

[Tableau de bord](#) / [Mes cours](#) / [INF1015 - Programmation orientée objet avancée](#) / [Examens](#) / [Contrôle - Hiver 2022 \(intra\)](#)**Commencé le** vendredi 11 mars 2022, 18:30**État** Terminé**Terminé le** vendredi 11 mars 2022, 19:30**Temps mis** 1 heure**Note** 20,00 sur 20,00 (100%)

## Description

- Pour un examen, le **plagiat** inclut le fait de demander de l'aide à quelqu'un pour répondre aux questions et le copier-coller venant de sources dont vous ne possédez pas tous les droits (i.e. TP fait en équipe, notes de cours, site Internet), sans en citer la source. Toute personne qui aide un autre étudiant pendant l'examen commet une **fraude**.
- En cas de doute sur le sens d'une question, énoncez clairement toutes suppositions que vous faites, soit en commentaires dans le code, soit dans la dernière question de cet examen (qui est un espace pour écrire vos commentaires). Nous ne répondrons pas aux questions.
- En cas de réel problème technique, vous pouvez demander au surveillant. Pour les cas qui font l'examen à distance, écrire en message direct à frboyer sur le Discord du cours.
- Vous avez droit à toutes les notes de cours et, pour les questions de «programmation» (où vous devez écrire un programme) à un compilateur, mais CodeRunner est suffisant pour répondre et les questions n'ont pas été particulièrement faites pour l'utilisation d'un autre compilateur. Vos programmes doivent passer les tests CodeRunner.
- La sauvegarde se fait automatiquement lorsque vous changez de page, elle est déclenchée quand le bouton « Suivant » dans le bas de la page est utilisé ou lorsque vous changez de page en utilisant le bloc «Navigation du test ». Cette sauvegarde permet de limiter la perte d'information en cas de coupure avec Internet ou autre problème technique et de garder la session active. Après deux heures d'inactivité (c.-à-d. aucune interaction avec le serveur), vous êtes automatiquement déconnecté(e) de Moodle.
- En cas de perte de connexion à la fin de l'examen, la tentative est envoyée automatiquement.
- L'examen est sur 20 points, le barème de chaque exercice est indiqué dans le bloc «Navigation du test ».

Bon travail !



## Question 1

Correct

Note de 1,50 sur 1,50

Soit la définition de la classe suivante et de son opérateur d'insertion dans un flux (affichage) :

```
struct Spam {  
    void print(ostream& out) const {  
        out << "Spam and eggs?";  
        nbAffichages++;  
    }  
  
    int nbAffichages = 0;  
};  
  
ostream& operator<<(ostream& out, Spam& spam) {  
    spam.print(out);  
    spam.nbAffichages++;  
    return out;  
}
```

Choisissez l'option qui décrit le mieux pourquoi le code ne compile pas.

- ☐ a. La fonction `operator<<` doit être une fonction constante pour appeler `Spam::print()` qui est une méthode constante
- ☒ b. La méthode `Spam::print()` ne peut pas incrémenter `nbAffichages` car c'est une méthode constante ✓
- ☐ c. La fonction `operator<<` ne peut pas accéder directement à `nbAffichages` car c'est une donnée membre

Votre réponse est correcte.

La réponse correcte est : La méthode `Spam::print()` ne peut pas incrémenter `nbAffichages` car c'est une méthode constante



## Question 2

Correct

Note de 1,50 sur 1,50

Soit le prototype et utilisation de la fonction `fn()` suivante :

```
void fn(string arg1, const string* arg2) {  
    // ...  
}  
  
int main() {  
    string foo = "henlo fren";  
    const string* bar = &foo;  
    fn(foo, bar);  
}
```

Cochez toutes les options qui sont vraies dans le contexte de l'utilisation dans le `main()`.

- ☐ a. On peut modifier la valeur de `foo` à travers `arg2`, car `foo` est modifiable
- ☒ b. `arg1` sera une copie de `foo` modifiable et locale à `fn()` ✓
- ☒ c. `arg2` pointera vers `foo` ✓
- ☐ d. Si on modifie `arg2` pour le faire pointer à une autre adresse, on change l'adresse pointée par `bar`.

Votre réponse est correcte.

