Durée 3 h Documents autorisés (polycopiés de cours uniquement)

Sujet d'examen 1

Pratique du C

Février 2012

Introduction

Écrivez lisiblement et n'hésitez pas à commenter votre code en langage C. Vous ne pouvez utiliser que les fonctions C dont le prototype est donné dans l'énoncé et celles dont vous donnez la définition dans vos copies.

Les sections sont indépendantes ; lisez l'énoncé complet avant de commencer à le résoudre.

1 Quizz

1. Qu'affiche le programme suivant :

```
#include<stdio.h>
#define PRECEDENTE 1
#define ACTUELLE 2
#define PROCHAINE 3
int
main
(void)
{
  int annee = ACTUELLE;
  switch (annee)
  {
   case PRECEDENTE: annee = 2007;
   case ACTUELLE: annee = 2008;
   case PROCHAINE: annee = 2009;
   default:annee = 2010;
  }
  printf("Bonne annee %d\n", annee);
  return 0;
}
```

Correction. Ce programme affiche

```
% ./a.out
Bonne annee 2010
%
```

En effet, aucun `break` n'est utilisé dans le `switch`. Donc, toutes les instructions après `ACTUELLE` sont exécutées.

2. Donnez l'affichage produit par l'exécution du programme suivant :

```
#include<stdio.h>
char *f0 (char *a[][3]){ return (**a); }
char *f1 (char *a[][3]){ return (*(*(a+1)+2)); }
int main (void){
  char *(*fcts[2]) (char *tableau[][3]);
  static char *tableau[][3] =
          {{"0","1","2"},
           {"3","4","5"},
           {"6","7","8"}};
  fcts[0] = &f0;
  fcts[1] = &f1;
  printf("%s\n", (**fcts)(tableau));
  printf("%s\n", (**(fcts+1))(tableau+1));
  return 0 ;
}
```

```
0
8
```

2 Exercice sur les classes d'allocations

Considérons le fichier source `aux.c` suivant :

```
#include <stdio.h>
int x;
static int y;
void affectation(void) { x = 2; y = 3; }
void choix(int a) {
    int x = 1;
    static int y;
    if (a == 0)
        y = 0;
    else y += x;
    printf("choix: %d %d\n", x, y);
}

void proc1(void) { printf("proc1: %d %d\n", x, y); }
```

et le fichier source `main.c`

```
#include <stdio.h>
extern int x;
static int y;

void proc2(void)
{ printf("proc2: %d %d\n", x, y); }

int main(void) {
    choix(0);
    affectation();
    choix(1);
    proc1();
    proc2();
    return 0 ;
}
```

Questions :

1. Que manque-t'il au fichier source `main.c` en vue de la production d'un code objet ?
2. Donnez un makefile permettant d'obtenir un fichier exécutable à partir des fichiers sources `main.c` et `aux.c`.
3. Donnez la sortie produite par l'exécution du fichier ainsi obtenu.

Correction.

1. il manque les déclarations des fonctions définies dans `aux.c`

```
extern void affectation(void) ;
extern void choix(int) ;
extern void proc1(void) ;
```

2.
```
executable: aux.o main.o
	gcc -o executable aux.o main.o
aux.o: aux.c
	gcc -c aux.c
main.o: main.c
	gcc -c main.c
```

3.
```
choix: 1 0
choix: 1 1
proc1: 2 3
proc2: 2 0
```

3 Une méthode de cryptage : le renversement des fréquences

Étant donné un texte codé par une chaîne de caractères `char *text`, on se propose de le coder par le remplacement de chacune des lettres du texte clair par un autre caractère fourni par une table (cf. figure 1 page 10). Contrairement au chiffrement de César, on souhaite faire disparaître les indications fournies par les fréquences d'apparitions des lettres.

3.1 Calcul des fréquence des lettres d'un texte

Pour commencer, on se propose de construire une liste chaînée dont chaque cellule contient un des caractères présents dans le texte et sa fréquence d'apparition (le nombre d'occurences divisé par le nombre total de caractères dans le texte, le tout multiplié par 100). Au final, cette liste chaînée doit être ordonnée : le premier (en tête) élément étant la cellule associée à la lettre la moins fréquente et le dernier (en queue) à la lettre la plus fréquente.

