

# Makefile, Listes et Dictionnaires

ENSIIE FISA IAP 2019-2020

## 1 Makefiles

### Exercice 1 — *Makefile simple*

On souhaite écrire un code qui calcule un PGCD entre deux entiers positifs. On va utiliser 3 algorithmes différentes pour le même résultat : un code naïf, un algorithme d'Euclide récursif, un algorithme d'Euclide itératif.

On rappelle que deux entiers sont **premiers entre eux** si leur PGCD vaut 1.

1. Rédigez un fichier interface `pgcd.h` qui déclare une fonction `pgcd` prenant en entrée deux entiers et qui renvoie un entier.
2. Rédigez un fichier `main.c` qui contient une fonction `main` doit vérifier ; à l'aide de la fonction `pgcd` déclarée dans `pgcd.h` si deux entiers sont premiers entre eux. Ces deux entiers sont donnés en entrée standard. Si c'est le cas, il affiche un message de succès en console, et sinon, un message d'erreur avec le `pgcd`.

**main ne doit pas contenir de code pour calculer le PGCD, il doit explicitement utiliser la fonction `pgcd` pour cela.**

3. Compilez `main.c` en un fichier `main.o`
4. Rédigez un fichier `pgcdSimple.c` qui contient une implantation de `pgcd` appliquant l'algorithme suivant :

**Entrées:** 2 entiers  $n$  et  $m$ , avec  $n \leq m$ .

**Sorties:** Le PGCD de  $n$  et  $m$

**Pour  $d$  de  $n$  à 1 Faire**

**Si  $d$  divise  $n$  et  $m$  Alors**

**Renvoyer  $d$ .**

5. Compilez `pgcdSimple.c` en un fichier `pgcdSimple.o`
6. Compilez `main.o` et `pgcdSimple.o` en un fichier `main` exécutable. Testez cet exécutable avec quelques entrées standard. Pour rappel, vous pouvez envoyer des éléments en entrée standard ainsi :

```
> machin:~$ main < 12 23
```

7. Rédigez un **Makefile** qui effectue ces compilations pour vous. Supprimez `main.o`, `pgcdSimple.o` et `main` pour tester votre Makefile.
8. Modifiez votre Makefile pour utiliser la variable  $\$(CC)$  à la place de `gcc -Wall -Wextra -std=c99`.
9. Modifiez votre Makefile pour utiliser les mots clefs  $\$ <$ ,  $\$ \wedge$  et  $\$ @$ .
10. Modifiez votre Makefile pour utiliser la règle implicite de conversion des fichiers `.c` en fichiers `.o`.
11. Rédigez un fichier `pgcdRecur.c` qui contient une implantation de `pgcd` appliquant l'algorithme suivant :

**function** EUCLIDREC( $n, m$ )

**Entrées:** 2 entiers  $n$  et  $m$ .

**Sorties:** Le PGCD de  $n$  et  $m$

**Si  $n = m$  Alors**

```

Renvoyer Renvoyer  $n$ 
Si  $m < n$  Alors
     $n, m \leftarrow m, n$ 
Renvoyer EuclidRec( $n, m - n$ )

```

12. Modifiez votre Makefile pour y ajouter la possibilité de compiler `main.c` en utilisant `pgcdRecur.c`. Votre Makefile devra pouvoir permettre de compiler avec la version `pgcdSimple` et avec la version `pgcdRecur`.

On rappelle que `make` applique par défaut la règle `all` mais qu'il est possible d'appliquer une règle spécifique. Par exemple `make main.o` applique la règle qui permet de créer `main.o`.

13. Rédigez un fichier `pgcdIter.c` qui contient une implantation de `pgcd` appliquant l'algorithme suivant :

```

function EUCLIDITER( $n, m$ )
Entrées: 2 entiers  $n$  et  $m$ .
Sorties: Le PGCD de  $n$  et  $m$ 
    Tant que  $n \neq m$  Faire
        Si  $m < n$  Alors
             $n, m \leftarrow m, n$ 
         $m \leftarrow m - n$ 
Renvoyer  $n$ 

```

14. Modifiez votre Makefile pour y ajouter la possibilité de compiler `main.c` en utilisant `pgcdIter.c`. Votre Makefile devra pouvoir permettre de compiler avec les 3 versions du `pgcd`.

