E.N.S.E.I.R.B. Année 2004-2005
Filière Électronique $1^{ère}$ année
Filière Informatique $1^{ère}$ année
Filière Télécom $1^{ère}$ année

PROGRAMMATION IMPÉRATIVE - LANGAGE C

EXAMEN

CORRIGÉ

N.B. : - Ceci doit être considéré comme un corrigé-type : les réponses qu'il contient sont justes, mais leur rédaction n'était pas la seule possible.
- Le barème est donné à titre définitif. Outre l'exactitude des réponses aux questions posées, il a été tenu compte de leur concision et, dans une moindre mesure, de la présentation.

Question 1 (2 points)

Le langage C peut être vu comme un langage de bas niveau parce que nombre de ses instructions sont destinées à être traduites presque littéralement en instructions en langage machine, comme par exemple les « `++` », les décalages de bits, etc. Il ne possède pas de types de données évolués tels que les chaînes de caractères. Il ne possède pas non plus de mécanismes avancés que l'on retrouve dans les langages de haut niveau, comme la gestion automatique de la mémoire par « ramasse-miettes » (« `garbage collector` »), le contrôle des débordements de tableaux ou les exceptions, par exemple.

Le langage C peut être également considéré comme un langage de haut niveau parce que c'est un langage fortement typé qui permet la programmation structurée. La possibilité de définir de nouveaux types et les pointeurs de fonctions permettent de mettre en oeuvre des méthodes dynamiques similaires aux appels de méthodes de la programmation objet.

Question 2 (3 points)

(a) La forme « `char * p[];` »est valide pour déclarer un paramètre de fonction, mais n'est pas utilisable pour définir une variable globale, car aucune taille n'est spécifiée entre les crochets. La forme la plus générale, qui fonctionne dans tous les cas même si elle n'est pas aussi explicite que la précédente, est « `char ** p;` ».

(b) « `int t[5] = { 1, 2, 3 };` ».

(c) « `int * f (int, double);` ».

(d) « `int t[3][2] = { { 0, 1 }, { 0, 1 }, { 0, 1 }};` ».

(e) « `int (* f) (double);` ».

(f) Même configuration que pour le premier cas. La forme « `void (* p[]) (int);` » est moins générale que la forme « `void (** p) (int);` ».

Question 3 (3 points)

`a = 1, b = 1` La variable `a` vaut 1 après l'appel de `f1`. La fonction `f2` renvoie la valeur de `i`, qui est 1, la post-incrémentation étant perdue.

`0` L'appel à `f3` fait afficher la valeur 0 par cette dernière, puisque `(1 == 0)` est faux. La valeur retournée par `f3` est 1, qui est toujours la valeur de `a` après l'appel de `f3`.

`0` L'appel à `f4` fait afficher la valeur 0 par cette dernière, puisque la valeur de l'expression `i = 0` est bien 0. De plus, la valeur retournée par `f4` est alors 0 et non pas 1.

`a = 1, b = 0` Les valeurs affichées sont les deux valeurs retournées respectivement par `f3` et `f4`.

`a = 4, b = 3` La fonction `f5` pré-incrémente `a` avant de retourner sa valeur, qui sera de nouveau mise dans la variable `a` au retour de la procédure. La nouvelle valeur de `a` sera donc 2. La fonction `f6` commence par incrémenter la valeur de `a`, qui vaut alors 3, puis retourne cette valeur qui sera mise dans `b`, avant de post-incrémenter à nouveau `a`, qui vaut alors 4. Lors de l'affichage qui suit, la valeur de `a` sera donc 4.

Question 4 (4 points)

Les erreurs du programme sont les suivantes :

- Ligne 5 : du fait que le « `*` » n'appartient qu'à la variable `chaine`, la variable `mot_de_passe` est en fait définie ici comme un simple caractère. Il aurait donc fallu écrire : « `char * chaine, * mot_de_passe =...` ».
- Ligne 6 : la variable `chaine` est définie comme un pointeur sur un caractère (donc est un pointeur sur une chaîne de caractères), mais aucun espace n'est réservé pour celle-ci. Il aurait fallu soit faire une allocation dynamique, soit écrire : « `char chaine[64];` ».
- Ligne 7 : la comparaison s'effectue sur les valeurs des pointeurs au lieu de s'effectuer sur les contenus. Il aurait fallu écrire : « `if (strcmp (chaine, mot_de_passe) == 0)` ».
- Ligne 8 : il manque le point-virgule à la fin de la ligne.
- Ligne 11 : il manque les accolades pour que le « `return (1)` » se fasse seulement dans le cas du « `else` ».
- Ligne 12 : il manque un « `return (0)` » pour que la fonction retourne toujours une valeur. Pour une meilleure compréhension, cette instruction peut se mettre dans le corps du `if` de la ligne 7, en ajoutant ici aussi des accolades.

Question 5 (8 points)

(5.1) (1 point)

Une définition du type structuré `Chrono` est :

```
typedef struct Chrono_ {
  int         minutes;          /* Domaine : [0-...] */
  int         secondes;         /* Domaine : [0-59]  */
  int         centiemes;        /* Domaine : [0-99]  */
} Chrono;
```

Définir le type `Crono` comme un tableau d'entiers aurait été moins expressif, car ici chaque valeur (minutes, secondes et centièmes) a un domaine de validité distinct des autres. Définir le type `Chrono` comme un pointeur sur une `struct Chrono_` aurait pu être dangereux, car on peut par la suite oublier que le type est en fait un pointeur, et que si, dans une fonction, on définit une variable de type `Chrono`, elle n'est pas directement utilisable car il faut ensuite allouer la place mémoire destinée au stockage des valeurs, et ne pas oublier de la libérer une fois qu'on en n'a plus besoin.

