Exercices de programmation en C- ue AAGB
Partie 1: syntaxe de base
Exercices provenant en partie du module C avancé LU2IN018

Revu dernièrement par Mathilde Carpentier

Version du 21 septembre 2022

Enseignants:

NOTEde 2021 POUR 2022 : mettre tous les exercices à faire au début, puis faire une seconde partie exercices supplémentaires. Cette année on a fait 1 séance de cours + 2 séances de TP (ça rentre presque, 3 séances serait peut-être mieux, mais pas sûr)

Note de 2022 : Modification faite.

Exercices à faire :

- Premier programme
- Types de base, affichage, scanf et division entière
- Comparaison de flottants
- Le type **char** est un entier comme les autres
- Première fonction : Convertisseur francs-euros

Fin séance 1

- Syntaxe des alternatives : qui est le plus grand ?
- Les 3 boucles : Factorielles
- Boucles imbriquées : Nombres premiers
- Boucle `while` avec condition complexe : Racine Carrée

Fin séance 2

Table des matières

Niveau des exercices

Les exercices sont répartis selon trois niveaux :
— base : ces exercices couvrent les notions de base et sont à faire obligatoirement.
— entraînement : ces exercices abordent des notions déjà présentées dans d'autres exercices. Il est recommandé de les faire pour s'exercer.
— approfondissement : ces exercices sont plus difficiles. Il est intéressant de les faire pour aller plus loin.

1 Premiers pas

Exercice 1 *(base)* – Premier programme

Notions abordées

- ✯ Ecrire un programme en C
- ✯ Compilation
- ✯ Les bibliothèques (`#include`)
- ✯ trouver quelle(s) bibliothèque(s) inclure

Pour commencer, ouvrez un Terminal et lancez un éditeur de texte (c'est-à-dire tapez gedit, emacs ou vim, sans oublier le `&` à la fin). Vous écrirez vos programmes C dans cet éditeur, en les sauvegardant sous un nom explicite se terminant toujours par l'extension `.c` (ou `.h`).

1. Recopiez le programme suivant, compilez le (il faudra taper la commande `gcc...` dans le terminal) et exécutez le (il faudra taper `./MonExecutable` dans le terminal).

```
1
2 int main() {
3   int annee=2019;
4
5       return 0;
6 }
```

Il faut ensuite compiler votre programme. Pour compiler vos programme, vous utiliserez le compilateur `gcc` comme suit :

```
gcc -Wall -o MonExecutable monProgramme.c
```

Le programme exécutable nommé `MonExecutable` est créé à partir de votre programme que vous avez écrit dans le fichier texte `monProgramme.c`. Vous pourrez l'executer en tapant dans le terminal `./MonExecutable`

2. Ajoutez la ligne printf ("Hello world in %d\n", annee); puis compilez à nouveau.
Le programme ne compile pas et l'erreur "`warning: implicit declaration of function 'printf'` " est affichée. Cela signifie que le compilateur ne trouve pas la fonction `printf` car la bibliothèque la définissant n'est pas incluse dans votre programme.

3. Tapez `man 3 printf` dans le terminal. Déterminez quelle bibliothèque est nécessaire puis ajoutez là à votre code. Compilez et exécutez à nouveau votre programme.

Solution :

```c
#include <stdio.h>
int main() {
  int annee=2019;
  printf("Hello world in %d\n", annee);
  return 0;
}
```

Exercice 2 *(base)* – Types de base, affichage, scanf et division entière

Notions abordées

- ✯ les types
- ✯ la fonction `printf`
- ✯ la fonction `scanf`
- ✯ la division entière

La fonction scanf permet de de récupérer ce qui est tapé au clavier par l'utilisateur. Son prototype est :

```c
int scanf(const char * restrict format, ...);
```

Le "format" est le même qu'avec la fonction `printf`, et il faut ensuite donner l'**adresse** des variables dans lesquelles il faut mettre les valeurs lues (nous y reviendront plus tard). Par exemple, pour lire un entier, la fonction `scanf` s'écrit :

```c
int i1;
scanf("%d", &i1);
```

L'opérateur & permet d'avoir l'adresse de la variable i1 ; il ne faut surtout pas l'oublier. Il est conseillé d'écrire des formats les plus simples possibles car sinon le comportement de la fonction scanf devient rapidement difficile à comprendre.

