Séance 1 : Introduction au langage C

L2 Informatique – Université Côte d'Azur

Exercice 1 — Compiler avec `gcc`

1. Créez un répertoire `ProgC`, puis, dans celui-ci, créez un repertoire `TP1`. Créez alors un fichier `exercice1.c` contenant le code ci-dessous. Vous pouvez utiliser l'éditeur de votre choix (gedit, GNU/Emacs, autre).

```c
#include <stdio.h>
int c = 3;
void main(void) {
    int u,d,c;
    u = 1;
    d = 2;
    int n = 100*c + 10*d + u; /* c vaut 3, d vaut 2 et u vaut 1 */
    printf("321 = %i\n",n); /* affiche « 321 = 321 » */
}
```

2. Dans un terminal, compilez le fichier avec la commande : «`gcc exercice1.c`». Quel est le nom de l'exécutable produit ? Lancez-le. Le résultat est-il celui attendu ?
3. Pour éviter ce genre de mauvaise surprise, durant cette UE, nous allons systématiquement compiler le code avec les options «`gcc -Wall -ansi -pedantic exercice1.c`». Corrigez le code jusqu'à ce que le compilateur n'affiche aucun *warning* et que le code s'exécute comme souhaité. Vous devez trouver trois erreurs.
4. Créer un fichier `compiler.sh` qui lance automatiquement la bonne commande de compilation avec les bonnes options pour créer un fichier `exercice1`.

Durant les TP, on se placera dans le répertoire `~/progC/tp1/` précédemment créé. On utilisera un fichier source par exercice (`exercice1.c`, `exercice2.c`, etc.) Il faudra ajouter au script `compiler.sh` deux nouvelles lignes pour chaque exercices : une ligne pour afficher «`Compilation de l'exercice1`» et une autre pour lancer effectivement la compilation avec `gcc`. Il est important, dans ce TP et les suivants, que la compilation se fasse sans aucun *warning* !

```
étudiant@valrose:~/ProgC/tp1/ % ./compiler.sh
Compilation de l'exercice 1
Compilation de l'exercice 2
Compilation de l'exercice 3
```

Exercice 2 — Quelques manipulations simples

1. Créez un fichier `exercice2.c` contenant une fonction `question1` affichant `Hello World`. Modifiez le fichier `./compilation.sh` pour qu'il compile aussi l'exercice 2.

```c
void question1(void) {
  /* À Compléter */
}

int main(void) {
   question1();
   …
   question6();
   return 0 ;
}
```

2. Écrivez dans une fonction `question2` un code C qui affiche sur une ligne les entiers de 65 à 90. Prenez l'habitude de bien séparer le code en différentes fonctions (en régle générale, une par question).
3. Écrivez un code C qui affiche sur une ligne les caractères dont le code ASCII va de 65 à 90.
4. Créez deux variables, une chaîne `s="Pi vaut"` et un nombre `x=3.1415926` puis affichez les avec `printf`.
5. Copiez et exécutez la fonction ci-dessous. Prenez un instant pour méditer sur ce code.

```c
void question5(void) {
  char c = 'a';
  int i = c + 1;
  printf("Valeur de c en caractère est '%c' et son équivalent décimal vaut %d\n",c,c);
  printf("Valeur de i en caractère est '%c' et son équivalent décimal vaut %d\n",i,i);
}
```

6. Écrivez le code qui affiche sur une ligne les caractères de `'a'` à `'z'` (en minuscule).

Exercice 3 — De l'entier à la division

1. En vous inspirant du cours et en utilisant la fonction `getchar()`, écrire une fonction qui lit un entier sur la sortie standard et renvoie sa valeur. Si la valeur lue n'est pas un entier, la fonction renverra `-1`. Pour plus de lisibilité, on utilisera comme invite de commande la chaîne `"entier>"`.
2. En utilisant la question précédente, écrire une procédure qui lit un entier sur la sortie standard et affiche sa parité. On traitera le cas où la valeur lue ne correspond pas à un nombre.

stdin/stdout

```
entier> saucisse
Ce n'est pas un entier.
entier> 23
L'entier 23 est impair.
entier> 18
L'entier 18 est pair.
```

3. De même, écrire une fonction `moyenne_stdin` qui lit des notes sur l'entrée standard et renvoie la moyenne. Si une note n'est pas valide (plus grande que 20), elle sera ignorée. On s'arrête lorsque la valeur lue n'est pas un entier.
4. Écrire une fonction qui prend en paramètre deux entiers, `a` et `b` et affiche le résultat de la division euclidienne. Par exemple avec `a=25` et `b=10` on affichera « `25 = 2×10 + 5` ». On traitera proprement la cas où `b` est nul.
5. Même question avec la division décimale. Par exemple avec `a=25` et `b=10` on affichera « `25 = 2.5 × 10` ».
6. Écrire une fonction affichant le résultat de la division, sous forme d'entier lorsque `a` est divisible par `b` et sous forme décimale sinon. On affichera soit une chaîne de la forme « `25/10 = 2.5 (approchée)` » ou « `30/10 = 3 (exacte)` ».

