TP 6 - les fonctions

Exercice 1

1) Écrivez une fonction `float fcalcul(int v1, int v2)` qui délivre l'hypoténuse d'un triangle rectangle de côtés v1 et v2.

Remarques :

- hypothénuse = $\sqrt{v1^2 + v2^2}$
- la fonction `sqrtf(a)` délivre la racine carrée de `a` sous forme d'un réel flottant. Pensez à inclure `<math.h>` au début de votre programme.

2) Écrivez un main qui demande à l'utilisateur de saisir deux valeurs entières et qui affiche le résultat de la fonction `fcalcul` correspondant. Testez votre programme avec plusieurs couples de valeurs.

Exercice 2

1°) Écrivez une fonction *factorielle* qui prend en paramètre d'entrée un nombre entier naturel, et qui délivre sa factorielle (*cf* TP3 d'AP1).

2) Écrivez un `main` qui demande à l'utilisateur de saisir un entier et qui affiche la factorielle de ce nombre. Vous pourrez tester votre programme avec les valeurs suivantes :

0 ! = 1
1 ! = 1
2 ! = 2
3 ! = 6
7 ! = 5040

Exercice 3 : tester une fonction

Tester une fonction c'est vérifier que le résultat délivré par la fonction correspond toujours au résultat souhaité. Comme il est impossible de tester tous les cas, on teste en général la fonction avec plusieurs exemples de valeur judicieusement choisies.
Mais tester la fonction en saisissant des valeurs (comme demandé dans les exercices précédents) peut s'avérer très fastidieux : le nombre de tests, donc de saisies, peut être important et en plus il faut saisir ces valeurs à chaque exécution.

Première solution ; rediriger le flux d'entrée

Une première solution consiste à préparer à l'avance, dans un fichier, les valeurs à saisir et à l'exécution utiliser le principe de redirection de linux. Par exemple si le fichier `valeurs.txt` contient les valeurs de test, on lance l'exécution par : `./exo < valeurs.txt`

À l'exécution, les instructions scanf "puiseront" les valeurs dans le fichier d'entrée indiquée (ici `valeurs.txt`) au lieu de les "puiser" dans le buffer d'entrée standard (alimenté par le clavier).

Pour la fonction de l'exercice 1, créez un fichier `test1.txt` qui contient les valeurs (une valeur par ligne) :

```
3
4
```

Exécutez votre programme avec ce fichier comme flux d'entrée : `./exo < test1.txt`

Deuxième solution : écrire une fonction de test

Une deuxième solution consiste à écrire (une fois pour toutes) tous les tests que l'on veut faire subir à notre fonction. Le principe est, pour chaque test, d'afficher sur la même ligne le résultat attendu et le résultat fourni par la fonction. À l'exécution il suffira de comparer, pour chaque ligne, ces deux résultats.

Pour la fonction de l'exercice 1, cela peut prendre la forme d'une procédure `test()` :

```
void test(){
    printf("valeur attendue : %.3f     valeur obtenue : %.3f\n",5.0,fcalcul(3,4));
    printf("valeur attendue : %.3f     valeur obtenue : %.3f\n",1.414,fcalcul(1,1));
    printf("valeur attendue : %.3f     valeur obtenue : %.3f\n",0.0,fcalcul(0,0));
    printf("valeur attendue : %.3f     valeur obtenue : %.3f\n",3.606,fcalcul(2,3));
    printf("valeur attendue : %.3f     valeur obtenue : %.3f\n",10.630,fcalcul(7,8));
}
```

1) Ajoutez cette procédure à votre premier programme et exécutez-la.

L'avantage ici est qu'en cas d'échec, c'est-à-dire lorsqu'une ligne n'a pas deux résultats identiques, on peut retrouver rapidement le cas à l'origine du problème. L'inconvénient c'est que la liste des résultats affichés peut être fastidieuse à analyser.

2) Reprenez maintenant l'exercice 2 et proposez une procédure de test basée sur ce principe.

Exercice 4 : calcul d'une expression réelle

Question 1 : Écrivez et testez 4 fonctions permettant de faire la somme, la division, la soustraction et la multiplication de deux nombres réels flottants.