1. Ecrire un programme qui demande un entier positif à l'utilisateur et l'affichage au format entier (digit %d), octal (%o et hexadécimal (%X.
2. Compléter le programme précédent pour qu'il demande un autre entier à l'utilisateur (différent de 0) et affiche le résultats de la division du premier par le second. Que remarquez vous ? Comment l'éviter ?

Solution :

```c
  int i1, i2;
  float f1, f2;

  printf("Entrer un entier\n");
  scanf("%d", &i1);
  printf("Formats d'affichages entier: decimal %d octal %o hexadecimal %X\n",i1
      , i1, i1);
```

```
7
8     printf("Entrer un autre entier (<>0)\n");
9     scanf("%d", &i2);
10    printf("%d/%d=%d\n",i1, i2, i1/i2);
11    /*C'est une division entiere
12    Voici comment l'eviter:*/
13    printf("%d/%d=%f\n",i1, i2, (float)i1/i2);
```

Exercice 3 *(base)* – Comparaison de flottants

Notions abordées

- ✯ comparaison de **float**

1. Savez vous pourquoi on ne peut tester l'égalité entre nombres de type **float** ? Nous allons l'illustrer avec un exemple. Compléter le programme précédent en déclarant 2 variables de type **float** initialisées à 0.1. Ecrire ensuite un **if** qui teste si la somme de ces 2 variables est égale à 0.2 ou non. Que remarquez vous ? Faites le même test avec des variables initialisées à 2 (et la somme à 4).

Solution :

```
1   f1=0.1;
2   f2=0.1;
3   if(f1+f2==0.2){
4     printf("La somme de %f et %f fait bien 0.2\n", f1,f2);
5   }else{
6     printf("La somme de %f et %f ne fait pas 0.2\n", f1,f2);
7   }
8
9   f1=2;
10  f2=2;
11  if(f1+f2==4){
12    printf("La somme de %f et %f fait bien 4\n", f1,f2);
13  }else{
14    printf("La somme de %f et %f ne fait pas 4\n", f1,f2);
15  }
```

2. Modifiez le programme précédent pour vérifier non plus que la somme des deux variables initialisées à 0.1 est égale à 0.2 mais plutôt que la différence est inférieure à 10^{-6} (par exemple). Le comportement e votre programme vous semble-t-il plus cohérent ? Ceci est la bonne manière de comparer des **float**.

Solution :

```
1   f1=0.1;
2   f2=0.1;
```

```
3     if(f1+f2 - 0.2 <1E-6){
4       printf("La somme de %f et %f fait bien 0.2 a 1E-6 pret\n", f1,f2);
5     }else{
6       printf("La somme de %f et %f ne fait pas 0.2\n", f1,f2);
7     }
```

Exercice 4 *(base)* – Le type char est un entier comme les autres

Notions abordées

- ✯ le type **char**
- ✯ le code ASCII

1. Ecrire un programme qui demande à l'utilisateur un caractère et affiche son code ASCII, ainsi que la caractère suivant et son code ASCII et le caractère précédent et son code ASCII. Dans ce programme, vous déclarerez aussi une autre variable de type char initialisée à 127 et vous l'afficherez comme un entier (%d) avant et après l'avoir incrémentée de 1. Que remarquez cous ?

