Série 2 :
Programmation *C* : structures de contrôles et fonctions

Buts

Le but de cette série d'exercices est de vous permettre de continuer à pratiquer les bases de la programmation en C sur des exemples plus avancés utilisant des structures de contrôle et des fonctions.

Rappel

Avez-vous pris connaissance des **conseils relatifs à ces séries d'exercices** ?

Exercice 1 : Les tables de multiplication (itération `for`, niveau 1)

Objectif

Écrivez un programme `tables.c` affichant les tables de multiplication de 2 à 10.

Votre programme devra produire la sortie suivante à l'écran :

```
Tables de multiplication

Table de 2 :
  1 * 2 = 2
  ...
  10 * 2 = 20
...
Table de 5 :
  1 * 5 = 5
  2 * 5 = 10
  ...
...
Table de 10 :
  1 * 10 = 10
  ...
  10 * 10 = 100
```

Méthode :

Utilisez deux structures d'itération `for` imbriquées l'une dans l'autre.

Exercice 2 : Rebonds de balles (itération `for`, niveau 2)

Objectif :

L'objectif de cet exercice est de résoudre le problème suivant :

Lorsqu'une balle tombe d'une hauteur initiale ***h***, sa vitesse à l'arrivée au sol est $v = \sqrt{2hg}$. Immédiatement après le rebond, sa vitesse est *v1=eps*v*

(où ***eps*** est une constante et ***v*** la vitesse avant le rebond). Elle remonte alors à la hauteur $h1 = \frac{v1^2}{2g}$.

Le but est d'écrire un programme (`rebonds1.c`) qui calcule la hauteur à laquelle la balle remonte après un nombre `NBR` de rebonds.

Méthode :

On veut résoudre ce problème, non pas du point de vue formel (équations) mais par **simulation** du système physique (la balle).

Utilisez une itération `for` et des variables `v`, `v1`, (les vitesses avant et après le rebond), et `h`, `h1` (les hauteurs au début de la chute et à la fin de la remontée).

Tâches :

Écrivez le programme `rebonds1.c` qui affiche la hauteur après le nombre de rebonds spécifié.

Votre programme devra utiliser la **constante** `g`, de valeur 9,81 et demander à l'utilisateur d'entrer les valeurs de

- **H0** (hauteur initiale, contrainte : H0 >= 0),
- **eps** (coefficient de rebond, contrainte 0 <= eps < 1)
- **NBR** (nombre de rebonds, contrainte : 0 <= NBR).

Essayez les valeurs H0 = 10, eps = 0.9, NBR = 20 :

```
Au 20ème rebond, la hauteur sera de 0.147809 m.
```

Remarque :

- Pour utiliser les fonctions mathématiques (comme `sqrt()`), ajoutez `#include <math.h>` au début de votre fichier source et l'option `-lm` à la fin de votre commande de compilation :
  ```
  gcc monfichier.c -o monfichier -lm
  ```
- Rappel (cf **exercice 1** et **exercice 3** semaine passée) : dans `scanf()`, on utilise `"%d"` pour lire un entier et `"%lf"` pour lire un `double`.

Exercice 3 : Rebonds de balles - le retour. (boucles do...while, niveau 2)

On se demande maintenant combien de rebonds fait cette balle avant que la hauteur à laquelle elle rebondit soit plus petite que (ou égale à) une

hauteur donnée `h_fin`.

Écrivez le programme `rebonds2.c` qui affiche le nombre de rebonds à l'écran.

Il devra utiliser une boucle `do...while`, et demander à l'utilisateur d'entrer les valeurs de :

- **H0** (hauteur initiale, contrainte : H0 >= 0),
- **eps** (coefficient de rebond, contrainte 0 <= eps < 1)
- **h_fin** (hauteur finale désirée, contrainte : 0 < h_fin < H0).

Essayez les valeurs H0=10, eps=0.9, h_fin=0.5 :

```
Nombre de rebonds : 15
```

Exercice 4 : Une histoire de prêt (boucle, niveau 2)

L'objectif de cet exercice est de résoudre le problème suivant :

> Une banque fait un prêt à une personne X pour un montant total de **S0** francs. X rembourse chaque mois un montant **r** et paye un intérêt **ir*S** où **S** est la somme restant à rembourser et **ir** le taux d'intérêt employé.

