MACS1 – Sup Galilée

Langage C avancé : Séance 2

Marc TAJCHMAN
e-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Les supports de ce cours sont disponibles sur Github

Ouvrir la page web : https://github.com/tajchman/MACS1_LC

Pointeurs

Définitions : zone mémoire, adresse

La mémoire de travail de l'ordinateur est constituée d'une suite d'unités de mémoire (octets ou bytes en anglais) numérotées (ou indicées) de 0 à N (où N est peut être très grand, actuellement souvent > 10^{10})

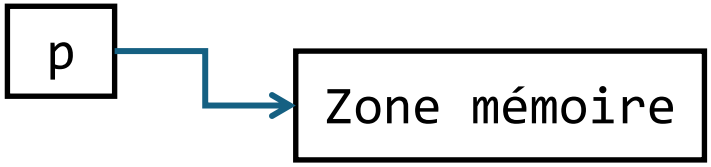
Une zone mémoire est un sous-ensemble d'unités de mémoire consécutives.

L'adresse de la zone mémoire est l'indice de la 1ère unité mémoire de la zone mémoire, c'est un nombre entier.

C définit un type d'entier spécifique, size_t, capable de représenter correctement une adresse.

Pointeurs

- Définition générale : un pointeur est une variable, cette variable occupe une zone mémoire et sa valeur est l'adresse d'une autre zone mémoire
- On peut représenter un pointeur p comme ci-dessous:



Pointeurs générique et typé

- Un pointeur peut contenir l'adresse d'une zone mémoire dont on ne connaît pas la structure : on parle d'un pointeur générique

```
void * v;
```

- Un pointeur peut contenir l'adresse d'une zone mémoire qui ne peut contenir que des valeurs d'un **type** choisi par le programmeur (int, double, char, struct, pointeur, ...)

```
int * p;
double * q;
```

Copies entre pointeurs générique et typé

- Il est possible de copier la valeur d'un pointeur typé ou d'un pointeur générique dans un autre pointeur générique:

```
int * p = …;
void * q = p;
```

- Par contre, on ne peut pas copier la valeur d'un pointeur générique dans un pointeur typé, sauf si on change le type du pointeur générique (et le compilateur ne fait aucune vérification):

```
void * q = …;
double * p = q; /* erreur */
double * r = (double *) q;
```

Copies entre pointeurs générique et typé

- De même, on ne peut pas copier la valeur d'un pointeur typé dans un pointeur d'un type différent:

```
int * q = …;
double * p = q; /* erreur */
```

Il est toujours possible de forcer la copie en changeant le type d'un des pointeurs, mais c'est dangereux (et rarement une bonne idée)

```
int * q = …;
double * p = (double *) q;
```

Exemples d'initialisation d'un pointeur générique

1. Un pointeur générique peut être initialisé avec une adresse spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse utilisable:

```
void * v;
v = NULL;
```

ou

```
void * v = NULL;
```

On pourra écrire, pour tester si un pointeur est utilisable:

```
if (v == NULL)
  printf("erreur v n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur générique (2)

2. Un pointeur générique peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc

```
void * v;
v = malloc(n);
```

ou

```
void * v = malloc(n);
```

Ici n représente la taille de la zone mémoire en octets.

La fonction malloc donne comme résultat un pointeur générique (void *) qui est copié directement dans v.

Exemples d’initialisation d’un pointeur générique (3)

Le résultat de malloc est NULL si la zone mémoire disponible n’est pas suffisante (n est trop grand) ou si n ≤ 0.

Une règle de bonne programmation est de vérifier que ce n’est pas le cas:

```
if (v == NULL) {
  printf("erreur: la mémoire demandée n’a pas pu"
         "être réservée\n");
  exit(-1);
}
```

Exemples d'initialisation d'un pointeur générique (4)

3. Un pointeur générique peut être initialisé avec l'adresse d'une variable existante (de type quelconque):