### Exercice 2 — Déblayer un peu

Ouvrez l'archive `errors.tar.gz` et corrigez le makefile qui s'y trouve pour pouvoir compiler correctement le fichier `main.c`.

Rappel, vous pouvez utiliser `make -n` pour afficher les lignes de compilations que `make` va essayer d'exécuter, sans les exécuter.

## 2 Listes chaînées

On va s'intéresser aux listes chaînées est un ensemble ordonné d'éléments. Une liste est représentée récursivement comme étant un élément (appelé la tête) suivit d'une liste (appelée la queue ou le reste).

Par exemple la liste `[2,5,1,3]` est constituée :

- d'une tête de valeur 2
- d'une queue de valeur égale à la liste `[5, 1, 3]` ; elle même constituée
  - d'une tête de valeur 5
  - d'une queue de valeur égale à la liste `[1, 3]` ; elle même constituée
    - d'une tête de valeur 1
    - d'une queue de valeur égale à la liste `[3]` ; elle même constituée
      - d'une tête de valeur 3
      - d'une queue de valeur égale à la liste vide

### Exercice 3 — Deux implantations d'une liste chaînée

1. Rédigez un fichier interface `list.h` contenant des déclarations (et donc pas les sources)
  - le type `element`, égal au type `int`
  - le type `list`, abstrait
  - une fonction `empty` qui prend en entrée une `list` et renvoie un entier. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie 1 si la liste est vide et 0 sinon.

- une fonction `head` qui prend en entrée une `list` et renvoie un `element e`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie l'élément `e` en tête de la liste.
  - une fonction `tail` qui prend en entrée une `list` et renvoie une `list`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie la queue de la liste.
  - une fonction `conse` sans argument qui renvoie une `list`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie la liste vide.
  - une fonction `cons` qui prend en entrée un `element e` et une `list` et renvoie une `list l`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie une liste dont l'élément en tête est `e` et dont la queue est `l`.
2. Implantez dans un fichier source `main.c` les fonctions **récur­sives** suivantes à l'aide des fonctions déclarées dans `list.h` :
- une fonction réursive `print_list` qui prend en entrée une `list` et affiche, dans l'ordre, tous les éléments de cette liste.
  - une fonction réursive `size` qui prend en entrée une `list` et renvoie sa taille.
  - une fonction réursive `last` qui prend en entrée une `list` et renvoie son dernier élément.
  - une fonction réursive `add` qui prend en entrée une `list` et un `int x` renvoie une `list` égale à cette liste où, à tous les éléments, on a ajouté `x`.
  - une fonction réursive `concat` qui prend en entrée deux `list l1`, `list l2` et renvoie une `list` égale à la concaténation de `l1` et `l2`.
  - une fonction réursive `reverse` qui prend en entrée une `list` et renvoie une `list` égale à cette liste inversée.
  - une fonction `main` qui servira à tester fonctions.
3. Compilez votre fichier `main.c` en un fichier `main.o` à l'aide d'un Makefile.
4. Rédigez une implantation de `list.h` dans un fichier `list_array.c` où le type `list` est implanté avec un type `int[101]` tel que si une liste est représentée par le tableau `list l` (ou de manière équivalent `int[101] l`), alors
- `l[0]` vaut 1 si la liste est non vide et 0 sinon.
  - si la liste est non vide `l[i]` est le  $i^{\text{e}}$  élément de la liste ; en particulier la tête est `l[1]`.
- . Modifiez le makefile pour compiler ce fichier et pour compiler `main.o` en un fichier exécutable `main`.
5. Rédigez une implantation de `list.h` dans un fichier `list_pointers.c` où le type `list` est implanté avec un type `struct s_list *`, avec `struct s_list {int h ; struct s_list * t }`. Dans ce cas,
- la liste vide est représentée par le pointeur `NULL`.
  - si une liste `l` est non vide alors sa tête est `l->h` et sa queue est `l->t`.
- . Modifiez le makefile pour compiler ce fichier et pour compiler `main.o` en un fichier exécutable `main`.

#### Exercice 4 — Algorithmes de tri de liste chaînée

On reprend l'exercice précédent avec son fichier `list.h` et l'une de ses deux implantations.

