IN103 : Sujet du TP2

Groupe ESIEE, Denis BUREAU, mars 2006.

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

1 Les objectifs

Être capable de réaliser des programmes en C définissant de nouveaux types et savoir utiliser le debugger.
Rappel : De nombreux liens sur C sont disponibles sur la page web de l'unité IN103.

Pour chaque exercice ci-après (sections 2, 3, 5, 6):

1. Créer un fichier `.c` différent pour chaque exercice ; il contiendra en plus du sous-programme demandé une fonction `main()` dont le schéma sera toujours le même :
   affichage d'un message, saisie de valeurs, appel de la fonction, affichage du résultat ;
   ou si c'est une procédure, affichage d'un message, saisie de valeurs, appel de la procédure.

2. Tester ensuite votre programme selon la procédure exposée sur la page web "Test des exercices", accessible à partir de la page web de l'unité à la fin du **paragraphe TP2**.

2 Exercice du TP2 : `rationnel.c`

1. Écrire la définition d'un type `Rationnel` composé de 2 entiers `num` et `den` .
2. Écrire deux fonctions de "construction" d'un rationnel (qui retournent un `Rationnel`), `rationnel2()` avec 2 paramètres d'entrée, et `rationnel1()` avec un seul paramètre (lorsque le dénominateur vaut 1). Chacune de ces 2 fonctions doit être utilisée à chaque fois que c'est possible dans la suite (notamment, l'une appelle l'autre).
3. Écrire une procédure `affiche` qui affiche le rationnel passé en paramètre sous la forme habituelle num/den .
4. Écrire un `main()` pour tester ce qui est déjà écrit (sans saisie ⇒ imposer les valeurs).
5. Écrire une fonction `saisit` (sans paramètre) qui permet de saisir un `Rationnel` sous la forme habituelle num/den , et qui retourne ce `Rationnel`.
6. Écrire une fonction `ajoute` qui retourne un rationnel égal à la somme des 2 rationnels passés en paramètres. La simplification du résultat n'est pas demandée.
7. Compléter le `main()` pour tester les 2 derniers sous-programmes.
   Exemple d'affichage souhaité; : `1/3 + 1/6 = 9/18`

3 Ligne de commande : `arguments.c`

En C comme en Java, il est possible de récupérer les arguments de la ligne de commande. L'entête du main doit alors s'écrire :
`int main( int argc, char* argv[] )` où `argc` pourrait être remplacé par `argumentsCount` ou `nombreDArguments`, et `argv` par `argumentsValues` ou `tableauDesArguments` ; pour le compilateur, `char*` signifie "chaîne de caractères" (explications lors du prochain cours).
Attention ! `argv[0]` est la chaîne de caractères contenant le nom du programme ; le premier argument commence donc à l'indice 1.
- Écrire un programme qui affiche tous les arguments sous la forme :

```
nomDuProgramme
1: argument1
.: ...
N: argumentN
```

4 Prise en mains du debugger

4.1 La compilation

Pour que le debugger puisse vous fournir un maximum d'aide, il faut ajouter à la commande habituelle de compilation l'option `-ggdb3` .
La commande d'édition de liens n'est pas modifiée.

4.2 L'exécution

Pour lancer le programme, taper `gdb hello` (si le programme s'appelle `hello`).

4.3 Les commandes de `gdb`

Toutes les commandes peuvent être abrégées tant qu'il n'y a pas d'ambiguité.

- `break fon` : met un point d'arrêt au début de la fonction `fon`
- `break num` : met un point d'arrêt à la ligne numéro `num`
- `set args a1 a2` : pour spécifier `a1` et `a2` comme arguments de la ligne de commande
- `run` : lance ou relance l'exécution du programme avec les mêmes arguments que précédemment
- `list` : affiche les instructions C autour du point d'arrêt
- `print e` : affiche la valeur de l'expression `e`
- `ptype e` : affiche la déclaration du type de l'expression `e`
- `watch e` : demande à être prévenu à chaque modification de la valeur de l'expression `e`
- `next` : exécute le programme jusqu'à la ligne de source suivante
- `step` : comme `next` mais entre à l'intérieur du sous-programme lorsqu'il y a un appel de sous-programme
- `continue` : continue l'exécution du programme jusqu'au prochain point d'arrêt
- `help` : pour avoir de l'aide sur ces commandes et découvrir toutes les autres
- `quit` : pour sortir du debugger

Pour plus de détails, voir le manuel de référence en ligne à partir de la page web de l'unité.

