IN103 : Sujet du TP4

Groupe ESIEE, Denis BUREAU, avril 2006.

Attention ! Le sujet peut être modifié jusqu'à la veille du TP.

1 Les objectifs

Être capable 1. de réaliser des programmes en C qui utilisent des fichiers de texte et 2. des programmes composés de modules compilés séparément, 3. d'automatiser la construction de ces programmes, 4. d'appeler du C depuis Java et d'échanger des données, et 5. de prendre conscience de l'importance de l'algorithmique.

2 Notions vues en cours

- Fichiers de texte en C (7 étapes)
- Compilation séparée, includes, pré-processeur
- `make`, `makefile`

3 Compléments sur la page web de l'unité

- Fichiers de textes en C
- Commande de compilation séparée
- Problèmes avec les includes
- Commande d'édition de liens
- `makefile` typique en C
- Appel de C en java
- `makefile` typique Java$<->$C
- Échanges de données Java$<->$C

4 Fichiers de texte en C

4.1 Création et écriture

1. Écrire un programme C (`creefich.c`) contenant juste une fonction `main()` affichant les 20 premiers entiers strictement positifs, à raison d'un par ligne. Compiler, tester.
2. En suivant toutes les étapes décrites en cours ou en suivant le lien "Fichiers de textes en C" ci-dessus, modifier le programme ci-dessus pour qu'au lieu d'afficher les entiers à l'écran, il les écrive dans un fichier `entiers.txt`, créé par votre programme.
3. Compiler, lancer le programme, et vérifier avec la commande `cat` le contenu du fichier `entiers.txt`.
4. Modifier ce programme pour qu'il demande à l'utilisateur le nom du fichier qu'il veut créer. Retester le programme comme précédemment.

4.2 Lecture et affichage

1. En suivant toutes les étapes décrites en cours ou en suivant le lien "Fichiers de textes en C" ci-dessus, écrire un autre programme C (`litfich.c`) contenant juste une fonction `main()` lisant par `fscanf()` les entiers (en nombre inconnu) dans un fichier de texte dont le nom est demandé à l'utilisateur et les affichant à l'écran, tous sur la même ligne, séparés par des virgules.
2. Compiler, lancer le programme, et vérifier l'affichage.

4.3 Formats de lecture (plus tard, en travail personnel)

1. Sauvegarder les 2 lignes suivantes dans le fichier `test.txt` (les 2 retours à la ligne doivent être présents) et lancer `litfich` sur ce fichier :
   ```
   123  456
   +789
   ```
2. Dans `litfich.c`, copier et coller en 4 exemplaires toute la partie lecture/affichage du fichier de texte (de l'ouverture à la fermeture).
3. Modifier le premier exemplaire pour qu'il lise le fichier caractère par caractère en remplaçant les `fscanf()` par des `fgetc()` et en affichant les caractères séparés par des virgules.
4. Modifier le deuxième exemplaire pour qu'il lise le fichier mot par mot en remplaçant le format des `fscanf()` par des `%s`, en réservant les tableaux de caractères nécessaires, et en affichant les mots séparés par des virgules.
5. Modifier le troisième exemplaire pour qu'il lise le fichier ligne par ligne en remplaçant les `fscanf()` par des `fgets()`, en réservant les tableaux de caractères nécessaires, et en affichant les lignes séparés par des virgules.
6. Compiler, lancer le programme sur `test.txt`, regarder les affichages, et comprendre pourquoi ces différents affichages.

5 Compilation séparée

1. Reprendre l'exercice 6 du TP précédent (Rationnel avec pointeurs) pour le découper en modules, mais s'assurer d'abord qu'il compile sans aucun warning et qu'il fonctionne bien. `main.c` contiendra évidemment la fonction `main()`, `e-s.c` contiendra les sous-programmes `affiche()` et `saisit()`, et `operations.c` contiendra toutes les autres fonctions. **Dans un premier temps**, aucun fichier `.h` ne sera créé (copier/coller le minimum nécessaire).
   Compiler séparément, faire l'édition de liens, tester.
2. **Dans un deuxième temps**, modifier les 3 modules pour qu'ils incluent tous un fichier `"declar.h"` incluant `<stdio.h>` & `<stdlib.h>` et contenant la définition des types et TOUS les prototypes.
   Compiler séparément, faire l'édition de liens, tester.
3. **Dans un troisième temps**, décomposer puis oublier `declar.h` : créer un fichier `types.h` contenant la définition des types, un fichier `e-s.h` contenant uniquement les prototypes des sous-programmes de `e-s.c` et incluant évidemment `types.h`, un fichier `operations.h` contenant uniquement les prototypes des fonctions de `operations.c` et incluant évidemment `types.h`, et un fichier `main.h` incluant les 3 `.h` précédents. Modifier les 3 fichiers `.c` pour qu'ils n'incluent plus que le strict nécessaire.
   Compiler séparément (résoudre le problème d'inclusion multiple), faire l'édition de liens, tester.

