Développement C **L2 CUPGE 2018–2019**

Travaux pratiques Tableaux

I Exercice : *Crible d'Érathostène Simple*

Pré-requis : Thème Entrées/Sorties, exercice 6.

L'algorithme du crible d'Érathostène est utilisé pour calculer en une fois tous les nombres premiers jusqu'à une certaine limite. On veut calculer un tableau `t` d'entiers tel que `t[i] = 1` si, et seulement si, `i` est un nombre premier. Tout le travail se fait dans le fichier `exercice1.c` à partir du programme de base.

(a) On commence par fixer la taille maximale qu'on accepte pour tester des nombres premiers. Ajouter en haut du fichier, après l'inclusion de `stdio.h` :

```
#define TAILLE 100000
```

(b) On va écrire une fonction `crible` calculant le crible. Pour celà :
- il faut déclarer dans la fonction `main` un tableau d'entiers t de `TAILLE` cases ;
- appeler la fonction `crible` avec l'adresse de ce tableau comme paramètre.

Commencez par écrire une fonction `crible` qui ne fait rien, mais est appelée avec les bons arguments, et vérifiez que le programme compile correctement.

(c) Lorsqu'on programme en C, il est recommandé de suivre la recommandation suivante :
- si une fonction n'a pas eu d'erreur, elle renvoie l'entier 0 ;
- sinon, elle renvoie un entier (le code d'erreur) qui indique quelle a été l'erreur.

Modifiez le programme pour qu'en cas d'erreur de la fonction `crible`, la fonction `main` s'arrête immédiatement en renvoyant un entier > 0.

(d) Dans la fonction `crible`, commencez par mettre les deux premières cases à 0 (car 0 et 1 ne sont pas premiers) et toutes les autres à 1. Compilez et vérifiez que le programme s'exécute correctement, même s'il ne fait rien pour l'instant.

(e) L'algorithme du crible d'Érathostène contient 2 boucles imbriquées :
- La première parcourt toutes les cases du tableau. Pour chaque case i :
  - Si $t[i] = 0$, passer à la case suivante ;
  - Si $t[i] = 1$, alors i est un nombre premier, et :
    - Pour tous les $j \geq 2$ tels que $i * j \leq$ `TAILLE`, mettre la case $i * j$ à 0 (car $i * j$ est un multiple de i).

(f) Modifiez la boucle externe pour ne pas parcourir toutes les cases jusqu'à `TAILLE` et vous arrêter avant.

(g) Dans la fonction `main`, faire une boucle infinie (dont la condition est toujours vraie) demandant à l'utilisateur un entier et faisant :
- Si l'entier est strictement négatif, on sort de la boucle avec `break` (c'est le seul moyen de sortir de cette boucle infinie) ;
- Si l'entier est plus grand ou égal à la taille, on ne peut pas faire de calcul, donc on passe à l'itération suivante avec `continue` ;
- Sinon, il faut afficher si l'entier i qui a été lu est premier en regardant la valeur de $t[i]$.

II Exercice : *Crible d'Érathostène utile*

Commencez par copier dans `exercice2.c` le programme de l'exercice précédent.

(a) Modifiez la fonction `crible` pour que la case $t[i]$ contienne :
— i si i est un nombre premier ;
— j qui est le plus grand diviseur premier de i sinon.
Pour $i = 1$, on prend $t[1] = 1$.

(b) Écrivez une fonction qui prend en entrée le crible et un entier, et affiche la décomposition en facteurs premiers de cet entier.

(c) Modifiez la fonction `main` pour qu'elle affiche la décomposition en facteurs premiers des entiers que donne l'utilisateur.

III Exercice : *Recherche de majorité*

Le but de cet exercice est de chercher dans un tableau d'entiers aléatoires si une des valeurs apparaît dans plus de la moitié des cases.

Obtenir un entier aléatoire. Pour obtenir un entier suffisament aléatoire, on va utiliser un *générateur de nombres pseudo-aléatoires*. Ce générateur doit être *initialisé* par un entier différent à chaque lancement du programme, sinon ce seront toujours les mêmes nombres qui seront choisis. Pour celà, on initialise le GNPA avec l'heure actuelle :

```
srand ( time ( NULL ) ) ;
```

Il est nécessaire d'inclure les bibliothèques `time.h` et `stdlib.h`. Ensuite, chaque fois qu'on a besoin d'un entier aléatoire, on appelle la fonction `rand`.

On fixe une taille de tableau à 10 éléments.

(a) Écrire un programme qui demande à l'utilisateur un entier positif n, et initialise un tableau `t` avec des entiers entre 0 et $n - 1$.