4.4 Mise en pratique

Essayer le debugger sur le programme `arguments`.
Ne pas hésiter à l'utiliser par la suite à chaque fois que la cause d'une erreur à l'exécution ou d'un résultat faux ne vous apparaît pas clairement.

5 Exercice du TD6 : `minimum.c`

- Définir un type tableau de `MAX` réels (`MAX` peut valoir 10 par un `#define`)
- Écrire une fonction `minimum` qui prend en paramètre un tableau de réels et le nombre d'éléments utiles, et qui retourne la plus petite valeur présente dans le tableau.
- Écrire un `main()` pour pouvoir tester la fonction (imposer les valeurs à la création d'un tableau).

6 Exercices du TP6

1. `palindrome.c`
   - Écrire une fonction `estPalindrome2` prenant en entrée une chaîne de caractères et retournant une valeur booléenne suivant que cette chaîne est un palindrome ou non (donc 2 nouveaux types à définir !)
   Rappel : un palindrome est un mot ou une phrase se lisant de la même manière de gauche à droite et de droite à gauche.
   Exemples : "Radar", "Esope reste ici et se repose."
   On peut donc constater d'après ces exemples que les minuscules et les majuscules sont équivalentes, et que seuls les caractères alphabétiques sont pris en compte.

   - Écrire un `main()` permettant de tester cette fonction.

   Aide : `ctype.h` contient la déclaration d'une fonction `isalpha(c)` qui retourne vrai si `c` est une lettre (minuscule ou majuscule), et faux sinon, ainsi que la déclaration d'une fonction `tolower(c)` qui retourne le caractère `c` en minuscule si c'était une majuscule, ou le caractère `c` inchangé sinon.
   Pour lire une phrase comportant des espaces, il faut taper : `fgets( tc, MAXCH, stdin );`
   si `tc` a été déclaré comme un tableau de `MAXCH+1` caractères.

2. `matrice.c`
   - Définir un type `Matrice` comme un tableau de 20 x 5 réels.
   - Écrire une procédure `initMat` qui initialise la matrice passée en paramètre comme suit : la $3^{ème}$ colonne contiendra les nombres de 1 à 20 ; les colonnes 1, 2, 4, et 5 contiendront respectivement la racine cubique, la racine carrée, le carré et le cube du nombre de la $3^{ème}$ colonne.
   Remarque : une seule boucle est nécessaire.
   Attention ! En C, un paramètre tableau est automatiquement considéré comme paramètre de sortie ; cela signifie que modifier des valeurs dans le tableau de la procédure modifie en fait aussi le tableau du programme principal.
   - Écrire une procédure `affMat` d'affichage de la matrice sur 20 lignes de 5 colonnes, avec seulement 3 chiffres après la virgule pour les racines.
   - Écrire le `main()` permettant de tester tout ça.

3. `eratosthene.c`
   Le crible d'Ératosthène est une méthode pour découvrir les nombres premiers. Il consiste à mettre tous les entiers (inférieurs ou égaux à une certaine `LIMITE`) dans une grille, puis à rayer méthodiquement tous les multiples des nombres non encore rayés.

   Exemple :

```
debut:  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
raye2:  2  3  /  5  /  7  /  9  / 11  / 13  / 15  / 17  / 19  / 21  / 23  / 25
raye3:  2  3     5     7     /    11    13     /    17    19     /    23    25
raye5:  2  3     5     7          11    13          17    19          23     /
raye7:  ...
```

   On constate alors qu'il ne reste plus que des nombres premiers dans la grille.
   En informatique, il est toutefois difficile de "rayer" un nombre. On choisit donc de déclarer un tableau de booléens dont chaque case représentera le nombre entier correspondant à son indice : si la valeur de la case est `true`, c'est que l'entier correspondant n'est pas rayé. Et rayer un nombre signifiera donc passer la case correspondante à `false`.

