TP1 – Les fonctions

Projet de programmation M1

30 Septembre 2014

Exercice 1. [Copie] Exécutez le programme suivant :

```c
#include <stdio.h>

void f(int y) {
  y = 0;
}

int main() {
  int x = 1;
  f(x);
  printf("%d", x);
  return 0;
}
```

1. Quelle est la valeur de x à la fin du programme ? Pourquoi ? Si vous trouvez cela choquant, appelez `f` sur `3*x`.

   Réponse. *x vaut toujours* 1 *à la fin du programme. En C, quand on appelle une fonction, seule la valeur des arguments est transmise à la fonction ; il n'y a donc aucune raison pour que la valeur de x en soit affecté. Et c'est naturel. Si vous appelez votre fonction sur* `3*x`*, vous voulez que la* valeur *de* `3*x` *soit transmise à la fonction, cela n'a rien à voir avec la variable* `x`*.*

2. Remplacez `x` par `y` dans la fonction `main`. Qu'affiche ce programme ? Si vous trouvez cela choquant, dites-vous que c'est la même notion de variables libres/liées qu'en mathématiques.

   Réponse. *De même, la valeur de* `y` *dans le* `main` *n'a pas changée et vaut* 1*. Cela vient du fait que le* `y` *du* `main` *et le* `y` *de* `f` *sont deux variables différentes. L'argument* `y` *de* `f` *est* local *au code de* `f`*.*

Exercice 2. [La machine à café] En C, on peut lire une entrée clavier avec la fonction `scanf` et la stocker dans une variable. Pour lire un entier, on peut par exemple faire :

```c
    scanf("%d", &x); // x est une variable
```

1. Écrire une fonction `somme` qui demande à l'utilisateur un entier jusqu'à ce qu'il entre l'entier 0. La fonction retourne alors la somme de tous les entiers entrés jusque-là.
2. Écrire une fonction `machine_a_cafe` qui prend en argument un entier prix. La fonction demande ensuite à l'utilisateur de rentrer des pièces (des entiers) jusqu'à ce que la valeur prix soit dépassée. La fonction retourne finalement la monnaie due.

   Réponse. *On donne ici directement les deux fonctions :*

```c
#include <stdio.h>

int somme() {
  int x, s=0;
  do {
    scanf("%d",&x);
    s = s+x;
  } while (x != 0);
  return s;
}

int machine_a_cafe(int prix) {
  int x, s=0;
```

```
  do {
    scanf("%d",&x);
    s = s+x;
  } while (s < prix);
  return s-prix;
}

int main() {
  printf("Bonjour, je vais faire une somme\n");
  printf("La somme de vaut entier est %d\n",somme());
  printf("Bonjour, le cafe coute 30:\n");
  printf("Je vous rends %d\n", machine_a_cafe(30));
  return 0;
}
```

Exercice 3. [Le digicode] On rappelle qu'en C, si `a` et `b` sont entiers, on a `a/b` et `a%b` qui valent respectivement le quotient et le reste de la division euclidienne de `a` par `b`.

1. Écrivez une fonction `unite` qui étant donné un entier, renvoie le chiffre de ses unités (en base 10).

   Réponse. *Le chiffre des unités d'un entier est simplement le reste de la division euclidienne de cet entier par 10 d'où :*

```
int unit(int n) {
  return n % 10;
}
```

2. Écrivez une fonction `digicode_idiot` qui prend un entier `code` en argument. Cet entier, écrit en base 10, représente un code si on le lit de droite à gauche. Par exemple 1664 représente le code $4, 6, 6, 1$. Votre fonction doit demander successivement les chiffres du code à l'utilisateur. Dès que celui-ci se trompe de chiffre, la fonction doit afficher une erreur et inviter l'utilisateur à recommencer depuis le début.

   Réponse. *À chaque fois que l'utilisateur rentre un chiffre, on regarde s'il correspond au chiffre des unités du code courant. Si oui, on divise code par 10 pour passer au chiffre suivant. Si non, on affiche une erreur et on réinitialise le code initial. Attention alors à avoir stocké au préalable ce code initial !*

```
void digicode_idiot(int code) {
  int code_courant = code, x;
  while(code_courant != 0) {
    scanf("%d",&x);
    if (x == code_courant % 10) {
      code_courant = code_courant/10;
    }
    else {
      code_courant = code;
      printf("Access denied\n");
    }
  }
  printf("Access granted\n");
}
```

3. Si vous ne connaissez pas la valeur de `code` dans la fonction `digicode_idiot`, en combien de coups (en fonction de la longueur du code) pouvez-vous la trouver ? Expliquez alors le nom de cette fonction.

