ECUE «Prog 1» - CC3
5 janvier 2017 - Bruno Bouzy
sans document - durée 1 heure 30

CORRIGE

Dans les exécutions de programmes, les entrées clavier sont en **gras** et les sorties en police normale.

Exercice 1 (4 points)

Donner la sortie du programme ci-dessous. Pour chacune des trois lignes avec commentaire, tenir compte de la couleur (Bleu, Rouge, Vert ou Jaune) de votre copie pour utiliser la valeur précisée dans le commentaire de la ligne à la place de la valeur donnée en gras dans la ligne.

```
#include <stdio.h>
int main() {
  int a = +2;                    // Bleu:-1, Rouge:+2, Vert:-3, Jaune:+2
  int * p = &a;
  int b = *p;
  printf("1: a = %d, b = %d, *p = %d.\n", a, b, *p);
  a *= -4;                       // Bleu:+5, Rouge:-2, Vert:+3, Jaune:-4
  printf("2: a = %d, b = %d, *p = %d.\n", a, b, *p);
  b += -1;                       // Bleu:+2, Rouge:-3, Vert:+4, Jaune:-1
  printf("3: a = %d, b = %d, *p = %d.\n", a, b, *p);
  int * q = &b;  printf("4: a = %d, b = %d, *p = %d, *q = %d.\n", a, b, *p, *q);
  *q += (*p)++;  printf("5: a = %d, b = %d, *p = %d, *q = %d.\n", a, b, *p, *q);
  *q *= ++(*p);  printf("6: a = %d, b = %d, *p = %d, *q = %d.\n", a, b, *p, *q);
  p = q;         printf("7: a = %d, b = %d, *p = %d, *q = %d.\n", a, b, *p, *q);
  q = &a;        printf("8: a = %d, b = %d, *p = %d, *q = %d.\n", a, b, *p, *q);
  return(0);
}
```

Bleu

```
1: a = -1, b = -1, *p = -1.
2: a = -5, b = -1, *p = -5.
3: a = -5, b = 1, *p = -5.
4: a = -5, b = 1, *p = -5, *q = 1.
5: a = -4, b = -4, *p = -4, *q = -4.
6: a = -3, b = 12, *p = -3, *q = 12.
7: a = -3, b = 12, *p = 12, *q = 12.
8: a = -3, b = 12, *p = 12, *q = -3.
```

Rouge

```
1: a = 2, b = 2, *p = 2.
2: a = -4, b = 2, *p = -4.
3: a = -4, b = -1, *p = -4.
4: a = -4, b = -1, *p = -4, *q = -1.
5: a = -3, b = -5, *p = -3, *q = -5.
6: a = -2, b = 10, *p = -2, *q = 10.
7: a = -2, b = 10, *p = 10, *q = 10.
8: a = -2, b = 10, *p = 10, *q = -2.
```

Vert

```
1: a = -3, b = -3, *p = -3.
2: a = -9, b = -3, *p = -9.
3: a = -9, b = 1, *p = -9.
4: a = -9, b = 1, *p = -9, *q = 1.
5: a = -8, b = -8, *p = -8, *q = -8.
6: a = -7, b = 56, *p = -7, *q = 56.
7: a = -7, b = 56, *p = 56, *q = 56.
8: a = -7, b = 56, *p = 56, *q = -7.
```

Jaune

```
1: a = 2, b = 2, *p = 2.
2: a = -8, b = 2, *p = -8.
3: a = -8, b = 1, *p = -8.
4: a = -8, b = 1, *p = -8, *q = 1.
5: a = -7, b = -7, *p = -7, *q = -7.
6: a = -6, b = 42, *p = -6, *q = 42.
7: a = -6, b = 42, *p = 42, *q = 42.
8: a = -6, b = 42, *p = 42, *q = -6.
```

0.5 point par ligne correcte.

Exercice 2 (7 points)

1) Ecrire une procédure `void affiche` prenant un tableau `t` d'entiers et une taille `n` du tableau, et affichant les éléments du tableau en respectant les sorties du haut de la page suivante. **(1 point)**

```
// 1 point
void affiche(int * t, int n) {
  int i;
  printf("{ ");
  for (i=0; i<n; i++) printf("%d ", t[i]);
  printf("}\n");
}
```

2) Ecrire une procédure `void init` prenant en entrée un tableau `t` d'entiers et une taille `n` du tableau, et initialisant `t` avec des nombres entrés au clavier compris entre `0` et `9`. Tant qu'un nombre entré au clavier n'est pas compris entre `0` et `9`, la procédure redemande le nombre. La procédure respectera les entrées sorties du haut de la page suivante. **(2 points)**

```
// 2 points
void init(int * t, int n) {
  int i;
  for (i=0; i<n; i++) {
    do {
      printf("v%d ? ", i); scanf("%d", &t[i]);
    } while ((t[i]<0) || (t[i]>9));
  }
}
```

