Université de Nice-Sophia Antipolis — POLYTECH Sophia
ELSE4-FISE — 2023–2024

Examen de C++/COO

Durée : 1h00
Aucun document autorisé
Mobiles interdits

▶ 1. Expliquez de façon claire et synthétique le patron de conception *itérateur*. Vous tracerez son diagramme de classes.

Cf. cours.

.....

▶ 2. Écrivez la fonction `créerIntervalle` qui prend en paramètre deux entiers `bInf` et `bSup` et qui renvoie un `std::vector` d'entiers formé de la suite d'entiers de `bInf` à `bSup` (*i.e.* `[bInf ; bSup]`).

```
/*
 * Antécédédent : bInfo ⩽ bSup
 * Rôle : renvoie un vecteur formé des entiers [bInf ; bSup]
 *
 */
std::vector<int> créerIntervalle(const int bInf, const int bSup) {
  assert(bInf<=bSup);
  std::vector<int> v;
  for (int i=bInf; i<=bSup; i++)
    v.push_back(i);
  //
  return v;
}
```

En utilisant `std::iota`, on veut également écrire cette fonction comme suit :

```
std::vector<int> créerIntervalle(const int bInf, const int bSup) {
  assert(bInf<=bSup);
  std::vector<int> v(bSup-bInf+1);
  std::iota(v.begin(), v.end(), bInf);
  return v;
}
```

.....

▶ 3. À l'aide d'un itérateur, écrivez la procédure `écrireIntervalle` qui écrit sur la sortie standard la suite d'entiers `[bInf ; bSup]` contenue dans le `std::vector` passé en paramètre.

```
/*
 * Rôle : écrit sur la S/S l'intervalle d'entiers contenu dans vecteur v
 */
void écrireIntervalle(const std::vector<int> &v) {
  std::cout << "[ ";
  for (std::vector<int>::const_iterator it=v.cbegin(); it!=v.cend(); it++)
    std::cout << *it << " ";
  std::cout << "]" << std::endl;
}
```

.....

▶ 4. À l'aide d'un *reverse* itérateur, écrivez la fonction `créerIntervalleImpair` qui prend en paramètre un `std::vector` formé de la suite d'entiers `[bInf ; bSup]` qui renvoie un `std::vector` formé des entiers impairs pris sur l'intervalle `[bSup ; bInf]`.

```
/*
 * Antécédent : inter contient la suite d'entiers [bInf ; bSup]
 * Rôle : renvoie le vecteur formé des entiers impairs pris sur
 * l'intervalle [bSup ; bInf].
 */
std::vector<int> créerIntervalleImpair(const std::vector<int> &inter) {
  std::vector<int> v;
  std::vector<int>::const_reverse_iterator it = inter.crbegin();
  // vérifier si la borne sup est paire
  if ((inter.back()&1)==0) it++;
  //
  for ( ; it<inter.crend() ; it+=2)
    v.push_back(*it);

  return v;
}
```

.....

▶ 5. Écrivez la fonction `main` qui lit sur l'entrée standard deux entiers a et b; ($b \geqslant a$) crée et écrit l'intervalle $[a;b]$; puis, à partir de cet intervalle, crée et écrit l'intervalle d'entiers impairs. Par exemple, avec les entiers $a=-7$ et $b=10$, le programme affiche :

```
[ -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 ]
[ 9 7 5 3 1 -1 -3 -5 -7 ]
```

```
int main () {
  int a, b;
  // lire les bornes inf et sup de l'intervalle
  std::cin >> a >> b;
  assert(b>=a);
  // les bornes sont valides ⇒ créer et écrire les intervalles
  std::vector<int> interAB;
  écrireIntervalle(interAB = créerIntervalle(a, b));
  écrireIntervalle(créerIntervalleImpair(interAB));
  //
  return EXIT_SUCCESS;
}
```

.....

La notion de liste linéaire est représentée par la classe abstraite `Liste` (voir annexe), et mise en œuvre par la classe `ListeChaînée` à l'aide d'une structure dynamique simplement chaînée (voir annexe). On souhaite programmer un itérateur pour la classe `ListeChaînée`. On donne ci-dessous le squelette de la classe `iterator` interne à `ListeChaînée` :

```
template <typename T>
class ListeChaînée : public Liste<T> {
public:
...
class iterator : public std::iterator<
  std::input_iterator_tag,    // iterator_category
  std::remove_cv_t<T>,        // value_type
  std::ptrdiff_t,             // difference_type
  T*,                         // pointer
  T&                          // reference
 >
 {
   using pointer = noeud<T>*;
   using reference = const T&;

  private:
    pointer position;
  public:
    iterator(const pointer &p) : position(p) {...}
    // incrémentation préfixe, i.e. ++p
    iterator &operator++() { ... }
    // incrémentation postfixe, i.e. p++
    iterator operator++(int) { ...}
    // accès
    reference operator*() const { .... }
    // opérateurs de comparaison
    bool operator==(const iterator& it) const {... }
    bool operator!=(const iterator& it) const { ... }
 }; // fin classe iterator
 ...
 iterator begin() noexcept { ... }
 iterator end() noexcept  { ... }
} // fin classe ListeChaînée
```

