université PARIS-SACLAY

# Correction TP de programmation n°8

**Cours de programmation modulaire**

*—Licence MI/IM/MNSI - Info 3—*

## Makefile, Classes

Dans cette séance de travaux pratiques, nous allons apprendre à construire et à utiliser des fichiers Makefile pour faciliter la compilation séparée (partie importante de la programmation modulaire). Nous allons aussi continuer à étudier les classes, les constructeurs, les méthodes, etc.

### ▸ Exercice 1. (Makefile)

Nous allons prendre un programme qui a été fait lors d'un exercice précédent, dont la correction est donnée dans un seul fichier `Pharmacie-un-fichier.cpp`. L'objectif sera de le découper en plusieurs fichiers, et de compiler ces fichiers grâce à Makefile. Il est conseillé de ne pas modifier le fichier fourni (copier-coller les éléments à reprendre), ou d'en faire une sauvegarde avant modification.

1. Compiler et exécuter le fichier fourni pour vérifier que tout se passe comme prévu.
2. Il s'agit dans un premier temps de découper ce fichier en cinq nouveaux fichiers dont les contenus respectifs sont indiqués ci-dessous :
   - `Medicament.hpp` : déclaration des classes `Medicament` et `Stock` ainsi que de la fonction `lireValeurBornee` que l'on utilise dans la méthode `Stock::ajouteMedicament`.
   - `Medicament.cpp` : définition des méthodes des classes précédentes et de la fonction `lireValeurBornee`.
   - `Prescription.hpp` : déclaration de la classe `Prescription`.
   - `Prescription.cpp` : définition des méthodes de la classe précédente.
   - `Pharmacie-main.cpp` qui ne contient que la fonction `main`, ainsi que la fonction `creeUnStockdeTest`.

Avant de passer à la suite, bien faire attention à
- ne pas mettre de définition dans un `.hpp` ;
- inclure le fichier `.hpp` dans le fichier `.cpp` de même nom ;
- inclure dans un fichier `.hpp` les bibliothèques et autres fichiers `.hpp` qu'il utilise – par exemple `Prescription.hpp` doit inclure `Medicament.hpp` car il utilise la classe `Stock`, déclarée dans ce dernier ;
- mettre les gardes d'inclusion multiple dans les fichiers `.hpp`

Il ne faudra pas oublier d'ajouter avec `git add` les fichiers créés afin qu'ils soient bien pris en compte dans la soumission (voir dernière question de l'exercice).

Pour compiler les fichiers obtenus, on va dans les questions suivantes créer au fur et à mesure différentes versions de `Makefile` en suivant la démarche du cours.

Normalement quand on lance la commande `make`, on tape

`make` *cible* ou bien `make`

et l'outil `make` va chercher la configuration dans le fichier `Makefile`. Dans ce TP, nous utilisons un nom différent pour le fichier `Makefile` car nous en aurons plusieurs (`Makefile.v1`, `Makefile.v2`, etc.). Il faut donc préciser à `make` dans quel fichier il doit aller chercher sa configuration. Pour ceci, chaque `Makefile` devra être lancé avec la commande

`make -r -f Makefile.v`*i* *cible*

L'option `-f` permet de sélectionner quel fichier remplace le `Makefile`. L'option `-r` élimine l'utilisation des règles implicites intégrées. Ainsi, seules les règles que vous écrirez seront appliquées.

Pour chaque Makefile réalisé, il faudra lancer votre Makefile puis vérifier que le nouvel exécutable obtenu a bien le même comportement que celui observé à la première question. Puis avant de passer au Makefile suivant, supprimer tous les exécutables et fichiers .o (soit à la main, soit en utilisant `make clean` si vous l'avez écrit) afin que le compilateur recompile effectivement avec le nouveau Makefile (sinon il ne se passera rien car les cibles auront déjà été créées par le précédent Makefile).

**Attention** : Si vous mettez des espaces à la place des tabulations, vous aurez le message d'erreur `*** missing separator. Stop`. Certains éditeurs, notamment celui de Jupyter, remplacent automatiquement les tabulations par des espaces. Si vous êtes en salle de TP, vous pouvez travailler en local et utiliser l'éditeur `jedit`. Sinon, selon l'éditeur, nous vous conseillons la configuration suivante :

`JupyterLab` :
— Cliquer sur l'onglet «paramètres» ;
— Aller dans la rubrique «Indentation de l'éditeur de texte» ;
— Cocher la case «Indenter avec les tabulations».

`Geany:`
— Cliquer sur le menu «Affichage» ;
— Cocher la case «Afficher les espaces».

