Université de Douala
École Supérieure Technique La Salle

Programmation Orientée Objet en langage C++

Fiches de travaux dirigés

Classes : **IIA4 & IE4**

Pr Moffo Lonla Bertrand

Année académique 2026–2027

Table des matières

Mode d'emploi **2**

TD 1 — Spécificités du langage C++ **3**

TD 2 — Classes et objets **5**

TD 3 — Construction, destruction et initialisation des objets **7**

TD 4 — Les fonctions amies **9**

TD 5 — Surdéfinition des opérateurs **10**

TD 6 — L'héritage et l'héritage multiple **11**

TD 7 — Le polymorphisme **13**

TD 8 — Les templates **15**

TD 9 — Révision générale : sujet type d'examen **16**

Mode d'emploi

Chaque fiche correspond à un chapitre du cours. Les exercices sont à préparer **avant** la séance de TD ; la séance est consacrée à la correction et aux difficultés rencontrées. Les exercices de programmation se compilent avec :

```
g++ -std=c++17 -Wall -Wextra -o exo exo.cpp
```

Fiche	Thème	Séance de TD prévue
TD 1	Spécificités du C++	travail personnel (séance 1)
TD 2	Classes et objets	travail personnel / TP séance 2
TD 3	Construction, destruction, initialisation	travail personnel / TP séance 3
TD 4	Fonctions amies	séance 4 (12/12/2026)
TD 5	Surdéfinition des opérateurs	séance 5 (19/12/2026)
TD 6	Héritage et héritage multiple	séance 6 (09/01/2027)
TD 7	Polymorphisme	travail personnel / TP séance 7
TD 8	Templates	séance 8 (23/01/2027)
TD 9	Révision générale : sujet type d'examen	séance 9 (30/01/2027)

TD 1 — Spécificités du langage C++

Exercice 1 — Pointeurs constants et fonctions `inline` questions de cours

1. Identifier l'assertion **erronée** au sujet des fonctions `inline` : (a) elles peuvent contenir plusieurs instructions ; (b) elles doivent être `static` ; (c) elles peuvent améliorer les performances du programme ; (d) elles sont expansées à l'endroit de chaque appel.
2. On considère les déclarations `int i = 10; const int IC = 20; const int *pic;`. Laquelle des instructions suivantes est illégale ? (a) `pic = &IC;` (b) `pic = &i;` (c) `*pic = IC;` (d) `i = *pic;`
3. On considère `int i = 10; int j;`. Laquelle des lignes suivantes est légale ? (a) `int * const pi = &j; *pi = i;` (b) `int * const pi; pi = &j;` (c) `int * const pi = &j; pi = &i;`
4. Quelle différence y a-t-il entre une fonction `inline` et une macro `#define CARRE(x) x*x` ? Que vaut `CARRE(a+1)` ?

Exercice 2 — Transmission par référence

1. Écrire une fonction `echange` qui permute deux entiers, d'abord en transmettant leurs **adresses** (style C), puis en les transmettant **par référence**. Comparer les deux appels dans `main`.
2. Écrire une fonction `void minMax(const int t[], int n, int &mini, int &maxi)` qui renvoie, par ses deux derniers arguments, le minimum et le maximum d'un tableau.
3. Pourquoi le prototype `int &plusGrand(int a, int b) { return a > b ? a : b; }` est-il dangereux ? Le corriger.

Exercice 3 — Surdéfinition et arguments par défaut

On dispose des déclarations suivantes :

```
void f(int);              // (1)
void f(double);           // (2)
void f(int, double = 0);  // (3)
void g(char, int = 1, double = 2.5);
```

1. Ces déclarations sont-elles toutes acceptées ? Pour chacun des appels `f(5.0)`, `f(3, 1.5)`, `f('a', 2)`, `f(3)`, dire quelle fonction est appelée ou pourquoi l'appel est rejeté.
2. Quels sont les appels corrects parmi : `g('x')`, `g('x', 5)`, `g('x', , 1.0)`, `g()` ? Peut-on déclarer `void h(int a = 0, int b);` ?

Exercice 4 — Allocation dynamique avec `new` et `delete`

Écrire un programme qui :

1. lit un entier `n`, puis alloue dynamiquement un tableau de `n` réels ;
2. lit les `n` notes d'une classe et les range dans ce tableau ;
3. affiche les notes à l'aide d'une fonction `void afficher(const double *t, int n, char sep = ' ')` (argument par défaut), puis affiche la moyenne et le nombre de notes supérieures à la moyenne ;

4. libère la mémoire.