   - Écrire une procédure `initV` qui initialise correctement les éléments du tableau passé en paramètre, pour signifier qu'a priori, tous les entiers à partir de 2 sont candidats à être premiers. La `LIMITE` est également passée en paramètre.

   - Écrire une procédure `raye` qui applique la méthode décrite ci-dessus en mettant `false` dans le tableau passé en paramètre pour chaque case correspondant à un nombre qu'elle doit "rayer" (inférieur ou égal à la `LIMITE` également passée en paramètre). .../...

Remarque : il est inutile de parcourir TOUS les nombres jusqu'à la `LIMITE`.
Question : quand peut-on arrêter la boucle principale ?

- Écrire une procédure `prepare` qui applique la méthode du crible d'Ératosthène au tableau de booléens passé en paramètre, en appelant les procédures `initV` et `raye`. La `LIMITE` est également passée en paramètre.

- Écrire une procédure `affiche` qui affiche l'intégralité des nombres premiers figurant dans le tableau passé en paramètre (jusqu'à la `LIMITE` également passée en paramètre), à raison de 10 par ligne.

- Écrire un `main` qui déclare puis "prépare" un tableau de booléens, demande quelle action on veut faire (voir ci-dessous), puis effectue l'action, ceci tant qu'on ne désire pas arrêter le programme.
Pour choisir l'action, on saisira un nombre de 0 à 3 : 0 pour sortir de la boucle et arrêter le programme, 1 pour appeler la procédure `affiche` (saisir la limite), 2 pour tester si un nombre est premier ou non (saisir ce nombre), 3 pour trouver le plus grand nombre premier inférieur ou égal à un nombre (saisir ce nombre). On supposera que l'utilisateur n'entre que des nombres entiers.

Exemple d'affichage (option 1) :

```
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229
...
```

4. `anagrammes.c`
Le but est de réaliser une fonction permettant de détecter si deux mots (ou deux phrases) sont des anagrammes ou non.
Rappel : Deux mots sont des anagrammes si et seulement s'ils s'écrivent avec exactement les mêmes lettres, chacune en même quantité.
Exemples : `Saintes` et `Tisanes` sont des anagrammes.
`tartines` et `satiner` ne sont pas des anagrammes.
`"Mer : Danger !"` et `"grand-mere"` sont des anagrammes.
Remarque : Les minuscules sont considérées identiques aux majuscules, et les caractères non alphabétiques sont ignorés dans la comparaison.

La méthode consiste, pour chaque phrase, à stocker dans un tableau le nombre de fois qu'apparaît chaque lettre de l'alphabet. Il suffit ensuite de comparer les deux tableaux : s'ils sont égaux, les deux phrases sont des anagrammes !

- Écrire la procédure `init0` qui initialise à 0 chaque case du tableau de comptage passé en paramètre (le rôle d'un tableau de comptage est de stocker pour chaque lettre de l'alphabet le nombre de fois qu'elle apparaît dans la phrase).

- Écrire la procédure `compte` qui prend une chaîne de caractères en 1^{er} paramètre et remplit le tableau de comptage passé en $2^{ème}$ paramètre.
Contrainte : tout `switch` est interdit.
Rappel : des méthodes utiles vous ont été indiquées à l'exercice sur les palindrômes.
Aide : $\forall x,\ x - x = 0$

- Écrire la fonction booléenne `anagramme` qui créera les deux tableaux de comptage (un pour chaque phrase) et qui appellera pour chacun d'eux la procédure `init0` pour initialiser ce tableau, puis la procédure `compte` pour compter dans ce tableau les occurences [1] de chaque lettre de l'alphabet dans la chaîne ; enfin, elle comparera les deux tableaux de comptage.

- Écrire le `main()` permettant de tester tout ça.

[1] apparitions