Sinon vous pouvez copier-coller une tabulation depuis un autre `Makefile` (par exemple un de ceux du cours).

3. Créer un premier `Makefile.v1` minimal qui permet de compiler les 5 fichiers découpés.

✂ ----------

```
1 # fichier executable
2 Pharmacie-main: Medicament.o Prescription.o Pharmacie-main.o
3         g++ -std=c++11 -Wall -o Pharmacie-main Medicament.o Prescription.o Pharmacie-main.o
4
5 # fichiers objets
6 Pharmacie-main.o: Pharmacie-main.cpp Prescription.hpp Medicament.hpp
7         g++ -std=c++11 -Wall -c Pharmacie-main.cpp
8 Prescription.o: Prescription.cpp Prescription.hpp Medicament.hpp
```

```
9         g++ -std=c++11 -Wall -c Prescription.cpp
10 Medicament.o: Medicament.cpp Medicament.hpp
11        g++ -std=c++11 -Wall -c Medicament.cpp
```

**Attention !** Les dépendances d'un fichier `.o` doivent contenir non seulement le fichier `.cpp` du même nom, mais aussi **tous les fichier `.hpp` qui on été inclus, directement ou indirectement.**

4. Modifier ce fichier pour avoir `Makefile.v2` en ajoutant les deux cibles "all" et "clean".

```
1  all: Pharmacie-main
2
3  # fichier executable
4  Pharmacie-main: Medicament.o Prescription.o Pharmacie-main.o
5          g++ -std=c++11 -Wall -o Pharmacie-main Medicament.o Prescription.o Pharmacie-main.o
6
7  # fichiers objets
8  Pharmacie-main.o: Pharmacie-main.cpp Prescription.hpp Medicament.hpp
9          g++ -std=c++11 -Wall -c Pharmacie-main.cpp
10 Prescription.o: Prescription.cpp Prescription.hpp Medicament.hpp
11         g++ -std=c++11 -Wall -c Prescription.cpp
12 Medicament.o: Medicament.cpp Medicament.hpp
13         g++ -std=c++11 -Wall -c Medicament.cpp
14
15 clean:
16         rm -f *.o Pharmacie-main
```

5. Modifier ce fichier pour avoir `Makefile.v3` en ajoutant les trois variables `CXX`, `CXXFLAGS` et `EXEC_FILES`.

```
1  CXX=g++
2  CXXFLAGS= -Wall -std=c++11 -g -O3
3  EXEC_FILES= Pharmacie-main
4
5  all: Pharmacie-main
6
7  # fichier executable
8  Pharmacie-main: Medicament.o Prescription.o Pharmacie-main.o
9          $(CXX) $(CXXFLAGS) -o Pharmacie-main Medicament.o Prescription.o Pharmacie-main.o
10
11 # fichiers objets
12 Pharmacie-main.o: Pharmacie-main.cpp Prescription.hpp Medicament.hpp
13         $(CXX) $(CXXFLAGS) -c Pharmacie-main.cpp
14 Prescription.o: Prescription.cpp Prescription.hpp Medicament.hpp
15         $(CXX) $(CXXFLAGS) -c Prescription.cpp
```

```
16 Medicament.o: Medicament.cpp Medicament.hpp
17         $(CXX) $(CXXFLAGS) -c Medicament.cpp
18
19 clean:
20         rm -f *.o $(EXEC_FILES)
```

---

6. Modifier ce fichier pour avoir `Makefile.v4` en ajoutant des variables dites internes, comme `$@`, `$^`, `$<`.

---

```
1  CXX=g++
2  CXXFLAGS= -Wall -std=c++11 -g -O3
3  EXEC_FILES= Pharmacie-main
4
5  all: Pharmacie-main
6
7  # fichier executable
8  Pharmacie-main: Medicament.o Prescription.o Pharmacie-main.o
9          $(CXX) -o $@ $^ $(LDFLAGS)
10
11 # fichiers objets
12 Pharmacie-main.o: Pharmacie-main.cpp Prescription.hpp Medicament.hpp
13         $(CXX) -o $@ -c $< $(CXXFLAGS)
14 Prescription.o: Prescription.cpp Prescription.hpp Medicament.hpp
15         $(CXX) -o $@ -c $< $(CXXFLAGS)
16 Medicament.o: Medicament.cpp Medicament.hpp
17         $(CXX) -o $@ -c $< $(CXXFLAGS)
18
19 clean:
20         rm -f *.o $(EXEC_FILES)
```