Question.

1. Donnez la déclaration `struct cellule_s` permettant de représenter une cellule décrite ci-dessus. Pour la suite du problème, on souhaite ajouter à cette déclaration un champs `struct cellulecode_s *listecirculaire` dont nous verrons l'usage dans la section 3.3.

   Correction.

   ```
   struct cellule_s
   {
     char caractere ;
     float frequence ;
     struct cellulecode_s *listecirculaire ;
     struct cellule_s *next ;
   }
   ```

2. Donnez la définition de la fonction de prototype :

   ```
   struct cellule_s *analyseDeFrequence(char *);
   ```

   qui prend en paramètres le texte à analyser et qui retourne la liste chaînée décrite ci-dessus non triée (comme décrit ci-dessus). Les pointeurs `listecirculaire` devront être affectés à `NULL`.

   Correction.

   ```
   struct cellule_s *
   analyseDeFrequence
   (char *txt)
   {
     register int  i ;
     float nbtotal ;
     char occurencelettres[255] ;
     struct cellule_s *head,*tmp ;

     /* on commence par faire des statistiques */
     for(i=0 ; i<256 ; i++)
       occurencelettres[i]=0;

     while(*txt)
       occurencelettres[(int) (*(txt++))]++ ;

     nbtotal = 0 ;
   ```

```
  for(i=0 ; i<256 ; i++)
    nbtotal += occurencelettres[i];

  /* on construit maintenant la chaine */
  head = 0 ;
  for(i=0 ; i < 256 ; i++)
    if(occurencelettres[i])
      {
tmp = (struct cellule_s *) malloc(sizeof(struct cellule_s));
tmp->next = head ;
tmp->caractere = (char) i ;
tmp->frequence = 100.*((float) occurencelettres[i])/nbtotal ;
tmp->listecirculaire = NULL ;
head = tmp ;
      }

  return head ;
}
```

3. Donnez la définition de la fonction de prototype :

   ```
   struct cellule_s *TriSuivantFrequence(struct cellule_s *);
   ```

   qui prend en paramètre une liste chaînée non triée et qui retourne une liste chaînée triée (comme décrit ci-dessus). (Vous êtes libre de la méthode ; si votre tri ne détruit pas la liste d'origine vous devrez tout de même libérer la mémoire correspondante).

   Correction.

```
struct cellule_s *
TriSuivantFrequence
(struct cellule_s **oldhead)
{
  struct cellule_s *newhead, *tmp, *newcell, *previouscell, *newend ;
  float minfreq ;
  newhead = NULL ;
  while(*oldhead)
    /* tant que la liste non triee est non vide */
    {
      tmp = *oldhead ;
      newcell=tmp ;
      previouscell= 0;
      /* on cherche la cellule de frequence minimale */
      minfreq = tmp->frequence ;
      while(tmp->next)
{
  if(tmp->next->frequence < minfreq)
    {
      minfreq = tmp->next->frequence ;
      previouscell = tmp ;
      newcell = tmp->next ;
    }
  tmp = tmp->next ;
}
      /* on la retire de la liste */
      if(previouscell)
previouscell->next = newcell->next ;
```

```
        else *oldhead = newcell->next ;
        /* on la rajoute a la fin de la liste triee */
        if(newhead)
    {
      newend -> next = newcell ;
      newend = newcell ;
    }
        else newend = newhead = newcell ;
       }
      return newhead ;
    }
```

3.2 Intermezzo : empilement d'une suite aléatoire

On suppose disposer d'une macro `MaxPile` et de ne pas avoir besoin de plus de `MaxPile` caractères (dans la suite cette macro désigne 255). Nous allons utiliser une pile codée par un tableau afin d'empiler des caractères. Cette pile utilise la structure :

```
struct pile_s{
      unsigned int sommet ;
      char tab[MaxPile] ;
} ;
```

La pile vide correspond à une valeur nulle de `sommet`.

Question.