(5.2) (2 points)

Pour plus d'efficacité, on passera tous les paramètres par référence, sous la forme de pointeurs

vers des structures `Chrono`. Afin de donner au code la plus grande souplesse possible, la structure destination de l'ajout est distincte des deux temps à ajouter, mais on a fait en sorte qu'elle puisse être égale à l'un des deux.

```
void
chronoAjoute (
Chrono *      dest,
Chrono *      src1,   /* On peut avoir dest = src1 */
Chrono *      src2)   /* On peut avoir dest = src2 */
{
  int           minutes;
  int           secondes;
  int           centiemes;

  minutes   = src1->minutes   + src2->minutes;
  secondes  = src1->secondes  + src2->secondes;
  centiemes = src1->centiemes + src2->centiemes;

  if (centiemes >= 100) {      /* À faire dans le bon ordre */
    centiemes -= 100;
    secondes  ++;
  }
  if (secondes >= 60) {
    secondes -= 60;
    minutes  ++;
  }
  dest->minutes   = minutes;   /* On peut modifier une des sources */
  dest->secondes  = secondes;
  dest->centiemes = centiemes;
}
```

Une autre possibilité, pour définir l'interface des routines de la classe `Chrono`, est de ne passer que des `Chrono` plutôt que des `Chrono *`. Dans ce cas, le prototype de la fonction serait : « `Chrono chronoAjoute (Chrono, Chrono) ;` », et il faudrait faire de même pour les routines suivantes, pour conserver l'homogénéité des interfaces.

(5.3) (2 points)

Une écriture possible de la fonction `chronoCompare()`, avec nos conventions de départ, est :

```
int
chronoCompare (
Chrono *      src1,
Chrono *      src2)
{
  if (src1->minutes > src2->minutes)
    return (1);
  if (src1->minutes < src2->minutes)
    return (-1);
  if (src1->secondes > src2->secondes)
    return (1);
  if (src1->secondes < src2->secondes)
    return (-1);
  if (src1->centiemes > src2->centiemes)
    return (1);
  if (src1->centiemes < src2->centiemes)
    return (-1);
  return (0);
}
```

Il existe aussi une autre solution, très élégante : convertir les deux temps en centièmes et les comparer sur la base d'une unique valeur entière. Pour limiter les risques de débordement, ces deux valeurs seront de type `long`, et l'on calculera la différence entre chaque paire de valeurs avant d'effectuer les multiplications :

```
int
chronoCompare (
Chrono *      src1,
Chrono *      src2)
```

```
{
  long          diffcentiemes;

  diffcentiemes = (long) (src1->minutes   - src2->minutes)  * 6000L +
                  (long) (src1->secondes  - src2->secondes) * 100L  +
                  (long) (src1->centiemes - src2->centiemes);

  if (diffcentiemes > 0)
    return (1);
  if (diffcentiemes < 0)
    return (-1)
  return (0);
}
```

(5.4) (3 points)

Comme on a à travailler sur quatre valeurs, il est préférable de faire une boucle ; cela simplifiera aussi grandement la recherche, au fur et à mesure, des temps minimum et maximum. Les processus qui se répèteront, et qu'il sera bon de déplacer dans des fonctions annexes pour la clarté du code sont :

– la saisie d'un temps ;

– l'affichage d'un temps.

En conservant les conventions de codage précédentes, on a donc :

```
void
chronoSaisit (
Chrono *      src)
{
  printf ("Entrez les minutes : ");   /* Aucune vérification faite */
  scanf  ("%d", &src->minutes);       /* À faire en option         */
  printf ("Entrez les secondes : ");
  scanf  ("%d", &src->secondes);
  printf ("Entrez les centièmes : ");
  scanf  ("%d", &src->centiemes);
}

void
chronoAffiche (
Chrono *      src)
{
  printf ("%d'%d\"%d\n",
          src->minutes,
          src->secondes,
          src->centiemes);
}

main ()
{
  Chrono        somme;
  Chrono        max;
  Chrono        min;
  int           i;              /* Boucle sur les relayeurs */

  printf ("Entrez le temps du relayeur numéro 1 :\n");
  chronoSaisit (&somme);        /* Initialisation des compteurs */
  min =
  max = somme;

  for (i = 2; i <= 4; i ++) {  /* Boucle sur les autres relayeurs */
    Chrono        courant;

    printf ("Entrez le temps du relayeur numéro %d :\n", i);
    chronoSaisit (&courant);
    chronoAjoute (&somme, &somme, &courant);
    if (chronoCompare (&courant, &min) == -1)
      min = courant;
    else if (chronoCompare (&courant, &max) == 1) /* Soit l'un, soit l'autre */
      max = courant;
```

```
  }
  printf ("Temps total : ");
  chronoAffiche (&somme);
  printf ("Meilleur temps : ");
  chronoAffiche (&min);
  printf ("Moins bon temps : ");
  chronoAffiche (&max);

  return (0);
}
```