Les réponses correctes sont : `arg1` sera une copie de `foo` modifiable et locale à `fn()`, `arg2` pointera vers `foo`



## Question 3

Correct

Note de 2,00 sur 2,00

Soit le code suivant :

```
1. class Roue {
2. public:
3.     Roue(int diametre)
4.         : diametre_(diametre) { }
5.
6. private:
7.     int diametre_;
8. };
9.
10. class Voiture {
11. public:
12.     Voiture(const Roue& modele)
13.         : roues_{modele, modele, modele, modele} { }
14.
15. private:
16.     Roue roues_[4];
17. };
18.
19. class Garage {
20. public:
21.     void ajouterVoiture(Voiture* voiture) {
22.         voitures_.push_back(voiture);
23.     }
24.
25. private:
26.     vector<Voiture*> voitures_;
27. };
```

En observant la possession des ressources, décrivez les relations entre les classes *Roue*, *Voiture* et *Garage*.

- ☒ a. *Roue* en composition dans *Voiture*, ✓  
*Voiture* en agrégation dans *Garage*
- ☐ b. *Voiture* hérite de *Roue*,  
*Voiture* en composition dans *Garage*
- ☐ c. *Roue* en composition dans *Voiture*,  
*Voiture* en composition dans *Garage*
- ☐ d. *Roue* en agrégation dans *Voiture*,  
*Voiture* en agrégation dans *Garage*

Votre réponse est correcte.

La réponse correcte est :

*Roue* en composition dans *Voiture*,  
*Voiture* en agrégation dans *Garage*

## Question 4


Correct

Note de 2,25 sur 2,25

Soit le code suivant :

```
1. template <typename T>
2. class Liste {
3. public:
4.     Liste(const vector<T>& elems)
5.     : elems_(elems) { }
6.
7.     T& get(int index) { return elems_[index]; }
8.
9.     friend ostream& operator<<(ostream& out, const Liste& liste) {
10.         for (auto&& e : liste.elems_)
11.             out << e << " ";
12.         return out;
13.     }
14.
15. private:
16.     vector<T> elems_;
17. };
18.
19.
20. int main() {
21.     struct Spam {
22.         int data;
23.     };
24.     Liste<int> foo({1, 2, 3});
25.     Liste<Spam> bar({Spam{1}, Spam{2}});
26.     foo.get(0) = 42;
27.     bar.get(1) = Spam{42};
28.     cout << foo << "\n";
29.     cout << bar << "\n";
30.     return 0;
31. }
```

Choisissez la déclaration qui décrit le mieux le code ci-dessus.

- ☐ a. Le code ne compile pas à cause de l'instanciation `Liste<Spam>` à la ligne 25. Le type générique de `Liste` doit **absolument** supporter l'opérateur d'affichage `<<`, sinon toute instanciation est invalide, qu'on se serve ou non de l'opérateur dans le `main()`. `Liste<int>` est toutefois valide.
- ☐ b. Le code ne compile pas à cause de l'instanciation `Liste<Spam>` à la ligne 25. Le type générique de `Liste` doit **absolument** supporter l'opérateur d'affichage `<<`, sinon toute instanciation est invalide, qu'on se serve ou non de l'opérateur. `Liste<int>` n'est pas valide non plus.
- ☒ c. Le `main()` ne compile pas **uniquement** à cause de la ligne 29 qui essaie d'utiliser l'opérateur `<<` pour `Liste<Spam>`. Hors,  celui-ci utilise l'opérateur `<<` pour `Spam` qui n'existe pas. Toutes les autres lignes du `main()` sont cependant valides.

Votre réponse est correcte.

La réponse correcte est : Le `main()` ne compile pas **uniquement** à cause de la ligne 29 qui essaie d'utiliser l'opérateur `<<` pour `Liste<Spam>`. Hors, celui-ci utilise l'opérateur `<<` pour `Spam` qui n'existe pas. Toutes les autres lignes du `main()` sont cependant valides.



## Question 5

Correct

Note de 2,75 sur 2,75

Cette question devrait être faite sans utiliser un compilateur.