Exercice 4 — Boucles imbriquées

Écrivez un programme qui affiche les nombres de 1 à 100 à raison de 10 par lignes. Les nombres devront être alignés correctement.

stdin/stdout

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88  89  90
 91  92  93  94  95  96  97  98  99 100
```

Exercice 5 — ASCII art

Écrivez une fonction `pyramide(int n)` qui affiche une pyramide de taille `n`. Écrivez ensuite une boucle qui demande à l'utilisateur de rentrer une valeur entière et trace la pyramide correspondante. On s'arrêtera lorsque l'utilisateur rentrera une valeur invalide.

stdin/stdout

```
entier> 3
  *
 * *
* * *
entier> 7
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
* * * * * * *
entier> q
```

Exercice 6 — Notre ami Fibonacci

1. Écrire, avec une boucle (et non en récursif) une fonction de signature « `int fibo_naif(int n)` » calculant le n^e élèment de la suite de Fibonacci.
2. Afficher la valeur pour `n=50`. Qu'observez-vous ? Que s'est-il passé ?
3. Écrire une fonction de signature« `int somme(int a, int b)` » qui renvoie `a+b` si possible et `-1` en cas de débordement. On utilisera la valeur `INT_MAX` de la bibliothèque `limits.h` (qu'il faut inclure en début de fichier de la même façon que l'on inclut déjà `stdio.h`).
4. Écrivez une nouvelle fonction `fibonnaci` semblable à celle de la question 1, mais qui renvoie `-1` en cas de débordement d'entier.
5. En utilisant cette fonction, affichez la première valeur de `n` pour laquelle `fibonnaci(n)` ne donne pas une valeur correcte.
6. Ajoutez à votre programme une boucle qui demande un entier à l'utilisateur et affiche soit la valeur associée dans la suite de Fibonacci soit un message d'erreur.

stdin/stdout

```
Question 2)
fibo naïf : 50 → -298632863

Question 5)
La première valeur de n qui ne marche pas est : <???>

Question 6)
entier> 10
fibo(10) = 55
entier> 100
Débordement d'entier
entier> quitter
```

Exercice 7 — Lire les flottants

1. Écrivez une fonction `lire_flottant` sur le modèle de `lire_entier`. Un flottant est de la forme :
   (a) un signe « - » (facultatif) ;
   (b) une suite d'entiers représentant la partie entière (si vide, la partie entière vaut 0) ;
   (c) un point suivi éventuellement d'une suite d'entiers représentant la partie décimale.

À partir du moment ou l'expression contient un point, la partie entière est facultative et vaut alors 0..

2. Testez votre fonction sur les exemples suivants : `-12.25 6.55957 0.17 52. .53 -54. -.55`
3. Modifiez votre fonction pour afficher un avertissement lorsque la valeur lue dépasse la taille maximale d'un flottant (`FLT_MAX` dans la bibliothèque `float.h`).
4. Modifier votre fonction pour afficher un avertissement lorsque le nombre de chiffres après la virgule dépasse la précision permise par les flottants.

stdin/stdout

```
flottant> -.43
La valeur lue est « -0.430000 »
flottant> 7894561231032454215345343215247878787878
Avertissement : nombre trop grand
La valeur lue est « 789456031954734652599503351915216896 00.000000 »
flottant> 0.7894561231032454215345343215245321324312547879
Avertissement : précision ignorée
La valeur lue est « 0.789456 »
```

Exercice 8 — Une racine qui flotte

1. Écrivez une fonction `absolue` renvoyant la valeur absolue d'un flottant `x`.
2. Écrivez une fonction `float racine(float x, float p)` calculant par dichotomie la racine carrée d'un nombre x avec une précision p. L'algorithme est le suivant : on part de deux nombres $a = 0$ et $b = x$, puis on pose $m = (a + b)/2$. Si m^2 est plus grand que x on recommence en posant $b = m$, sinon, on pose $a = m$. On s'arrête lorsque $|m^2 - x| < p$.
3. Utilisez la fonction en demandant deux flottants sur l'entrée standard correspondant respectivement à x et p.

stdin/stdout

```
flottant> 25
flottant> 0.0001
4.99999
```

Exercice 9 — Les nombres particuliers

1. Un nombre est dit premier s'il est différent de 1, et s'il n'admet comme diviseurs que 1 et lui-même.
   (a) Écrivez une fonction qui détermine si un nombre entier est premier.
   (b) Écrivez un programme C qui lit sur l'entrée standard un nombre entier k et affiche les k premiers nombres premiers.
2. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs autres que lui-même. 6 est parfait, il admet comme diviseurs 1, 2, 3 et 6. La somme de ses diviseurs autre que lui-même est $1 + 2 + 3 = 6$. 28 est un autre nombre parfait.
   (a) Écrivez une fonction qui détermine si un nombre entier est parfait.
   (b) Écrivez un programme C qui affiche tous les nombres parfaits compris entre 2 bornes lues sur l'entrée standard.
3. Un nombre est dit doublon si le produit de ses diviseurs est un multiple de la somme de ses diviseurs. 6 est doublon, il admet comme diviseurs 1, 2, 3 et 6. La somme de ses diviseurs est $1 + 2 + 3 + 6 = 12$, le produit de ses diviseurs est $1 \times 2 \times 3 \times 6 = 36$, et 36 est un multiple de 12.
   (a) Écrivez une fonction qui détermine si un nombre entier est doublon.
   (b) Écrivez un programme C qui affiche tous les doublons compris entre 2 bornes lues sur l'entrée standard.
4. Écrivez un programme qui permet de tester les 3 fonctions précédentes. Il propose à l'utilisateur de taper :
   0 pour arrêter le programme
   1 pour calculer et afficher les k premiers nombres premiers
   2 pour calculer et afficher tous les nombres parfaits compris entre 2 bornes
   3 pour calculer et afficher tous les doublons compris entre 2 bornes