TD 2 — Classes et objets

Exercice 1 — Notion d'objet : le compte bancaire

On veut modéliser un compte bancaire.

1. Donner des exemples d'**état**, de **comportement** et d'**identité** pour un objet « compte bancaire ».
2. Écrire la déclaration d'une classe `CompteBancaire` ayant deux attributs privés, `titulaire` (`string`) et `solde` (`double`), et les méthodes publiques :
   - `void initialiser(const string &nom, double montant)` ;
   - `void deposer(double montant)` ;
   - `bool retirer(double montant)`, qui refuse le retrait (et renvoie `false`) si le solde est insuffisant ou le montant négatif ;
   - `void afficher() const` ;
   - les accesseurs `getTitulaire()` et `getSolde()`, et un mutateur `setTitulaire()`.
3. Définir les fonctions membres hors de la classe.
4. Dans `main`, instancier un objet automatique, puis un objet alloué dynamiquement, et tester toutes les méthodes. Ajouter une méthode `bool virer(CompteBancaire &dest, double montant)`.
5. Pourquoi n'a-t-on pas prévu de mutateur `setSolde()` ?

Exercice 2 — Recherche d'erreurs

Le programme suivant contient **six** erreurs de compilation. Les repérer et les corriger.

```
class Rectangle {
    double largeur, hauteur;
public:
    void fixer(double l, double h);
    double surface() const;
    void agrandir(double k) const { largeur *= k; hauteur *= k; }
}
void fixer(double l, double h) { largeur = l; hauteur = h; }
double Rectangle::surface() { return largeur * hauteur; }
int main() {
    Rectangle r;
    r.fixer(2, 3);
    r.largeur = 5;
    std::cout << Rectangle.surface();
}
```

Exercice 3 — Le pointeur `this` et les méthodes surdéfinies

Écrire une classe `Point` (coordonnées `x`, `y` privées) comportant :

1. une méthode `initialise(double x = 0, double y = 0)` dont les arguments portent le même nom que les attributs ;
2. deux méthodes `deplace`, l'une recevant deux réels `dx`, `dy`, l'autre un `Point` dont elle ajoute les coordonnées ;

3. une méthode `bool coincide(const Point &) const` ;
4. une méthode `Point &symetrique()` qui transforme le point en son symétrique par rapport à l'origine et renvoie l'objet lui-même, de sorte que l'on puisse écrire `p.symetrique().affiche();`.

Exercice 4 — Membres statiques : numérotation automatique

On veut que chaque objet `Etudiant` reçoive automatiquement un matricule unique, de la forme `IIA4-001`, `IIA4-002`, etc., au moment de son initialisation.

1. Quel type de membre faut-il utiliser pour mémoriser le dernier numéro attribué ? Où doit-il être initialisé ?
2. Écrire la classe `Etudiant` (nom, matricule) avec une méthode `inscrire(const string &nom)`, une méthode `afficher()` et une fonction membre statique `effectif()` qui renvoie le nombre d'étudiants inscrits.
3. Peut-on appeler `effectif()` avant d'avoir créé le moindre étudiant ? Une fonction membre statique peut-elle utiliser l'attribut `nom` ?

TD 3 — Construction, destruction et initialisation des objets

Exercice 1 — Prévoir l'ordre des appels

Sans l'exécuter, donner l'affichage produit par le programme suivant. Justifier chaque ligne.

td3_ex1.cpp

```cpp
#include <iostream>
using namespace std;

class T {
    int n;
public:
    T(int v) : n(v) { cout << "+T(" << n << ") "; }
    ~T()            { cout << "-T(" << n << ") "; }
};

T a(1);

void f() {
    static T s(2);
    T c(3);
}

int main() {
    cout << "\n[main] ";
    T b(4);
    f();
    f();
    {
        T d(5);
    }
    T *p = new T(6);
    T *q = new T(7);
    delete p;
    (void)q;              // q n'est jamais libéré
    cout << "\n[fin main] ";
    return 0;
}
```

Exercice 2 — Une pile d'entiers de taille variable

On souhaite réaliser une classe `PileEntiers` dont la capacité est fixée à la création de l'objet. Les éléments sont rangés dans un tableau alloué dynamiquement.