1. Donnez la définition de la fonction de prototype :

   ```
   int empile(char, struct pile_s *) ;
   ```

   qui empile l'entier donné en premier paramètre sur la pile fournie en second. Cette fonction retourne 0 si la pile est pleine, 1 sinon.

   Correction.

   ```
   int
   empile
   (char entier, struct pile_s *mapile)
   {
     if(mapile->sommet==MaxPile)
       return 0  ;
     mapile->tab[mapile->sommet] = entier ;
     mapile->sommet++ ;
     return 1 ;
   }
   ```

2. Donnez la définition de la fonction de prototype :

   ```
   int depile(struct pile_s *,char *) ;
   ```

   qui dépile l'entier du sommet de la pile fournie en paramètre et le retourne dans son second paramètre. Cette fonction retourne 0 si la pile est vide, 1 sinon.

   Correction.

   ```
   int
   depile
   (struct pile_s *mapile, char *entier)
   {
   ```

```
  if(!mapile->sommet)
    return 0 ;
  mapile->sommet-- ;
  *entier = mapile->tab[mapile->sommet] ;
  return 1 ;
}
```

On suppose que la fonction `int constrand(struct pile_s *)` remplit la pile par des entiers aléatoires tous distincts en retournant 1 si c'est possible, 0 sinon.

3.3 Construction et utilisation d'une table de symboles de substitution

Il nous faut maintenant construire la clef de chiffrement qui est le secret utilisé pour coder et décoder le texte. Étant donnée une liste chaînée `struct cellule_s *code`, nous allons adjoindre à chaque lettre i.e. à chaque cellule une liste circulaire composée de cellule de déclaration `struct cellulecode_s`. Ces structures sont composées de 2 champs : un caractère n et un pointeur `struct cellulecode_s *`.

Une liste circulaire est une liste dans laquelle on a remplacé le pointeur NULL du dernier élément de la liste par un pointeur vers la tête de liste (le premier élément).

Question.

1. Donnez la définition de `struct cellulecode_s`.

   Correction.

   ```
   struct cellulecode_s
   {
     char n ;
     struct cellulecode_s * next;
   } ;
   ```

2. Donnez la définition de la fonction de prototype :

   ```
   void construirecode(struct pile_s *,struct cellule_s *) ;
   ```

   qui prend en argument la pile d'entiers aléatoires construite dans la section 3.2 et la liste chaînée d'analyse des fréquences construites dans la section 3.1.

   Cette fonction parcours la liste d'analyse et à chaque cellule c :
   – détermine l'entier n immédiatement supérieur à la fréquence de c (on rappelle que la conversion de type explicite de flottant machine à entier revient à supprimer la partie décimale dans notre contexte) ;
   – construit une liste circulaire de n cellules, chaque cellule contenant un entier dépilé de la pile des entiers aléatoires (si cette pile est vide, le programme se termine par un `exit(2)`) ;
   – fait pointer `listecirculaire` de c sur une cellule de la liste circulaire ainsi construite.

   Correction.

   ```
   void
   construirecode
   (struct pile_s *pile, struct cellule_s *head)
   {
     int n ;
     struct cellulecode_s *tmp ;

     while(head)
       {
         n = 1+(int) head->frequence ;
         while(n--)
   ```

```
{
  tmp = (struct cellulecode_s *) malloc(sizeof(struct cellulecode_s));
  if(!depile(pile,&(tmp->caractere)))
    exit(2) ;

  if(!head->listecirculaire)
    {
      head->listecirculaire=tmp ;
      tmp->next = head->listecirculaire ;
    }
  else
    {
      tmp->next = head->listecirculaire->next ;
      head->listecirculaire->next = tmp ;
    }
}
      head=head->next ;
    }
}
```

3. Donnez le prototype et la définition de la fonction `clean` qui libère la mémoire associée à une clef de chiffrement (représentée ici par la liste chaînée construite par la fonction `construirecode`).