Quelle est la somme des intérêts encaissés par la banque quand X a remboursé la totalité de son prêt ?

Écrivez le programme `pret.c` qui calcule la somme des intérêts encaissés et la durée en mois du remboursement, puis qui affiche ces informations à l'écran.

Contraintes :

- Votre programme devra utiliser une boucle `while`.
- Les valeurs `S0`, `r` et `ir` seront demandées à l'utilisateur avec les contraintes : `S0 >= 0 , r > 0, et 0<=ir<1 .`

Testez votre programme avec les valeurs suivantes: S0=30000, r=1300, ir=0.01 (i.e. 1%) :

```
Somme des intérêts encaissés : 3612.00 (sur 24 mois).
```

Exercice 5 : Nombres premiers (structures de contrôle, niveau 2)

Écrivez le programme `premier.c` qui demande à l'utilisateur d'entrer un entier **n** strictement plus grand que 1, puis décide si ce nombre est premier ou non.

Algorithme :

1. Vérifier si le nombre **n** est pair (si oui, il n'est pas premier sauf si c'est 2).
2. Pour tous les nombres impairs inférieurs ou égaux à la racine carrée de **n**, vérifier s'ils divisent **n**. Si ce n'est pas le cas, alors **n** est premier.

Tâches :

- Si **n** n'est pas premier, votre programme devra afficher le message:
  ```
  Le nombre n'est pas premier, car il est divisible par D
  ```
  où *D* est un diviseur de n autre que 1 et n.
- Sinon, il devra afficher le message:
  ```
  Je crois fortement que ce nombre est premier
  ```

Testez votre programme avec les nombres : 2, 16, 17, 91, 589, 1001, 1009, 1299827 et 2146654199. Indiquez ceux qui sont premiers.

Exercice 6 : Expressions arithmétiques (niveau 3)

Soient les expressions suivantes :

- $\frac{x}{1-e^x}$;
- $x\log(x)e^{\frac{2}{(x-1)}}$;
- $\frac{-x-\sqrt{x^2-8x}}{2-x}$;
- $\sqrt{(\sin x - \frac{x}{20})(\log\sqrt{x^2-\frac{1}{x}})}$

On rappelle que le logarithme est défini sur les réels strictement positifs, la racine carrée sur les réels positifs ou nuls, la fraction 1/x sur les réels non nuls. Les autres fonctions sont définies sur l'ensemble des réels.

Écrivez un programme `formules.c` qui :

1. demande à l'utilisateur d'entrer un réel ;
2. enregistre la réponse de l'utilisateur dans une variable *x* de type réel ;
3. teste pour chacune des expressions ci-dessus si elle est définie pour *x* :
   - si oui, le programme calcule le résultat de l'expression puis l'affiche ;
   - sinon, le programme affiche :
     ```
     Expression indéfinie : i
     ```
     où *i* est le numéro de l'expression considérée.

Pour utiliser les fonctions mathématiques, vous devez ajouter en début de programme la ligne :

```
#include <math.h>
```

et l'option `-lm` à la fin de votre commande de compilation :

```
gcc monfichier.c -o monfichier -lm
```
(ce genre d'instructions seront expliquées plus tard dans le cours).

Vous pouvez alors utiliser les fonctions `log` pour le logarithme, `sqrt` pour la racine carré (*SQuare RooT*), `exp` pour l'exponentielle, et `sin` pour le sinus.
Pour l'élévation au carré, utilisez plutôt la multiplication que la fonction `pow` (`man pow`), qui est plus lente.

