Makefile, Listes et Dictionnaires

ENSIIE FISA IAP 2019-2020

1 Makefiles

Dans cet exercice, on va utiliser des fichier `.h` et des fichiers `.o` pour compiler plusieurs fichiers `.c` en un seul exécutable proprement.

Un fichier `.h` ne contient que des `#include`, `#define`, des types (`typedef et struct`) et des déclarations de fonctions. Il peut aussi contenir des variables globales mais on ne devrait pas en avoir besoin ici. Une fonction déclarée ne contient pas de code, juste la signature de la fonction (`typeretour nom(parametres);`). Par exemple `void f(int x);`. Ces fonctions sont codées dans un fichier `.c`.

Supposons donc qu'on dispose d'un fichier `f.h` qui contient juste la déclaration de la fonction `void f(int x);`. On dispose également d'un fichier `f.c` qui contient le code de la fonction `void f(int x){...}`.

Un fichier `.h` ne contient donc pas de code, juste suffisamment d'information pour pouvoir utiliser des fonctions. Lorsqu'un programme inclue un fichier `.h` (par exemple `#include <math.h>`), il obtient juste ces informations. On ne peut donc pas le compiler en un exécutable, mais juste vérifier qu'il n'y a pas d'erreur de typage. C'est pour ça qu'il faut ajouter `-lm` lorsqu'on compile un fichier qui inclue `math.h`, afin de préciser à `gcc` où est le code de la bibliothèque `math.h`.

Par exemple si j'appelle la fonction `f` mentionnée plus haut avec un `char*` en entrée, je sais qu'il s'agit d'une erreur. Par contre si je n'ai pas le code de la fonction `f` je ne peux pas finir la compilation. On peut donc faire une première partie de la compilation (vérification des erreurs) mais pas la fin (création de l'exécutable).

Supposons donc que j'ai un fichier `test.c` qui inclue `f.h` et qui utilise `f` correctement. Je peux précompiler mon fichier `test.c` avec l'option`-c` de `gcc` et j'aurais une vérification syntaxique de mon programme `test.c`. Le résultat est un fichier `objet` nommé `test.o`.

Le fichier `test.o` n'est pas un exécutable. Pour pouvoir finir la compilation, j'ai besoin du code de `f.c`. Je peux également compiler mon fichier `f.c` en un fichier `f.o` avec la même technique. Je peux enfin compiler `test.o` et `f.o` en un seul exécutable avec la commande `gcc ... test.o f.o -o test` qui me produit l'exécutable `test`. L'intérêt est de pouvoir compiler du code écrit par des personnes différentes dans des fichiers différents. Notez par exemple que la personne qui a codé `f.c` l'a peut être fait 2 semaines après que `test.c` soit codé. Car tout ce dont on a besoin pour coder `test.c` c'est `f.h`. Par contre on ne peut pas tester le résutlat tant que `f.c` n'est pas codé.

Pour finir, si on commence à avoir beaucoup de fichiers, exécuter de nombreux `gcc` en cascade devient difficile à faire à la main. Une bonne pratique consiste à construire un `makefile`. Sans rentrer dans les détails, un `makefile` vous permets simplement de définir un script de compilation de vos fichiers.

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. Cette fonction ne doit pas être codée, juste déclarée.
2. Rédigez un fichier `main.c` qui contient une fonction main doit vérifier ; à l'aide de la fonction `pcgd` 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 fonction `pgcd` ou de code pour calculer le PGCD, il doit explicitement utiliser la fonction `pgcd` de `pgcd.h` pour cela.

3. Compilez `main.c` en un fichier `main.o` avec la commande `gcc -Wall -Wextra -std=c99 -c main.c -o main.o`
4. Rédigez un fichier `pcgdSimple.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.
   Renvoyer -1
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. Le makefile suivant devrait suffire. (Attention, il faut mettre un TAB avant le `gcc` et non une série d'espaces).

```
all : main

main.o : main.c
	gcc -Wall -Wextra -std=c99 -c main.c -o main.o


pgcdSimple.o : pgcdSimple.c
	gcc -Wall -Wextra -std=c99 -c pgcdSimple.c -o pgcdSimple.o

main : main.o pgcdSimple.o
	gcc -Wall -Wextra -std=c99 main.o pgcdSimple.o -o main
```

8. Cherchez en ligne comment modifier votre Makefile pour utiliser la variable $(CC) à la place de `gcc -Wall -Wextra -std=c99`.
9. Cherchez en ligne comment modifier votre Makefile pour utiliser les mots clefs $ <, $∧ et $@.
10. Cherchez en ligne comment modifier votre Makefile pour utiliser la règle implicite de conversion des fichiers `.c` en fichiers `.o` (notations `%.o : %.c`).
11. Rédigez un fichier `pcgdRecur.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`.

    Remarque : `make` applique par défaut la première régle, souvent nommée `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 `pcgdIter.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'annexe `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 ; avec `typedef struct s_list* list;` Le type `struct s_list` sera définit plus tard. On en a pas besoin pour le moment.
   - 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 récursive `print_list` qui prend en entrée une `list` et affiche, dans l'ordre, tous les éléments de cette liste.
   — une fonction récursive `size` qui prend en entrée une `list` et renvoie sa taille.
   — une fonction récursive `last` qui prend en entrée une `list` et renvoie son dernier élément.
   — une fonction récursive `add` qui prend en entrée une `list` et un `int x` et renvoie une `list` égale à cette liste où, à tous les éléments, on a ajouté `x`.
   — une fonction récursive `concat` qui prend en entrée deux `list l1, list l2` et renvoie une `list` égale à la concaténation de `l1` et `l2`.
   — une fonction récursive `reverse` qui prend en entrée une `list` et renvoie une `list` égale à cette liste inversée.
   — une fonction `main` qui servira à tester les 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`. Dans cette version, le type `struct s_list` est une structure contenant un champ `struct s_list {int[101] t}` tel que, dans une liste `list l`
   — `l->t[0]` vaut la taille de la liste.
   — si la liste est non vide `l->t[i]` est le i^{e} élément de la liste ; en particulier la tête est `l->t[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 récursive 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 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 **Alors**
   Renvoyer la liste vide
   Sinon
   $h \leftarrow$ `head`(l)

$t \leftarrow$ `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 insert `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`, ecrire 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`.