```
int k;
void * v;
v = &k;
```

ou

```
int k;
void * v = &k;
```

&k représente l'adresse de la variable k.

On ne doit pas tester si le pointeur est NULL puisque toute variable existante a forcément une adresse valable.

Exemples d'initialisation d'un pointeur typé

1. Un pointeur typé peut être initialisé avec une valeur spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse:

```
double * d;
d = NULL;
```

ou

```
double * d = NULL;
```

On pourra donc tester si un pointeur typé est utilisable ou non dans la suite du code:

```
if (d == NULL)
  printf("erreur d n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur typé (2)

2. Un pointeur typé peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc:

```
int * i;
i = (int *) malloc(n * sizeof(int));
```

ou

```
int * i = (int *) malloc(n * sizeof(int));
```

Dans ce cas, le pointeur i contient l'adresse d'une suite de n éléments du type choisi (ici int).

Il faut donc fournir à malloc la taille cette zone mémoire: n x la taille d'une valeur du type choisi (en bleu dans les exemples ci-dessus).

Exemples d'initialisation d'un pointeur typé (3)

La fonction malloc a pour résultat un pointeur générique void * et il faut changer son type pour initialiser le pointeur typé (ici int *):

```
int * i;
i = (int *) malloc(n * sizeof(int));
```

C'est un des seuls changements de type de pointeurs qu'il est « raisonnable » de s'autoriser.

Ne pas oublier de tester si le pointeur initialisé par malloc n'est pas NULL.

Exemples d'initialisation d'un pointeur typé (4)

3. Un pointeur typé peut être initialisé avec l'adresse d'une variable existante du même type :

```
int k;
int * v;
v = &k;
```

ou

```
int k;
int * v = &k;
```

&k représente l'adresse de la variable k.

4. On peut aussi définir et initialiser une zone mémoire de plusieurs valeurs du même type, et en même temps, définir un pointeur vers le début de cette zone:

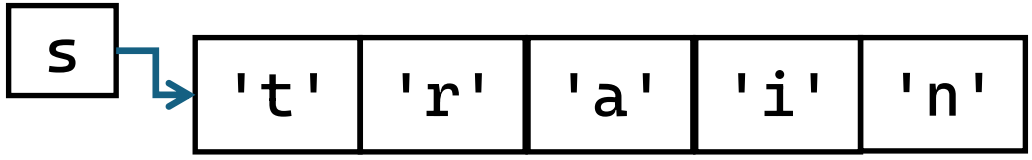
```
double * v = {1.2, 3.4, 4.5};
```

Remarque: traitement particulier des caractères

Notez la différence entre :

- Un vecteur de caractères (une suite de caractères quelconques)

```
char * s = {'t', 'r', 'a', 'i', 'n'};
```

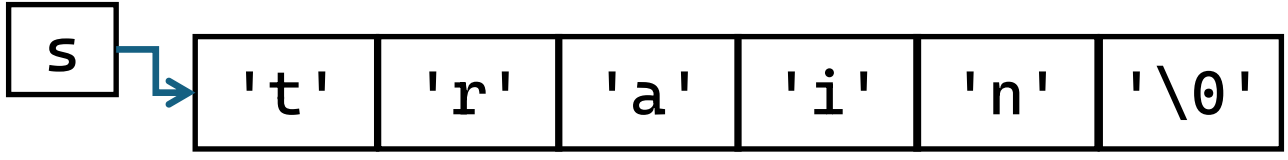


- Une chaîne de caractères (une suite de caractères dont le dernier est le caractère spécial '\0')

```
char * s = "train";
```