Testez votre programme avec les valeurs : -1, 0, 1, 2, 3, 8

Résultas attendus:

```
Entrez un nombre réel : -1
Expression 1 : -1.581977
Expression 2 : non définie
Expression 3 : -0.666667
Expression 4 : non définie

Entrez un nombre réel : 0
Expression 1 : non définie
Expression 2 : non définie
Expression 3 : -0.000000
Expression 4 : non définie

Entrez un nombre réel : 1
Expression 1 : -0.581977
Expression 2 : non définie
Expression 3 : non définie
Expression 4 : non définie

Entrez un nombre réel : 2
Expression 1 : -0.313035
Expression 2 : 10.243407
Expression 3 : non définie
Expression 4 : 0.711989

Entrez un nombre réel : 3
Expression 1 : -0.157187
Expression 2 : 8.959013
Expression 3 : non définie
Expression 4 : non définie

Entrez un nombre réel : 8
Expression 1 : -0.002685
Expression 2 : 22.137106
Expression 3 : 1.333333
Expression 4 : 1.106779
```

Exercice 7 : Résolution d'une équation du 3e degré (variables, expressions arithmétiques, branchements conditionnels; niveau 2)

On veut maintenant faire un programme qui demande trois valeurs (a0, a1, a2) à l'utilisateur et affiche la (ou les) solution(s) réelle(s) *z* de l'équation :

$$z^3 + a_2 z^2 + a_1 z + a_0 = 0$$

Indications - commencer par calculer :

$$Q = \frac{3a_1 - {a_2}^2}{9}$$

$$R = \frac{9a_2a_1 - 27a_0 - 2{a_2}^3}{54}$$

$D = Q^3 + R^2$

Démonstration des formules à la page : **http://mathworld.wolfram.com/CubicEquation.html**

Si D < 0, on calcule les trois solutions réelles ainsi :

$$\theta \equiv \cos^{-1}\left(\frac{R}{\sqrt{-Q^3}}\right).$$

$\cos^{-1}$ est la fonction `acos` de C, fournie par `<math.h>`, i.e. il faut inclure ce fichier en début de programme

$$z_1 = 2\sqrt{-Q}\cos\left(\frac{\theta}{3}\right) - \tfrac{1}{3}a_2$$

$$z_2 = 2\sqrt{-Q}\cos\left(\frac{\theta + 2\pi}{3}\right) - \tfrac{1}{3}a_2$$

$$z_3 = 2\sqrt{-Q}\cos\left(\frac{\theta + 4\pi}{3}\right) - \tfrac{1}{3}a_2.$$

Sinon, on calcule :

S = $\sqrt[3]{R+\sqrt{D}}$ la racine cubique de x est obtenue par "`pow(x,1.0/3.0)`" en C.
Notez que la racine cubique de (`-x`) est l'opposé de la racine cubique de `x`.
Il faut en effet traiter séparément le cas où `x < 0` du cas `x >= 0`, car C ne l'accepte pas dans la fonction `pow`.

T = $\sqrt[3]{R-\sqrt{D}}$,

Si D=0 et S+T != 0, il y a 2 racines :

$z_1 = -\frac{1}{3}a_2 + (S+T)$

$z_2 = -\frac{1}{3}a_2 - \frac{1}{2}(S+T)$ (racine double)

Sinon, il y a une racine unique : z_1 ci-dessus.

Exercice 8 : Prototypes (fonctions, niveau 1, puis 2 pour les points 4 & 5)

Écrivez un programme `proto.c` dans lequel vous définissez une fonction `demander_nombre()` respectant le prototype suivant :

```
int demander_nombre(void);
```

Cette fonction doit demander un entier à l'utilisateur et retourner sa valeur.

1. Placez la **définition** de la fonction avant le `main()`.
   Faites appel à la fonction dans le `main()` et affichez le résultat renvoyé.
2. Que se passe-t-il si l'on déplace la **définition** de la fonction `demander_nombre()` *après* le `main()` et que l'on recompile le programme ? (faites-le et vérifiez si le compilateur réagit comme vous vous y attendiez).
3. Ajoutez ensuite le **prototype** de la fonction `demander_nombre()` *avant* `main()` et recompilez le programme.
4. Modifiez maintenant la fonction pour qu'elle prenne 2 paramètres : les bornes minimale et maximale entre lesquelles on veut que l'utilisateur entre le nombre.
   La fonction doit boucler tant que l'utilisateur ne donne pas un chiffre valide (compris entre ces deux bornes).
   Pensez aussi à vérifier que la borne minimale est plus petite que la borne maximale ! Si ce n'est pas le cas, il faut inverser leur rôle.
5. Raffinez encore le programme de sorte que si la borne maximale est inférieure ou égale à la borne minimale, elle ne soit pas prise en compte (c'est-à-dire que l'on peut entrer n'importe quel chiffre plus grand que la borne minimale). Faites en sorte de spécifier à l'utilisateur quelle(s) borne(s) il doit respecter.