Pour la division, on admet que c'est l'appelant qui doit vérifier que son paramètre effectif correspondant au diviseur est non nul.

Question 2 : Exploitez les fonctions définies précédemment pour calculer en une seule instruction l'expression arithmétique suivante :

((4.2 + (5.3 * (((4.5 + 1.3) / 2.0) + 1.1))) - 1.0)

Pour tester votre programme, affichez le résultat attendu (que vous aurez calculé vous-même) et le résultat obtenu.

Exercice 5

Décrivez en une phrase ce que fait cette fonction.

```
#include <stdlib.h>
#include <time.h>
int aleaInfBorne(int borne){
      srand(time(NULL));
      return rand() % borne;
}
```

Consultez le manuel unix pour connaître l'utilisation de *srand* et *rand (man 3 rand* et *man 3 srand).*

Aide :

```
#include <time.h>
#include <stdlib.h>

srand(time(NULL));  //initialisation du générateur à partir de
                    // l'heure machine
int nbAleatoire = rand();  //récupération d'un nombre pseudo-aléatoire
```

Essayez cette fonction `aleaInfBorne` avec un *main* qui y fait appel.

Exercice 6 : *jeu de devinette*

Spécification du problème :

Un joueur tente de deviner un nombre généré pseudo-aléatoirement par l'ordinateur. Le programme à réaliser doit donc générer ce nombre secret et ensuite proposer au joueur de le deviner en moins de dix essais (10 est un exemple).

À chaque essai, le programme indique au joueur si le nombre proposé est égal, inférieur ou supérieur au nombre secret à deviner.

Méthode :

1) Proposez et testez dans un *main()* une fonction *generer(...)* qui « retourne » un entier généré de manière pseudo-aléatoire compris entre 0 et une borne supérieure passée en paramètre d'entrée. Cette fonction fera elle-même appel à la procédure *srand* pour initialiser le générateur de nombres pseudo-aléatoires et à la fonction *rand* pour l'appeler (cf exercice précédent).

2) Proposez et testez dans un *main()* une fonction *comparer(...)* qui prend en paramètre d'entrée deux nombres entiers et retourne le résultat de la comparaison de ces nombres. Le résultat vaut 0 si les deux nombres sont égaux, 1 si le premier nombre est supérieur au deuxième et -1 si le premier nombre est inférieur au deuxième.

3) Proposez et testez dans un *main()* la fonction *jeu (....)* qui permet à un joueur de faire ses essais et qui affiche les indications à chaque essai (nombre supérieur, inférieur, ou égal au nombre secret. La fonction prend en paramètre d'entrée le nombre secret et retourne le nombre d'essais qui ont été nécessaires pour que l'utilisateur découvre ce nombre secret.

4) Proposez et testez dans un *main()* une procédure afficherResultat(...) qui affiche le résultat (« perdu » ou « gagné en *n* coups ») en fonction du nombre *n* d'essais effectué, passé en paramètre d'entrée.

5) En utilisant les procédures et fonctions définies précédemment, écrivez et testez le programme qui permet de jouer à ce jeu.

Exercice 7 : Jeu Pierre-Feuille-Ciseau

On veut écrire un programme permettant de jouer à « pierre-feuille-ciseau » contre l'ordinateur.

Question 1 : Écrivez et testez une fonction prenant en paramètre deux actions : l'action1 et l'action 2. Ces actions sont matérialisées par des caractères 'P' pour pierre, 'F' pour feuille et 'C' pour ciseau.

La fonction retourne le caractère 'G' si l'action 1 est gagnante par rapport à l'action 2, 'P' si l'action 1 est perdante par rapport à l'action 2, et 'N' si le match est nul (actions identiques).

Question 2 : Écrivez et testez une fonction qui retourne l'action proposée par l'ordinateur ('P', 'C' ou 'F'). Pour cela, il faut utiliser le générateur pseudo-aléatoire et donc faire appel à la fonction de l'exercice 4.