6 Automatisation

1. Dans le répertoire de l'exercice précédent, ne conserver que les fichiers `.c` et les fichiers `.h` (détruire les autres fichiers).
2. Récupérer le fichier `makefile` en suivant le lien "`makefile` typique en C" section 3 ci-dessus (par téléchargement, pas par copier/coller), et le modifier pour l'adapter à l'exercice précédent.
3. Tester la commande `make`, puis lancer le programme. Taper `make` une seconde fois pour voir le message.
4. Modifier le fichier `types.h` OU BIEN simuler une modification de ce fichier par la commande `touch types.h` qui change la date de dernière modification du fichier sans vraiment le modifier. `make`: vérifier à l'écran que tous les modules sont recompilés.
5. Modifier le fichier `main.c` (ou `touch`); `make`: vérifier à l'écran que seul ce fichier est recompilé.

7 Appel Java $->$ C

1. En vous reportant fréquemment aux documents des 3 derniers liens de la section 3 ci-dessus, faire fonctionner l'exemple décrit dans le lien "Appel de C en java", en exécutant toutes les commandes (en jaune) des paragraphes 1.A à 1.E ; les fichiers Java et C sont fournis au 1.F. Cela donne-t-il bien les résultats attendus ?

2. Dans un autre répertoire et en suivant les 5 mêmes étapes qu'à l'exercice précédent, réaliser un programme démontrant la possibilité d'utiliser en Java les formatages fournis par C (voir exemple d'exécution au 7.6 ci-dessous). Commencer par compléter la classe Java ci-dessous :

```
// MyFormat.java
public class MyFormat
{ // declaration des 3 sous-programmes C appeles ci-dessous
  // (lengthInt, formatInt, et formatInteger)
  // qui seront a realiser en C dans le fichier MyFormat.c

  static { System.loadLibrary( "MyFormat" ); } // chargera libMyFormat.so
  public static void main( String [] args )
  { int len;   String s;
    len = lengthInt( "[%5d]", 12 ); // nb de caracteres de la String a afficher
    System.out.println( "aff1=" + len );
    len = lengthInt( "[%04d]", 12 );
    System.out.println( "aff2=" + len );
    s = formatInt( "[%5d]", 12 ); // String formatee d'un int
    System.out.println( "aff3=" + s );
    s = formatInt( "[%04d]", 12 );
    System.out.println( "aff4=" + s );
    s = formatInteger( "[%5d]", new Integer(12) ); // String formatee d'un Integer
    System.out.println( "aff5=" + s );
    s = formatInteger( "[%04d]", new Integer(12) );
    System.out.println( "aff6=" + s );
  } // main()
} // MyFormat
```

3. Compiler la classe (jusqu'à ce qu'il n'y ait plus d'erreur) et générer le `.h` .
   Pour simplifier, il est possible de ne s'occuper que de `lengthInt()` dans un premier temps.

4. Reprendre le processus utilisé au 7.1 ci-dessus, et lire le dernier lien section 3 ci-dessus.

5. Écrire le fichier `MyFormat.c` avec les contraintes suivantes :
   - Utiliser `sprintf()` pour les 3 fonctions ; taper `man sprintf` pour obtenir de l'aide ou consulter le poly papier ou en ligne.
   - Penser à convertir une `jstring` en `const char *` avant de l'utiliser, et à rendre la mémoire après.
   - Penser à convertir une `const char *` en `jstring` avant de la retourner vers Java.
   - Allouer dynamiquement toute chaîne de caractères C et libérer la mémoire après.
   - Accéder à la valeur de l'`Integer` à travers sa méthode `intValue()`.