   Correction.

```
void
clean
(struct cellule_s *head)
{
  struct cellulecode_s *tmp  ;
  if(head)
    {
      clean(head->next) ;
      if(head->listecirculaire)
while(head->listecirculaire->next!=head->listecirculaire)
  {
    tmp = head->listecirculaire->next ;
    head->listecirculaire->next = tmp->next ;
    free(tmp) ;
  }
      free(head->listecirculaire) ;
      free(head) ;
    }
}
```

4. Donnez la définition de la fonction de prototype :

   `char *codage(char *,struct cellule_s **) ;`

   qui prend en paramètre un texte et un pointeur pour stocker le code associé (avant l'appel de cette fonction, il n'y a pas de mémoire réservée pour ce code).
   Cette fonction regroupe les étapes de ce problème (analyse, construction de pile d'entier, construction de la clef). Elle retourne un pointeur sur un texte chiffré par la méthode proposée (bien que le pointeur retourné soit de type `char *`, ce texte est 4 fois plus long que l'original car physiquement constitué d'entiers `int`).
   Si la clef est donnée par la grille de la figure 1 page 10, le mot *abracadabra* sera chiffré par la suite d'entiers "33 24 92 15 03 96 61 55 24 21 57" i.e. chaque fois que l'on désire coder une

lettre, on utilise un entier de la liste circulaire et au prochain codage de cette même lettre, on utilise l'entier suivant.

Correction.

```
char *
codage
(char *texte, struct cellule_s **clef)
{
  int i ;
  struct cellule_s *grille, *foo ;
  char * res,*tmp ;
  struct pile_s *pile = initpile() ;
  for(i=0; texte[i] ; i++) ;
  res = (char *) malloc(i) ;

  for(i=33; i<MaxPile; i++)
    empile(i,pile) ;

  grille = analyseDeFrequence(texte) ;
  grille = TriSuivantFrequence(&grille) ;
  construirecode(pile,grille) ;

  affichegrille(grille) ;
  *clef = grille ;

  /* le codage proprement dit */
  tmp = res ;
  while(*texte)
    {
      foo=grille ;
      while(foo->caractere!=*texte)
foo=foo->next ;
      *tmp = foo->listecirculaire->caractere ;
      foo->listecirculaire=foo->listecirculaire->next ;
      tmp++ ;
      texte++ ;
    }
  return res ;
}
```

5. Critiquez l'usage de liste circulaire et proposez un codage plus simple en le justifiant.

   Correction. L'utilisation de liste chaînée en général dans cet exercice est purement pédagogique, puisque l'on connaît a priori le nombre de caractères a utiliser dans chaque situation, on devrait plutôt utiliser des tableaux (un calcul modulaire sur l'indice permettant d'implanter une liste circulaire avec un tableau).

Fonctions auxiliaires. L'allocation mémoire se fait par le biais de la fonction `malloc` et la désallocation par le biais de la fonction `free`.

Une grille [1] de substitution pour la langue française qui utilise tous les nombres de 00 à 99.

Lettre	Fréquence	Symboles de substitution
a	8.40 %	33, 15, 37, 96, 55, 57, 72, 91
b	1.06 %	24
c	3.03 %	03, 39, 67
d	4.18 %	61, 04, 43, 88
e	17.26 %	08, 12, 20, 46, 47, 48, 53, 59, 64, 76, 79, 80, 81, 85, 90, 94, 97
f	1.12 %	40
g	1.27 %	29
h	0.92 %	05
i	7.34 %	14, 45, 50, 73, 82, 93, 99
j	0.31 %	11
k	0.05 %	77
l	6.01 %	01, 16, 26, 60, 71, 98
m	2.96 %	34, 87
n	7.13 %	06, 17, 22, 30, 31, 49, 58
o	5.26 %	02, 10, 41, 66, 89
p	3.01 %	13, 18, 83
q	0.99 %	36
r	6.55 %	92 21, 25, 65, 68, 95
s	8.08 %	00, 28, 51, 52, 63, 74, 78, 84
t	7.07 %	07, 19, 23, 35, 38, 54, 70
u	5.74 %	09, 32, 42, 69, 75
v	1.32 %	44
w	0.04 %	56
x	0.45 %	86
y	0.30 %	62
z	0.12 %	27

FIGURE 1 – Grille de codage