TABLEAUX, STRUCTURES ET FICHIERS

2012-12-03

© Emmanuel Chieze
Département d'informatique, UQAM
INF3135

Plan

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Tableaux multidimensionnels

3

- Déclaration

```
int b[3][2];
```

  - pas d'initialisation par défaut si automatique (i.e. non statique)
  - pas de limites sur le nombre de dimensions

- Déclaration et initialisation

```
int a[3][2] = { {1, 2 }, {3, 4}, {5,
  6} };
```

- Accès à un élément

```
i = a[2][1]; /* i vaut 6 */
```


Tableaux multidimensionnels

4

- Les éléments sont rangés selon l'ordre obtenu en faisant varier le dernier indice en premier
- Pour un tableau à 2 dimensions :
  - interprétation classique matrice [ligne][col]
  - les éléments sont rangés par lignes
- Vérification :

```
int a[3][2] = { {1, 2 }, {3, 4}, {5, 6} };
int i, j;

for (i=0; i < 3; i++)
   for (j=0; j < 2; j++)
        printf("%p : %4d\n",&a[i][j],
        a[i][j]);
```

```
ffbee788 :    1
ffbee78c :    2
ffbee790 :    3
ffbee794 :    4
ffbee798 :    5
ffbee79c :    6
```

Tableaux multidimensionnels

5

- Un tableau à n dimensions est un vecteur de tableaux à n-1 dimensions (n >= 2)
- a+i désigne l'adresse du (i+1)ème tableau à N-1 dimensions
- a[i] équivaut à &a[i][0] : adresse du premier élément du (i+1)ème vecteur

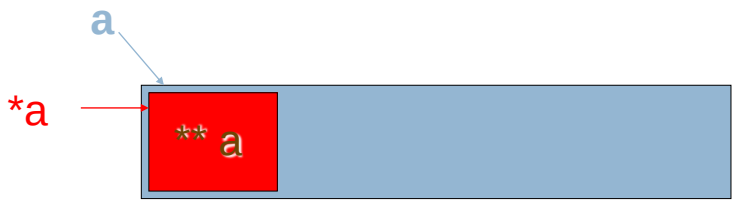
```
int a[3][2] = { {1, 2 }, {3, 4}, {5, 6} };
int *pvi = a[1];
printf("%d, %d, %d, %d\n",
  *(pvi + 1),
  *(pvi + 2),
  (*(a + 1))[1],
  *(*(a + 1) + 1)); /* Affiche 4, 5, 4, 4 */
```


Tableaux multidimensionnels

- Équivalences entre notations

```
int a[3][2];
```

- a et *a désignent la même valeur, mais ne sont pas du même type
- *a et a[0] désignent la même valeur et sont de même type



Tableaux multidimensionnels

- Équivalences entre notations

type int* [2]	a		a+1		a+2	
type int *	a[0] *a &a[0][0]	a[0]+1 *a + 1 &a[0][1]	a[1] *(a+1) &a[1][0]	a[1]+1 *(a+1)+1 &a[1][1]	a[2] *(a+2) &a[2][0]	a[2]+1 *(a+2)+1 &a[2][1]
type int	a[0][0] **a	a[0][1] *(*a + 1)	a[1][0] **(a+1)	a[1][1] *(*(a+1)+1)	a[2][0] **(a+2)	a[2][1] *(*(a+2)+1)

Tableaux multidimensionnels

- Assignations équivalentes

```
int a[3][2] = { {1, 2 }, {3, 4}, {5,
  6} };
int a[3][2] = { 1, 2 , 3, 4, 5, 6 };
```

- Les assignations suivantes sont-elles équivalentes ?

```
int a[3][2] = { {1 }, {3, 4}, {5} };
int a[3][2] = { 1, 3, 4, 5};
```

Tableaux multidimensionnels

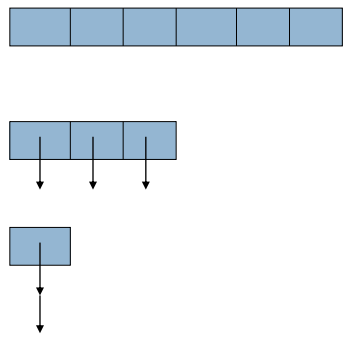
9

- 3 types de déclaration distincts
  1. `int a[3][2];`
     - réserve 6 emplacements contigus de taille *int* en mémoire
     - `(int *) a == a[0]` est vrai
  2. `int *a[3];`
     - alloue 3 emplacements contigus pour des pointeurs.
     - Si chaque pointeur pointe sur un vecteur de taille 2, on occupe 6 emplacements de taille int, plus 3 emplacements de pointeurs.
     - permet d'avoir des lignes de taille variable
     - Chacune des lignes peut être placée n'importe où en mémoire
     - `(int *) a == a[0]` est faux
  3. `int **a;`
     - pointeur vers un pointeur vers un entier
     - Sert à spécifier un tableau bidimensionnel
     - Ou à passer un pointeur par adresse (cf. strtol)
- Les trois types de déclaration permettent un adressage a[i][j]