Pour chaque case de cette question, indiquez le type qu'il faut mettre devant le nom de variable dans la déclaration pour que l'affectation soit correcte et qu'elle ne change pas le type de la valeur (ex. : `bool x = 2;` est une affectation correcte mais change le 2 d'un entier à un booléen, donc pas accepté comme réponse). Aucune des réponses n'est "const" ni une référence à quelque chose qui pouvait être affecté par copie.

Indiquez 'X' comme type si l'expression ne peut pas être correcte.

Vous n'obtenez aucun point pour une réponse *auto*.

Réponse : (régime de pénalités : 0 %)

```
// Soient les déclarations:
struct C { char a; float b; string** c; };
struct B { int a; C b; C** c; B* d; };
struct A { double* a; short b; B** c; };
B* a(int x);
B b;
A c;
```

// Indiquez les bons types devant les variables suivantes:

X  d = b.c->c;

B  e = \*(a(3));

C  f = a(1)->d->b;

B  g = \*\*(c.c);

B\*  h = a(5);

char\*  i = new char;

C  j = b.d->c[0][0];

char\*  k = &(b.c[0]->a);

double  m = c.a[0];

C\*  n = b.d->c[0];

Réponse bien enregistrée: X, B, C, B, B\*, char\*, C, char\*, double, C\*

Ce message vert n'indique pas si la réponse est bonne ou non, seulement que toutes les cases ont été remplies:

Tous les tests ont été réussis ! ✔

Correct

Note pour cet envoi : 2,75/2,75.



## Description

Soit la classe générique `Liste` suivante. Elle représente un tableau dynamique d'éléments contigus, un peu comme `std::vector`. Elle est composée d'une capacité (`capacite` : le nombre total de cases qui peuvent être remplies), d'un nombre d'éléments (`nbElems` : les éléments concrètement utilisés) et d'un tableau alloué dynamiquement (`elements`). Le tableau sous-jacent est sous la forme d'un pointeur intelligent d'éléments (`unique_ptr<T[]>`).

```
1. template <typename T>
2. class Liste
3. {
4. public:
5.     Liste() = default;
6.     Liste(const Liste& autre);
7.
8.     ~Liste();
9.
10.    Liste& operator=(const Liste& autre);
11.
12.    int getNbElems() const { return nbElems; }
13.    int getCapacite() const { return capacite; }
14.    T& operator[](int index) { return elements[index]; }
15.    const T& operator[](int index) const { return elements[index]; }
16.
17.    void ajouter(const T& elem);
18.    void changerCapacite(int nouvelleCapacite);
19.
20.    template <typename>
21.    friend ostream& operator<<(ostream& out, const Liste& liste);
22.
23. private:
24.     int nbElems = 0;
25.     int capacite = 0;
26.     unique_ptr<T[]> elements;
27. };
```

Dans les questions qui suivent, vous devez implémenter certaines méthodes de cette classe en respectant strictement la déclaration ci-dessus. Vous faites vos définitions à l'extérieur de la classe et vous ne pouvez pas changer les signatures des méthodes ou en ajouter de nouvelles. Notez que dans les tests qu'on vous fournit, il semble y avoir des utilisations illégales de membres privés. C'est fait exprès pour alléger l'écriture des tests et permettre à ceux-ci d'inspecter les données internes de la classe.

Vous pouvez réouvrir cette page dans un autre onglet ou fenêtre puisque vous aurez à vous y référer souvent pour les prochaines questions.



Question **6**

Correct

Note de 3,50 sur 3,50

À partir de la déclaration fournie de la classe générique `Liste`, écrivez la définition de la méthode `changerCapacite()` qui redimensionne la liste selon une nouvelle capacité. On veut changer la taille du tableau dynamique alloué, mais en conservant les éléments déjà présents. Donc, si on a une liste taille 10 et capacité 10 et qu'on fait `.changerCapacite(15)`, on aurait maintenant une liste de taille 10 et de capacité 15 et les 10 éléments conservés. Si on fait par la suite `.changerCapacite(5)`, on aurait une liste de taille 5 et de capacité 5, on a donc perdu les 5 derniers éléments.

Pour simplifier le code, on fait toujours la réallocation, même si la capacité diminue. Vous devez utiliser les pointeurs intelligents, vous ne pouvez pas utiliser `new` et `delete`. Votre code correspond à la définition de la méthode à l'extérieur de la déclaration de la classe.

