Durée 1 h 30 Documents non autorisés

Sujet d'examen 1

Pratique du C

Octobre 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. Quel est l'affichage produit par l'exécution du fichier produit par la compilation du source suivant :

```
#include<stdio.h>
enum bool {false, true } ;

int main(void){
  enum bool test ;
  int i,j ;
  for (i = 2; i < 10; i++){
   test = false ;
   j    = 2     ;
    while (!test && j < i){
      if (i % j == 0)
        test = true;
      j++;
    }
    if (!test)
      putchar(i+'0') ;
  }
  putchar('\n');
  return 0 ;
}
```

Correction. La correction est binaire, soit on trouve :

2357,

soit c'est 0.

2. (a) Que retourne au shell l'exécutable produit par le code source suivant :

```
void CalculAireRectangle (int longueur, int largeur, int AireRectangle){
AireRectangle = longueur * largeur;
}

int main (void){
     int longueur = 5;
     int largeur = 4;
     int AireRectangle = 0;
     CalculAireRectangle(longueur, largeur, AireRectangle);
     return AireRectangle ;
}
```

   (b) Modifier ce programme pour que sa valeur de retour corresponde bien à l'aire du rectangle définit par les longueur et largeur spécifiées.

   (a) Ce code retourne la valeur 0.

   (b) il suffit de faire un passage par adresse

```
void CalculAireRectangle
(int longueur, int largeur, int *AireRectangle)
{
*AireRectangle = longueur * largeur;
}

int main (void){
     int longueur = 5;
     int largeur = 4;
     int AireRectangle = 0;
     CalculAireRectangle(longueur, largeur, &AireRectangle);
     return AireRectangle ;
}
```

2 Nombres de Catalan

La suite de Catalan $(C_j)_{j\in\mathbb{N}}$ intervient dans de nombreux problèmes de combinatoire. Ces nombres vérifient — entre autres — la récurrence de Segner (présentée en 1758, cette récurrence n'est pas la plus simple expression pour les nombre de Catalan) pour $n > 0$:

$$C_0 := 1, \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}.$$

1. Construire une fonction récursive `Catalan` qui prend en argument un entier non signé n et qui retourne l'entier non signé C_n.

   Correction.

```
long unsigned int
catalan
(unsigned int n)
{
long unsigned int res ;
        register unsigned int i ;
fprintf(stderr,"calcul de catalan(%d)\n",n) ;
if(!n)
```

```
return 1 ;

for(res=i=0;i<n;i++)
res += catalan(i)*catalan(n-1-i) ;

return res ;
}
```

Malheureusement, une approche naïve se heurte à une trop grande complexité (le nombre d'appel est trop grand). Pour résoudre ce problème, on se propose d'utiliser un entier `next2compute` initialisé à 0 et un tableau `catalannumbers` stockant les `next2compute` valeurs déjà calculées (on suppose que la taille maximale de ce tableau est 37 — au delà, les nombres de Catalan dépassent la taille maximale d'un entier machine non signé).

2. Construire une fonction récursive `Catalan2` qui prend en argument un entier non signé n et qui retourne l'entier non signé C_n mais cette fois en utilisant le tableau `catalannumbers` :
   – si le nombre de Catalan utilisé est déjà calculé, on le cherche dans le tableau ;
   – sinon, on fait un appel récursif pour que la fonction le calcule.

   Correction.

```
#define NB 37

long unsigned int
catalan2
(unsigned int n)
{
long unsigned int res ;
        register unsigned int i ;
static long unsigned int catalannumber[NB] ;
static unsigned int next2compute ;

if(!n)
res = 1 ;
else
res = 0 ;

for(i=0;i<n;i++)
res += ((i<next2compute)?catalannumber[i]:catalan(i))
*((n-1-i<next2compute)?catalannumber[n-1-i]:catalan(n-1-i)) ;

catalannumber[next2compute++] = res ;
return res ;
}
```

Information culturelle. Étant donné un produit de n nombres, il y C_n façons de parenthéser le produit en regroupant deux termes consécutifs (sans changer l'ordre) parmi les n donnés (deux termes ou plus ne peuvent rester sans parenthèses).

3 Carré magique

Un carré magique d'ordre n est composé des n^2 premiers entiers (distincts) écrits sous la forme d'une matrice carrée et tels que la sommes des nombres soient égales sur chaque lignes, colonnes et sur les 2 diagonales principales. Par exemple, on a :

$$\begin{array}{cccccc} & & 2 & 7 & 6 & \rightarrow & 15 \\ & & 9 & 5 & 1 & \rightarrow & 15 \\ & & 4 & 3 & 8 & \rightarrow & 15 \\ & \swarrow & \downarrow & \downarrow & \downarrow & \searrow & \\ 15 & & 15 & 15 & 15 & & 15 \end{array}$$

Donnez la définition de la fonction de prototype :

```
int EstMagique(unsigned int ** tab, unsigned int n) ;
```

qui retourne 1 si le carré magique `tab` d'ordre `n` est magique et qui retourne 0 sinon.