---

7. Modifier ce fichier pour avoir `Makefile.v5` en ajoutant une règle générique.

---

```
1  CXX=g++
2  CXXFLAGS= -Wall -std=c++11 -g -O3
3  EXEC_FILES= Pharmacie-main
4
5  all: Pharmacie-main
6
7  # Regle generique
8  %.o: %.cpp
9          $(CXX) -o $@ -c $< $(CXXFLAGS)
10
11 # fichier executable
12 Pharmacie-main: Medicament.o Prescription.o Pharmacie-main.o
13         $(CXX) -o $@ $^ $(LDFLAGS)
```

```
14
15 # fichiers objets
16 Pharmacie-main.o: Pharmacie-main.cpp Prescription.hpp Medicament.hpp
17 Prescription.o: Prescription.hpp Medicament.hpp
18 Medicament.o:  Medicament.hpp
19
20 clean:
21         rm -f *.o $(EXEC_FILES)
```

**Attention !** Ne pas laisser de tabulation toute seule sur une ligne blanche – que vous avez peut-être copié-collé mais qui peut être invisible selon votre éditeur. En effet, `make` comprend que la commande à exécuter est vide et n'applique pas la règle générique.

8. Pour que les fichiers nouvellement créés (Makefiles, entêtes et sources) soient pris en compte par `submit`, il ne faut pas oublier de les ajouter avec la commande

   ```
   git add NomDuFichier
   ```

   Après les avoir ajoutés, vérifier qu'ils sont bien pris en compte en tapant la commande

   ```
   git status
   ```

   Les fichiers non pris en compte se trouvent en dessous de

   ```
   Untracked files:
     (use "git add <file>..." to include in what will be committed)
   ```

   Il est normal que les exécutables et les fichiers .o ne soient pas pris en compte, mais les Makefile et les fichiers .hpp et .cpp doivent être pris en compte.

▸ **Exercice 2. (Les tours de Hanoi)** Si vous n'aviez pas fini cet exercice la semaine dernière, le finir maintenant. Se reporter au sujet de la semaine dernière pour l'énoncé.

▸ **Exercice 3. (Séquences génomiques)**

Dans le contexte de la biologie moléculaire, on s'intéresse à la modélisation de séquences d'ADN. Il n'est pas nécessaire d'avoir des connaissances en biologie pour résoudre cet exercice. Toutes les informations nécessaires vous seront données dans l'énoncé.

L'ADN est une séquence de nucléotides (bases) qui sont : adénine (A), thymine (T), guanine (G), cytosine (C). C'est la succession de ces bases résultant de l'enchaînement des nucléotides qui constitue le message génétique.

Dans cet exercice, nous modéliserons les séquences d'ADN sous la forme d'une classe `ADN` contenant un unique attribut `sequence` qui est un vecteur de caractères.

1. Dans la classe `ADN` du fichier `Genome.cpp` fourni, déclarez l'attribut `sequence` puis définissez un constructeur à partir d'un vecteur donné.

```
13
14  struct ADN {
15    vector<char> sequence;
16
17    // Constructeurs
18    //
19    ADN(const vector<char> &seq) : sequence{seq} {};
20    ADN(int n);  // Ajouté en question 5
21    //
```

2. Coder une méthode `int nb_nucleotides()` qui retourne le nombre de nucléotides d'une séquence.

```
26    int nb_nucleotides() const { return sequence.size(); }
```

3. Coder une méthode `int occurence_nucleotide(char c)` qui prend en paramètre un caractère et qui retourne son nombre d'occurences dans une séquence.

```
37  int ADN::occurence_nucleotide(char c) const {
38    //
39    int nb = 0;
40    for (size_t i = 0; i < sequence.size(); i++)
41      if (c == sequence[i]) nb++;
42    return nb;
43   // Fincorrection
44
45  }
```

4. Proposer des tests pour `occurence_nucleotide`.

```
49 TEST_CASE("occurence_nucleotide") {
50   //
51   CHECK(ADN{{}}.occurence_nucleotide('A') == 0);
52   CHECK(ADN{{'A', 'A', 'F'}}.occurence_nucleotide('A') == 2);
53   CHECK(ADN{{'A', 'C', 'C'}}.occurence_nucleotide('G') == 0);
54   //
55 }
```

5. Déclarez et définissez un constructeur à partir d'un entier donné qui représente la taille de la séquence. Le constructeur choisit au hasard les nucléotides.

```
59 ADN::ADN(int n) : sequence{} {
60   //
61   int nuc;
62   while (n != 0) {
63     nuc = rand() % 4;
64     n--;
65     switch (nuc) {
66     case 0:
67       sequence.push_back('A');
68       break;
69     case 1:
70       sequence.push_back('C');
71       break;
72     case 2:
73       sequence.push_back('G');
74       break;
75     case 3:
76       sequence.push_back('T');
77       break;
78     }
79   }
80   //
81 }
```