Par exemple:

Test	Résultat
<pre>// Vérification de changement de taille/capacité Liste&lt;double&gt; foo; foo.changerCapacite(4); foo.nbElems = 3; cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.changerCapacite(2); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.changerCapacite(3); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n";</pre>	<pre>nbElems=3 capacite=4 nbElems=2 capacite=2 nbElems=2 capacite=3</pre>

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
1 // By the way, C'est quoi le STEP ?
2 template <typename T>
3 void Liste<T>::changerCapacite(int nouvelleCapacite)
4 {
5     unique_ptr<T[]> nouveauxElements = make_unique<T[]>(nouvelleCapacite);
6     nbElems = min(nouvelleCapacite, nbElems);
7     capacite = nouvelleCapacite;
8
9     for (int i : range(nbElems))
10    {
11        nouveauxElements[i] = elements[i];
12    }
13
14    elements = move(nouveauxElements);
15 }
```

Test	Résultat attendu	Résultat obtenu	
------	------------------	-----------------	--





	Test	Résultat attendu	Résultat obtenu	
✓	<pre>// Vérification de changement de taille/capacité Liste&lt;double&gt; foo; foo.changerCapacite(4); foo.nbElems = 3; cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.changerCapacite(2); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.changerCapacite(3); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n";</pre>	<pre>nbElems=3 capacite=4 nbElems=2 capacite=2 nbElems=2 capacite=3</pre>	<pre>nbElems=3 capacite=4 nbElems=2 capacite=2 nbElems=2 capacite=3</pre>	✓
✓	<pre>// Vérification des valeurs d'éléments correctes Liste&lt;double&gt; foo; foo.changerCapacite(2); foo.nbElems = 2; foo[0] = 420.69; foo[1] = 1337.42; cout &lt;&lt; foo &lt;&lt; "\n"; foo.changerCapacite(3); foo.nbElems = 3; foo[2] = 9000; cout &lt;&lt; foo &lt;&lt; "\n"; foo.changerCapacite(1); cout &lt;&lt; foo &lt;&lt; "\n";</pre>	<pre>[420.69 1337.42] [420.69 1337.42 9000] [420.69]</pre>	<pre>[420.69 1337.42] [420.69 1337.42 9000] [420.69]</pre>	✓
✓	<pre>// Vérification de l'allocation Liste&lt;double&gt; foo; foo.changerCapacite(4); cout &lt;&lt; "nullptr? " &lt;&lt; (foo.elements.get() == nullptr) &lt;&lt; "\n"; auto* old = foo.elements.get(); foo.changerCapacite(2); cout &lt;&lt; "Changement? " &lt;&lt; (foo.elements.get() != old) &lt;&lt; "\n"; old = foo.elements.get(); foo.changerCapacite(3); cout &lt;&lt; "Changement? " &lt;&lt; (foo.elements.get() != old) &lt;&lt; "\n";</pre>	<pre>nullptr? false Changement? true Changement? true</pre>	<pre>nullptr? false Changement? true Changement? true</pre>	✓

Tous les tests ont été réussis ! ✓

### Solution de l'auteur de la question (Cpp):