Solution :

```
1  #include <stdio.h>
2
3  int main() {
4    char c1;
5    char c3=127;
6
7    printf("Entrer un caractere\n");
8    scanf("%c", &c1);
9    printf("c1: %c %d, Caractere suivant %c %d, caractere precedent %c %d\n", c1,
         c1, c1+1, c1+1, c1-1, c1-1);
10
11   printf("c3: %d\n", c3);
12   c3++;
13   printf("c3+1: %d\n", c3);
14   return 0;
15 }
```

2 Les fonctions

Exercice 5 *(base)* – Première fonction : Convertisseur francs-euros

1. Écrivez une fonction convertir qui prend en argument un nombre flottant et un caractère (e ou f) et qui renvoie la valeur du nombre convertie de francs en euros (resp. d'euros en francs) lorsque le caractère est f (resp. e).
   Vous écrirez une fonction main pour la tester

Solution :

```c
#include <stdio.h>
#include <stdlib.h>

float EUR=6.55957;

float convertir(float x, char y)
{
  if ( y == 'f') {
    return x / EUR;
  } else {
    if (y=='e') {
      return x * EUR;
    } else {
      printf("ATTENTION ! Devise inconnue : %c\n", y);
      exit(1);
    }
  }
}

int main()
{
  printf("100 FRF = %.2f EUR\n", convertir(100, 'f'));
  printf("10 EUR = %.2f FRF\n", convertir(10, 'e'));
  return 0;
}
```

3 Alternatives, Boucles

Exercice 6 *(base)* – Syntaxe des alternatives : qui est le plus grand ?

Notions abordées

- ✯ `if` et `else`
- ✯ la fonction `scanf`

Prenons un programme manipulant deux variables de type `int` a et b :

```c
#include <stdio.h>

int main() {
int a, b;

printf("Entrez une valeur pour a\n");
```

```
scanf("%d",&a);
printf("Entrez une valeur pour b\n");
scanf("%d",&b);

return 0;
}
```

1. Complétez ce programme pour qu'il affiche "a est plus grand que b" si a est supérieur à b.
2. Complétez le programme précédent pour qu'il affiche "b est au moins aussi grand que a" si a n'est pas supérieur à b.

Solution :

```
#include <stdio.h>

int main()
{
  int a, b;

  printf("Entrez une valeur pour a\n");
  scanf("%d",&a);
  printf("Entrez une valeur pour b\n");
  scanf("%d",&b);

  if (a > b) {
    printf("a %d  est le plus grand\n", a);
  } else {
    if (b > a) {
      printf("b %d est le plus grand\n", b);
    } else {
      printf("a et b %d sont egaux\n", a);
    }
  }
  return 0;
}
```

Exercice 7 *(base)* – Les 3 boucles : Factorielles

La factorielle du nombre n, notée $n!$, est définie par $n! = 1 \times 2 \times ... \times (n-1) \times n$. Par convention, $0! = 1$.

1. Écrivez un programme qui calcule la factorielle du nombre 10 en utilisant une boucle for.

Solution :

```
#include <stdio.h>
int main()
{
  int i;
```

```
5     int fact = 1;
6     for (i = 2; i <= 10; i++) {
7       fact = fact * i;
8     }
9     printf("fact %i vaut %i\n", 10, fact);
10    /*ou*/
11    for (i = 10; i > 1; i--) {
12      fact = fact * i;
13    }
14    printf("fact %i vaut %i\n", 10, fact);
15
16
17    return 0;
18  }
```

2. Écrivez un programme qui calcule la factorielle du nombre 10 en utilisant une boucle while.

Solution :

```
1   #include <stdio.h> 2
2
3
4    int main(){
5     int i = 2;
6     int fact = 1;
7
8       while (i <= 10) {
9       fact = fact * i;
10      i++;
11    }
12     printf("10! vaut %d\n", fact);
13
14  /*ou*/
15
16    i = 10;
17    fact = 1;
18    while (i > 1) {
19      fact = fact * i;
20      i--;
21    }
22    printf("10! vaut %d\n", fact);
23
24
25     return 0;
26
27  }
```

3. Écrivez un programme qui calcule la factorielle du nombre 10 en utilisant une boucle do-while.

