Solution de l'Atelier 5 : Concepts Avancés de C++ (Templates, Exceptions, Surcharge)

Exercice 1 : Fonction de recherche générique

Cet exercice introduit le concept de **templates de fonctions** et l'utilisation des **itérateurs** pour créer une fonction de recherche universelle.

Concepts C++ requis

- **Templates de fonctions** (template <typename T>)
- **Itérateurs** (généralisation des pointeurs)

Code C++ (Exercice1_RechercheGenerique.cpp)

```
template <typename Iterator, typename T>
```

```
bool rechercher(Iterator debut, Iterator fin, const T& valeur) {
    for (Iterator it = debut; it != fin; ++it) {
        if (*it == valeur) {
            return true; // Valeur trouvée
        }
    }
    return false; // Valeur non trouvée
}
```

```
// ... (Programme principal pour les tests)
```

Exercice 2 : Gestion des exceptions

Cet exercice vise à implémenter une gestion robuste des erreurs en utilisant le mécanisme try-catch et en lançant des exceptions standard (std::out_of_range, std::runtime_error).

Concepts C++ requis

- **Exceptions** (try, catch, throw)
- **Exceptions standard** (de la bibliothèque <stdexcept>)

Code C++ (Exercice2_Exceptions.cpp)

```
#include <iostream>
```

```
#include <vector>
#include <stdexcept>

class Test {
public:
  static int tableau(int indice, int diviseur) {
    std::vector<int> tab = {17, 12, 15, 38, 29, 157, 89, -22, 0, 5};

    // 1. Vérification de l'indice
    if (indice < 0 || indice >= tab.size()) {
      throw std::out_of_range("Indice hors limites du tableau.");
    }

    // 2. Vérification du diviseur
    if (diviseur == 0) {
      throw std::runtime_error("Division par zéro impossible.");
    }

    return tab[indice] / diviseur;
  }
};
```

```
// ... (Programme principal avec try-catch)
```

Exercice 3 : Classe vect et surcharge d'opérateurs

Cet exercice se concentre sur la gestion de la mémoire dynamique pour une classe de type tableau (vect) et l'implémentation de la **Règle de Trois** (Constructeur de copie, Opérateur d'affectation, Destructeur) pour garantir une **copie profonde**.

Concepts C++ requis

- **Surcharge d'opérateurs** (operator[], operator=)

- **Règle de Trois** (gestion de la mémoire dynamique)
- **Copie profonde**

Code C++ (Exercice3_Vect.cpp)

```
class vect {

protected:
  int n;
  int* ptr;

public:
  // Constructeur, Destructeur (libère ptr)

  // Constructeur de copie (Copie Profonde)
  vect(const vect& autre) : n(autre.n) {
    ptr = new int[n];
    for (int i = 0; i < n; ++i) {
      ptr[i] = autre.ptr[i];
    }
  }

  // Opérateur d'affectation (Copie Profonde)
  vect& operator=(const vect& autre) {
    if (this != &autre) {
      delete[] ptr; // Libération de l'ancienne mémoire
      n = autre.n;
      ptr = new int[n];
      // Copie des données
      for (int i = 0; i < n; ++i) {
        ptr[i] = autre.ptr[i];
      }
    }
    return *this;
  }

  // Opérateur d'indexation
  int& operator[](int indice) {
    // ... (vérification des limites)
    return ptr[indice];
  }

};
```

Exercice 4 : Constructeurs et Destructeurs

Cet exercice teste la compréhension de l'ordre d'appel des constructeurs et destructeurs dans le cas de l'**héritage multiple**.

Ordre d'appel

L'ordre d'appel des constructeurs est le suivant :

1. Constructeurs des classes de base (dans l'ordre de déclaration dans la classe dérivée : B puis A).
2. Constructeur de la classe dérivée (C).

L'ordre d'appel des destructeurs est l'inverse :

3. Destructeur de la classe dérivée (C).
4. Destructeurs des classes de base (dans l'ordre inverse de la construction : A puis B).

Résultats attendus

Pour l'instruction C c1(10, 12, 5, 5.0f); (en supposant l'ajout d'un 3ème argument int pour correspondre à la signature du constructeur C):

Action	Sortie
Construction	** construction objet B : 3 5
	** construction objet A : 22 0.833333
	** construction objet C : 5
Destruction	** destruction objet C : 5
	** destruction objet A : 22 0.833333
	** destruction objet B : 3 5

Exercice 5 : Template de fonction

Création d'un template de fonction simple pour calculer le carré d'une valeur, démontrant la capacité du template à fonctionner avec différents types de données.

Code C++ (Exercice5_TemplateFonction.cpp)

```cpp
template <typename T>
```

```cpp
T carre(T valeur) {
    return valeur * valeur;
}
```

```cpp
// ... (Programme principal pour les tests)
```

Exercice 6 : Template de classe

Création d'un template de classe point pour gérer des coordonnées de n'importe quel type (int, float, etc.).

Concepts C++ requis

- **Templates de classes** (template <class T>)

Code C++ (Exercice6_TemplateClasse.cpp)

```cpp
template <class T>
```

```cpp
class point {
private:
    T abs;
    T ord;

public:
    point(T x, T y) : abs(x), ord(y) {}

    void affiche() const {
        std::cout << "Coordonnees : " << abs << " " << ord << std::endl;
    }
};
```

```cpp
// ... (Programme principal pour les tests)
```

Exercice 7 : Gestion des exceptions (suite)

Analyse du comportement des exceptions, y compris le lancement d'une exception personnalisée et la **relance** d'une exception (throw;).

Résultats attendus

L'appel à g() lance une exception qui est capturée dans g(), affichant "dans g() : 999", puis relancée (throw;). Elle est ensuite capturée dans main(), affichant "dans main : 999". Le deuxième bloc try-catch capture l'exception directement, affichant "dans f : 999".

Action	Sortie
Bloc 1	dans g() : 999
	dans main : 999
Bloc 2	dans f : 999

Exercice 8 : Classe Stack et surcharge d'opérateurs

Implémentation d'une classe Stack (Pile) avec gestion de la mémoire dynamique et surcharge des opérateurs << (empiler) et >> (dépiler).

Concepts C++ requis

- **Surcharge d'opérateurs** (operator<<, operator>>, operator++, operator--)
- **Règle de Trois** (Constructeur de copie et Opérateur d'affectation pour la copie profonde)

Code C++ (Exercice8_Stack.cpp)

```
class Stack {

private:
   int* elements;
   int tailleMax;
```

```
    int sommet;

public:
    // Constructeur, Destructeur, Constructeur de copie, Opérateur d'affectation (Copie Profonde)

    // Opérateur << (Empiler)
    Stack& operator<<(int val) {
        if (sommet < tailleMax - 1) {
            elements[++sommet] = val;
        } else {
            // Gestion de la pile pleine
        }
        return *this;
    }

    // Opérateur >> (Dépiler)
    Stack& operator>>(int& val) {
        if (sommet >= 0) {
            val = elements[sommet--];
        } else {
            // Gestion de la pile vide
        }
        return *this;
    }

    // Opérateur ++ (Vérifie si la pile est pleine)
    int operator++() {
        return (sommet == tailleMax - 1) ? 1 : 0;
    }

    // Opérateur -- (Vérifie si la pile est vide)
    int operator--() {
        return (sommet == -1) ? 1 : 0;
    }
```

```
};
```