```
1 | template <typename T>
2 | void Liste<T>::changerCapacite(int nouvelleCapacite) {
3 |     auto nouvelleListe = make_unique<T[]>(nouvelleCapacite);
4 |
5 |     if (elements != nullptr) {
6 |         nbElems = min(nouvelleCapacite, nbElems);
7 |         for (int i = 0; i < nbElems; i++)
8 |             nouvelleListe[i] = move(elements[i]);
9 |     }
10 |
11 |     elements = move(nouvelleListe);
12 |     capacite = nouvelleCapacite;
13 | }
14 |
```

Correct

Note pour cet envoi : 3,50/3,50.

Commentaire : Question liste\_changerCapacite. 3.5/3.5



Question **7**

Correct

Note de 2,00 sur 2,00

À partir de la déclaration fournie de la classe générique `Liste`, écrivez la définition de la méthode `ajouter()` qui ajoute un élément à la fin de la liste. On fait la réallocation du tableau en doublant sa capacité s'il ne reste pas de place en s'assurant qu'il y a au moins une capacité d'un élément (sinon, le double de zéro resterait zéro). On ne fait pas de réallocation s'il y a suffisamment de place.

Pour ce numéro, la méthode `changerCapacite()` est implémentée pour vous et vous pouvez l'utiliser.

Par exemple:

Test	Résultat
<pre>// Vérification du doublage de capacité à partir de 0. Liste&lt;int&gt; foo; cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(1); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(2); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(3); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(4); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(5); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n";</pre>	<pre>nbElems=0 capacite=0 nbElems=1 capacite=1 nbElems=2 capacite=2 nbElems=3 capacite=4 nbElems=4 capacite=4 nbElems=5 capacite=8 foo: [1 2 3 4 5]</pre>

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
1 // By the way, C'est quoi le STEP ?
2 template <typename T>
3 void Liste<T>::ajouter(const T& elem)
4 {
5     if (nbElems + 1 > capacite)
6     {
7         changerCapacite(max(capacite*2, 1));
8     }
9
10    elements[nbElems++] = elem;
11 }
```

Test	Résultat attendu	Résultat obtenu
------	------------------	-----------------



	Test	Résultat attendu	Résultat obtenu	
✓	<pre>// Vérification du doublage de capacité à partir de 0. Liste&lt;int&gt; foo; cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(1); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(2); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(3); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(4); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(5); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n";</pre>	<pre>nbElems=0 capacite=0 nbElems=1 capacite=1 nbElems=2 capacite=2 nbElems=3 capacite=4 nbElems=4 capacite=4 nbElems=5 capacite=8 foo: [ 1 2 3 4 5]</pre>	<pre>nbElems=0 capacite=0 nbElems=1 capacite=1 nbElems=2 capacite=2 nbElems=3 capacite=4 nbElems=4 capacite=4 nbElems=5 capacite=8 foo: [ 1 2 3 4 5]</pre>	✓
✓	<pre>// Vérification du doublage de capacité à partir d'une taille quelconque. Liste&lt;int&gt; foo; foo.changerCapacite(10); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; for (int i = 0; i &lt; 10; i++)     foo.ajouter(42); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(42); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n"; foo.ajouter(42); cout &lt;&lt; "nbElems=" &lt;&lt; foo.getNbElems() &lt;&lt; " capacite=" &lt;&lt; foo.getCapacite() &lt;&lt; "\n";</pre>	<pre>nbElems=0 capacite=10 nbElems=10 capacite=10 nbElems=11 capacite=20 nbElems=12 capacite=20</pre>	<pre>nbElems=0 capacite=10 nbElems=10 capacite=10 nbElems=11 capacite=20 nbElems=12 capacite=20</pre>	✓
✓	<pre>// Vérification d'allocations inutiles. Liste&lt;double&gt; foo; foo.ajouter(1); cout &lt;&lt; "nullptr? " &lt;&lt; (foo.elements.get() == nullptr) &lt;&lt; "\n"; auto* old = foo.elements.get(); foo.ajouter(2); cout &lt;&lt; "Changement? " &lt;&lt; (foo.elements.get() != old) &lt;&lt; "\n"; old = foo.elements.get(); foo.ajouter(3); cout &lt;&lt; "Changement? " &lt;&lt; (foo.elements.get() != old) &lt;&lt; "\n"; old = foo.elements.get(); foo.ajouter(4); cout &lt;&lt; "Changement? " &lt;&lt; (foo.elements.get() != old) &lt;&lt; "\n"; old = foo.elements.get();</pre>	<pre>nullptr? false Changement? true Changement? true Changement? false</pre>	<pre>nullptr? false Changement? true Changement? true Changement? false</pre>	✓

Tous les tests ont été réussis ! ✓

### Solution de l'auteur de la question (Cpp):

```
1 template <typename T>
2 void Liste<T>::ajouter(const T& elem) {
3     if (nbElems == capacite)
4         changerCapacite(max(1, capacite * 2));
5     elements[nbElems++] = elem;
6 }
7
```



Correct

Note pour cet envoi : 2,00/2,00.