Solution :

```c
#include <stdio.h>

int main(){
  int i = 1;
  int fact = 1;
  do {
    fact = fact * i;
    i++;
  } while (i <= 10);
  printf("fact %i vaut %i\n", 10, fact);
  /*ou*/
  i = 10;
  fact = 1;
  do {
    fact = fact * i;
    i--;
  } while (i >= 1);
  printf("fact %i vaut %i\n", 10, fact);


  return 0;
}
```

4. Y a-t-il une différence à l'exécution entre les boucles for/while et la boucle do-while ?

Solution :
Dans la boucle `do-while`, la condition est évaluée après le premier passage dans la boucle. La boucle sera donc toujours exécutée au moins une fois. Dans les boucles for et `while` au contraire, si la condition est évaluée initialement, si elle n'est pas vérifiée, la boucle n ?est jamais exécutée.

5. Dans cet exemple de calcul de la factorielle, quel type de boucle vous semble le plus approprié ?

Solution :
La boucle ici consiste à faire évoluer un compteur entre deux bornes connues : 2 et N. La boucle `for` est la plus naturelle dans ce cas.

Exercice 8 *(base)* – Boucles imbriquées

1. Ecrivez un programme qui affiche toutes les tables de multiplications de 1 à 10 de manière lisible.

Exercice 9 *(base)* – Boucle `while` avec condition complexe : Racine Carrée

L'algorithme de Héron d'Alexandrie (ou de Newton) propose une méthode itérative pour calculer la racine carrée d'un nombre strictement positif a. Soit x la racine carrée de ce nombre a. On a :

$$\begin{aligned} x^2 &= a \\ x^2 + x^2 &= x^2 + a \\ 2x &= (x^2/a)/x \\ x &= (x^2 + a)/2x \end{aligned}$$

$$x_{n+1} = \frac{x_n + \frac{A}{x_n}}{2}$$

D'après le théorème du point fixe [1], on obtient alors la suite : $x_{i+1} = (x_i + a/x_i)/2$.

Le calcul peut être commencé avec n'importe quelle valeur x_0 différente de zéro. Par exemple, pour la racine de 2 ($a = 2$) en partant de $x_0 = 1$:

$$\begin{aligned} x_0 &= 1 \\ x_1 &= (1/2)(1 + 2/1) = 3/2 = 1,5 \\ x_2 &= (1/2)(3/2 + 2/(3/2)) = 17/12 = 1,4166666 \\ x_3 &= (1/2)(17/12 + 2/(17/12)) = 577/408 = 1,41421568 \end{aligned}$$

1. Écrivez un programme qui calcule la racine carrée d'une valeur a demandée à l'utilisateur en appliquant cet algorithme en calculant jusqu'à x_{10}.
2. On souhaite arrêter le calcul lorsque la différence relative entre deux valeurs calculées (c'est-à-dire $x_{i+1} - x_i)/x_i$ est inférieure en valeur absolue à une précision p que l'on s'est fixée (NB : $|x| < p$ équivaut à $p < x < p$).
   Modifier votre programme pour que le nombre d'itérations correspondent à cette condition.

Solution :

```c
#include <stdio.h>

int main()
{
  float a;     /* le nombre dont on calcule la valeur
          * absolue */
  float precision;
  float xi = 1.0;   /* la valeur initiale de la suite */
  float xj;   /* la valeur d indice (i+1) */
  float erreur;   /* la difference relative */

  printf("Entrez le nombre dont vous voulez la racine");
  scanf("%f", &a);
  printf("Entrez la precision voulue");
  scanf("%f", &precision);

```

1. Soit h une fonction dérivable, de fonction dérivée h' continue sur un voisinage V d'un point fixe b de h, c'est-à-dire tel que $h(b) = b$ et vérifiant : $Sup_{x \in V}|h'(x)| < 1$, alors la suite (x_n) définie par $x_{n+1} = h(x_n)$ converge vers b dès qu'il existe N tel que x_N est dans V.
Ici, $h(x) = (x + a/x)/2$ et le point fixe est $b = \sqrt{a}$.

```
17    do {
18      xj = (xi + a / xi) / 2;
19      erreur = (xj - xi) / xi;
20      xi = xj;
21    } while ((erreur >= precision) || (erreur <= -precision));
22
23    printf("la racine de %f a une precision de %E est %f\n", a, precision, xj);
24
25    return 0;
26  }
```