L'action de l'ordinateur est obtenue à partir de l'entier pseudo-aléatoire compris entre 0 et 2 :

0 -> 'P', 1 -> 'F', 2 -> 'C'.

Question 3 : Écrivez et testez une fonction (sans paramètres) permettant de jouer contre l'ordinateur à Pierre-Feuille-Ciseau. Le gagnant est celui arrivant à 3 points. Cette fonction demande à l'utilisateur son choix parmi 'P' pierre, 'F' feuille, 'C' ciseau. La fonction retourne 'H' si c'est le joueur humain qui gagne et 'O' si c'est l'ordinateur. Évidemment, il faut utiliser les fonctions définies dans les questions 1 et 2.

Question 4 : Écrivez un *main()* affichant qui, de l'Homme ou de la Machine, a gagné la partie, suite à l'appel à la fonction de la question 3.

Testez le tout.

Exercice 8 : l'additionneur

Les additionneurs sont des composants électroniques qui permettent d'additionner des nombres codés en binaire, chiffre par chiffre.
Par exemple, l'addition des nombres 0011 et 0101 se fait suivant le tableau :

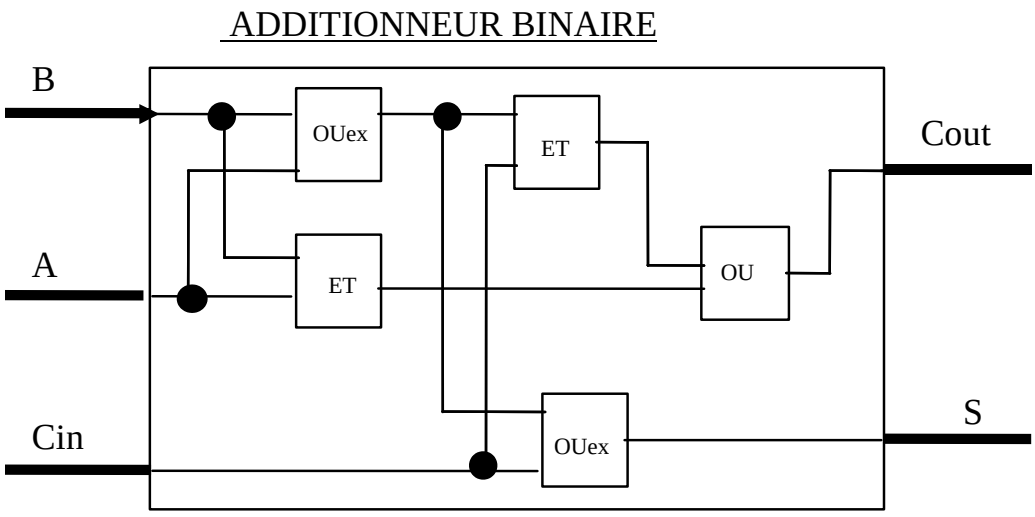
retenue (Cin/Cout)	D = 0	1	1	1	0
1° nombre (A=A4A3A2A1)		0	0	1	1
2° nombre (B=B4B3B2B1)		0	1	0	1
résultat (S=S4S3S2S1)		1	0	0	0

Ce TP est partagé en procédure, fonctions, et un programme principal.

On veut réaliser un additionneur de nombres de quatre chiffres binaires ; pour cela, il est indispensable de réaliser un additionneur de chiffres binaires.

Dans notre cas, un chiffre binaire sera un entier (type int) qui ne peut prendre que les valeurs 0 ou 1 (attention à la lecture des nombres qui doit se faire chiffre par chiffre).

Un additionneur de chiffres binaires à la forme suivante :



1°) Écrivez et **testez** (dans un programme de test affichant la valeur attendue et la valeur obtenue pour chacun des 4 cas) les fonctions ET, OU, OUEX.

