IN204

Programmation Orientée Objet – Examen de mise en œuvre des notions de C++

Examen du 16 novembre 2020

B. Monsuez

|  |  |
| --- | --- |
| NOM : |  |
| PRENOM : |  |

**Question n°1 : Constructeurs**

Nous nous intéressons à la classe suivante qui associe à une clé une valeur :

class key\_value\_pair

{

private:

int key;

std::string value;

public:

key\_value\_pair();

key\_value\_pair(int theKey, std::string theValue);

key\_value\_pair(const key\_value\_pair& anotherPair);

};

**Question n°1.1 :**

Pour chacun des constructeurs, expliquer la fonction du constructeur et dites si le constructeur est un constructeur que C++ génère automatiquement en son absence ou pas.

key\_value\_pair();

key\_value\_pair(int theKey, std::string theValue);

key\_value\_pair(const key\_value\_pair& anotherPair);

**Question n°1.2 :**

Ecrivez le code d’initialisation qu’effectue le constructeur.

key\_value\_pair();

key\_value\_pair(int theKey, std::string theValue);

key\_value\_pair(const key\_value\_pair& anotherPair);

**Question n° 2 : Accéder aux données internes stockées dans les champs**

**Question n° 2.1 : Champ privé**

Est-il possible d’accéder aux champs key et value ? Pourquoi donc ?

**Question n° 2.2 : Proposez une méthode pour accéder aux données stockées dans la classe** key\_value\_pair.

En lecture pour le champ key.

En lecture et écriture pour le champ value.

**Question n° 3 : Opérateurs de comparaison**

Nous considérons que les objets de type key\_value\_pair sont ordonnée en fonction de la clé (champ key).

class key\_value\_pair

{

private:

int key;

std::string value;

public:

key\_value\_pair();

key\_value\_pair(int theKey, std::string theValue);

key\_value\_pair(const key\_value\_pair& anotherPair);

...

bool **operator** == (**const** key\_value\_pair&) **const**;

bool **operator** != (**const** key\_value\_pair&) **const**;

};

**Question 3.1**

Proposer le code pour les deux opérateurs suivants :

bool **operator** == (**const** key\_value\_pair&) **const**;

bool **operator** != (**const** key\_value\_pair&) **const**;

**Question n°3.2 :**

Ajouter à la classe key\_value\_pair les opérateurs <, <=, > et >=

**Question n° 4 : Patrons**

La classe key\_value\_pair est définie pour des clés de type entier (int key) et pour des valeurs de type valeur (std::string value).

class key\_value\_pair

{

private:

int key;

std::string value;

public:

key\_value\_pair();

key\_value\_pair(int first, int second);

key\_value\_pair(const key\_value\_pair& anotherPair);

...

bool **operator** == (const key\_value\_pair&) **const**;

bool **operator** != (const key\_value\_pair&) **const**;

};

Nous souhaitons définir une classe qui pourrait être paramétrisée par le type des clés (par exemple le type keyT pour le type de la clé, et pour le type valueT de la valeur).

**Question n° 4.1 :**

Proposer une transformation de la classe key\_value\_pair en une classe paramétrisée par keyT et valueT.

**Question n° 4.2 :**

Est-ce que l’expression

key\_value\_pair<std::complex, std::string> keyValuePair;

compile (La classe std::complex n’implante pas les opérateurs <, <=, > et >=) ? Expliquer pourquoi ?

**Question n°5 : Utilisation de la classe** key\_value\_pair.

Nous souhaitons utiliser la classe key\_value\_pair pour stocker des paires associant à un identifiant (la clé ayant pour type std::string) à une fréquence (la valeur a pour type float) dans un vecteur.

std::vector<key\_value\_pair<std::string, float>> listOfIdentifiers;

listOfIdentifiers.push\_back(key\_value\_pair<std::string, float>(“mot”, 10);

listOfIdentifiers.push\_back(key\_value\_pair<std::string, float>(“le”, 100);

listOfIdentifiers.push\_back(key\_value\_pair<std::string, float>(“la”, 80);

listOfIdentifiers.push\_back(key\_value\_pair<std::string, float>(“du”, 40);

...

**Question n° 5.1 :**

Expliquer ce que fait le code et précédent et quel est le contenu du vecteur listOfIdentifiers à la fin de la séquence.

**Question n° 5.2 :**

Nous nous proposons de trier les données dans le vecteur en fonction de la clé. Comment faisons-nous ? (remarque : nous avions utilisé dans un TD des fonctions de tri).

**Question n°6 : Opérateurs surchargés & Flux**

Nous souhaitons écrire sur un flux de type. Pour ce faire, nous envisageons de définir un opérateur << qui a la signature suivante :

template<class charT, class traits>

std::basic\_ostream<charT, traits>& operator << (std::basic\_ostream<charT, traits>& aStream, key\_value\_pair<keyT, valueT> thePair);

**Question n°6.1 :**

Pourquoi cet opérateur ne peut pas être défini comme opérateur dans la classe key\_value\_pair ?

**Question n°6.2 :**

Proposer une implantation de l’opérateur qui pour l’objet key\_value\_pair<std::string, float>(“du”, 40) retourne l’affichage suivant :

du => 40

**Question n°7 :**

Nous souhaitons dériver de la classe key\_value\_pair une classe key\_defined\_value qui ajoute un champ complémentaire

bool is\_void;

Ce champ indique si la valeur est définie ou n’est pas définie.

**Question 7.1**:

Proposer une classe key\_defined\_value<keyT, valueT> en tant qu’extension de la classe key\_value\_pair<keyT, valueT>.