ESIEE IN4A11

Programmation

Algorithmique
Pointeurs, listes et arbres

Jean-Claude GEORGES

Année scolaire 2007-2008

1 Mémoire et pointeurs (1) (*)

Soit le morceau de programme suivant :

```
int a;
int *pa;
double x;
int **p;
a=8;
pa=&a;
x=3.14159;
p=&pa;
**p=281;
```

En supposant que la mémoire soit structurée en octets, que les entiers soient codés sur 4 octets, les pointeurs sur 4 octets et que la zone de mémoire automatique soit située en début d'exécution à l'adresse 2364, représentez la mémoire finale de ce programme.

2 Mémoire et pointeurs (2) (*)

Soit la fonction suivante :

```
void echange1 (int x, int y) {
  int z;
  z=x;x=y;y=z;
}
```

Pourquoi ne fonctionne-t-elle pas lorsqu'on l'appelle avec par exemple
`a=2 ;b=3 ;echange1(a,b)` ?
Représentez la mémoire lors de l'exécution de ce morceau de programme.

3 Mémoire et pointeurs (3) (*)

Soit la fonction suivante :

```
void echange2 (int *x, int *y) {
  int *z;
  *z=*x;*x=*y;*y=*z;
}
```

Pourquoi **risque-t-elle** ne pas fonctionner lorsqu'on l'appelle avec par exemple
`a=2 ;b=3 ;echange2(&a,&b)` ?
Représentez la mémoire lors de l'exécution de ce morceau de programme.

4 Mémoire et pointeurs (4) (*)

Soit la fonction suivante :

```
void echange3 (int *x, int *y) {
  int z;
  z=*x;*x=*y;*y=z;
}
```

et l'appel suivant :
`a=2 ;b=3 ;echange3(&a,&b)`
Représentez la mémoire lors de l'exécution de ce morceau de programme.

5 Mémoire et pointeurs (5) ... et références C++ (*)

Soit la fonction suivante :

```
/* Attention : C++ */
void echange4 (int &x, int &y) {
  int z;
  z=x;x=y;y=z;
}
```

et l'appel suivant :
`a=2 ;b=3 ;echange4(a,b)`
Représentez la mémoire lors de l'exécution de ce morceau de programme.

6 Mémoire et pointeurs (6) ... et représentation des nombres (**)

Soit la fonction suivante

```
void echange(type *x, type *y) {
  *y=*x + *y;
  *x=*y - *x;
  *y=*y - *x;
}
```

ou `type` peut être `double` ou `int`. Faites tourner cette fonction à la main avec par exemple :
`int a,b ; a=2 ;b=3 ;echange(&a,&b)`
Fonctionne-t-elle dans tous les cas ?
Donnez un exemple avec des `double` et un exemple avec des `int` où elle est incorrecte.

7 Pointeurs vs. tableau (**)

Soit les deux déclarations suivantes :

```
char tabString[]="toto" ;
char *ptrString="Titi" ;
```

Essayez d'en trouver les différences de comportement. En particulier, notez les possibilités d'exécuter les instructions suivantes :
`tabString[2]='e' ;`
`ptrString[2]='e' ;`
`tabString=ptrString ;`
`ptrString = tabString ;`
Lesquelles provoquent une erreur à la compilation ? à l'exécution ? Faites un dessin représentant la mémoire.

