université PARIS-SACLAY

# Travaux Pratiques de programmation n°6

**Cours de programmation modulaire**

*—Licence MI/IM/MNSI - Info 3—*

## Les classes

Dans cette séance de travaux pratiques, nous allons apprendre à structurer un code en utilisant les `classes`.

---

▸ **Exercice 1. (État civil)**

Dans cet exercice nous allons reprendre le code de l'exercice de l'état civil de la semaine 5 et le restructurer en utilisant une classe `EtatCivil`. Le fichier à compléter est le fichier `ClasseEtatCivil.cpp` fourni. Pour gagner du temps et vous concentrer sur les notions de classes et méthodes, vous pouvez consulter votre fichier de la semaine dernière ou le fichier `EtatCivil_correction.cpp` fourni (ne pas modifier ce fichier). Nous gardons la structure Personne et l'énumeration Genre comme la semaine dernière et nous allons modifier uniquement la structure `EtatCivil`.

1. Déclarer dans la classe `EtatCivil` la méthode `void initialise(string titreEtatCivil);` qui initialise un état civil, puis la définir hors de la classe en utilisant la syntaxe

   ```
   void EtatCivil::initialise(string titreEtatCivil) {
     code de la fonction....
   }
   ```

2. Consulter le `main`, mettre en commentaire les 3 appels à des méthodes non encore écrites (il faudra penser à les décommenter plus tard), exécuter et vérifier que l'affichage obtenu est bien le bon.
3. Déclarer dans la classe EtatCivil la méthode `int cherche(string nom)` qui recherche le nom d'une personne et retourne l'indice de la personne dans l'état civil si elle la trouve ou -1 sinon. Puis définir cette méthode. N'oubliez pas d'ajouter le mot clé `const` après la liste des paramètres pour indiquer que la méthode ne modifie pas l'objet.
4. Tester le bon fonctionnement de la méthode `cherche`. Utiliser la fonction `creeEtatCivildeTest` pour génerer un exemple d'un état civil pour les tests.
5. Déclarer et définir les méthodes `imprimePersonne` et `imprimeEtatCivil`. La méthode `imprimePersonne` affiche dans le terminal une personne dont le nom est passé en paramètre. La méthode `imprimeEtatCivil` affiche dans le terminal un état civil (son nom et sa table). Notez que les deux fonctions membres auront le mot clé `const` dans leurs déclarations puisqu'elles ne modifient pas l'objet EtatCivil.

   ```
   void imprimePersonne(string nom) const;
   void imprimeEtatCivil() const;
   ```

   Facultatif : Si c'est plus simple pour vous, on pourra utiliser une fonction auxilliaire `imprimeIndPersonne` qui prend l'indice d'une personne dans l'état civil et affiche les informations de cette personne.

   ```
   void imprimeIndPersonne(int ind) const;
   ```

   Sinon, commenter l'appel de `imprimeIndPersonne` dans le main.

6. Déclarer, définir et tester la méthode `int personne(string sonNom, Genre s)` qui ajoute une nouvelle personne à l'état civil. Cette méthode modifie l'objet EtatCivil donc elle ne prend pas la mention `const`. La méthode lève une exception au cas où le nom serait vide et où la personne existe déjà dans l'état civil. Sinon, elle renvoie l'indice de la personne dans l'état civil.
   Pour tester la méthode, on pourra utiliser la fonction de `doctest`

   ```
   CHECK_THROWS_WITH_AS(codeATester, "message de l'exception", type_exception);
   ```

   Cette fonction vérifie à la fois que l'exception levé dans le méthode est la même que celle indiqué à la place de `exception` mais également le message de l'exception.

▸ **Exercice 2. (Autres méthodes)** Cet exercice est à faire dans le même fichier que le précédent. Il n'est pas indispensable, donc si vous êtes plutôt lents, passez directement à l'exercice suivant (qui lui est indispensable et devra être terminé d'ici le prochain TP).

Rédiger la déclaration et la définition des méthodes `mariage`, `naissance`, `ascendantI` et `ascendantR` vues dans le TP de la semaine dernière et faire les modifications nécessaires pour les tests.

7. mariage : méthode qui enregistre le mariage de deux personnes dont on passe les noms en paramètre. La fonction renvoie `true` si le mariage est possible et `false` sinon. On impose que les deux personnes soient enregistrées et ne soient pas déjà mariées.