## Question 8

Correct

Note de 3,50 sur 3,50

À partir de la déclaration fournie de la classe générique `Liste`, écrivez la définition du constructeur de copie et de l'opérateur d'affectation de copie. Ceux-ci doivent faire des copies profondes des tableaux sous-jacents. Pour l'opérateur d'affectation, il faut réallouer seulement si c'est nécessaire. Par exemple, si on affecte une liste de 10 éléments à une liste existante qui a une capacité de 15, aucune réallocation est nécessaire et on ne fait que copier les éléments.

Pour ce numéro, les méthodes `changerCapacite()` et `ajouter()` sont implémentées pour vous et vous pouvez les utiliser. N'oubliez pas que l'opérateur d'affectation peut être réutilisé dans le constructeur de copie ou *vice-versa*.

Par exemple:

Test	Résultat
<pre>// Vérification de copie d'éléments Liste&lt;int&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;int&gt; bar = foo; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n"; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; Liste&lt;int&gt; qux; qux.ajouter(11); qux.ajouter(12); bar = qux; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; cout &lt;&lt; "qux: " &lt;&lt; qux &lt;&lt; "\n";</pre>	<pre>foo: [1 2 3] bar: [1 2 3] bar: [11 12] qux: [11 12]</pre>

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
1 // By the way, C'est quoi le STEP ?
2 template <typename T>
3 Liste<T>::Liste(const Liste& autre)
4 {
5     *this = autre;
6 }
7
8 template <typename T>
9 Liste<T>& Liste<T>::operator=(const Liste& autre)
10 {
11     if (this != &autre)
12     {
13         if (autre.nbElems > capacite)
14         {
15             changerCapacite(autre.capacite);
16         }
17
18         for (int i : range(autre.nbElems))
19         {
20             elements[i] = autre[i];
21         }
22         nbElems = autre.nbElems;
23     }
```

Test	Résultat attendu	Résultat obtenu	
------	------------------	-----------------	--



	Test	Résultat attendu	Résultat obtenu	
✓	<pre>// Vérification de copie et d'affectation de listes vides Liste&lt;double&gt; foo; Liste&lt;double&gt; bar = foo; cout &lt;&lt; "nullptr? " &lt;&lt; (foo.elements == nullptr) &lt;&lt; "\n"; cout &lt;&lt; "nullptr? " &lt;&lt; (bar.elements == nullptr) &lt;&lt; "\n"; foo = bar; cout &lt;&lt; "nullptr? " &lt;&lt; (bar.elements == nullptr) &lt;&lt; "\n";</pre>	<pre>nullptr? true nullptr? true nullptr? true</pre>	<pre>nullptr? true nullptr? true nullptr? true</pre>	✓
✓	<pre>// Vérification de copie d'éléments Liste&lt;int&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;int&gt; bar = foo; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n"; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; Liste&lt;int&gt; qux; qux.ajouter(11); qux.ajouter(12); bar = qux; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; cout &lt;&lt; "qux: " &lt;&lt; qux &lt;&lt; "\n";</pre>	<pre>foo: [1 2 3] bar: [1 2 3] bar: [11 12] qux: [11 12]</pre>	<pre>foo: [1 2 3] bar: [1 2 3] bar: [11 12] qux: [11 12]</pre>	✓
✓	<pre>// Vérification de copie profonde Liste&lt;int&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;int&gt; bar = foo; cout &lt;&lt; "Meme pointeurs? " &lt;&lt; (foo.elements.get() == bar.elements.get()) &lt;&lt; "\n"; Liste&lt;int&gt; qux; qux.ajouter(11); qux.ajouter(12); bar = qux; cout &lt;&lt; "Meme pointeurs? " &lt;&lt; (bar.elements.get() == qux.elements.get()) &lt;&lt; "\n";</pre>	<pre>Meme pointeurs? false Meme pointeurs? false</pre>	<pre>Meme pointeurs? false Meme pointeurs? false</pre>	✓
✓	<pre>// Vérification d'allocations inutiles Liste&lt;int&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;int&gt; bar; bar.changerCapacite(3); auto* old = bar.elements.get(); bar = foo; cout &lt;&lt; "Changement? " &lt;&lt; (bar.elements.get() != old) &lt;&lt; "\n";</pre>	<pre>Changement? false</pre>	<pre>Changement? false</pre>	✓
✓	<pre>// Vérification de l'enchaînement d'affectation Liste&lt;int&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;int&gt; bar; bar.ajouter(10); bar.ajouter(20); bar.ajouter(30); Liste&lt;int&gt; qux; qux.ajouter(40); qux.ajouter(60); bar = foo = qux; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n"; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; cout &lt;&lt; "qux: " &lt;&lt; qux &lt;&lt; "\n";</pre>	<pre>foo: [420 69] bar: [420 69] qux: [420 69]</pre>	<pre>foo: [420 69] bar: [420 69] qux: [420 69]</pre>	✓
✓	<pre>// Vérification de copie d'éléments d'un type spécial Liste&lt;MonType&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); Liste&lt;MonType&gt; bar = foo; cout &lt;&lt; "foo: " &lt;&lt; foo &lt;&lt; "\n"; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; Liste&lt;MonType&gt; qux; qux.ajouter(11); qux.ajouter(12); bar = qux; cout &lt;&lt; "bar: " &lt;&lt; bar &lt;&lt; "\n"; cout &lt;&lt; "qux: " &lt;&lt; qux &lt;&lt; "\n";</pre>	<pre>foo: [1 2 3] bar: [1 2 3] bar: [11 12] qux: [11 12]</pre>	<pre>foo: [1 2 3] bar: [1 2 3] bar: [11 12] qux: [11 12]</pre>	✓

	Test	Résultat attendu	Résultat obtenu	
✓	<pre>Liste&lt;MonType&gt; foo; foo.ajouter(1); foo.ajouter(2); foo.ajouter(3); MonType::reset(); foo = foo; cout &lt;&lt; MonType::nCopies &lt;&lt; ' ' &lt;&lt; MonType::nMoves;</pre>	0 0	0 0	✓

Tous les tests ont été réussis ! ✓

### Solution de l'auteur de la question (Cpp):