La recherche d'un candidat majoritaire se fait en 2 phases :

1. d'abord, on parcourt une fois toutes les cases du tableau pour trouver un candidat possible. Pour celà :
   — si la majorité actuelle est de 0, l'élément courant devient candidat avec une majorité de 1 ;
   — si la majorité actuelle est strictement plus grande que 0 :
      — si l'élément courant est égal au candidat courant, on augmente la majorité de 1 ;
      — sinon, on diminue la majorité de 1

   Cet algorithme garantit que **s'il existe un élément majoritaire**, alors le candidat est cet élément ;
2. Ensuite, on parcourt le tableau une deuxième fois, et on compte le nombre de fois où le candidat apparaît.
   — Si ce nombre est strictement supérieur à `TAILLE / 2` alors on a trouvé l'élément majoritaire ;
   — Sinon, aucune valeur n'est majoritaire.

(b) Écrire la fonction `trouve_candidat` qui trouve le candidat qui peut être majoritaire :

```
void trouve_candidat ( int * t , int * candidat ) ;
```

avec `t` l'adresse de la première case d'un tableau de `TAILLE` entiers, et `candidat` l'adresse de la variable dans laquelle il faut mettre la valeur trouvée.

(c) Écrire la fonction `verifie_candidat` qui vérifie que le candidat trouvé est majoritaire.

```
int verifie_candidat ( int * t , int candidat ) ;
```

(d) Afin de vérifier le résultat, écrire une fonction qui affiche le contenu du tableau d'entiers.

(e) Complétez la fonction `main` et vérifiez que votre programme marche. Il est conseillé de prendre de choisir $n = 2$ ou $n = 3$ pour avoir de bonnes chances d'avoir un élément majoritaire.

IV Exercice : *Demi-additionneur et additionneur*

On va voir dans cet exercice comment additionner deux entiers ayant un nombre arbitraire de chiffres. Le principe est simple :

1. Chaque entier en entrée est codé par un tableau de chiffres ;
2. On calcule la solution chiffre par chiffre, en n'oubliant pas la retenue.

Pour le 1. on pourrait choisir des chiffres dans n'importe quelle base (au fond, c'est l'ordinateur qui fait les additions), mais pour avoir des exemples faciles à construire, on va utiliser la base 256, donc :

— Chaque chiffre est un *unsigned char* entre 0 et 255 ;
— Un entier est un tableau de *unsigned char*.

Pour les exemples, on va fixer la taille de ces tableaux à 10, mais n'importe quelle valeur est possible. Pour pouvoir parcourir les tableaux à partir de l'indice 0, on suppose en plus que les chiffres sont écrits dans l'ordre inversé (le chiffre des unités dans la case 0, puis celui des "dizaines", etc.)

(a) Dans la fonction *main*, définissez 3 tableaux de type *unsigned char*, et initialisez les 2 premiers. Vérifiez que le code est correct en compilant et en exécutant le programme (il ne fait rien).

```
unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
    t3[10] ;
```

Solution

```
#include <stdio.h>

int
main ( int argc , unsigned char * argv[] )
{
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
```

```
        t3[10] ;
  return 0 ;
}
```

(b) Écrire une fonction qui prend en entrée l'adresse d'un tableau d'*unsigned char* et son nombre de cases, et qui affiche le contenu du tableau. Modifiez la fonction *main* pour afficher les 3 tableaux.

Solution

```
#include <stdio.h>

void print_tableau_c ( unsigned char * t , int n )
{
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}

int
main ( int argc , unsigned char * argv[] )
{
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
      t3[10] ;
      print_tableau_c ( t1 , 3 ) ;
      print_tableau_c ( t2 , 3 ) ;
      print_tableau_c ( t3 , 3 ) ;
  return 0 ;
}
```

(c) Écrire une fonction *additionneur* qui prend en entrée :

- l'adresse de deux *unsigned char*, `resultat` et `retenue`, à laquelle elle mettra les valeurs de la somme et de la retenue de la somme suivante ;
- 2 *unsigned char* dont elle doit faire la somme avec la valeur, lors de l'appel de la fonction, contenue à l'adresse `retenue`.

Cette fonction ne renvoie pas de résultat.

Solution

```
void additionneur (
        unsigned char * retenue ,
        unsigned char * resultat ,
        unsigned char c1 ,
        unsigned char c2 )
```

```
{
  int tmp ;
  tmp = c1 + c2 + * retenue ;
  *retenue = tmp / 256 ;
  *resultat = tmp % 256 ;
}
```

(d) Écrire une fonction *addition_tableau* qui prend en entrée les adresses de 3 tableaux d'*unsigned char* et leur nombre de cases, et qui écrit dans le premier le résultat de l'addition des 2 autres. Cette fonction renvoie 1 si la retenue finale est 1 (c'est un cas d'erreur), et 0 sinon. Appliquez cette fonction sur les tableaux de la fonction *main*, et affichez le résultat.

