TP 10 : Templates

Imad Kissami

21 Avril 2025

Objectif

- Créer un `Makefile` pour compiler tous les fichiers `.cpp`.
- Tous les fichiers doivent être regroupés dans un dossier `TP10_Nom_Prénom`.
- Maîtriser les concepts des templates en C++ (fonctions templates, classes templates, spécialisation, déduction de types, `if constexpr`, variadic templates, méta-programmation).

Exercice 1 : Fonction template pour le maximum

Écrivez une fonction template `maximum` qui prend deux paramètres de type générique et retourne le plus grand des deux. Implémentez une spécialisation pour les chaînes de caractères (`std::string`) qui compare les chaînes lexicographiquement.

Instructions :

- La fonction template `maximum` utilise un type générique `T` et compare les valeurs avec l'opérateur `>`.
- Une spécialisation explicite pour `std::string` utilise `std::string::compare` pour une comparaison lexicographique.
- Le programme principal teste la fonction avec des entiers, des nombres flottants, et des chaînes de caractères.

Exemple du main.cpp :

```cpp
int main() {
  std::cout << maximum(5, 10) << std::endl; // Affiche 10
  std::cout << maximum(3.14, 2.71) << std::endl; // Affiche 3.14
  std::cout << maximum(std::string("chat"), std::string("chien")) << std::endl; // Affiche chien

  return 0;
}
```

Exercice 2 : Classe template Conteneur

Créez une classe template `Conteneur` qui gère une liste d'éléments de type `T` avec une capacité maximale fixée à la construction. Implémentez des méthodes pour ajouter (`ajouter`) et obtenir (`obtenir`) des éléments.

Instructions :

- La classe utilise un `std::vector<T>` pour stocker les éléments et vérifie que la capacité n'est pas dépassée.
- Si la capacité est dépassée, affichez un message d'erreur sans lever d'exception.
- Le programme principal teste la classe avec des entiers et des chaînes de caractères.

Exemple du main.cpp :

```
int main() {
  Conteneur<int> c1(3); // Capacité de 3
  c1.ajouter(10);
  c1.ajouter(20);
  c1.ajouter(30);
  c1.ajouter(40); // Capacité dépassée
  std::cout << c1.obtenir(0) << "␣" << c1.obtenir(1) << std::endl; // Affiche 10 20

  Conteneur<std::string> c2(2);
  c2.ajouter("Bonjour");
  c2.ajouter("Monde");
  std::cout << c2.obtenir(0) << "␣" << c2.obtenir(1) << std::endl; // Affiche Bonjour Monde

  return 0;
}
```

Exercice 3 : Classe template Paire avec spécialisation et if constexpr

Implémentez une classe template `Paire` qui stocke deux valeurs de types potentiellement différents (`T` et `U`). Ajoutez une méthode `afficher` pour afficher les valeurs. Implémentez :

- Une spécialisation partielle pour `Paire<T, int>` qui affiche un message spécifique.
- Une méthode `afficher` utilisant `if constexpr` pour gérer les booléens différemment.

Testez la classe dans un conteneur hétérogène de paires.

Instructions :

- La classe `Paire` stocke `premier` (type `T`) et `second` (type `U`).
- La spécialisation partielle pour `Paire<T, int>` affiche un message indiquant que le second paramètre est un entier.
- `if constexpr` dans `afficher` affiche les booléens comme "vrai" ou "faux".
- Utilisez un `std::vector<std::unique_ptr<PaireBase>>` pour stocker des paires hétérogènes, avec une classe de base `PaireBase`.

Exemple du main.cpp :

```cpp
int main() {
  std::vector<std::unique_ptr<PaireBase>> paires;

  paires.push_back(std::make_unique<Paire<double, int>>(3.14, 42));
  paires.push_back(std::make_unique<Paire<int, bool>>(10, true));
  paires.push_back(std::make_unique<Paire<std::string, double>>("Test", 2.71));

  for (const auto& p : paires) {
    p->afficher();
  }

  return 0;
}
```

Exercice 4 : Template Matrix avec méta-programmation

Implémentez une classe template `Matrix` qui représente une matrice de taille fixe (`M` lignes, `N` colonnes) d'éléments de type `T`. Ajoutez une méthode pour l'addition de matrices et une spécialisation partielle pour gérer les cas où les dimensions sont incompatibles.

Instructions :

- La classe utilise un tableau `std::array` pour stocker les éléments et des paramètres non-types `M` et `N` pour les dimensions.
- Implémentez une méthode `add` pour additionner deux matrices de mêmes dimensions.
- Utilisez `static_assert` pour vérifier la compatibilité des dimensions lors de l'addition.
- Implémentez une spécialisation partielle via une classe `MatrixAdder` pour gérer les cas où les dimensions des matrices sont incompatibles, en affichant un message d'erreur et en retournant une matrice vide.
- Le programme principal teste l'addition de matrices avec des entiers et des flottants.

Exemple du main.cpp :

```cpp
int main() {
  Matrix<int, 2, 2> m1;
  m1.set(0, 0, 1); m1.set(0, 1, 2);
  m1.set(1, 0, 3); m1.set(1, 1, 4);

  Matrix<int, 2, 2> m2;
  m2.set(0, 0, 5); m2.set(0, 1, 6);
  m2.set(1, 0, 7); m2.set(1, 1, 8);

  auto m3 = m1.add(m2);
  std::cout << m3.get(0, 0) << "␣" << m3.get(0, 1) << std::endl; // Affiche 6 8
  std::cout << m3.get(1, 0) << "␣" << m3.get(1, 1) << std::endl; // Affiche 10 12

  Matrix<int, 2, 3> m4; // Incompatible
  // m1.add(m4); // Erreur de compilation (static_assert)

  return 0;
}
```