```

1  template <typename T>
2  Liste<T>::Liste(const Liste& autre) {
3      *this = autre;
4  }
5
6  template <typename T>
7  Liste<T>& Liste<T>::operator=(const Liste& autre) {
8      if (&autre == this)
9          return *this;
10     if (autre.nbElems > capacite)
11         changerCapacite(autre.nbElems);
12     for (int i = 0; i < autre.nbElems; i++)
13         elements[i] = autre.elements[i];
14     nbElems = autre.nbElems;
15     return *this;
16 }
17
```

Correct

Note pour cet envoi : 3,50/3,50.

Commentaire : Question liste\_copie. 3.5/3.5

#### Question 9

Correct

Note de 1,00 sur 1,00

En prenant en compte tous les requis demandés dans les questions précédentes sur la classe `Liste`, quelle affirmation décrit le mieux ce qui doit être fait dans le destructeur de `Liste`?

- ☐ a. Pour libérer les ressources contenues dans `elements` et éviter les fuites, il faut faire manuellement `delete[] elements.get()` dans le corps du destructeur, car c'est un tableau dynamique (`delete` ne fonctionnerait pas).
- ☒ b. Il n'y a rien à écrire manuellement dans le destructeur, on peut le mettre = `default` ou lui donner un corps vide. ✓
- ☐ c. Pour libérer les ressources contenues dans `elements` et éviter les fuites, il faut faire manuellement `elements.reset()` dans le corps du destructeur. Sans cela, le tableau dynamique ne sera pas libéré.

Votre réponse est correcte.

La réponse correcte est : Il n'y a rien à écrire manuellement dans le destructeur, on peut le mettre = `default` ou lui donner un corps vide.



## Question 10

Terminer

Non noté

Sur mon honneur, je (écrire votre nom) , affirme que j'ai fait  cet examen par moi-même, sans communication avec personne (autre que les enseignants indiqués en première page en cas de problème), et selon les directives identifiées dans cet énoncé d'examen.

## Question 11

Terminer

Non noté

Si nécessaire, inscrivez vos suppositions ici, en précisant pour chaque supposition le numéro de la question concernée.

[◀ Réponses aux questions de pratique](#)[Quiz 3-Partie 1-Programmation qui doit passer les tests, Hiver 2022 ►](#)