Dans un fichier `main.c`, utilisez les fonctions déclarées dans `list.h` pour répondre aux questions suivantes.

1. Codez une fonction réursive qui, connaissant une `list l` *supposée triée* et un entier `x`, renvoie la liste triée contenant tous les éléments de `l` et de `x`.
2. Codez une fonction réursive qui trie une liste avec l'algorithme suivant :

**Entrées:** une liste `l`

**Sorties:** une liste contenant tous les éléments de `l` triée

**Si** `l` est vide **Alors**

**Renvoyer** la liste vide

**Sinon**

$h \leftarrow \text{head}(l)$

$t \leftarrow \text{tail}(l)$

Trier  $t$

Insérer  $h$  dans  $t$

**Renvoyer** la liste résultante.

3. Codez une fonction récursive **fusion** qui, connaissant deux listes **list l1 list l2** supposément triées renvoie une liste triée contenant tous les éléments de  $l1$  et tous les éléments de  $l2$ . *Vous pouvez coder cette fonction avec une complexité  $O(|l1| + |l2|)$ .*
4. Codez une fonction récursive qui trie une liste avec l'algorithme suivant :

**Entrées:** une liste  $l$

**Sorties:** une liste contenant tous les éléments de  $l$  triée

**Si**  $l$  est vide ou  $l$  ne contient qu'un seul élément **Alors**

**Renvoyer** la liste  $l$

**Sinon**

$l1, l2 \leftarrow l$  coupée en deux listes de tailles égales ou dont les tailles diffèrent de 1.

Trier  $l1$  et  $l2$

Fusionner  $l1$  et  $l2$

**Renvoyer** la liste résultante.

### Exercice 5 — *Modifications en place*

On reprend l'exercice précédent avec son fichier **list.h** et l'une de ses deux implantations.

1. Modifiez **list.h** et les deux implantations pour permettre d'utiliser deux nouvelles fonctions :
  - **push** qui prend en entrée un pointeur **list \* l** et un entier **x** et modifie  $l$  pour insérer  $x$  en tête de  $l$ .
  - **pop** qui prend en entrée un pointeur **list \* l**, modifie  $l$  pour supprimer sa tête et la renvoyer.

Dans un fichier **main.c**, utilisez les fonctions déclarées dans **list.h** pour répondre aux questions suivantes.

2. Codez une fonction **clear** qui prend en entrée un pointeur **list \* l** et qui vide la liste.
3. Codez une fonction **insert** qui prend en entrée un pointeur **list \* l**, un entier **int x** et un entier **int index** (entre 0 et la taille de la liste  $|l|$ ) et qui insère  $x$  de sorte qu'il soit en position **index** de la liste. Si **index** n'est pas entre 0 et la taille de la liste  $|l|$  alors la fonction ne fait rien.
4. Codez une fonction **delete** qui prend en entrée un pointeur **list \* l** et un entier **int index** (entre 0 et la taille de la liste  $|l| - 1$ ) et qui supprime l'élément d'indice **index** de la liste. Si **index** n'est pas entre 0 et la taille de la liste  $|l|$  alors la fonction ne fait rien.
5. Codez une fonction **remove** qui prend en entrée un pointeur **list \* l** et un entier **int x** et qui supprime toutes les occurrences de  $x$  de  $l$ .

## 3 Tables d'associations

### 3.1 Code non générique

On aura besoin d'une implantation de liste chaînée de type spécial dans la suite.

#### Exercice 6 — *Préambule*

Dupliquez votre code **list.h** et l'une des deux implantations de sorte à disposer de 3 types de listes chaînées : une liste dont les éléments sont des entiers, une liste dont les éléments sont des chaînes de caractères et une liste dont les éléments sont de type **struct couple {char\* k; int v}**.

## 3.2 Tables d'associations

On va s'intéresser aux tables d'associations ou dictionnaires qui sont des moyens d'associer à des **clefs** des **valeurs**.

Par exemple la table âge associe à des chaînes de caractères un entier qui indiquent respectivement des initiales et l'âge de la personne dont on a noté les initiales : ["DW" -> 4, "CM" -> 18, "JF" -> 25, ...]. On dira alors que les chaînes sont des **clefs** et les entiers des **valeurs** associées à ces clefs. Une chaîne ne peut être associée qu'à un entier. Si on réassocie une chaîne à un entier, alors on remplace l'ancienne association par la nouvelle.