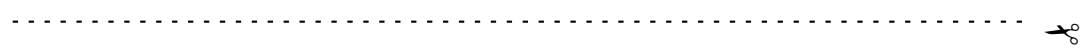
6. Coder une méthode `bool estADN()` qui teste si une séquence est une séquence d'ADN correctement formée, c'est-à-dire composée uniquement des caractères A, T, G et C. La fonction retournera `true` si la séquence est un ADN correctement formé, `false` sinon. Vous devrez essayer de faire en sorte que la fonction s'arrête à partir du moment où elle a trouvé un caractère différent de A, T, G et C, car ce n'est alors plus la peine de regarder le reste de la séquence. Proposer par la suite des tests pertinents.

```
bool ADN::estADN() const {
  //
  int i = 0;
  while (i < nb_nucleotides()) {
    if (sequence[i] != 'A' and sequence[i] != 'C' and sequence[i] != 'G' and
        sequence[i] != 'T') {
      return false;
    }
    i++;
  }
  if (i == 0)
    return false;
  else
    return true;
  //
}

TEST_CASE("estADN") {
  //
  CHECK_FALSE(ADN{{}}.estADN());
  CHECK_FALSE(ADN{{'A', 'C', 'F'}}.estADN());
  CHECK(ADN{{'A', 'T', 'G'}}.estADN());
  //
}
```

7. Pour les organismes vertébrés, le rapport entre le nombre de nucléotides de type C et G cumulé et le nombre de nucléotides total doit être compris entre 0,40 et 0,45 (exclus). En suivant ce critère, coder une méthode `bool estADNVertebre()` qui teste si une séquence est une séquence d'ADN d'un vertébré. La fonction retournera `true` si la séquence est un ADN d'un vertébré, `false` sinon. Proposer par la suite des tests pertinents.

```
bool ADN::estADNVertebre() const {
  //
  bool resultat = false;
  if (estADN()) {
    int cETg = occurence_nucleotide('C') + occurence_nucleotide('G');
    float rapport = float(cETg) / float(nb_nucleotides());
    resultat = (rapport > 0.40) and (rapport < 0.45);
  }
  return resultat;
  //
}

TEST_CASE("estADNVertebre") {
  //
  CHECK_FALSE(ADN{{}}.estADNVertebre());
```

```
138   CHECK_FALSE(ADN{{'A', 'A', 'F'}}.estADNVertebre());
139   CHECK_FALSE(ADN{{'A', 'C', 'A'}}.estADNVertebre());
140   CHECK(ADN{
141     {'A', 'C', 'A', 'A', 'G', 'T', 'A', 'G', 'G', 'T', 'G',
142      'A'}}.estADNVertebre());
143   //
144 }
```



**Exercice 4. (Encore plus de nombres !)**

Voici quelques suggestions de travaux intéressants pour ceux qui vont vite :

La semaine dernière nous avons travaillé sur les entiers naturels. Vous pouvez reprendre ce travail et l'utiliser dans une nouvelle classe `Relatif` qui code un entier relatif (positif ou négatif). Cette classe a deux attributs, la valeur absolue qui est de type `Naturel` et le signe qui sera d'un nouveau type énuméré.

**Attention :** il n'y a qu'un seul zéro et il est positif !

1. Créer deux fichiers `.hpp` et `.cpp` pour les entiers naturels à partir de ce que vous avez fait la semaine dernière.
2. Créer deux nouveaux fichiers `.hpp` et `.cpp` pour les entiers relatifs. On définira les mêmes méthodes et opérateurs que pour les naturels. Bien tout tester rigoureusement.
3. Plus difficile : surcharger l'opérateur de multiplication pour les naturels puis pour les relatifs. Bien tester.
4. Encore plus difficile : surcharger les opérateurs de division et de reste pour les naturels puis pour les relatifs. Bien tester.
5. En utilisant l'algorithme d'euclide (voir le fichier du cours sur les rationels), vous pouvez maintenant calculer les pgcd et les ppcm !
6. Si vous êtes arrivé ici, vous pouvez maintenant reprendre le fichier sur les rationnels vu en cours et l'adapter pour qu'il marche avec vos nombres naturels et relatifs : le numérateur est un relatif et le dénominateur un naturel non nul.