Exercice 9 : Calcul approché d'une intégrale (fonctions niveau 1)

On peut montrer que pour une fonction suffisamment régulière (disons ici C-infinie), on a la majoration suivante :

$$\left|\int_a^b f(u)\,\mathrm{d}u - \frac{b-a}{840}\left[41f(a)+216f(\frac{5a+b}{6})+27f(\frac{2a+b}{3})+272f(\frac{a+b}{2})+27f(\frac{a+2b}{3})+216f(\frac{a+5b}{6})+41f(b)\right]\right| \leq M_8\cdot(b-a)^9\frac{541}{315\cdot(9!)\cdot 2^9}$$

où M_8 est un majorant de la dérivée huitième de f sur le segment [a,b].

Écrivez un programme calculant la valeur approchée d'une intégrale à l'aide de cette formule, c'est-à-dire par :

$$\frac{b-a}{840}\left[41f(a)+216f(\frac{5a+b}{6})+27f(\frac{2a+b}{3})+272f(\frac{a+b}{2})+27f(\frac{a+2b}{3})+216f(\frac{a+5b}{6})+41f(b)\right]$$

Où les valeurs `a` et `b` sont entrées par l'utilisateur.

Pour cela écrivez 3 fonctions :

1. Une fonction `f` de votre choix, qui correspond à la fonction dont vous souhaitez calculer l'intégrale. (Essayez plusieurs cas avec x^2, x^3, ..., sin(x), 1/x, etc. Il faudra bien sûr recompiler le programme à chaque fois). Pour utiliser les fonctions mathématiques, n'oubliez pas d'ajouter
   ```
   #include <math.h>
   ```
   en début de programme et compiler avec l'option `-lm`.
2. Une fonction `integre` qui, à partir de deux arguments correspondant à *a* et *b*, calcule la somme ci-dessus pour la fonction `f` .
3. Une fonction qui demande à l'utilisateur d'entrer un nombre réel (similaire à `demander_nombre` dans l'exercice 1). Utilisez cette fonction pour demander à l'utilisateur les bornes *a* et *b* de l'intégrale.

Utilisez ces fonctions dans le `main()` pour réaliser votre programme.

Note : Vous pourrez trouver **ici une démonstration de la formule** donnée plus haut.

Question subsidiaire (Niveau 3) :
Comment faire pour ne pas recompiler le programme pour chaque nouvelle fonction ?

En tant que tel, c'est-à-dire laisser l'utilisateur saisir lui-même sa formule, cela est beaucoup trop compliqué pour ce cours d'introduction.
Mais si on simplifie le problème en donnant la possibilité de choisir parmi un ensemble de fonctions préfétinies (inclues alors dans le programme), alors c'est faisable en utilisant des **pointeurs** sur des fonctions. Les *pointeurs* seront introduits en semaine 5 dans le cours, et

les pointeurs sur fonction seront vus en semaine 7. Il faudra donc patienter encore un peu...

Exercice 10 : La fonction cosinus (définition et appel de fonction, niveau 2)

Écrivez le programme `cos.c` qui calcule une approximation de la fonction cosinus *cos(x)* (pour *x* dans [0, 2*pi])

Méthode

Pour calculer *cos(x)*, utilisez la série définie par :

$$\cos(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!}$$
$$= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

Tâches

Écrivez les fonctions (appellées depuis la fonction `cos(x)`)

- `double factorielle(int k)` qui calcule la factorielle d'un nombre *k* (et la retourne au format `double`)
- `double somme_partielle(double x, int N)` qui calcule la somme partielle de la série au rang *N* (somme des *N* premiers termes de la série).