Correction.

```
#include "CarreMagique.h"

int
EstMagique
(unsigned int tab[TABSIZE][TABSIZE], unsigned int n)
{
  unsigned int h,i,j,res ;

  /* v\'erifions que toutes les sommes sont \'egales */

  /* commen\c{c}ont par une diagonale */

  for(res=i=0;i<n;i++)
    res +=tab[i][i] ;

  /* v\'erifions avec l'autre diagonale */
  for(j=i=0;i <n;i++)
    j += tab[n-1-i][i] ;

  if(j!=res)
    return 1==0 ;

  /* puis les lignes  */
  for(i=0;i<n;i++)
    {
      for(h=j=0;j<n;j++)
h+=tab[i][j] ;
      if(h!=res)
return 1==0 ;
    }

  /* puis les colonnes  */
  for(i=0;i<n;i++)
    {
      for(h=j=0;j<n;j++)
h+=tab[j][i] ;
```

```
      if(h!=res)
return 1==0 ;
    }

  /* v\'erifions que tous les nombre de 1 \`a n^2 sont pr\'esents */
  /* soit on fait du malloc
     int * complement = (int *) malloc(sizeof(int)*n*n) ;
     for(i=0;i<n*n;i++)
       complement[i]=0 ;
     for(i=0;i<n;i++)
       for(j=0; j<n; j++)
          complement[tab[i][j]] = 1 ;

     for(j=1,i=0;i<n*n;i++)
         j = j || complement[i] ;

     if(!j)
        return 1==0 ;

     soit on ne fait pas du malloc et on met de la complexit\'e pour compenser l'espace
   */
  for(res=h=0;h<n*n ; h++)
    {
      for(i=0 ;i<n ;i++)
for(j=0 ;j<n; j++)
  if(tab[i][j] == h+1)
    res++ ;
    }
  if(res!=n*n)
    return 1==0 ;


  return 1==1 ;
}
```

4 Utilisation du clavier d'un téléphone en mode SMS

Énoncé

Vous avez déjà remarqué que le clavier d'un téléphone outre la touche #, ressemblait à :

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ

Cette disposition permet à l'utilisateur de taper du texte. Par exemple, presser la touche 7 une fois correspond à la lettre P alors que presser cette touche 4 fois correspond à la lettre S. La touche # du téléphone sert à séparer les lettres et peut être omise lorsqu'il est clair qu'une lettre se termine et l'autre commence.

Par exemple, la séquence

`777666222559996#6668866#8244466`

correspond donc au mot

`ROCKYMOUNTAIN`

Question. Donnez une fonction qui prend en paramètre une chaîne de caractères — stockées dans un tableau — correspondant au texte tapé en mode SMS (comme la séquence ci-dessus) et qui affiche le texte correspondant sur la sortie standard

Indications. Le prototype de la fonction `int putchar(int)` est défini dans le fichier d'entête `#include <stdio.h>`. Cette fonction prend en argument un entier, le converti en caractère et l'affiche sur la sortie standard et retourne ce caractère en cas de succès ou EOF en cas d'échec.

Correction.

Proposition de correction

En un seul bloc, cela donne

```
#include <stdio.h>
struct data_m
{
char chiffre ;
int iteration ;
char modulo ;
};
typedef struct data_m data_t ;
void
Traduction
(char *texte)
{
data_t data ;
char tab[10] = "  ADGJMPTW" ;
if(!*texte)
   return ;
if((*texte<'1' || *texte>'9') && *texte!='#')
   Traduction(texte+1) ;
data.chiffre = *texte;
data.iteration = 0 ;
if(*texte=='1')
  data.modulo = 1 ;
else
  data.modulo=(*texte=='9' || *texte=='7')?4:3 ;
texte++ ;
while(1)
{
if(!*texte)
{
putchar(tab[data.chiffre-'0']+(data.iteration % data.modulo)) ;
break ;
}
if((*texte<'1' || *texte>'9') && *texte!='#')
{
  texte++ ;
```

```
  continue ;
}
if(*texte==data.chiffre)
{
  data.iteration++ ;
  texte++;
  continue ;
}
while(*texte=='#')
  texte++ ;
if(!*texte)
  continue ;
putchar(tab[data.chiffre-'0']+(data.iteration % data.modulo)) ;
if(*texte>'0' && *texte<'9'+1)
{
data.chiffre = *texte;
data.iteration = 0 ;
if(*texte=='1')
  data.modulo = 1 ;
else
  data.modulo=(*texte=='9' || *texte=='7')?4:3 ;
}
texte++ ;
}
putchar('\n');
}
```

À vous d'estimer comment la réponse de l'étudiant remplit les objectifs.

Détails de cette correction

Pour définir le code, nous allons suivre les étapes :

⟨*⟩≡
  ⟨*Entête*⟩
  ⟨*Structure de donnée*⟩
  ⟨*FonctionDeTraduction*⟩

Il ne reste plus qu'à écrire la fonction de traduction en se basant sur le squelette suivant :

⟨*FonctionDeTraduction*⟩≡
  ⟨*EntêteDeFonction*⟩
  {
  ⟨*Variables automatiques*⟩
  ⟨*conversion*⟩
  putchar('\n');
  }

Cette fonction d'identificateur `Traduction` prend une chaîne de caractères en paramètre et ne retourne rien :

⟨*EntêteDeFonction*⟩≡
  void
  Traduction
  (char *texte)

Gestion de l'entrée

On commence par s'assurer que l'entrée correspond bien à nos attentes.