8 Un programme à étudier (**)

Étudiez le programme suivant. Modifiez-le pour que les gestion d'arbres et de liste puissent prendre en compte les doublons (les contenus du maillon et du nœud seront modifiés en un couple <valeur,fréquence>.)

```
/* liste_et_arbre.c
* Exemple de programme de base de gestion de listes chaînées
* et d'arbres binaires en C
* JCG écrit le 22/03/2004 modifié le 1/09/2006
*/
#include <stdio.h>
#include <stdlib.h>
/*************************************************************/
/*                                                           */
/*                          Listes                           */
/*                                                           */
/*************************************************************/
/*
* le maillon d'un liste : un contenu et un chaînage vers le maillon suivant
*/
typedef struct _maillon{
  int contenu;
  struct _maillon * suivant;
} maillon;
/*
* une liste est l'adresse du premier maillon d'une chaîne (tête de liste)
*/
typedef maillon *liste;
/*
* afficher une liste, c'est afficher le contenu de son premier élément,
* puis afficher la queue de la liste
*/
void afficheListe(liste l) {
```

```
  if (l!=NULL) {
    printf("%5d",l->contenu);
    afficheListe(l->suivant);
  }
}
/*
* juste un affichage avec retour ligne
*/
void afficheListeLN(liste l) {
  afficheListe(l); printf("\n");
}
/*
* libérer une liste, c'est libére la queue de la liste, puis
* libérer son premier élément
*/
void libereListe(liste l) {
  if (l!=NULL) {
    libereListe(l->suivant);
    free(l);
  }
}
/*
* insére un maillon à sa place (ordre croissant) dans une liste
* et retourne la nouvelle liste
*
* si le nouveau maillon est destiné à être tête de liste,
*    lui chaîner la liste actuelle
*    retourner son adresse
* sinon
*   chaîner à l'actuelle tête de liste l'insertion du nouveau maillon
*    dans la queue de la liste
*    retourner l'adresse de la tête de liste actuelle
*/
liste insereListe(maillon *m, liste l) {
  if (l==NULL || m->contenu<=l->contenu) {
    m->suivant=l;
    return m;
  }
  else {
    l->suivant=insereListe(m,l->suivant);
    return l;
  }
}
/*************************************************************/
/*                                                           */
/*                    Arbres binaires                        */
/*                                                           */
/*************************************************************/
/*
* le noeud d'un arbre : un contenu, un chaînage vers le fils gauche et
* un chaînage vers le fils droit
*/
typedef struct _noeud{
  int contenu;
```

```
  struct _noeud * fg, *fd;
} noeud;
/*
* un arbre est l'adresse de son premier noeud (racine)
*/
typedef noeud *arbre;
/*
* afficher un arbre (affichage infixe), c'est afficher son fils gauche,
* afficher le contenu de sa racine, puis afficher son fils droit
*/
void afficheArbre(arbre a) {
  int i;
  if (a!=NULL) {
    afficheArbre(a->fg);
    printf("%5d",a->contenu);
    afficheArbre(a->fd);
  }
}
/*
* juste un affichage avec retour ligne
*/
void afficheArbreLN(arbre a) {
  printf("\n");
  afficheArbre(a);
  printf("\n");
}
/*
* libérer un arbre (affichage infixe), c'est libérer son fils gauche,
* libérer son fils droit, puis libérer sa racine
*/
void libereArbre(arbre a) {
  if (a!=NULL) {
    libereArbre(a->fg);
    libereArbre(a->fd);
    free(a);
  }
}
/*
* insérer un nouveau noeud dans un arbre binaire de recherche :
*   retourne l'adresse de la racine de l'arbre après insertion
*
* si l'arbre est vide,
*   la racine après insertion est le nouveau noeud.
* sinon
*   si contenu nouveau noeud  <= contenu racine, alors
*     l'arbre fils gauche de la racine sera égal à l'arbre modifié
*     par l'insertion du nouveau noeud dans l'arbre fils gauche,
*   sinon
*     l'arbre fils droit de la racine sera égal à l'arbre modifié
*     par l'insertion du nouveau noeud dans l'arbre fils droit
*/
arbre insereArbre(noeud *n, arbre a ) {
  if (a==NULL) {
    n->fg = n->fd=NULL;
```

```
    return n;
  }
  else {
    if (n->contenu<=a->contenu)
      a->fg=insereArbre(n, a->fg);
    else
      a->fd=insereArbre(n, a->fd);
    return a;
  }
}
/*****************************************************************/
/*                                                               */
/*                    Utilitaires divers                         */
/*                                                               */
/*****************************************************************/
/*
* principe d'un générateur de nombre pseudo aléatoire
*  x <-- (ax+b) mod m, avec a, b et m judicieusement choisis
*  cf Numerical Recipes in C
*/
int aleasuivant() {
  static int x=1;
  return x=(17*x+7)%128;
}
/*****************************************************************/
/*                                                               */
/*                         Le main                               */
/*                                                               */
/*****************************************************************/
/*
* un  main de test où l'on crée une liste de 128 éléments (pseudo aléatoires)
*/
int main()
{
  liste l=NULL; arbre a=NULL;
  maillon *m;   noeud *n;
  int i;
  int x; /* contenu */
  for (i=0;i<200;++i){
    m=malloc(sizeof(maillon));
    n=malloc(sizeof(noeud));
    n->contenu=m->contenu=x=aleasuivant();
    printf("%5d",x);
    l=insereListe(m,l);
    a=insereArbre(n,a);
  }
  printf("\n\n");
  afficheListeLN(l);
  libereListe(l);
  afficheArbreLN(a);
  libereArbre(a);
  system("pause");
  return 0;
}
```

9 Arbres et listes chaînées (***)

Écrivez le programme complet suivant :
- demande à l'utilisateur la saisie d'une chaîne de caractères ;
- crée à partie de la chaîne un arbre binaire de recherche ordonné alphabétiquement dont chaque noeud contient le caractère et sa fréquence dans la chaîne ;
- crée à partir de cet arbre une liste chaînée composée de mailons <caractère, fréquence> triée par ordre de fréquence ;
- affiche l'arbre et la liste.

Note : Ne pas oublier de libérer l'arbre et la liste en fin de programme.