1. Déclarer la classe avec les attributs `capacite`, `nb` (nombre d'éléments présents) et `tab` (`int*`).
2. Écrire un constructeur recevant la capacité (20 par défaut), utilisant une liste d'initialisation, et le destructeur.
3. Écrire les méthodes `bool empiler(int)`, `bool depiler(int &)`, `bool vide() const` et `bool pleine() const`.

4. Que se passe-t-il si l'on écrit `PileEntiers q = p;` sans avoir défini de constructeur de recopie ? Écrire ce constructeur.
5. Tester la classe avec un objet automatique, un objet dynamique et une fonction recevant une pile **par valeur**.

Exercice 3 — Objets membres : la classe `Personne`

1. Écrire une classe `Date` (jour, mois, année) munie d'un constructeur à trois arguments et d'une méthode `afficher()`. Le constructeur et le destructeur affichent un message.
2. Écrire une classe `Personne` ayant pour attributs un nom (`string`) et une date de naissance (objet membre de type `Date`). Son constructeur reçoit le nom, le jour, le mois et l'année.
3. Pourquoi est-on **obligé** d'utiliser une liste d'initialisation dans le constructeur de `Personne` ?
4. Dans quel ordre sont appelés les constructeurs et les destructeurs ? Vérifier en exécutant le programme.
5. Déclarer un tableau de 2 personnes initialisé à la déclaration.

TD 4 — Les fonctions amies

Exercice 1 — Fonction indépendante amie

Écrire une classe `Personne` ayant deux données membres privées, `nom` et `age`, et un constructeur avec paramètres. Écrire ensuite, à l'extérieur de la classe, une fonction amie indépendante `afficher` qui affiche les informations d'une personne, et une fonction amie `plusAgee` qui reçoit deux personnes et renvoie le nom de la plus âgée.

Exercice 2 — Fonction amie de deux classes

On dispose d'une classe `Point` (coordonnées privées `x`, `y`) et d'une classe `Cercle` (centre `xc`, `yc` et rayon `r`, privés).

1. Écrire une fonction `bool interieur(const Point &, const Cercle &)` qui indique si un point est à l'intérieur d'un cercle (ou sur le cercle). Elle doit accéder directement aux membres privés des deux classes.
2. Pourquoi faut-il une déclaration anticipée de l'une des deux classes ?
3. Remplacer ensuite cette fonction par une fonction **membre** `bool Cercle::contient(const Point &) const`, amie de la classe `Point`. Quelle déclaration faut-il placer dans `Point` ? Dans quel ordre doit-on écrire le code ?

Exercice 3 — Propriétés de l'amitié

questions de cours

Répondre par vrai ou faux, en justifiant.

1. Une fonction peut se déclarer elle-même amie d'une classe.
2. Si la classe `B` est amie de la classe `A`, alors `A` est amie de `B`.
3. Si `B` est amie de `A` et `C` amie de `B`, alors `C` est amie de `A`.
4. Une fonction amie peut utiliser le pointeur `this`.
5. La position de la déclaration `friend` (partie `public` ou `private`) n'a pas d'importance.
6. Une classe dérivée d'une classe amie de `A` est elle aussi amie de `A`.

TD 5 — Surdéfinition des opérateurs

Exercice 1 — La classe `Fraction`

Une fraction est représentée par un numérateur et un dénominateur entiers. Elle est toujours stockée sous forme **irréductible**, avec un dénominateur strictement positif.

1. Écrire la classe `Fraction` avec un constructeur `Fraction(int num = 0, int den = 1)` qui simplifie la fraction (à l'aide d'une fonction privée `pgcd`) et traite le signe.
2. Surdéfinir, sous forme de fonctions **amies**, les opérateurs `+`, `-`, `*`, `/` et `==`.
3. Surdéfinir, sous forme de fonctions **membres**, `+=` et les deux formes de `++` (qui ajoutent 1).
4. Surdéfinir `<<` pour afficher `3/4` (ou `2` si le dénominateur vaut 1).
5. L'expression `1 + f`, où `f` est une `Fraction`, est-elle acceptée ? Le serait-elle si `operator+` était une fonction membre ?

Exercice 2 — Un vecteur dynamique sûr (règle des trois)

On reprend la classe `Vecteur` du cours : `n` réels rangés dans un tableau dynamique.