4 Exercices supplémentaires

Exercice 10 *(approfondissement)* – Syntaxe des alternatives et `switch` : base complémentaire

Notions abordées

- ✯ `switch`
- ✯ `scanf`

Soit un programme demandant une lettre à l'utilisateur.

```
#include <stdio.h>

int main() {
char a;

printf("Donnez un nucléotide: \n");
scanf("%c",&a);

return 0;
}
```

1. Modifiez ce programme pour qu'il affiche la base complémentaire de celle entrée par l'utilisateur et une message d'erreur ("base invalide") si ce n'est pas une base nucléotidique. Vous utiliserez des alternatives (`if`).
2. Ecrivez le même programme cette fois en utilisant un `switch`

Exercice 11 *(entrainement)* – Boucles imbriquées : Nombres premiers

Nous souhaitons afficher les nombres premiers inférieurs à N et supérieurs à 1.

Un nombre premier est un nombre exclusivement divisible par 1 et lui-même. Pour décider si un nombre M est premier, nous allons utiliser une variable entière `premier` initialisée à 1. Nous allons parcourir tous les nombres entre 2 et M. Si l'on trouve parmi ces nombres un diviseur de M, alors la variable `premier` est mise à 0. Si la variable `premier` a toujours la valeur 1 à la fin du parcours, alors M est premier.

Cet ensemble d'opérations doit être exécuté pour tous les nombres M entre 2 et N.

1. Pour commencer, écrivez un programme C qui permet de déterminer si M, nombre demandé à l'utilisateur, est premier ou non, en recherchant un diviseur entre 2 et M. Votre programme affichera à l'écran "M est premier" ou "M n'est pas premier" selon la valeur de premier à l'issue de votre boucle.

Solution :

```c
#include <stdio.h>

int main()
{
  int i, M;
  int premier = 1;

  printf("Entrez un nombre entier");
  scanf("%d", &M);

  for (i = 2; i < M; i++) {
    /* i est-il un diviseur de M ? */
    if ((M % i) == 0) {
      premier = 0;
    }
  }
  /* A l'issue de cette boucle, on sait si M est premier ou non */
  if (premier == 1) {
    printf("%d est premier\n", M);
  } else {
    printf("%d n est pas premier\n", M);
  }
  return 0;
}
```

2. Traduisez maintenant l'algorithme complet en un nouveau programme C qui affiche tous les nombres premiers compris entre 2 et N (N est demandé à l'utilisateur).
   Hint : Le nombre M variera alors entre 2 et N.

Solution :

```c
#include <stdio.h>

int main()
{
  int i, m;
  int N;
  int premier;

  printf("Entrez un nombre entier");
  scanf("%d", &N);

  for (m = 2; m <= N; m++) {
```

```
13      /* m est-il premier ? */
14      premier = 1;
15      for (i = 2; i < m; i++) {
16        /* j est-il un diviseur de i ? */
17        if ((m % i) == 0) {
18          premier = 0;
19        }
20      }
21      if (premier == 1) {
22        printf("%d est premier\n", i);
23      }
24    }
25    return 0;
26  }
```

3. Dans notre programme, la boucle for parcourt tout l'intervalle $[2, N[$ à la recherche d'un diviseur, alors qu'il serait plus judicieux de s'arrêter dès qu'on a trouvé un diviseur. Et même lorsque N est premier, il n'est pas nécessaire de vérifier sur tout l'intervalle : si on n'a pas trouvé de diviseur plus petit que N, alors N est premier.

   Modifiez votre programme afin de prendre en compte ces remarques. et comparez les temps d'exécution avec ceux obtenus à la question précédente, et vérifiez que ce nouveau programme est plus efficace.