N sera demandé en entrée à l'utilisateur dans la fonction `main()` (vous pouvez utiliser ici la fonction `demander_nombre` de l'exercice 1), puis tant que l'utilisateur n'entre pas 0.0, le programme calcule et affiche le cosinus du nombre entré.

Remarques :

- La fonction factorielle renvoie facilement des nombres très grands, qui dépassent les capacités de précision de l'ordinateur. Pour un ordre de grandeur, avec le type `double`, la plus grande valeur permise sera k=170 (170! = 7.25742e+306). La valeur retournée pour des nombre supérieurs à ces valeurs limites sera `inf`.
  Évitez donc de tester le programme avec de trop grandes valeurs de N.
- Vous pouvez aussi raffiner votre programme de sorte que *N* soit inférieur ou égal à 86 (par exemple en utilisant la borne maximale dans `demander_nombre`).
- Notez que pour afficher plus de décimales dans le résultat, vous pouvez utiliser par exemple `%.12f` comme argument de `printf`. La valeur entre le `.` et le `f` indique le nombre de décimales.

Exercice 11 : Histoires de dates (structures de contrôle, fonctions, niveau 1)

[proposé par A. Perrin, 2009]

Le 31 Décembre 2008, la version 30 Go du Zune player, un lecteur audio de Microsoft, se bloquait au démarrage. La solution officielle fût d'attendre le 1er janvier. Un bug à été découvert dans la gestion des années bissextiles.

Voici le code qui se trouvait dans la version buggée du Zune player. Le nombre de jours écoulés depuis le 01/01/1980 (« l'epoch » Microsoft), à compter de 1, est enregistré dans une puce. Au démarrage, le Zune player lit le nombre de jours et le convertit en date.

```
int days = 10593;       /* 31 décembre 2008, normalement lu depuis la puce */
int year = ORIGINYEAR;  /* 1980, l'epoch pour Microsoft */

while (days > 365) {
    if (IsLeapYear(year)) { /* 2008 est une année bissextile */
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

1. Trouvez l'erreur et proposez une solution (supposez que la fonction `IsLeapYear` est correcte).
2. Implémentez et vérifiez votre solution dans un nouveau programme nommé `zune.c` qui demande un entier supérieur ou égal à 1, représentant le nombre de jours écoulés depuis le 31/12/1979 (1 correspond au 01/01/1980 et 10593 au 31 décembre 2008), puis affiche la date correspondante sur la sortie standard.

   Exemples d'exécution :

```
 $ ./zune
Entrez etc. : 10593
31/12/2008

$ ./zune
Entrez etc. : 42
11/02/1980

$ ./zune
```

```
Entrez etc. : 1
01/01/1980

$ ./zune
Entrez etc. : 1337
29/08/1983
```

Indices :

- écrivez la fonction `int IsLeapYear(int y)`, qui renvoie 0 si `y` n'est pas une année bissextile, et un entier non nul autrement ;
- écrivez la fonction `DaysForMonth`, qui renvoie le nombre de jours du mois donné en argument. L'année doit aussi être donnée en paramètre (pourquoi ?) :

  ```
  int DaysForMonth(int year, int month)
  ```

- utilisez ces deux fonctions pour trouver la date à partir du nombre de jours écoulés depuis « l'epoch » Microsoft.

3. Pour continuer sur le même thème...

   Le «*timestamp*» UNIX est le nombre de secondes écoulées depuis le 01/01/1970 à 00:00. En utilisant la fonction `time` de la bibliothèque `time.h` («`man 2 time`» pour plus de détails), écrivez un programme `unix-time.c` qui affiche la date et l'heure actuelle.

   Exemple d'exécution :

   ```
   $ ./unix-time
   1235583855 secondes se sont ecoulees depuis le 1.1.1970 a minuit.
   Nous sommes donc le 25/02/2009 a 17:44:15.
   ```

   Comparez la sortie de votre programme à la date actuelle. Y'a-t-il une différence ? Si oui, expliquez pourquoi.

Dernière mise à jour le 25 février 2022
Last modified: Fri Feb 25, 2022