3) Ecrire une procédure `void taprosca` prenant en entrée deux tableaux `u` et `v` d'entiers et la taille `n` des deux tableaux, n'affichant rien, et donnant en sortie les trois produits scalaires `u.u`, `v.v`, et `u.v`, où les tableaux `u` et `v` sont considérés comme des vecteurs. **(2 points)**

Rappel mathématique: `u.v = Σu[i]v[i]`

```
// 2 points
void taprosca(int * u, int * v, int n, int * uu, int * vv, int *
uv)
{
  int i;
  *uu=0; *vv=0; *uv=0;
  for (i=0; i<n; i++) {
    *uu += u[i]*u[i];
    *vv += v[i]*v[i];
    *uv += u[i]*v[i];
  }
}
```

4) Ecrire un programme `main` déclarant deux tableaux `A` et `B` de taille `3`, initialisés avec `init`, affichés avec `affiche`, calculant avec `taprosca` les produits scalaires `A.A`, `B.B` et `A.B`, où les tableaux `A` et `B` sont considérés comme des vecteurs, et affichant les résultats. Le programme respectera les entrées-sorties du haut de la page suivante. **(2 points)**

```
// taprosca.c

#include <stdio.h>

#define TAILLE 3

// ici les fonctions

int main() {
  int A[TAILLE], B[TAILLE], aa, bb, ab;

  // 1 point
  printf("A ?\n");
  init(A, TAILLE);
  printf("A = ");
  affiche(A, TAILLE);
  printf("B ?\n");
  init(B, TAILLE);
  printf("B = ");
  affiche(B, TAILLE);

  // 1 point
  taprosca(A, B, TAILLE, &aa, &bb, &ab);
  printf("A.A = %d,\nB.B = %d,\nA.B = %d.\n", aa, bb, ab);
  return 0;
}
```

```
A ?
v0 ? 10
v0 ? 3
v1 ? -1
v1 ? 4
v2 ? 7
A = { 3 4 7 }
B ?
v0 ? 5
v1 ? 6
v2 ? 2
B = { 5 6 2 }
A.A = 74,
B.B = 65,
A.B = 53.
```

Exercice 3 (9 points)

La fonction de Ackermann `A` est définie comme suit. Pour `m` et `n`, deux entiers positifs ou nuls,

Si `m=0`, alors	`A(0,n)`	=	`n+1`
sinon si `n=0`, alors	`A(m,0)`	=	`A(m-1,1)`
sinon	`A(m, n)`	=	`A(m-1, A(m, n-1))`

1° Calculer `A(1,0)`, `A(1,1)`, `A(2,0)`. **(1 point)**

```
A(1, 0) = 2
A(1, 1) = 3
A(2, 0) = 3
0.33 point par résultat correct.
```

2° Avec la définition, programmer la fonction `ackermann` de manière récursive. **(1 point)**

```
int ackermann(int m, int n) {
  if (m==0) return n+1;
  else if (n==0) return ackermann(m-1, 1);
  else return ackermann(m-1, ackermann(m, n-1));
}
1 point pour l'ensemble
```

3° Programmer la fonction `ackermann_2` de manière récursive et analogue à la fonction `ackermann` de la question 2° en lui rajoutant un paramètre de sortie `int * count` donnant le nombre d'appels récursifs effectués. **(1 point)**

```
int ackermann_2(int m, int n, int * cnt) {
  int a, r1;
  (*cnt)++;
  if (m==0) {
    return n+1;
  }
  else if (n==0) {
    a = ackermann_2(m-1, 1, cnt);
    return a;
  }
  else {
    r1 = ackermann_2(m, n-1, cnt);
    a = ackermann_2(m-1, r1, cnt);
    return a;
  }
}
// 1 point pour utilisation correct du paramètre cnt
```

4° Calculer `A(1,2)` et `A(1,3)` et donner le nombre d'appels pour chaque calcul. **(1 point)**

```
A(1,2) = 4 et 6 appels
A(1, 3) = 5 et 8 appels

0.5 point par calcul et nombre d'appels corrects
```

