Programmation Générique
M1 Informatique — Examen 27 avril 2007
2 heures

Les documents, hors livres, sont autorisés. Les **4** exercices sont indépendants.

Exercice 1 :

Qu'affiche le programme ci-dessous ? Justifier.

```
#include<iostream>

struct A{
  int child;

  A(): child(0) {}

  virtual void foo(){
    std::cout << "foo "
              << child << std::endl;
  }
};

struct B{
  int child;

  B(): child(0) {}

  void bar(){
    std::cout << "bar "
              << child << std::endl;
  }
};

template<int N>
struct C: public A, virtual public B{
  C(){ A::child = B::child = N; }

  void foo(){ A::foo(); }
  void bar(){ B::bar(); }
};

struct D: public C<1>, public C<2>{};

int main(){
  D d;
  d.C<1>::foo();     d.C<1>::bar();
  return 0;
}
```

L'objet d étant déclaré comme précédemment, lesquelles des initialisations suivantes sont-elles correctes ? Justifier.

- `A& a=dynamic_cast<A&>(d);`
- `C<1>& c1=dynamic_cast<C<1>&>(d);  A& a=dynamic_cast<A&>(c1);`
- `B& b=dynamic_cast<B&>(d);`
- `C<1>& c1=dynamic_cast<C<1>&>(d);  B& b=dynamic_cast<B&>(c1);`

Exercice 2 :
La fonction d'Ackerman est définie de la façon suivante :

$$\mathsf{Ack}(n,m) = \begin{cases} n+1 & \text{si } m = 0, \\ \mathsf{Ack}(1, m-1) & \text{si } n = 0 \text{ et } m > 0, \\ \mathsf{Ack}(\mathsf{Ack}(n-1,m), m) & \text{sinon .} \end{cases}$$

Écrire les classes nécessaires pour que la compilation du programme contenant la méthode `main` suivante :

```
int main(){
  aff<ack<5,3>::val > a;
  a.res();
  return 0;
}
```

affiche un message d'erreur du type

```
In function ‘int main()’:
 error: ’struct aff<253>’ has no member named ’res’
```

Où 253 est justement la valeur cherchée : $\mathsf{Ack}(5,3)$.

Exercice 3 :

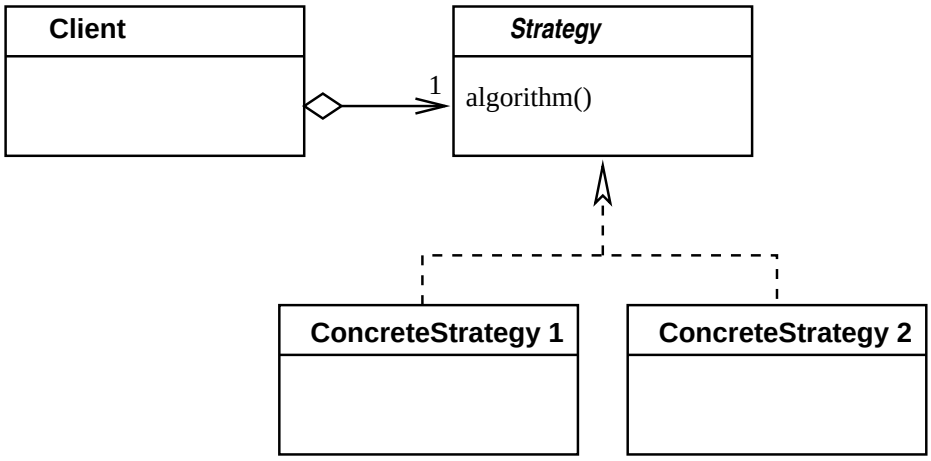
On souhaite utiliser des collections de manière analogue à Java. Pour cela, on crée un type `Collection` paramétré par le type des éléments stockés et par le conteneur (au sens de la bibliothèque standard) qui implémente la collection. La valeur par défaut de ce second paramètre est `std::list<T>`. On veut pouvoir par exemple écrire le morceau de code suivant :

```
int main(){
  Collection<int> y;
  for(int x=0;x<10;x++){
    y.add(x*5);
  }
  Collection<int>::Iterator it=y.iterator(); \\(*)
  while(it.hasNext()){
    std::cout << it.next() << std::endl;
  }
}
```

Pour simplifier, on supposera que la classe Collection ne contient que les méthodes `add` et `iterator`, et que les Iterator ne contiennent que les méthodes `hasNext` et `next`. Écrire les définitions de la classe `Collection`, de la classe correspondant au type `Collection<T>::Iterator`, ansi que de leurs méthodes. (On supposera que le conteneur sous-jacent dispose d'une méthode `push_back`.) Dans le cas d'un appel de `next` alors que l'itérateur est à la fin de la collection, lancer une exception (à définir) `NoSuchElementException`.

Discuter (10 lignes maximum) la possibilité d'écrire `Iterator<int> it=y.iterator();` à la ligne (*).

Exercice 4 :



Le design pattern *Strategy* est illustré par la figure ci-dessus. Un client peut utiliser dans l'une ou l'autre de ses méthodes différentes versions d'un algorithme selon la *Strategy* avec lequel il est composé.

On désire appliquer ce design pattern à la validation de documents. On suppose l'existence d'une structure `ficheLogin` dont les champs sont `Nom`, `Prenom`, `Telephone` et `Login`. On suppose que la classe *Client* est une fabrique `registerLogin` qui dispose d'une méthode `newLogin` qui demande d'écrire sur l'entrée standard les renseignements nécessaires et retourne un objet `ficheLogin`. Cette méthode vérifie en particulier que le format du numéro de téléphone est correct en appelant une méthode `isValid` dont l'implémentation dépend d'un objet de type `Validator` (jouant ici le rôle de *Strategy*). On suppose en outre que cet objet fournit une méthode `correctForm` qui indique quel est le format attendu.

1) Écrire en langage objet classique en C++ la classe `registerLogin` (contenant la méthode `newLogin`), l'interface `Validator` ainsi qu'une classe `StandardValidator` qui permet de valider le numéro de téléphone si celui-ci est une chaîne de caractères de 10 chiffres sans espace. `registerLogin` devra aussi fournir une méthode permettant de fixer un `Validator`.

On prêtera attention au fait que toutes les instances d'une même implémentation de `Validator` se comportent de la même manière ; on mettra donc en place dans `StandardValidator` les mécanismes qui permettent d'assurer qu'on ne peut créer qu'une instance de `StandardValidator`.

2) On suppose maintenant que le `Validator` utilisé est connu dès la déclaration de l'objet `registerLogin` et ne change pas.

Modifier la classe `registerLogin` de sorte que le type de `Validator` en soit un paramètre. Mettre en place les mécanismes permettant de vérifier à la compilation que le type paramètre est bien un `Validator`.

Comme le `Validator` est un paramètre de `registerLogin`, on n'a uniquement besoin des classes `Validator`, et non des objets. Indiquer les modifications que ceci implique.