1. Écrire le constructeur `Vecteur(int n, double val = 0)`, le constructeur de recopie et le destructeur.
2. Surdéfinir l'opérateur d'affectation `=`. Expliquer le test `if (this != &v)` et la valeur de retour.
3. Surdéfinir l'opérateur `[]` en deux versions (objet modifiable et objet constant). Il doit vérifier l'indice et lever une exception `std::out_of_range` en cas de dépassement.
4. Surdéfinir `==`, `+` (somme de deux vecteurs de même taille) et `<<`.

Exercice 3 — Choisir la forme d'un opérateur

questions de cours

Pour une classe `Heure` (heures, minutes), indiquer pour chacun des opérateurs suivants s'il doit, peut ou ne peut pas être une fonction membre, et donner son prototype :

```
h1 + h2   h += 15   h1 < h2   cout << h   cin >> h   h = h2   h[0]   -h
```

TD 6 — L'héritage et l'héritage multiple

Exercice 1 — Personnel d'une école

1. Écrire une classe `Personne` avec deux attributs **protégés** `nom` et `prenom`, un constructeur à deux arguments et une méthode `afficher()`.
2. Dériver publiquement de `Personne` une classe `Etudiant` (attributs `matricule` et `filiere`) et une classe `Enseignant` (attributs `grade` et `volumeHoraire`). Chaque constructeur transmet le nom et le prénom au constructeur de `Personne`.
3. Redéfinir `afficher()` dans chaque classe dérivée en réutilisant la version de `Personne`.
4. Ajouter dans `Enseignant` une méthode `double prime(double tauxHoraire) const`.
5. Ajouter des messages dans les constructeurs et destructeurs pour vérifier l'ordre des appels.

Exercice 2 — Droits d'accès et modes de dérivation

Pour chacune des lignes marquées `(n)`, dire si elle est acceptée par le compilateur.

```
class A {
    int pv;
protected:
    int pt;
public:
    int pb;
};
class B : public A {
public:
    void f() { pv = 1;   /* (1) */  pt = 2;  /* (2) */  pb = 3;  /* (3) */ }
};
class C : private A {
public:
    void g() { pt = 4;   /* (4) */  pb = 5;  /* (5) */ }
};
class D : public C {
public:
    void h() { pb = 6;   /* (6) */ }
};
int main() {
    B b;  C c;
    b.pb = 7;   // (7)
    b.pt = 8;   // (8)
    c.pb = 9;   // (9)
    A *pa = &b; // (10)
    A *pc = &c; // (11)
}
```

Exercice 3 — Héritage multiple : horloge-calendrier

1. Écrire une classe `Horloge` (heures, minutes) avec un constructeur, une méthode `avancer(int minutes)` qui renvoie le nombre de jours écoulés (passages par minuit), et une méthode `afficher()`.

2. Écrire une classe `Calendrier` (jour, mois, année) avec un constructeur, une méthode `lendemain()` (on supposera tous les mois de 30 jours) et une méthode `afficher()`.
3. Dériver de ces deux classes une classe `HorlogeCalendrier` dont la méthode `avancer(int minutes)` fait avancer l'heure *et* la date. Redéfinir `afficher()`.
4. Dans quel ordre les constructeurs des classes de base sont-ils appelés ? Comment lever l'ambiguïté sur `afficher()` ?

Exercice 4 — Le losange : héritage virtuel

Une école emploie des étudiants comme moniteurs : un `Moniteur` est à la fois un `Etudiant` et un `Salarie`, qui dérivent tous deux de `Personne`.

1. Combien de sous-objets `Personne` contient un `Moniteur` si les dérivations sont ordinaires ? Quel problème cela pose-t-il pour l'accès à `nom` ?
2. Modifier les déclarations pour qu'un `Moniteur` ne contienne qu'un seul sous-objet `Personne`. Écrire les constructeurs et vérifier, par des messages, le nombre d'appels au constructeur de `Personne`.

TD 7 — Le polymorphisme

Exercice 1 — Liaison statique ou dynamique ?