**Exercice 7 — Deux implantations d'une table d'associations** On va utiliser 2

manières pour implanter un dictionnaire. Comme pour l'exemple, on associera des entiers à des chaînes de caractère.

1. Rédigez un fichier interface `dico.h` contenant des déclarations (et donc pas les sources)
  - le type `key` égal à `char*`.
  - le type `value` égal à `int`.
  - le type `dico`, abstrait
  - une fonction `empty` qui prend en entrée un `dico` et renvoie un entier. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie 1 si le dictionnaire est vide et 0 sinon.
  - une fonction `exist` qui prend en entrée un `dico d` et une clef `key c` et renvoie un entier. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie 1 si la clef `c` existe dans `d` et 0 sinon.
  - une fonction `get` qui prend en entrée un `dico d`, une clef `key c` et renvoie un `value`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie la valeur associée à `c` dans `d`.
  - une fonction `put` qui prend en entrée un pointeur `dico* d`, une clef `key c` et une valeur `value x` et ne renvoie rien. Vous pouvez rajouter un commentaire expliquant que cette fonction associe `x` à `c` dans `d`. Si l'association existe déjà, elle est remplacée par cette nouvelle association.
  - une fonction `remove_d` qui prend en entrée un pointeur `dico* d`, une clef `key c`. Vous pouvez rajouter un commentaire expliquant que cette fonction supprime de `d`, si elle existe, l'association dont la clef est `c`. Si l'association existe déjà, elle est remplacée par cette nouvelle association.
  - une fonction `conse_d` sans argument qui renvoie un `dico`. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie un dictionnaire vide.
  - une fonction `keys` qui renvoie une liste chaînée de chaînes de caractères. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie la liste des clefs.
  - une fonction `values` qui renvoie une liste chaînée d'entiers. Vous pouvez rajouter un commentaire expliquant que cette fonction renvoie la liste des valeurs.
2. Codez un fichier source `main.c` qui utilise les fonctions déclarées dans `dico.h` pour implanter les fonctions suivantes :
  - une fonction `print_dico` qui prend en entrée un `dico` et affiche l'ensemble des paires clef-valeurs de ce dictionnaire.
  - une fonction `size` qui prend en entrée un `dico` et renvoie le nombre de clefs qu'il contient.
  - une fonction récursive `add` qui prend en entrée un `dico* d` et un `int x` modifie `d` pour augmenter de `x` toutes les valeurs de `d`.
  - une fonction `concat` qui prend en entrée deux `dico d1`, `dico d2` et renvoie un `dico` égal à la fusion de `d1` et `d2`.
  - une fonction `main` qui servira à tester fonctions.
3. Compilez votre fichier `main.c` en un fichier `main.o` à l'aide d'un Makefile.

4. Rédigez une implantation de `dico.h` dans un fichier `dico_arrays.c` où le type `dico` est implanté avec un type `struct s_dico {int size, char* [100] keys, int[100] values}` tel que si on dispose d'un dictionnaire `dico d` alors

- `d` contient `size` clefs, toutes écrites dans `d.keys` entre les cases 0 et `d - 1`.
- `d.keys[i]` est associé à la valeur `d.values[i]`.

Modifiez le `makefile` pour compiler ce fichier et pour compiler `main.o` en un fichier exécutable `main`.

5. Dans un fichier `dico_hash.c`, écrire une fonction `hash` qui prend en entrée une chaîne de caractères et qui renvoie en sortie la somme des valeurs ASCII des caractères de la chaîne modulo 1024. (On rappelle qu'un caractère en C est aussi un entier égale à sa valeur ASCII.)
6. Rédigez une implantation de `dico.h` où le type `dico` est implanté avec un type `list_couple[1024]` où `list_couple` est à remplacer par le type de listes de couples codé dans le préambule.

Dans ce cas, si `d` est de type `dico` alors

- la liste chaînée `d[i]` contient des couples (clefs, valeur) dont toute clef `c` vérifie `hash(c) = i`.

. Modifiez le `makefile` pour compiler ce fichier et pour compiler `main.o` en un fichier exécutable `main`.