Si la chaîne donnée en paramètre est vide, on quitte la fonction :

⟨*Initialisation*⟩≡
```
if(!*texte)
   return ;
```

L'expression booléenne suivante :

⟨*caractère invalide*⟩≡
```
(*texte<'1' || *texte>'9') && *texte!='#'
```

est vrai si le caractère considéré est invalide.

On va chercher le premier caractère valide de la chaîne en utilisant une récursion :

⟨*Initialisation*⟩+≡
```
if(⟨caractère invalide⟩)
   Traduction(texte+1) ;
```

4.0.1 Structure de donnée

Nous allons utiliser la structure suivante :

⟨*Structure de donnée*⟩≡
```
struct data_m
{
char chiffre ;
int iteration ;
char modulo ;
};
```

et pour faire simple, on défini le type suivant :

⟨*Structure de donnée*⟩+≡
```
typedef struct data_m data_t ;
```

Notre implantation se base principalement sur la manipulation de la variable automatique suivante :

⟨*Variables automatiques*⟩≡
```
data_t data ;
```

qui est modifiée à chaque nouveau caractère rencontré suivant les règles que nous allons implanter ci-dessous.

Pour bien comprendre cette structure de donnée, nous allons tout de suite voir comment l'utiliser lors de l'affichage de la conversion.

Affichage

Pour l'affichage, on va se servir du tableau suivant alloué automatiquement :

⟨*Variables automatiques*⟩+≡
```
char tab[10] = "  ADGJMPTW" ;
```

L'affichage proprement dit correspond au code suivant :

⟨*afficher*⟩≡
```
putchar(tab[data.chiffre-'0']+(data.iteration % data.modulo)) ;
```

qui explicite l'usage de notre structure `data`.

Le prototype de la fonction `putchar` utilisée est déclaré dans le fichier d'entête suivant :

⟨*Entête*⟩≡
```
#include <stdio.h>
```

Affectation de `data`

On initialise l'ensemble des champs de la variable `data` en traitant le premier caractère de la chaîne donnée en paramètre :

⟨*Affectation de data*⟩≡
```
data.chiffre = *texte;
data.iteration = 0 ;
if(*texte=='1')
  data.modulo = 1 ;
else
  data.modulo=(*texte=='9' || *texte=='7')?4:3 ;
```

La conversion commence par l'initialisation de `data` et la prise en compte du caractère suivant :

⟨*conversion*⟩≡
```
⟨Initialisation⟩
⟨Affectation de data⟩
texte++ ;
```

Le reste correspond l'application des règles de conversion :

⟨*conversion*⟩+≡
```
while(1)
{
⟨règles de conversion⟩
}
```

définies dans la section suivante.

Règles de conversion

Notre conversion obéit aux règles suivantes :

1. si le caractère considérée est le caractère de terminaison de chaîne (de code ASCII 0), on affiche la traduction en cours (si le caractère précédent n'est pas un dièse) et on sort de la boucle :

   ⟨*règles de conversion*⟩≡
   ```
   if(!*texte)
   {
   ⟨afficher⟩
   break ;
   }
   ```

2. on passe les caractères non valides :

   ⟨*règles de conversion*⟩+≡
   ```
   if(⟨caractère invalide⟩)
   {
     texte++ ;
     continue ;
   }
   ```

3. si on considère de nouveau le même chiffre, on enregistre une itération supplémentaire, on passe ce caractère et on recommence la boucle :

   ⟨*règles de conversion*⟩+≡
   ```
   if(*texte==data.chiffre)
   {
     data.iteration++ ;
     texte++;
     continue ;
   }
   ```

4. si le caractère considéré est le dièse, on le passe :

   ⟨*règles de conversion*⟩+≡
   ```
   while(*texte=='#')
     texte++ ;
   ```

   Remarquons que l'on peut tomber sur une chaîne avec un dièse terminal. La règle suivante gère ce cas :

   ⟨*règles de conversion*⟩+≡
   ```
   if(!*texte)
     continue ;
   ```

5. la dernière règle est implicite — pas besoin de conditionnelle puisque qu'on se trouve forcement dans ce cas — et correspond à la prise en compte d'un nouveau chiffre. Dans ce cas, il faut afficher et réactualiser data :

   ⟨*règles de conversion*⟩+≡
   ⟨*afficher*⟩
   ⟨*réactualiser data*⟩

Réactualisation de data

En ce qui concerne la réactualisation, on ne modifie `data` que face à un chiffre compris entre 2 et 9 inclus.

⟨*réactualiser data*⟩≡
```
if(*texte>'0' && *texte<'9'+1)
{
```
⟨*Affectation de data*⟩
```
}
texte++ ;
```