Nom : Prénom :

Université de Nice-Sophia Antipolis POLYTECH
ELEC4 2019–2020

Examen de Programmation C++

Durée : 1h
Aucun document autorisé
Mobiles interdits

Exercice 1

RLE (*Run Length Encoding*) est une technique de compression de données, anciennement utilisée pour la compression d'images. Le principe est simplissime : une séquence de caractères c identiques est *remplacée* par :

c **marqueur** L

où **marqueur** est un caractère spécial, si possible, peu fréquent dans la suite de caractères à comprimer et L la longueur de la séquence de caractères c codée sur 1 seul caractère. Puisque L est codée sur un caractère, cela veut dire que ce codage ne comprime qu'une suite d'au plus 9 caractères identiques. Si la suite fait plus de 9 caractères, les 9 premiers seront comprimés, puis la suite de la séquence sera considérée comme une nouvelle séquence à coder. Lorsque le marqueur apparaît dans les données à comprimer, il est remplacé par « **marqueur** 0 ».

Par exemple, si on choisit comme marqueur le caractère #, la suite `bbbbb##aaaaaaaaaaabb#xxx` sera codée `b#5#0#0a#9a#2b#2#0x#3`.

On définit la classe (*incomplète*) `rle` :

```
class rle {
private :
  const int lg_max = 9;
  char marqueur;
public:
  rle(const char c) :  marqueur(c) {}
  std::string comprimer(const std::string &s) const ;
  std::string decomprimer(const std::string &s) const;
};
```

▶ 1. Écrivez la méthode `decomprimer` qui décode la chaîne de caractères codée passée en paramètre. Cette fonction devra émettre des exceptions que vous définirez par héritage de la classe `std::exception` pour les cas d'erreurs possibles. En cas d'erreur, l'exception émise devra contenir :

- un message décrivant l'erreur ;
- le fragment de la chaîne déjà décomprimé ;
- le reste de la chaîne à partir de la position erronée.

Vous pourrez utiliser la méthode `substr` de la classe `string` qui renvoie une sous-chaîne de la chaîne courante :

- `substr(pos,len)` renvoie la sous-chaîne entre `[pos ; pos+len]` ;
- `substr(pos)` renvoie la sous-chaîne de l'indice `pos` jusqu'à la fin de la chaîne courante.

```
#pragma once

#include <string>
#include <iostream>
#include <exception>

class RLE_Exception : public std::exception {
protected:
  std::string msg;
public:
  RLE_Exception(std::string m, std::string av, std::string ap)
  { this->msg = m + ", " + av + "[" + ap +"]";}

  RLE_Exception(const RLE_Exception &e) : msg(e.msg) {}

  // redéfinition de what
  virtual const char* what() const noexcept override {
    return msg.c_str();
  }
};

class rle {
private :
  const int lg_max = 9;
  char marqueur;

  char toInt(char c) const {
    return c-'0';
  }

  void decoder(std::string &s, const char c, const int lg) const {
    for (int i=0; i<lg ; i++)
      s.push_back(c);
  }

  void decoderMarqueur(std::string &s) const {
    s.push_back(marqueur);
  }
```

```
public:
  rle(const char c) :  marqueur(c) {}

  std::string +decomprimer(const std::string &sc) const {
    std::string s;
    int sc_size = sc.size(), i=0;
    //
    while (i<sc_size) {
      if (sc[i]==marqueur) {
        if (toInt(sc[i+1])!=0)
          throw RLE_Exception("0 attendu",
                              sc.substr(0,i+1), sc.substr(i+1));
        decoderMarqueur(s);
        i+=2;
      }
      else {
        // s[i] caractère à décoder => vérifier la présence du marqueur
        if (sc[i+1]!=marqueur)
          throw RLE_Exception("marqueur attendu",
                              sc.substr(0,i+1), sc.substr(i+1));
        // vérifier la validité de la longueur
        int lg = toInt(sc[i+2]);
        if (lg<1||lg>lg_max)
          throw RLE_Exception("1<=longueur<=9",
                              sc.substr(0,i+1), sc.substr(i+1));
        // ok
        decoder(s, sc[i], toInt(sc[i+2]));
        i+=3;
      }
    }
    return  s;
  }
};
```

.....

▶ 2. Écrivez un fragment de code C++ qui crée un objet de type `rle`, qui décomprime une chaîne de caractères codée, et qui affiche sur la sortie d'erreur standard les éventuelles exceptions.

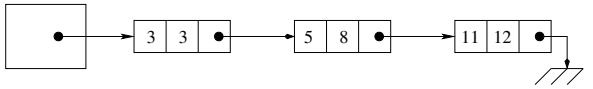
```
rle c('#');
std::string s = "b#5xx#0#0a#9a#2b#2#0x#3";

try {
   std::cout << c.decomprimer(s) << std::endl;
}
catch (const RLE_Exception &e) {
    std::cerr << e.what() << std::endl;
}
```

.....

Exercice 2

On souhaite représenter des grands ensembles (au sens mathématique) de taille quelconque dont les d'éléments sont de type quelconque. Ces grands ensembles sont implémentés par une suite d'intervalles alloués dynamiquement et simplement chaînés. Chaque intervalle possède une borne inférieure, une borne supérieure et un lien sur l'intervalle suivant. Les intervalles sont en ordre croissant, et la borne supérieure d'un intervalle doit toujours être inférieure à la borne inférieure de l'intervalle suivant :

Par exemple, l'ensemble d'entiers $\{3, 5, 6, 7, 8, 11, 12\}$ est représenté par :



▶ 3. Écrivez la classe générique `intervalle` avec :
- 3 variables d'instance `binf`, `bsup` et `suivant` ;
- un constructeur pour initialiser les deux bornes de l'intervalle ;
- 3 accesseurs pour accéder aux 3 variables d'instance.

```
template<typename T>
class intervalle {
protected :
  T binf, bsup; // inf <= sup
  intervalle<T>* suivant=nullptr;
public:
  intervalle(const T &x, const T &y) :
    binf(x), bsup(y), suivant(n) { assert(x<=y); }
  const T &getBInf() const { return this->binf ;}
  const T &getBSup() const { return this->bsup ;}
  intervalle<T> *getSuivant() const { return this->suivant;}
};
```

.....

▶ 4. Par héritage de la classe `intervalle`, écrivez la classe générique `singleton` qui représente un intervalle dont la borne inférieure et supérieure sont identiques.

```
template<typename T>
class singleton : public intervalle<T> {
public :
  singleton(const T &x) : intervalle<T>(x,x) {}
};
```

▶ 5. Écrivez la classe générique `hlm` qui représente les grands ensembles. Cette classe contiendra :
- un constructeur pour créer un ensemble vide ;
- la méthode booléenne `vide` qui teste si l'ensemble courant est vide ou non ;
- la méthode booléenne `appartient` qui teste si la valeur de son paramètre `x` appartient à l'ensemble `hlm` courant.

```
template<typename T>
class hlm {
private :
  intervalle<T> *tete;
public:
  hlm() : tete(nullptr) {}
  bool vide() {
    return tete == nullptr;
  }
  /*
   * Rôle : appartient(x) = x ∈ e
   */
  bool appartient(const T &x) {
    intervalle<T> * e = this->tete;
    while (e != nullptr && x >= e->getBInf()) {
      if (x>=e->getBInf()>=0 && x<=e->getBSup())
        // x ∈ *this
        return true;
      // else
      e = e->getSuivant();
    }
    // x ∉ *this
    return false;
  }
};
```