Tableaux multidimensionnels

10

- `int a[3][2];`
- `int *a[3];`
- `int **a;`



Tableaux multidimensionnels

11

- Exemple

```
char *a[] = { "riri", "toto" };
char **p ;
p = a;
printf("%c, %c, %s, %s\n",
        **p,
        *a[0],
        *(p+1),
        a[1]); /* Affiche r, r, toto, toto */
```


Passage en arguments de tableaux multidimensionnels

12

- nécessaire de spécifier la taille de chaque dimension sauf la première dans la déclaration de la fonction
- pour permettre le calcul d'adresse par le compilateur
  - Tableau à 2 dimensions : &a[i][j] = *a + (i*NCOL+j)
  - Tableau de pointeurs : &a[i][j] = a[i] + j
- cf. exemple5.2.c

Passage en arguments de tableaux multidimensionnels

Exemple		Argument	Paramètre
exemple5.12a.c	OK	float a[][NBCOLMAX]	float a[][NBCOLMAX]
exemple5.12b.c	NOK	float a[][5]	float a[][NBCOLMAX]
exemple5.12c.c	NOK	float *a[]	float a[][NBCOLMAX]
exemple5.12d.c	NOK	float **a	float a[][NBCOLMAX]
exemple5.12e.c	NOK	float a[][NBCOLMAX]	float *a[]
exemple5.12f.c	OK	float *a[]	float *a[]
exemple5.12g.c	OK	float **a	float *a[]
exemple5.12h.c	NOK	float a[][NBCOLMAX]	float **a
exemple5.12i.c	OK	float *a[]	float **a

Tableaux multidimensionnels

- **a peut désigner
  - un tableau à 2 dimensions
    - mais il faut allouer explicitement l'emplacement mémoire associé
  - un pointeur vers un pointeur
    - pour permettre le passage d'un pointeur par adresse
    - afin d'en modifier la valeur
    - cf. fonction strtol(), exemple5.13.c

Tableaux multidimensionnels

15

- `main(int argc, char *argv[], char *env[])`
  - argv et env sont des pointeurs vers un tableau de chaînes de caractères
  - argv[argc] est le pointeur NULL
  - **argv équivaut à *argv[] dans un prototype de fonction
  - **argv équivaut à *argv[0] dans une expression
  - cf. exemple5.1.c : que se passe-t-il lorsque l'on tape la commande `exemple5.1 toto 43` ?
  - utilisation de *env[] : cf. exemple5.14.c

Tableaux multidimensionnels

16

- Distinguer
  - `int *a[13]` (ou `int *(a[13])`)

  (tableau de 13) pointeurs d'entiers
  - `int (*a)[13]`

  (pointeur) sur un tableau de 13 entiers

Plan

17

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Structures

18

- Structure = enregistrement
- Regroupe des données de types différents
- Permet de les traiter comme un tout
- Ou de les traiter séparément
- Définit un nouveau type de données

Structures

19

- Exemple : définition d'un type struct adresse

```
struct adresse {
   char numero[6];
   char rue[30];
   int appt;
   char ville[30];
   char cp[6];
};
```

- Déclaration de variables

```
struct adresse adr1={"4400A", "St-Laurent", 3, "Montreal",
   "H0H 0H0"};
struct adresse adr3;
```

- On peut combiner définition, déclaration et initialisation

Structures

20

- Copie en bloc d'une structure

```
adr3 = adr1;
```

- Passage d'une structure par valeur
- Accès aux membres d'une structure

```
void affiche(struct adresse adr, char *s) {
        sprintf(s, "%s-%d %s\n%s %s\n", adr.numero,
  adr.appt, adr.rue, adr.ville, adr.cp);
}
```

Structures

21

- Pour passage d'un argument struct … par adresse, utilisez les pointeurs
  - plus efficace pour les grosses structures
- Exemple :

```
void change_ville1(struct adresse *adr) {
        strcpy(adr->ville , "Quebec");
}
```

- `adr-> champ` équivaut à `(*adr).champ`
- Voir exemple5.3.c
- Une fonction peut retourner une structure

Structures

22

- Imbrication de structures

```
struct adresse {
        char numero[7];
        char rue[30];
        int appt;
        char ville[30];
        char cp[8];
};

struct individu {
        char nom[30];
        char prenom[30];
        struct adresse adr;
};
```