Solution :

```
1   #include <stdio.h>
2
3   int main()
4   {
5     int i, m;
6     int N;
7     int premier;
8
9     printf("Entrez un nombre entier");
10    scanf("%d", &N);
11
12    for (m = 2; m <= N; m++) {
13      /* m est-il premier ? */
14      premier = 1;
15      for (i = 2; i < m && premier == 1; i++) {
16        /* j est-il un diviseur de i ? */
17        if ((m % i) == 0) {
18          premier = 0;
19        }
20      }
21      if (premier == 1) {
22        printf("%d est premier\n", i);
23      }
24    }
25    return 0;
26  }
```

Exercice 12 *(entrainement)* – Boucle while et traduire un algorithme : Calcul de PGCD

On souhaite déterminer le PGCD de deux nombres en appliquant l'algorithme suivant :

```
tant que (nombre_1 different de nombre_2)
faire
   si nombre_1 > nombre_2
   alors
 nombre_1=nombre_1-nombre_2
   sinon
 nombre_2=nombre_2-nombre_1
   fin si
fin tant que
```

1. Ècrivez un programme C calculant le PGCD de deux nombres $N1$ et $N2$.

Solution :

```c
#include<stdio.h>
int main()
{
  int nombre_1 = 1468;
  int nombre_2 = 48;
  while (nombre_1 != nombre_2) {
    if (nombre_1 > nombre_2) {
      nombre_1 = nombre_1 - nombre_2;
    } else {
      nombre_2 = nombre_2 - nombre_1;
    }
  }
  printf("PGCD(1468, 48) = %d\n",  nombre_1);
  return 0;
}
```

Exercice 13 *(entrainement)* – Calcul de pi

1. Le nombre π peut être calculé en appliquant la suite de Leibniz : $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - ...$
   Supposons que l'on veuille calculer π avec une précision p (*i.e.* nous souhaitons calculer une approximation pi de π telle que $\pi - p \leq pi \leq \pi + p$). Quel type de boucle vous semble la plus appropriée pour calculer π ?

Solution :
La suite tend progressivement vers π au travers de valeurs approchées par défaut et par excès. Deux valeurs successives de la suite encadrent donc π et leur différence indique donc l'intervalle d'incertitude restant sur p. Connaissant p, on peut poursuivre les améliorations de la valeur approchée de π jusqu'à ce que la valeur absolue de la différence entre deux valeurs successives soit inférieure à $2 * p$. On va donc ici aussi utiliser de préférence une boucle **do–while**, qui est particulièrement adaptée lorsque le nombre d'itération n'est pas connu à l'avance et que l'on souhaite exécuter la boucle au moins une fois.

2. Quel algorithme proposez-vous ? Ecrivez-le en C.

Solution :
On calcule ici une somme dont on ne connait pas le nombre de termes a priori. On procède de manière itérative, en ajoutant un terme à la fois et en regardant après chaque ajout si la précision souhaitée est atteinte.
Pour gérer l'alternance des signes dans la somme, on exprime le terme ajouté sous la forme $signe/nombre$, où $signe$ vaut alternativement 1 ou -1 : à chaque itération, on inverse sa valeur par l'opération signe = −signe.
La dernière difficulté est que C effectue des divisions entières lorsque les deux opérandes sont des nombres entiers. Ainsi, si l'on écrit signe/nombre avec $signe = \pm 1$ et $nombre$ deux variables entières, on obtient 0 (et non pas $\pm\frac{1}{nombre}$). La solution consiste à faire de signe une variable flottante (qui vaudra alternativement 1.0 et -1.0).

```c
#include <stdio.h>

int main()
{
  float pidiv4 = 0.0;
  float pidiv4_i;
  float erreur;
  int nombre = 1;
  float signe = 1.0;
  float PRECISION=0.000001;
  /* la valeur initiale (d indice i) */
  /* la valeur d indice (i+1) */
  /* pour le terme a ajouter */
  /* pour le signe du terme a ajouter */
  do {
    pidiv4_i = pidiv4 + signe / nombre;
    erreur = (pidiv4 - pidiv4_i) / pidiv4;
    signe = -signe;
    nombre = nombre + 2;
    pidiv4 = pidiv4_i;
  } while ((erreur >= PRECISION) || (erreur <= -PRECISION));
  printf("PI vaut %f\n", pidiv4 * 4.0);
  return 0;
}
```