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();
```

Constructeur par default, Oui c++ le génère automatiquement, il initialise les différents champs à des valeurs defaults généralement 0.

```
key_value_pair(int theKey, std::string theValue);
```

Constructeur qui permet d'initialiser les champs d'un objet de la classe. C++ ne le genere pas automatiquement.

```
key_value_pair(const key_value_pair& anotherPair);
```

Constructeur de copie, il permet de copier un objet dans un autre objet, il n'est pas genere automatiquement.

Question n°1.2 :

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

```
key_value_pair();
```

key_value_pair() : key(0), value(NULL){}

```
key_value_pair(int theKey, std::string theValue);
```

key_value_pair(int theKey, std::string theValue) : key(theKey), value(theValue) {}

```
key_value_pair(const key_value_pair& anotherPair);
```

key_value_pair(const key_value_pair& anotherPair) : key(anotherPair.key), value(anotherPair.value) {}

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 ?

A l'extérieure de la class non, il sont privé et donc masqué de l'utilisateur.

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`.

```
int getKey() const
{
   return key;
}

std::string getValue() const
{
   return value;
}

void setValue(const std::string& Valeur)
{
   value=Valeur;
}
```

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;
```

```
bool operator == (const key_value_pair& Objet) const
{
  if(key==Objet.key && value==Objet.value)
  {
    return true;
  }
  else
  {
    return false;
  }
}
bool operator != (const key_value_pair& Objet) const
{
  if(key!=Objet.key || value!=Objet.value)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Question n°3.2 :

Ajouter à la classe key_value_pair les opérateurs <, <=, > et >=

```
bool operator < (const key_value_pair& Objet) const
{
   if(key < Objet.key)
   {
      return true;
   }
   else
   {
      return false;
   }
}

bool operator <= (const key_value_pair& Objet) const
{
   if(key <= Objet.key)
   {
      return true;
   }
   else
   {
      return false;
   }
}

bool operator > (const key_value_pair& Objet) const
{
   if(key > Objet.key)
   {
      return true;
   }
   else
   {
      return false;
   }
}

bool operator >= (const key_value_pair& Objet) const
{
   if(key >= Objet.key)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

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.

```
template<class keyT , class valueT>
class key_value_pair
{
private:
   keyT key;
   valueT value;
public:
   key_value_pair() : key(0), value(NULL){}
   key_value_pair(keyT theKey, valueT theValue) : key(theKey), value(theValue) {}
   key_value_pair(const key_value_pair& anotherPair) : key(anotherPair.key), value(anotherPair.value) {}

   keyT getKey() const
   {
      return key;
   }

   valueT getValue() const
   {
      return value;
   }

   void setValue(const std::string& Valeur)
   {
      value=Valeur;
   }

   bool operator == (const key_value_pair& Objet) const
   {
```

```
    if(key==Objet.key && value==Objet.value)
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool operator != (const key_value_pair& Objet) const
{
    if(key!=Objet.key || value!=Objet.value)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool operator < (const key_value_pair& Objet) const
{
    if(key < Objet.key)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool operator <= (const key_value_pair& Objet) const
{
    if(key <= Objet.key)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool operator > (const key_value_pair& Objet) const
{
    if(key > Objet.key)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
    bool operator >= (const key_value_pair& Objet) const
    {
        if(key >= Objet.key)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
};
```

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 ?

Oui elle compile, mais il fait quand même écrire cela :

key_value_pair<std::complex<int ou double ou n'importe quelle type>, std::string> keyValuePair;

L'expression avec les <= est differente.

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.

Il crée un tableau nommée listOfIdentifiers et ajoute 4 objets appartenant à la class key_value_pair, dont la clé est une chaine de caractère string et la valeur une float.

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).

```
template<class keyT , class valueT>
void simple_sort(std::vector<key_value_pair<keyT, valueT>>& Tableau)
{
   for (int i = 0; i<Tableau.size(); i++)
      {
         for (int j = i + 1; j< Tableau.size(); j++)
         {
            if (Tableau[i]>=Tableau[j])
            {
               key_value_pair<keyT, valueT> Temp = Tableau[i];
               Tableau[i] = Tableau[j];
               Tableau[j] = Temp;
            }
         }
      }
}
```

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 ?

Car il prend en argument un key_value_pair<keyT, valueT> .

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
```

```
template<class keyT , class valueT>
class key_value_pair
{
….
template<class charT, class traits>
   friend std::ostream& operator << (std::ostream& aStream, const key_value_pair<keyT, valueT>&  thePair);
};
template<class keyT , class valueT>
inline std::ostream& operator << (std::ostream& aStream, const key_value_pair<keyT, valueT>& thePair)
{
   aStream<< thePair.getKey() <<" => "<< thePair.getValue() << std::endl;
   return aStream;
}
```

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>.

```
template<class keyT , class valueT>
class key_defined_value : public key_value_pair<keyT, valueT>
{
private:
   bool is_void;
public:
   key_defined_value() : key_value_pair<keyT, valueT>(), is_void(true){}
   key_defined_value(keyT theKey, valueT theValue, bool theis_void) : key_value_pair<keyT,
valueT>(theKey,theValue),is_void(theis_void)  {}
   key_defined_value(const key_defined_value& anotherPair) :  key_value_pair<keyT, valueT>(anotherPair.key,
anotherPair.value), is_void(anotherPair.is_void){}
};
```