Donner l'affichage du programme suivant, puis expliquer chaque ligne.

td7_ex1.cpp

```cpp
#include <iostream>
using namespace std;

class A {
public:
    void qui() const          { cout << "A::qui  "; }
    virtual void quoi() const { cout << "A::quoi" << endl; }
    void montre() const       { quoi(); }
    virtual ~A() { }
};

class B : public A {
public:
    void qui() const           { cout << "B::qui  "; }
    void quoi() const override { cout << "B::quoi" << endl; }
};

void appel(A x) { x.qui(); x.quoi(); }

int main() {
    A a; B b;
    A *pa = &b;
    A &ra = b;
    cout << "1) "; b.qui();   b.quoi();
    cout << "2) "; pa->qui(); pa->quoi();
    cout << "3) "; ra.qui();  ra.quoi();
    a = b;
    cout << "4) "; a.qui();   a.quoi();
    cout << "5) "; appel(b);
    cout << "6) "; pa->montre();
    return 0;
}
```

Exercice 2 — Classe abstraite : la paie des employés

Une entreprise emploie trois catégories de personnel :

— les **permanents**, payés par un salaire mensuel fixe ;

— les **vacataires**, payés à l'heure (nombre d'heures × taux horaire) ;

— les **commerciaux**, qui reçoivent un fixe plus une commission de 5 % de leur chiffre d'affaires.

1. Écrire une classe abstraite `Employe` contenant le nom de l'employé, une fonction virtuelle pure `double salaire() const`, une fonction virtuelle `string categorie() const` et une méthode non virtuelle `afficher()` qui affiche le nom, la catégorie et le salaire.
2. Écrire les classes concrètes `Permanent`, `Vacataire` et `Commercial`. On pourra dériver `Commercial` de `Permanent`.

3. Dans `main`, ranger plusieurs employés dans un `std::vector<Employe*>`, afficher la fiche de chacun et la masse salariale totale, puis libérer la mémoire.
4. Pourquoi le destructeur de `Employe` doit-il être virtuel ?
5. Que se passe-t-il si l'on écrit `Employe e("X");` ? Et si `Vacataire` oublie de redéfinir `salaire()` ?

Exercice 3 — Destructeur virtuel

Le programme suivant a une fuite de mémoire. L'exécuter, expliquer l'affichage obtenu, puis le corriger.

```
class Base {
public:
    ~Base() { std::cout << "~Base" << std::endl; }
};
class Tampon : public Base {
    char *buf;
public:
    Tampon(int n) : buf(new char[n]) { }
    ~Tampon() { delete[] buf; std::cout << "~Tampon" << std::endl; }
};
int main() {
    Base *p = new Tampon(1024);
    delete p;
}
```

TD 8 — Les templates

Exercice 1 — Fonctions templates

1. Écrire une fonction template `carre` qui calcule le carré d'une valeur de type quelconque (le résultat a le même type). La tester avec un `int`, un `double` et un objet `Complexe` (on rappelle que `Complexe` surdéfinit $*$).
2. Écrire une fonction template `echanger(T &a, T &b)`.
3. Écrire une fonction template `T maximum(const T t[], int n)` qui renvoie le plus grand élément d'un tableau. Quelle opération le type `T` doit-il fournir ?
4. Pourquoi l'appel `maximum(t, 3)` avec `const char *t[] = {"pomme", "ananas", "zeste"}` ne donne-t-il pas le résultat attendu ? Écrire une spécialisation qui corrige ce cas.
5. L'appel `carre<double>(3)` est-il correct ? Et `echanger(n, x)` avec `int n; double x;` ?

Exercice 2 — Classe template `Tableau`

On veut une classe générique `Tableau<T>` représentant un tableau dynamique d'éléments de type `T`, de taille fixée à la construction.

1. Écrire la classe avec un constructeur `Tableau(int n, const T &val = T())`, le constructeur de recopie, l'affectation et le destructeur.
2. Ajouter l'opérateur `[]` (avec contrôle de l'indice), une méthode `taille()` et une méthode `T somme() const`. Définir `somme()` **hors** de la classe.
3. Surdéfinir `<<` pour afficher le tableau.
4. Tester avec `Tableau<int>`, `Tableau<double>` et `Tableau<string>`. Que représente `T()` pour chacun de ces types ? Que signifie la somme pour des `string` ?

Exercice 3 — Utiliser la bibliothèque standard

On dispose d'une classe `Etudiant` (nom, moyenne).

