda simple
auto add = [](int a, int b) { return a + b; };
int sum = add(5, 3);  // 8

// Avec capture
int factor = 2;
auto multiply = [factor](int x) { return x * factor; };
multiply(4);  // 8

// Par référence
auto increment = [&factor]() { factor++; };
increment();  // factor devient 3

// Lambda générique (C++14)
auto maximum = [](auto a, auto b) { return a > b ? a : b; };
maximum(3, 7);     // 7
maximum(3.2, 2.1); // 3.2

// Lambda récursif (C++23)
auto factorial = [](this auto&& self, int n) {
    if (n <= 1) return 1;
    return n * self(n - 1);
};
factorial(5);  // 120
```

std::function

```
// Stockage d'un callable
std::function<int(int, int)> operation;
operation = [](int a, int b) { return a + b; };
operation(5, 3);  // 8

// Réaffectation
operation = [](int a, int b) { return a * b; };
operation(5, 3);  // 15

// Alternative au polymorphisme virtuel
void processWithLogger(int x, const std::function<void(std::string)>& logger) {
    logger("Processing " + std::to_string(x));
}

// Utilisation
processWithLogger(42, [](const std::string& msg) {
    std::cout << "LOG: " << msg << std::endl;
});
```

Templates

Template de fonction

```
template<typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

// Utilisation
int i = max(42, 73);  // 73
double d = max(3.14, 2.72);  // 3.14
```

Template de classe

```
template<typename T, int SIZE = 10>
class Array {
    T data[SIZE];
public:
    T& operator[](int index) {
        return data[index];
    }

    int size() const {
        return SIZE;
    }
};

// Utilisation
Array<int, 5> intArray;
Array<std::string> stringArray;  // Taille par défaut (10)
```

Spécialisation de template

```
template<typename T>
class IsPointer {
public:
    static constexpr bool value = false;
};

// Spécialisation pour pointeurs
template<typename T>
class IsPointer<T*> {
public:
    static constexpr bool value = true;
};

// Utilisation
static_assert(IsPointer<int*>::value == true);
static_assert(IsPointer<int>::value == false);
```

RAII (Resource Acquisition Is Initialization)

```
class FileHandle {
    FILE* file;
public:
    FileHandle(const char* filename, const char* mode) {
        file = fopen(filename, mode);
        if (!file) throw std::runtime_error("Cannot open file");
    }

    ~FileHandle() {
        if (file) fclose(file);
    }

    // Empêcher copie (ressource unique)
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Autoriser déplacement
    FileHandle(FileHandle&& other) noexcept : file(other.file) {
        other.file = nullptr;
    }

    FileHandle& operator=(FileHandle&& other) noexcept {
        if (this != &other) {
            if (file) fclose(file);
            file = other.file;
            other.file = nullptr;
        }
        return *this;
    }

    // Interface de fichier
    void write(const char* data) {
        if (file) fputs(data, file);
    }
};

// Utilisation
void process() {
    FileHandle f("data.txt", "w");  // Acquisition de ressource
    f.write("Hello, World!");
    // f est automatiquement fermé à la fin de la portée
}
```

Bonnes pratiques

- Utilisez les initialisations de liste pour les constructeurs
- Respectez la règle des 0/3/5 pour la gestion des ressources
- Préférez `unique_ptr` à `shared_ptr` quand possible
- Utilisez `virtual` pour le destructeur dans les classes de base
- Utilisez `make_unique`/`make_shared` plutôt que `new`
- Utilisez `override` pour les méthodes virtuelles
- Évitez les conversions implicites avec `explicit`
- Appliquez le principe RAII pour la gestion des ressources
- Utilisez std::move pour les rvalues et std::forward pour les références universelles
- Préférez les références constantes aux pointeurs constants quand possible