5° Calculer `A(2,1)`. Pour ce calcul, `ackermann_2` effectue-t-elle deux fois certains calculs ? Lesquels ? **(1 point)**

```
A(2, 1) = 5
0.5 point.

Pour effectuer ce calcul, on a :
A(2, 1) = A( 1, A(2, 0))
A(2, 0) = A(1, 1) = 3

puis
A(1, 3) = A(0, A(1, 2))
A(1, 2) = A(0, A(1, 1))
A(1, 1) = 3
A(1, 2) = A(0, 3) = 4
A(1, 3) = A(0, 4) = 5

finalement
A(2, 1) = A(1, 3) = 5

Pour faire ce calcul, on a donc appelé - calculé - deux fois
A(1,1).

0.5 point
```

Pour éviter de faire faire deux fois certains calculs, on va utiliser un tableau d'entiers `tab` déclaré dans le `main` et stockant les valeurs de Ackermann déjà calculées jusque là. Au début de la fonction (nommée `ackermann_3` et appelée avec les paramètres `m` et `n`, le compteur `cnt_3` et le tableau `tab`), la fonction teste si `tab` contient déjà la valeur `A(m, n)`.

Si oui, elle retourne la valeur. Sinon, elle s'exécute et, avant de faire le `return`, elle stocke la valeur calculée dans le tableau `tab`.

On stocke `A(m, n)` dans `tab[N_MAX*m+n]` où `N_MAX` est une constante fixée à `200000` avec un `#define`. Avant l'appel initial de la fonction, le tableau `tab` est initialisé avec des `0`.

(Les valeurs de Ackermann étant strictement positives, une valeur de case du tableau strictement positive correspondra à une valeur déjà calculée, et une valeur nulle à une valeur non calculée.)

6° Programmer la fonction `ackermann_3` en rajoutant un paramètre en entrée `int * tab` où `tab` est le tableau contenant les valeurs déjà calculées jusque là.
(2 points)

```
#define GET_INDEX(X, Y)   N_MAX*X + Y
// si programmation directe sans #define get index, compter ok.

int ackermann_3(int m, int n, int * cnt, int * tab) {

  if (tab[GET_INDEX(m, n)]!=0) return tab[GET_INDEX(m, n)];

  (*cnt)++;

  if (m==0) {
    tab[GET_INDEX(m, n)] = n+1;
  }
  else if (n==0) {
    tab[GET_INDEX(m, n)] = ackermann_3(m-1, 1, cnt, tab);
  }
  else {
    int r1 = ackermann_3(m, n-1, cnt, tab);
    tab[GET_INDEX(m, n)] = ackermann_3(m-1, r1, cnt, tab);
  }
  return tab[GET_INDEX(m, n)];
}

// 0.5 point pour signature correcte de la fonction
// 0.5 point pour premier if
// 1 point pour deuxième if else if else et return corrects
```

7° Programmer un `main` déclarant les entiers `m`, `n`, `a`, `i`, `cnt_2`, `cnt_3`, le tableau d'entiers `tab` de taille `1000000` (définie par `TAILLE_TAB`), demandant au clavier les valeurs de `m` entre `0` et `3` (définie par `M_SUP`) et de `n` entre `0` et `12` (définie par `N_SUP`), appelant `ackermann_2` et `ackermann_3` de manière adéquate et affichant les résultats : `A(m, n)`, le nombre d'appels avec ou sans utilisation d'un tableau. Le programme respectera les entrées-sorties suivantes : **(2 points)**

```
// ackermann+tab.c
#include <stdio.h>
#include <stdlib.h>

#define M_SUP  3
#define N_SUP  12
#define N_MAX  200000
#define TAILLE_TAB 1000000
#define GET_INDEX(X, Y)   N_MAX*X + Y

// ici les fonctions.

int main() {
  int m, n, a, i, cnt_2=0, cnt_3=0;
  int A[TAILLE_TAB];

  for (i=0; i<TAILLE_TAB; i++) A[i]=0;

  do { printf("m ? "); scanf("%d", &m);
  } while ((m<0) || (m>M_SUP));

  do { printf("n ? "); scanf("%d", &n);
  } while ((n<0) || (n>N_SUP));

  a = ackermann_2(m, n, &cnt_2);

  a = ackermann_3(m, n, &cnt_3, A);

  printf("A(%d, %d) = %d, \ncnt sans = %d, \ncnt avec = %d.\n", m,
n, a, cnt_2, cnt_3);
  return 0;
}

// 1 point pour les appels corrects aux 2 fonctions
// 1 point pour l'ensemble du reste
```

```
m ? 1                m ? 2                m ? 3
n ? 0                n ? 2                n ? 6
A(1, 0) = 2,         A(2, 2) = 7,         A(3, 6) = 509,
n appels sans = 2,   n appels sans = 27,  n appels sans = 172233,
n appels avec = 2.   n appels avec = 15.  n appels avec = 1277.
```