1. Ranger quelques étudiants dans un `vector<Etudiant>`.
2. Surdéfinir l'opérateur `<` pour comparer deux étudiants par moyenne, puis trier le vecteur par moyenne croissante avec `std::sort`.
3. Trier ensuite par ordre alphabétique des noms en passant à `std::sort` une fonction de comparaison (fonction ordinaire ou lambda).
4. Compter avec `std::count_if` le nombre d'étudiants admis (moyenne ≥ 10).

TD 9 — Révision générale : sujet type d'examen

Durée : 2 heures	Documents non autorisés
Barème indicatif : 6 + 4 + 6 + 4 = 20 points	

Exercice 1 — Notion d'objet : le thermostat

6 points

Les questions 2 à 7 font partie d'un même programme C++.

1. Donner, pour un objet « thermostat d'ambiance », des exemples d'état, de comportement et d'identité.
2. Fournir une déclaration de la classe `Thermostat`, ayant pour attributs privés une température de consigne `consigne` (`double`) et un indicateur `marche` (`bool`).
3. Écrire un constructeur recevant la consigne initiale (20 °C par défaut) ; le thermostat est à l'arrêt à sa création.
4. Instancier un objet automatique de consigne 22 °C, puis allouer dynamiquement un objet de consigne par défaut et le libérer.
5. Coder l'accesseur et le mutateur de `consigne`. Le mutateur refuse toute valeur hors de l'intervalle [5 ; 30].
6. Coder la définition de la méthode publique `reglerConsigne`, qui reçoit en argument un écart (positif ou négatif) à ajouter à la consigne et renvoie un booléen indiquant si le réglage a été accepté.
7. Ajouter un attribut permettant de connaître à tout moment le nombre de thermostats existants. Que faut-il ajouter au constructeur et au destructeur ?

Exercice 2 — Questions à choix multiple

4 points

Pour chaque question, une seule réponse est exacte.

1. Le prototype correct d'un constructeur de recopie de la classe `X` est : (a) `X(X);` (b) `X(const X &);` (c) `void X(const X &);` (d) `X &X(X *);`
2. Une fonction amie d'une classe : (a) est une fonction membre de la classe ; (b) dispose du pointeur `this` ; (c) a accès aux membres privés de la classe ; (d) est héritée par les classes dérivées.
3. Une classe abstraite : (a) ne contient que des fonctions virtuelles pures ; (b) contient au moins une fonction virtuelle pure ; (c) ne peut pas avoir de constructeur ; (d) ne peut pas servir de type de pointeur.
4. Avec `template <typename T> T plus(T a, T b) { return a + b; }`, quel appel est refusé à la compilation ? (a) `plus(2, 3)` (b) `plus(2.5, 1.0)` (c) `plus(2, 1.5)` (d) `plus<double>(2, 1.5)`

Exercice 3 — Héritage et polymorphisme : chaîne d'acquisition

6 points

Une carte d'acquisition lit des capteurs de natures différentes. Chaque capteur fournit une grandeur brute (tension ou durée) que l'on convertit en grandeur physique.

1. Écrire une classe abstraite `Capteur` ayant un attribut protégé `nom`, un constructeur, une fonction virtuelle pure `double mesure() const` (grandeur physique), une fonction virtuelle pure `string unite() const`, une méthode non virtuelle `afficher()` et un destructeur virtuel.
2. Écrire la classe `CapteurTemperature`, de type LM35 : il reçoit la tension lue en millivolts, et $T = V/10$ (en °C).
3. Écrire la classe `CapteurDistance`, de type ultrason HC-SR04 : il reçoit la durée de l'écho en microsecondes, et $d = t/58$ (en cm).
4. Dans `main`, ranger trois capteurs dans un tableau de pointeurs `Capteur*`, afficher leurs mesures dans une boucle, puis libérer la mémoire.
5. Expliquer pourquoi la boucle de la question 4 n'aura pas à être modifiée si l'on ajoute plus tard une classe `CapteurHumidite`.

Exercice 4 — Templates

4 points

1. Créer une fonction template `moyenne` qui calcule la moyenne des `n` éléments d'un tableau de type quelconque. Le résultat est toujours un `double`.
2. Créer une classe template `Paire<T>` contenant deux valeurs de type `T`, avec un constructeur, une méthode `plusGrand()`, une méthode `echanger()` qui permute les deux valeurs, et l'opérateur `<<`.
3. Écrire un petit programme utilisant ces patrons avec des `int`, des `double` et des `string`.