Solution

```
int addition_tableau (
unsigned char * res ,
unsigned char * t1 ,
unsigned char * t2 ,
int n )
{
  int i ;
  unsigned char retenue = 0 ;
  for ( i = 0 ; i < n ; i++ )
    additionneur ( & retenue , res + i , t1[i], t2[i] ) ;
  return retenue ;
}
```

(e) **Attention, ce qui suit peut être perturbant !**
De la même manière que précédement, créez et initialisez 3 tableaux d'entiers dans la fonction main, écrivez une fonction d'affichage des tableaux d'entiers, et *appelez la fonction addition_tableau* sur ces tableaux (attention au nombre de cases, cette fonction a besoin du nombre de cases de type *unsigned char*. Affichez le résultat. Que remarquez-vous ? Est-ce normal ?

Solution

Pour la solution complète, on utilise des demi-additionneurs, qui sont plus proches des circuits utilisés dans les processeurs (notez le *ou logique* pour la retenue finale). De plus, on teste que toutes adresses sont possibles (la seule réellement impossible étant l'adresse `NULL`)

```
#include <stdio.h>


int demi_additionneur (
        unsigned char * retenue ,
        unsigned char * resultat ,
        unsigned char c1 ,
        unsigned char c2 )
{
```

```
  int tmp ;
  if ( retenue == NULL )
    return 1 ;
  if ( resultat == NULL )
    return 1 ;
  * resultat = ( c1 + c2 ) % 256 ;
  * retenue = ( c1 + c2 ) / 256 ;
  return 0 ;
}

int additionneur (
       unsigned char * retenue ,
       unsigned char * resultat ,
       unsigned char c1 ,
       unsigned char c2 )
{
  unsigned char retenue1 , retenue2 ;
  if ( retenue == NULL )
    return 1 ;
  if ( resultat == NULL )
    return 1 ;
  if ( demi_additionneur ( & retenue1 , resultat , c1 , c2 ) )
    return 1 ;
  if ( demi_additionneur ( & retenue2 , resultat , * resultat , * retenue ) )
    return 1 ;
  * retenue = retenue1 || retenue2 ;
  return 0 ;
}



int addition_c (
unsigned char * res ,
unsigned char * t1 ,
unsigned char * t2 ,
int n )
{
  int i ;
  unsigned char retenue = 0 ;
  for ( i = 0 ; i < n ; i++ )
    if ( additionneur ( & retenue , res + i , t1[i], t2[i] ) )
      return 1 ;
  if ( retenue )
    return 2 ;
  return 0 ;
}

void print_tableau_c ( unsigned char * t , int n )
{
```

```
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}
void print_tableau_i ( unsigned int * t , int n )
{
  int i ;
  printf ( "[ %3d " , t[0] ) ;
  for ( i = 1 ; i < n ; i ++ )
    printf ( ", %3d " , t[i] ) ;
  printf ( "]\n" ) ;
}

int calcule ( char * t3 , char * t1 , char * t2 , int n )
{
  int i ;
  print_tableau_c ( t1 , n ) ;
  print_tableau_c ( t2 , n ) ;
  switch ( addition_c ( t3 , t1 , t2 , n ) )
    {
    case 1:
      printf ( "Erreur lors de l'addition.\n" ) ;
      return 1 ;
    case 2:
      printf ( "Il reste une retenue (dépassement).\n" ) ;
      return 2 ;
    default:
      print_tableau_c ( t1 , n ) ;
      print_tableau_c ( t2 , n ) ;
      print_tableau_c ( t3 , n ) ;
    }
}

int
main ( int argc , unsigned char * argv[] )
{
  unsigned int
    i1[3] = { 12345 , 12345 , 0 } ,
    i2[3] = { 66666 , 44444 , 0 } ,
      i3[3] ,
i ;
  unsigned char
    t1[10] = { 56 , 125 , 234 , 12 , 124 , 0 } ,
    t2[10] = { 34 , 131 , 20 , 244 , 200 , 0 } ,
      t3[10] ;
      calcule ( t3 , t1 , t2 , 10 ) ;
      calcule ( ( unsigned char * ) i3 , ( unsigned char * ) i1 , ( unsigne
```

```
        print_tableau_i ( i1 , 3 ) ;
        print_tableau_i ( i2 , 3 ) ;
        print_tableau_i ( i3 , 3 ) ;
    return 0 ;
}
```

Le résultat attendu est que l'addition est correcte pour les tableaux d'entiers. Quand on y pense ; c'est très perturbant, car normalement, dans chaque case d'entier, c'est comme-ci on faisait l'addition de la gauche vers la droite (on avait inversé l'ordre des chiffres dans le codage en tableau).

La raison est que les processeurs Intel ont évolué à partir d'ordinateurs (les processeurs 8086) qui ne faisaient d'additions que *char* par *char*. Par conservatisme, même maintenant, les entiers continuent d'être stockés dans l'ordre inverse, appelé en anglais *Big Endian* : le dernier char dans la représentation est celui qui contient les chiffres les plus importants. À l'inverse, les tablettes et téléphones portables, qui utilisent des processeurs de type ARM, stockent les entiers dans l'ordre naturel (*Little Endian*, les chiffres les moins importants sont à la fin de la représentation).

V Exercice : *Tri lent*

VI Exercice : *Tri rapide*