Elles réalisent les opérations suivantes :

0 ET 0 = 0 1 ET 0 = 0 0 ET 1 = 0 1 ET 1 = 1
donc a ET b ~ a * b

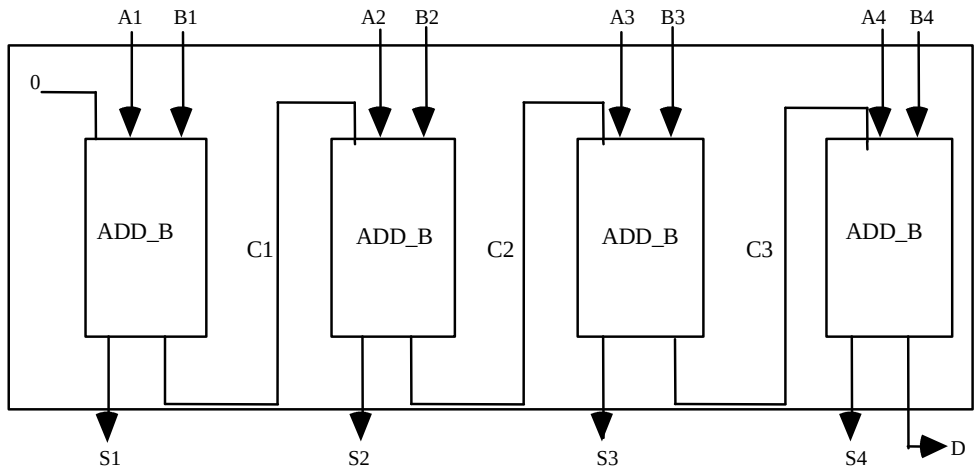
0 OUex 0 = 0 1 OUex 0 = 1 0 OUex 1 = 1 1 OUex 1 = 0
donc a OUex b ~ (a + b) - 2 * (a*b)

0 OU 0 = 0 1 OU 0 = 1 0 OU 1 = 1 1 OU 1 = 1
donc a OU b ~ (a + b) - (a*b)

2°) Écrivez et **testez** (dans un programme de test affichant la valeur attendue et la valeur obtenue pour chacun des 8 cas) la procédure add_b, qui réalise l'addition de deux chiffres binaires avec retenue (donc trois chiffres en entrée, deux en sortie).

3°) Écrivez et **testez** un programme qui réalise un additionneur de 2 nombres de 4 chiffres binaires.

L'additionneur de nombres binaires, pour lequel nous nous restreignons à 4 chiffres binaires par nombre, est spécifié par :



Remarques :

Cet additionneur réalise l'addition des deux nombres binaires A4 A3 A2 A1 et B4 B3 B2 B1 et donne comme résultat (D , S4 S3 S2 S1). Le chiffre binaire D est dit registre de débordement.

Exercices complémentaires

Exercice 9 : (*cf* TP3 exercice 13)

1) Écrivez une fonction `heron(entF ent a : réel) délivre réel`, qui « retourne » l'approximation de la racine carrée de `a` par la méthode d'Héron d'Alexandrie.

Cet algorithme repose sur le principe suivant :
La suite U telle que $U_0 = 1$ et $U_n = \frac{1}{2}(U_{n-1} + a/U_{n-1})$ tend vers racine carrée de a.

On arrêtera la boucle quand la différence en valeur absolue entre le carré de l'approximation et `a` sera inférieure à un *EPSILON* fixé au départ.

La fonction `fabs` (qui délivre un double) demande la directive `#include <math.h>`

2) Écrivez un programme, utilisant la fonction ci-dessus, demandant à l'utilisateur un réel et affichant l'approximation de la racine carrée de ce réel.

Exercice 10 :

1) Écrivez une fonction qui « retourne » la somme des *n* premiers termes de la série :

$$1+1/2+1/3 + \ldots + 1/(n-1)+1/n$$

2) Écrivez un programme de test de cette fonction, faisant appel à celle-ci en lui donnant quelques paramètres effectifs judicieusement choisis, et affichant pour chaque cas testé le résultat attendu et le résultat obtenu.

3) Une fois la fonction validée, écrivez un programme, utilisant la fonction ci-dessus, demandant à l'utilisateur un entier *n* et affichant la somme des *n* premiers termes de la série précédemment définie.

Exercice 11 : Testez les exercices du TD6.