   Réponse. *Il faut* $10n$ *coups pour trouver le digicode où* n *est la taille du digicode. Il suffit pour cela de trouver le premier chiffre en au plus* 10 *coups (on sait que c'est le bon chiffre quand le programme n'affiche pas d'erreur), puis le deuxième etc. C'est donc assez rapide à trouver, d'où le nom.*

4. Écrire une fonction `digicode` qui prend un entier `code` en argument dont tous les chiffres sont distincts et accepte dès que le code apparaît dans la suite de chiffres entrés par l'utilisateur. Votre fonction marche-t-elle encore si `code = 421321` (donc le digicode est 1,2,3,1,2,4) ? Pourquoi ?

   Réponse. *Il suffit d'enlever l'indication d'erreur dans* `digicode_idiot` *pour que votre programme fonctionne sur des digicodes dont on sait que les chiffres seront distincts deux à deux. Cependant,*

si ce n'est pas le cas, `digicode_idiot` *ne fonctionnera pas. En effet, la séquence 1,2,3,1,2,3,1,2,4 devrait accepter alors que* `digicode_idiot` *va la refuser puisque on repart du début quand on entre le deuxième 3. Il faudrait recommencer à tester l'entrée au milieu du digicode.*

5. (difficile, à faire quand vous avez fini le reste du TP. Appelez-moi avant) Écrire une fonction `digicode` qui marche dans tous les cas.

   Réponse. *Nous y reviendrons quand nous ferons les chaînes de caractères. Si vous voulez un début de réponse, vous pouvez vous inspirer de l'algorithme KMP.*

Exercice 4. [Récurrez !] Une fonction est dite récursive si elle s'appelle elle-même. Vous pouvez rapprocher cela des suites définies par récurrence. Bien sûr, ces fonctions ont toujours des "cas de base" pour lesquels elles ne s'appellent pas elles-même, sinon le calcul ne se terminerait jamais. Pour résoudre des problèmes avec une fonction récursive, on fait souvent un raisonnement proche de celui qu'on fait en raisonnant par récurrence en mathématiques : si je sais faire ça pour tous les cas plus petits, comment puis-je résoudre ce cas-là ?

1. En remarquant que $0! = 1$ et $n! = n \times (n-1)!$ pour $n \geq 1$, écrivez une fonction `factorielle` qui calcule $n!$ sans utiliser de boucles. Faites de même pour `pgcd`.

   Réponse. *On écrit tout simplement la définition :*

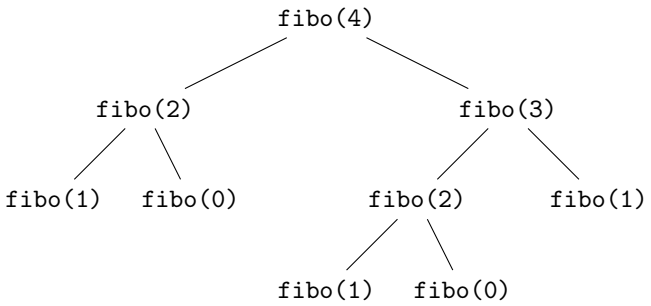
```c
int fact(int n) {
  if (n==0) return 1;
  else return (n*fact(n-1));
}
int pgcd(int a, int b) {
  if (b == 0) return a;
  else return pgcd(b, a%b);
}
```

2. Écrire `fibonacci` en récursif aussi et exécutez votre fonction pour $u_0 = u_1 = 1$ et $n = 40$. Que remarquez-vous par rapport à la version impérative ? À votre avis, pourquoi ? Écrivez `fibonacci` avec un seul appel récursif (indice : vous pouvez le faire en 3 lignes, pensez aux autres arguments !). Qu'en est-il des performances de cette nouvelle version ?