- Voir exemple5.4.c
- Possibilité d'avoir des tableaux de structures …

Typedef

23

- Définition de types synonymes
- Permet de simplifier et de clarifier le code
- Exemples :

```
typedef char NAS [9];
typedef struct individu Individu;
```

- On peut combiner définition de synonyme et de structure

```
typedef struct {
        char nom[30];
        char prenom[30];
        struct adresse adr;
} Individu;
```

- Voir exemple5.5.c

Union

24

- Permet de créer des variables associées à différents types de données selon le contexte
- Variable créée avec une taille assez grande pour contenir le type le plus volumineux
- Même syntaxe que pour les structures : union = structure sans décalage entre les composantes

```
typedef union {
        int toto ;
        float titi ;
} Bidon;
 Bidon v1, v2;
 v1.toto = 5;
 v2.titi = 10.4;
 printf("%d %f\n", v1.toto, v2.titi); /* Affiche : 5 10.4 */
 printf("%d %f\n", v2.toto, v1.titi); /* Valeurs bidon */
```

Union

25

- Initialisation en bloc des unions : seul le premier membre peut être initialisé

```
Individu personne1 ={"Tremblay", "Jean", {"4400A",
"St-Laurent", 3, "Montreal", {"H0H 0H0"},
"canada"}};
/* Individu personne2 ={"Clinton", "Bill", {"1",
"Pennsylvania Av.", 0, "Washington", {23943},
"etats-unis"}};  Ne fonctionne pas */
Individu personne2 ={"Clinton", "Bill",  {"1",
"Pennsylvania Av.", 0, "Washington", {""}, "etats-
unis"}};
personne2.adr.cp.cpIntl = 34214;
```

- Voir exemple5.6.c

Portée de struct, typedef

26

- Si définition dans une fonction, portée limitée à la fonction
- Si définition hors de toute fonction, portée jusqu'à la fin du fichier
- Impossible d'associer extern à une structure ou un typedef, car aucune adresse associée
- Solution:
  - Utiliser un fichier d'en-tête où l'on définit la structure et les synonymes de type
  - L'inclure dans les fichiers ayant besoin de ces définitions

Plan

27

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Fichiers

28

- Type prédéfini FILE dans <stdio.h>
- Déclaration d'un fichier

```
FILE * fic;
```

- Ouverture d'un fichier
  - selon le mode, le fichier doit exister ou est créé

```
FILE * fopen(char * nom_fichier, char * mode)
```

- Fermeture d'un fichier
  - force l'écriture sur disque du tampon associé au fichier

```
fclose(FILE *fic);
```

E/S séquentielle

29

- Lecture séquentielle
  - `fread(pointeur vers un bloc, sizeof(bloc), nbblocs, FILE *fic)`
  - `feof(FILE *fic)`
  - cf. exemple5.7.c
- Écriture séquentielle
  - `fwrite(pointeur vers un bloc, sizeof(bloc), nbblocs, FILE *fic)`
  - cf. exemple5.8.c

Accès direct

30

- À chaque fichier est associé un pointeur de fichier, qui de déplace lors de chaque lecture/écriture
- Possible d'accéder directement à une adresse dans le fichier (le xème octet du fichier)
- Détermination de la position courante avec `ftell(FILE *fic)`.
- Déplacement du pointeur avec `fseek(FILE *fic, emplacement, mode)`
  - renvoit 0 si positionnement OK
  - renvoit une autre valeur sinon
- Mode
  - SEEK_SET : adresse absolue (depuis le début de fichier)
  - SEEK_CUR : adresse relative à la position courante
  - SEEK_END : adresse relative à la fin du fichier
- Voir exemple5.9.c

Modes d'accès aux fichiers

31

- r : lecture seulement *.
- w : écriture seulement. Efface le contenu du fichier lors de son ouverture**.
- a : écriture seulement, en fin de fichier**.
- r+ : lecture et écriture. Utiliser fseek entre une série de lectures et une série d'écritures *.
- w+ : idem à r+ **.
- a+ : idem à w+, mais le pointeur est positionné à la fin du fichier lors de son ouverture**.

- * : le fichier doit exister
- ** : le fichier est créé s'il n'existe pas

Entrées-sorties formatées

32

- `fscanf(FILE fic*, format, liste d'adresses)`
- `fprintf(FILE fic*, format, liste d'expressions)`
- `fgetc(FILE fic*)`
- `fputc(char, FILE fic*)`
- `fgets(char *, taillemax, FILE fic*)`
- `fputs(char*, FILE fic*)`

Canaux standards

33

- Fichiers prédéfinis
  - `stdin`
  - `stdout`
  - `stderr`
- Pas besoin de les ouvrir ni de les fermer
- Exemples
  - `fprintf(stderr, "Erreur %d : %s", noErr, messageErreur);`
  - `fgets(ligne, taillemax, stdin)` permet de contourner la limite de `gets(ligne)` où l'on ne peut spécifier la taille maximale de la ligne lue.

