Université de Nice-Sophia Antipolis — POLYTECH
ELEC4 — 2024–2025

Examen de COO

Durée : 1h00
Aucun document autorisé
Mobiles interdits

► 1. Expliquez de façon claire et synthétique la notion *générale* d'itérateur et celle, particulière, de C++.

Cf. cours

.....

► 2. Récrivez la fonction `find_if` qui recherche dans un conteneur un élément qui vérifie un prédicat booléen. La fonction recherche entre deux itérateurs, et renvoie l'itérateur qui désigne le premier élément qui vérifie le prédicat. Si aucun élément ne vérifie le prédicat, la fonction renvoie l'itérateur de fin. L'en-tête de cette fonction est le suivant :

```
auto find_if(auto itDébut, auto itFin, auto prédicat)
```

```
auto find_if(auto itDébut, auto itFin, auto prédicat) {
  for (auto it=itDébut; it!=itFin; it++)
    if (prédicat(*it))
      return it;
  // aucune valeur ne vérifie le prédicat
  return itFin;
}
```

.....

► 3. Avec votre fonction `find_if`, écrivez le fragment de code qui recheche dans tout un vecteur `v` de `std::string` la première chaîne de caractères de taille paire. Si elle est trouvée, vous l'écrirez sur la sortie standard. Vous écrirez le prédicat booléen sous forme d'une *fonction anonyme*.

```
// v est un std:vector<std::string>
auto it = find_if(v.begin(), v.end(),
                  [] (const std::string &s) { return (s.size()&1)==0; });
// écrire sur la S/S la 1ère occurrence dans v d'une chaîne de caractères de longueur paire
if (it!=v.end())
       std::cout << *it << std::endl;
```

.....

On désire écrire une application qui compte le nombre d'occurrences des mots contenus dans un fichier de texte. Le programme écrit sur la sortie standard la liste des mots lus en ordre alphabétique avec, pour chaque mot, son nombre d'occurrences.

Pour mémoriser les mots et leur nombre d'occurrences, on définit la classe `Dico` qui héritera de la classe générique `std::map`. Dans cette classe, vous écrirez la méthode `insérer` pour ajouter un mot dans le dictionnaire, la méthode `toString` et l'opérateur `<<` pour écrire tous les mots du dictionnaire dans l'ordre l'alphabétique avec leurs nombres d'occurrences.

► 4. Écrivez la classe `Dico`.

```
class Dico : private std::map<std::string, int> {
public:
  // Rôle : insère dans le Dico courant le mot m
  void insérer(const std::string &m) {
    (*this)[m]++;
  }
  // Rôle : renvoie le Dico courant sous forme d'une std::string avec la
  // liste des mots en ordre alphabétique avec leur nb d'apparitions
  std::string toString() const {
    std::ostringstream s;
    for (const auto& [mot, nbOcc] : *this)
     s << mot << " : " << nbOcc << "\n";
    return s.str();
  }
  // Rôle : écrit le Dico d sur l'ostream f
  friend std::ostream &operator<<(std::ostream &f, const Dico& d) {
    return f << d.toString();
  }
};
```

.....

► 5. On veut écrire une application qui permet à son utilisateur de trouver le chemin qui relie deux lieux (représentés par des `std::string`) dans une ville, qu'il soit à pied, en bus ou en voiture. L'application suivra le patron de conception `stratégie`. Elle est formée de 5 classes : `stratégie`, `enVoiture`, `ÀPied`, `EnBus`, et `Navigateur`. Le programme principal est le suivant :

```
void gps(const stratégie &s, const std::string &from, const std::string &to) {
  Navigateur(s).trouverLeChemin(from, to);
}

int main() {
  ÀPied àPied;
  EnVoiture voiture;
  EnBus bus;

  gps(àPied, "Maison", "Piscine");
  gps(bus, "Maison", "Polytech'Sophia");
  gps(voiture, "Musée", "12 Rue St Paul");

  return EXIT_SUCCESS;
};
```

Il produit sur la sortie standard le résultat suivant :

```
à pied de Maison à Piscine, il faut passer par ...
en bus de Maison à Polytech'Sophia, il faut passer par ...
en voiture de Musée à 12 Rue St Paul, il faut passer par ...
```

On donne la classe `stratégie` :

```
class  stratégie {
public:
  virtual void trouverLeChemin(const std::string &a, const std::string &b)
  const =0;
};
```

► 6. Dessinez le diagramme de classes en UML, et complétez les classes données ci-dessous :


```
/**
 *
 */
class  EnVoiture ...
```

```
};
```

```
/**
 *
 */
class  ÀPied ...
```

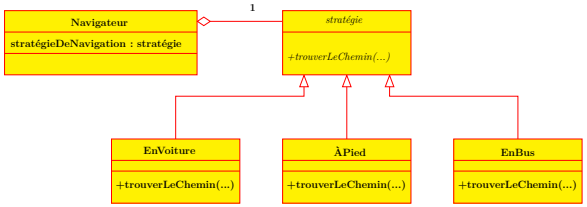
```
};
```

```
/**
 *
 */
class  EnBus ...
```

```
};
```

```
/**
 *
 */
class Navigateur ...
```

```
};
```




```
class  stratégie {
public:
  virtual void trouverLeChemin(const std::string &a, const std::string &b) const =0;
};

/*
 * Cette classe définit la stratégie pur trouver le chemin en voiture entre les points a et b
 */
class EnVoiture : public stratégie {
  virtual void trouverLeChemin(const std::string &a, const std::string &b) const overr
    // algorithme pour trouver le chemin en voiture entre les points a et b
    std::cout << "en voiture de " << a << " à " << b << ", il faut passer par ..." << std
  }
};

/*
 * Cette classe définit la stratégie pur trouver le chemin à pied entre les points a et b
 */
class ÀPied : public stratégie {
  virtual void trouverLeChemin(const std::string &a, const std::string &b) const overr
    // algorithme pour trouver le chemin à pied entre les points a et b
    std::cout << "à pied de " << a << " à " << b << ", il faut passer par ..." << std::enc
  }
};

/*
 * Cette classe définit la stratégie pur trouver le chemin en bus entre les points a et b
 */
class EnBus : public stratégie {
  virtual void trouverLeChemin(const std::string &a, const std::string &b) const overr
    // algorithme pour trouver le chemin en bus entre les points a et b
    std::cout << "en bus de " << a << " à " << b << ", il faut passer par ..." << std::enc
  }
};

/**
 * La classe Navigateur permet de trouver le chemiin entre les points a et b
 * selon la stratégie donnée au constructeur
 */
class Navigateur {
private:
```



```
  const stratégie & stratégieDeNavigation;
public:
  Navigateur(const stratégie &s) : stratégieDeNavigation(s) {}
  void trouverLeChemin(const std::string &a, const std::string &b) {
    this->stratégieDeNavigation.trouverLeChemin(a, b);
  }
};
```

.....