► 6. Écrivez le constructeur, les opérateurs `++` (préfixe et postixe), `*`, `==` et `!=` de la classe `iterator`, ainsi que les méthodes `begin` et `end` de la classe `ListeChaînée`.

Dans la classe interne `iterator` :

```
 // le constructeur
iterator(const pointer &p) : position(p) {}

// incrémentation préfixe, i.e. ++p
iterator &operator++() {
  assert(this->position!=nullptr);
  this->position = this->position->getSuivant();
   return *this;
}

// incrémentation postfixe, i.e. p++
iterator operator++(int) {
  assert(this->position!=nullptr);
  iterator i=*this;
  this->position = this->position->getSuivant();
  return i;
}

// accès à l'éléments
reference operator*() const {
  assert(this->position!=nullptr);
  return this->position->getÉlément();
}

// opérateurs de comparaison
bool operator==(const iterator& it) const { return this->position == it.position; }
bool operator!=(const iterator& it) const { return this->position != it.position; }
```

Dans la classe `ListeChaînée` :

```
iterator begin() noexcept { return iterator(this->tête->getSuivant()); }
iterator end() noexcept  { return iterator(nullptr); }
```

.....

► 7. Dans votre fonction `main` déclarez la liste chaînée d'entiers `lce` de type `ListeChaînée`. Insérez successivement les entiers -3, -9, 15 et 0, pour obtenir la liste $< 15, -3, -9, 0 >$. Déclarez l'itérateur `it` pour la liste `lce`. Par son intermédiaire, accédez au deuxième élément et écrivez sa valeur sur la sortie standard. Enfin, toujours avec l'itérateur, écrivez à l'aide d'un énoncé itératif, le reste de la liste (donc à partir du 3ème élément) sur la sortie standard.

```
ListeChaînée<int> l;
l.insérer(1, -3); l.insérer(2, -9);  l.insérer(1, 15);  l.insérer(4, 0);
//
ListeChaînée<int>::iterator it=l.begin();
std::cout << "le 2ème élément : " << *++it << "" << std::endl;
//
while (++it !=l.end())
    std::cout << *it << " ";
std::cout << std::endl;
```

.....

Annexe

Liste.hpp

```
template <typename T>
class Liste {
public:
  virtual int longueur() const =0;
  virtual T ième(const int r) const =0;
  virtual void insérer(const int r, const T &e) =0;
  virtual void supprimer(const int r) =0;
};
```

ListeChaînée.hpp

```
template <typename T>
class ListeChaînée : public Liste<T> {
private:
  noeud<T> *tête;
  int lg;
public:
  ListeChaînée() : lg(0) {
    this->tête = new noeud<T>(T());
  }
  virtual int longueur() const override {
    return this->lg;
  }
  T ième(const int r) const {
    assert(r>=1 && r<=this->lg);
    noeud<T> *p = this->tête->getSuivant();
    for (int i = 1; i<r; i++)
      p = p->getSuivant();
    return p->getÉlément();
  }
  void insérer(const int r, const T &e) override {
    assert(r>=1 && r<=this->lg+1);
    noeud<T> *n = new noeud<T>(e);
    noeud<T> *q = this->tête;
    for (int i=1; i<r; i++)
      q = q->getSuivant();
    n->setSuivant(q->getSuivant());
    q->setSuivant(n);
    this->lg++;
  }
  void supprimer(const int r) override  {
    assert(r>=1 && r<=this->lg);
    noeud<T> *p ;
    noeud<T> *q = this->tête;
    for (int i=1;  i<r; i++)
      q = q->getSuivant();
    p = q->getSuivant();
    q->setSuivant(p->getSuivant());
    delete p;
    this->lg--;
  }
};
```


noeud.hpp

```
template <typename T>
class noeud {
private:
  T elt;
  noeud<T> *suivant;
public:
  noeud(const T e, noeud<T> *s=nullptr) : elt(e), suivant(s) {}
  const T &getÉlément() const { return this->elt; }
  noeud<T>* getSuivant() const { return this->suivant; }
  void setÉlément(const T &x) { this->elt = x; }
  void setSuivant(noeud<T> *s) { this->suivant = s; }
};
```