   Réponse. *On a naïvement envie d'appliquer la définition et d'écrire :*

```c
int fibo(int n, int u0, int u1) {
  if (n==0) return u0;
  else if (n==1) return u1;
  else return fibo(n-1,u0,u1)+fibo(n-2,u0,u1);
}
```

On remarque que pour $n = 40$ le programme met quelques secondes à calculer alors que la version impérative de la semaine dernière donne le résultat immédiatement. Pourquoi une telle différence de comportement ? Cela vient du fait qu'on calcule plusieurs fois les mêmes valeurs. En effet, dans l'appel récursif, `fibo(n-1)` *va à nouveau appeler* `fibo(n-2)`*. On calculera donc deux fois la valeur de* `fibo(n-2)` *alors que dans la version impérative, elle n'était calculée qu'une fois. Et c'est le cas pour toute les valeurs.* `fibo(n-3)` *va être appelée trois fois etc. Regardons par exemple ce qu'il se passe pour* `fibo(4)` *:*



En tout, on appellera la fonction `fibo` 2^n *fois, ce qui est bien plus que les n opérations que faisaient le programme impératif de la semaine dernière. Pour éviter cela, on peut écrire cette fonction plus intelligemment :*

```
int fibomalin(int u0,int u1,int n){
  if (n == 0)
    return u0;
  else
    return fibomalin(u1,u0+u1,n-1);
}
```

3. Écrire une fonction récursive qui étant donné n et b, écrit à l'écran $a_0 \dots a_k$ où $n = (a_k \dots a_0)_b$ est l'écriture de n en base b, c'est-à-dire $a_k \neq 0$, $0 \leq a_i < b$ et $\sum_{i=0}^{k} a_i b^i = n$.

   Réponse.

```
  void reverse(int n,int b) {
    if (n != 0) {
      printf("%d", n%b);
      reverse(n/b,b);
    }
  }
```

4. Les tours d'Hanoi. On se propose de jouer au jeu suivant : on a 3 piquets et sur le piquet le plus à gauche, on a empilé n anneaux de diamètres différents, par ordre croissant. On a le droit de déplacer n'importe quel anneau qui est en haut de la pile d'un piquet vers un autre piquet, mais on ne doit jamais poser un anneau sur un anneau plus petit. Le but du jeu est de déplacer les anneaux du piquet de gauche au piquet de droite. Écrivez une fonction qui, étant donné n, écrit une suite d'opérations à effectuer pour résoudre le problème. Combien d'opérations votre solution nécessite-t-elle ? Est-ce optimal ?

   Réponse. *Les tours d'Hanoi sont un exercice classique. Ils montrent à quel point on peut écrire facilement certaines fonctions en récursif. Pour résoudre les tours d'Hanoi, il faut d'abord observer que pour bouger le plus gros anneau à droite, il faut avoir bougé les $n-1$ anneaux au-dessus et il faut que le piquet de droite soit libre. Il faut donc dans tous les cas bouger les $n-1$ anneaux à gauche vers le milieu. Puis bouger les $n-1$ anneaux du milieu vers la droite. On va juste reprendre cette observation pour résoudre le problème en écrivant une première fonction* `hanoi_(int n, int d, int i, int a)` *qui déplacent n anneaux du piquet d vers le piquet a en utilisant le piquet i comme intermédiaire ; on suppose que nous numérotons les piquets de 1 à 3 où 1 est le piquet de gauche, 2 est le piquet du milieu et 3 celui de droite. Il suffit désormais d'échanger les rôles des piquets et d'appliquer l'observation précédente pour avoir :*

```
#include <stdio.h>

void hanoi_(int n, int d, int i, int a) {
  if (n > 0) {
    hanoi_(n-1,d,a,i);
    printf("Move %d to %d \n", d, a);
    hanoi_(n-1,i,d,a);
  }
}

void hanoi(int n) {
  hanoi_(n, 1, 2, 3);
}

int main() {
  hanoi(6);
  return 0;
}
```

   Cette stratégie est optimale puisque dans tous les cas, il faudra déplacer le gros anneaux de la gauche vers la droite et donc avoir stocké temporairement les $n-1$ autres sur le piquet du milieu. Si on note x_n le nombre de mouvements effectués pour résoudre le problème à n anneaux, on a $x_0 = 0$ et $x_{n+1} = 2x_n + 1$ d'où si on pose $y_n = x_n + 1$ on a : $y_{n+1} = 2y_n$ et $y_0 = 1$ d'où $y_n = 2^n$ et $x_n = 2^n - 1$.