Travaux Dirigés Programmation C++ : Feuille 2

Informatique 2ème année.

—*Julien Allali* - `allali@enseirb.fr` —

Le but de ce TD est l'apprentissage des mécanismes de gestion mémoire, l'utilisation des références et des opérateurs.

Repartir des fichiers sources : chaine2.tar

1 Références et const.

▸**Exercice 1.** *Passage par référence*

Modifier la fonction `annexe` *dans le fichier* `Programme.cpp` *pour que l'instance de Chaine soit prise par référence.*

Compiler et executer. Combien d'instances sont créées au finale ?

▸**Exercice 2.** *Passage par référence constante*

Dans la fonction `annexe`*, afficher la valeur de retour de la méthode* `taille` *en utilisant* `printf`*.*

Compiler, executer.

Modifier la référence pour qu'elle soit constante :

```
void annexe(const Chaine &s)
```

Compiler et lire **attentivement** *le message du compilateur.*

Pour finir, dans la classe `Chaine`*, indiquer que la méthode* `taille` *ne modifiera pas les attributs de* `this`*. Pour cela, ajouter le mot clé* `const` *après les arguments :*

```
unsigned int taille() const;
```

Ce qualificatif fait parti de la signature de la méthode, il faut donc l'indiquer dans le fichier `Chaine.hpp` *et* `Chaine.cpp`*.*

Compiler. Qu'en déduisez vous sur le fonctionnement de `const`*.*

Régles de programmation : D'une façon générale, il faut toujours mettre le `const` sur la méthode et les références. La question que vous devez vous poser est : "vais-je modifier l'objet ?". Si vous répondez oui, alors on ne met pas le `const`, si vous répondez non, alors on laisse le `const` (puisqu'on venait de le mettre pas reflexe).

Régles de programmation : Dans le cas des méthodes d'instance, il peut arriver que l'on souhaite avoir deux comportements différents selon que la source (l'objet sur lequel on appelle la méthode) est constante ou non. Cela ne pose pas de problème étant donné que l'attribut `const` des méthodes fait parti de la signature : on écrira donc deux versions de la méthode, une `const` et l'autre non.

Régles de programmation : À partir de maintenant tous les arguments d'une fonction seront soit des références, soit des pointeurs, soit des types primitifs (int, char, ...).

---

2 Gestion de la recopie des objets

▸**Exercice 3.** *Recopie*

Dans la classe `Chaine`*, écrire une méthode* `debug()` *qui affiche l'attribut* `_donnees`*.*

Écrire le `main` *suivant :*

```
int main(){ //1
   Chaine s1("une chaine"); //2
   Chaine s2(s1); //3
   Chaine s3=s1; //4

   s1.debug();
   s2.debug();
   s3.debug();
}
```

Observer attentivement les affichages produit par l'execution. Qu'en déduisez vous sur les instanciations faites en ligne 3 et ligne 4 ? Utiliser valgrind pour vérifier la libération. Tracer les adresses des attributs des objets en utilisant `gdb`.

▸**Exercice 4.** *Constructeur par copie*
Ajouter un constructeur par recopie dans la classe `Chaine`*, c'est à dire un constructeur prenant en argument une instance de* `Chaine`.

▸**Exercice 5.** `explicit`
Déclarer le constructeur par recopie comme `explicit` *puis tester le code suivant :*

```
Chaine print(Chaine s){
  s.debug();
  return s;
}
...
Chaine c("message");
print(c);
```

Si la fonction a besoin d'une copie locale de la chaîne passée en argument, comme doit-elle procéder ?
Quand est-il de la valeur de retour ?

3 Opérateurs

La classe Chaine représente maintenant une chaine constante.

▸**Exercice 6.** *Opérateur d'affectation*
Écrire le `main` *suivant :*

```
int main(){
   Chaine s1("une chaine"); //1
   Chaine s2("Coucou"); //2

   s2 = s1; //3

   s2.debug();
}
```

Que se passe-t-il en 3 ? Vérifier la mémoire avec valgrind. Inspecter les instances en utilisant `gdb`.
Pour redéfinir l'opérateur d'affectation, il y a deux prototypes :

```
  const Chaine& operator=(const Chaine& );
//ou
  Chaine operator=(const Chaine& );
```

— *Pourquoi est-il préférable de choisir le premier ?*
— *Dans notre cas, que fait le code de l'opérateur ? Implémenter celui-ci.*
— *Vérifier votre code final avec valgrind.*

▸**Exercice 7.** *Opérateur []*
Nous voulons écrire le code suivant :

```
int main(){
   Chaine s("une chaine"); //1

   for (int i = 0; i < s.taille; i++)
     printf(''%c\n'', s[i]);

}
```

Pour cela, il est nécessaire de redéfinir l'opérateur[]. Il y a deux prototypes :

```
  char operator[](unsigned int indice);
// ou
  char& operator[](unsigned int indice);.
```

Dans quel cas peut-on écrire l'instruction suivante `s[3] = 'Z';` *? Pourquoi ? Dans* **notre** *cas (chaine constante) quel prototype est-il préférable ? Voyez-vous un troisième prototype possible ?*

Essayer d'utiliser l'opérateur dans la fonction annexe de l'exercice 2. Modifier.

▸**Exercice 8.** *Opérateur de conversion*
Nous voulons pouvoir écrire le code suivant :

```
int main(){
   Chaine s(''une chaine''); //1

   printf(''%s\n'', (const char *) s);
}
```

Pour cela, il est nécessaire de redéfinir l'opérateur de conversion. La valeur de retour de cet opérateur pose-t-elle problème ?

▸**Exercice 9.** *Opérateur + de concaténation*
Écrire dans la classe `Chaine` *l'opérateur d'addition effectuant la concaténation de deux chaînes. Cet opérateur ne modifiera pas les deux chaînes concaténées et renverra donc une nouvelle chaîne.*

Surcharger cet opérateur tel que l'on puisse concaténer une chaîne et un entier. Pour cette version, vous pourrez utiliser une variable static interne servant de buffer. Quels sont les dangers d'une telle solution ?