Plan

34

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Pointeurs vers des fonctions

35

- `double (*f) (double x)` est un pointeur vers une fonction de type
  `double f (double x)`
- Ne pas confondre avec `double *f (double x)`
  - fonction qui retourne un pointeur vers un double
- Utilité
  - passer des fonctions en argument de fonctions
  - créer des tableaux de fonctions
  - cf. exemple5.10.c (calcul de la dérivée d'une fonction en un point)

Plan

36

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Types enum

37

- Types définis par énumération

```
#include <stdio.h>
main (){
  typedef enum sexe { M, F} sexe;
  sexe s = M, t = F;

  printf("%d %d\n", s, t);
}
```

affiche

```
0 1
```


Types enum

38

- *enum* <*nom*> définit des symboles équivalents à des entiers
- Ces symboles ne peuvent servir à nommer des variables
- Par défaut, le premier symbole équivaut à 0, le suivant à 1 …
- On peut contrôler les valeurs affectées :

```
typedef enum sexe { M = 2, F = 1};
```

- Permet de rendre le code plus lisible

Types enum

39

- *enum <nom>* ne permet pas de définir de nouveaux types de données, ni non plus des domaines de int

```
#include <stdio.h>
main (){
   typedef enum sexe { M = 2, F = 1} sexe;
   sexe s = 8;
   int t = M;

   printf("%d %d\n", s, t);
}
```

affiche

```
8 2
```


Plan

40

- Tableaux multidimensionnels
- Structures
- Fichiers
- Compléments sur les fonctions
- Types enum
- Tableaux dynamiques

Tableaux dynamiques

41

- C impose de fixer la taille d'un tableau à la compilation
- Il n'est pas toujours acceptable de prendre une taille limite arbitraire
- Solution : allouer/réallouer un espace mémoire à un pointeur au fur et à mesure des besoins

Allocation de mémoire

42

- Opérateur **sizeof** : retourne l'espace nécessaire au stockage d'une donnée élémentaire
  - sizeof peut prendre comme argument :
    - le nom d'un type de données : sizeof(int)
    - une valeur constante : sizeof("toto")
    - le nom d'une variable : sizeof(toto)
  - Expression évaluée à la compilation

```
int a[5], *b = (int *) malloc ( 5 * sizeof (int));
printf ("%d %d\n", sizeof a, sizeof b);
```

Affiche

```
20  4
```

Allocation de mémoire

43

- Fonctions d'allocation de mémoire (stdlib.h) :
  - `void *malloc(int nboctets)` : réserve un espace mémoire contigu à un pointeur
  - `void *calloc(int nbelements, int tailleelement)` : initialise tous les octets de la zone à 0
- retournent NULL si l'allocation échoue

Allocation de mémoire

44

- Exemple 1

```
int *pi, *pj;
pi = (int*) malloc(sizeof(int));
/* pi = (int*) malloc(4) ;
équivalent de la ligne précédente, mais
  moins lisible et moins portable */
pj = (int*) calloc(10, sizeof(int));
```

Allocation de mémoire

45

- Exemple 2

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char *pc, c[] = "Bonjour";
  pc = (char*) malloc(sizeof(c));
  /* ou pc = (char*) malloc((strlen(c)+1) *
  sizeof(char)); */
  strcpy(pc,c);
  printf("%s\n%s\n", c, pc);
}
```


Allocation de mémoire

46

- Exemple 3 : Allocation pour un tableau d'entiers bidimensionnel dynamique : exemple6.8.c

Allocation de mémoire

47

- `void *realloc(void *p, int nouvelle_taille)` : réalloue une zone mémoire à un pointeur déjà associé à une zone, ou à NULL
  - transfert les données de l'ancienne zone à la nouvelle si nécessaire
  - si nouvelle_taille >= ancienne_taille, les nouveaux octets sont indéterminés
  - si nouvelle_taille < ancienne_taille, les octets sont préservés
  - renvoie NULL en cas d'erreur d'allocation (p est alors inchangé)

Allocation de mémoire

48

- La fonction standard **free** permet de libérer l'espace précédemment réservé par malloc, calloc ou realloc

```
free(pi);
```

  - un free pour un malloc/calloc

Allocation de mémoire

49

- Exemple de réallocation dynamique pour un tableau : exemple6.1.c
- Une fonction peut retourner un pointeur qu'elle a créé si allocation dynamique d'espace : exemple5.15.c