   ```
   bool EtatCivil::mariage(string lun, string lautre);
   ```

8. naissance : méthode qui enregistre la naissance d'une personne. Son en-tête est :

   ```
   bool EtatCivil::naissance(string qui, Genre s, string p1, string p2);
   ```

   Les paramètres sont le nom de l'enfant, son genre, les noms des parents ; les parents doivent être enregistrés et être conjoints. Si les conditions ne sont pas remplies l'enfant n'est pas enregistré. La fonction renvoie true ou false selon que la filiation a pu être enregistrée ou non.

9. ascendantI : méthode (version itérative) qui retourne si une personne est un ancêtre au sens large d'une personne. On pourra se servir d'un vecteur auxiliaire (géré en pile) dans lequel on stockera les indices des personnes dont on doit vérifier si elles sont ou non des ancêtres de la personne concernée, en commençant par ses parents. On rappelle que la méthode `pop_back()` permet de supprimer le dernier élément d'un vecteur. En-tête :

   ```
   bool EtatCivil::ascendantI(string qui, string ancetre);
   ```

10. ascendantR : version récursive de ascendantI

▸ **Exercice 3. (Gestion du stock d'une Pharmacie)**

Un pharmacien souhaite informatiser le traitement des prescriptions de ses clients. Un médicament est représenté par son nom, le nombre de comprimés par boîte, le prix de la boîte et le nombre de boîtes en stock. L'ensemble des médicaments existants est stocké dans le tableau `table` de la classe `Stock`. Chaque prescription est composée d'un nom de médicament, d'un nombre de comprimés à prendre par jour (au plus 6 comprimés par jour), pendant un certain nombre de jours (au plus 31 jours). Les structures de données choisies sont donc les suivantes :

```
struct Medicament {
  string nom;
  int nbBoites;
  int nbParBoite;
  float prixBoite;
};
```

```
struct Stock {
  vector<Medicament> table;
};
```

```
struct Prescription {
  string med;
  int nbCparJour;
  int nbJours;
};
```

**On testera toutes les fonctions après les avoir écrites, et on créera un TEST CASE à chaque fois que c'est possible.**

1. Dans le fichier `Pharmacie.cpp` fourni, écrire et tester la méthode `float prixComprime()` de la classe `Medicament` qui renvoie le prix d'un seul comprimé du médicament.
2. Écrire la méthode `void changePrix(float nouvPrix)` de la classe `Medicament` qui prend en paramètre le nouveau prix par boîte d'un médicament et qui modifie son prix par boîte.
3. Réaliser la méthode `int indiceMedicament(string nomMedicament)` de la classe `Stock` qui permet de chercher un médicament avec son nom dans la base de données du stock et renvoie son indice dans le tableau ou -1 si elle ne le trouve pas. Tester la fonction.
4. Ecrire la méthode `void ajouteMedicament()` de la classe `Stock` qui permet d'ajouter un nouveau médicament à la base de données du stock, à partir de données entrées au clavier par l'utilisateur. Vérifier que le médicament n'existe pas déja.

   Vous pouvez utiliser la fonction

   ```
   int lireValeurBornee(string texteAEcrire, int min, int max);
   ```

   fournie pour vous vous assurez que le nombre de comprimés à prendre par jour ne dépasse pas 6 comprimés et la durée du traitement ne dépasse pas un mois.
5. Ecrire la méthode `void lirePrescription(Stock s)` de la classe `Prescription` qui permet de saisir au clavier les informations relatives à une prescription (nom du médicament, nombre de comprimés par jour et durée du traitement). Sachant que le pharmacien n'accepte pas des préscriptions des médicaments qu'il n'a pas dans sa base de données, il faut donc chercher le nom du médicament dans le stock.
6. Écrire et tester la méthode `int nbBoites` de la classe `Prescription` qui renvoie le nombre de boîtes nécessaires pour couvrir la prescription. Par exemple, si le médicament est vendu par boîte de 20 comprimés, il ne faut qu'une boîte pour couvrir une prescription de 6 comprimés par jour pendant 3 jours, mais il faut 2 boîtes si le traitement dure 4 jours.
7. Écrire et tester la méthode `float coutTotal` de la classe `Prescription` qui renvoie le prix total des boîtes nécessaires pour couvrir la prescription et met à jour la quantité du médicament dans le stock. Si la quantité dans le stock ne couvre pas le nombre des boîtes necessaires, on donne quand même au patient le nombre de boîtes présentes en affichant un message d'avertissement.