6. Exécuter le programme. L'affichage devrait être :

```
aff1=7
aff2=6
aff3=[   12]
aff4=[0012]
aff5=[   12]
aff6=[0012]
```

8 Un peu d'algorithmique en C

1. Écrire une fonction `inclus1()` capable de vérifier si un entier est ou non présent dans un tableau d'entiers supposé trié et sans doublons qu'on appellera "ensemble d'entiers".
   Pour cela, définir le type tableau d'entiers comme un pointeur d'entiers pour pouvoir l'allouer dynamiquement, et passer en paramètres d'entrée l'entier recherché, l'ensemble d'entiers, et le nombre d'éléments de cet ensemble. On définira également le type booléen pour pouvoir retourner un résultat vrai ou faux.
2. Écrire un `main()` de test qui créera un ensemble d'entiers par exemple par l'instruction
   `int e1[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };`
   qui saisira une valeur au clavier, qui appellera `inclus1()` sur cette valeur, et qui affichera le résultat en français.
3. Ajouter à ce programme une fonction `inclus()` capable de vérifier si un ensemble d'entiers est inclus dans un autre. Elle appellera "naturellement" la fonction `inclus1()`.
4. Modifier le `main()` pour tester cette nouvelle fonction : remplacer la saisie d'un entier par la création d'un second ensemble d'entiers par exemple par l'instruction
   `int e2[] = { 4, 6, 14, 16, 18};`
   Remplacer aussi l'appel d'`inclus1()` par le bon appel d'`inclus()`.
5. On désire maintenant pouvoir tester la fonction `inclus()` sur des ensembles de taille plus conséquente, mais sans avoir à inclure dans le `.c` ni à saisir au clavier des milliers de valeurs. Il faut donc lire les valeurs à partir d'un fichier de texte. Pour cela, nous allons écrire 2 nouvelles fonctions `openR()` et `lire()`.
6. `openR()` prendra en paramètres une chaîne de caractères contenant le nom du fichier et retournera la variable fichier obtenue après ouverture en lecture et test d'ouverture.
7. `lire()` prendra en paramètre d'entrée la variable fichier, en paramètre de sortie l'ensemble d'entiers (puisque le tableau sera alloué dynamiquement), et retournera le nombre d'éléments.
   Pour pouvoir allouer le tableau, il faut connaître le nombre d'éléments, mais pour cela, il faut d'abord lire une première fois tout le fichier en comptant les éléments, puis le lire une seconde fois pour stocker chaque élément dans le tableau ... sauf si on utilise certaines fonctions de `<stdio.h>` que vous pouvez découvrir dans la documentation.
   Malgré tout pour vous éviter d'avoir à fermer et rouvrir le fichier, voici l'instruction qui permet de revenir au début après la première lecture : `rewind( vf );` où `vf` est la variable fichier.
8. Modifier le `main()` pour tester ces nouvelles possibilités : remplacer l'initialisation des 2 tableaux par 2 appels à `openR()`, 2 appels à `lire()`, puis 2 fermetures.
   D'autre part, utilisez 2 arguments de la ligne de commande pour récupérer le nom des 2 fichiers à lire.
9. Tester votre programme sur les 2 ensembles stockés dans les fichiers `E1.dat` et `E2.dat` .
   `inclus E1.dat E2.dat` est très rapide, mais `inclus E2.dat E1.dat` l'est beaucoup moins, non ?
10. Essayer maintenant le programme `inclusMfN` dans ces 2 mêmes cas. La différence sur le nombre de comparaisons effectuées explique facilement la différence de temps d'exécution.
    Il est facile de rendre le second cas encore plus rapide en n'effectuant toujours qu'une seule comparaison ; voyez-vous comment ?
    Par contre, il est beaucoup plus difficile de rendre le premier cas significativement plus rapide. Pourtant, si vous essayer le programme `inclusMpN` vous constaterez que c'est possible.
11. `inclusMfN` et votre programme effectuent de l'ordre de $m \times n$ comparaisons alors que `inclusMpN` n'effectue que $m + n$ comparaisons. Lorsque ces valeurs sont élevées, les différences deviennent significatives.
    Ce n'est pas en cherchant quelques "optimisations" du programme que l'on peut gagner autant de temps, mais en changeant radicalement l'agorithme. Ici, il faudra supprimer la fonction `inclus1()` et inventer un nouveau traitement "global" dans la fonction `inclus()` ; voyez-vous comment ?