ou

```
s = {'t','r','a','i','n','\0'};
```



Accès à la valeur pointée par le pointeur

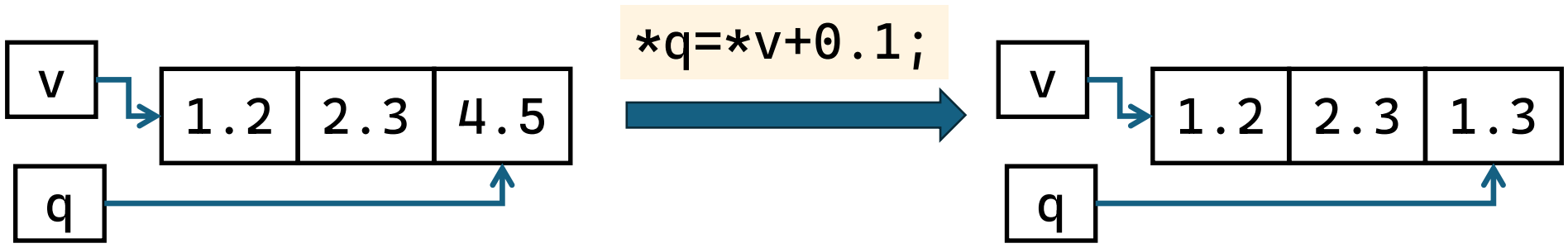
➢Pour utiliser ou modifier la valeur contenue dans la mémoire à l'adresse contenue dans le pointeur, on utilise l'opérateur d'indirection *.

Par exemple

```
double v[] = {1.2, 2.3, 4.5};
double *q = v + 2;
*q = *v + 0.1;
```

L'effet est le même que si on avait écrit:

```
v[2] = v[0] + 0.1
```



Caractère * dans le code source

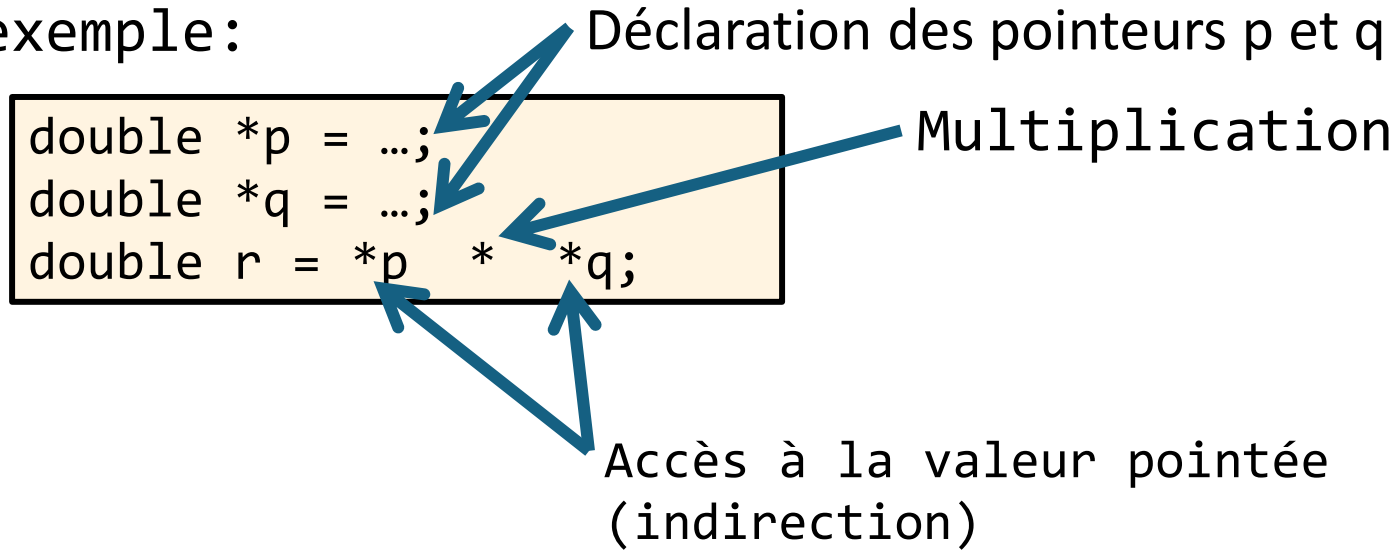
Noter que, en C, le caractère * a plusieurs significations, notamment:

- Opérateur de multiplication entre des nombres
- Bornes d'un commentaire /* … */
- Définition d'un pointeur (int *p = …)
- Accès à la valeur pointée par un pointeur ou indirection (*p = 3)

Un caractère * dans une instruction à l'une des significations ci-dessus en fonction du contexte.

Caractère * dans le code source (2)

Par exemple:

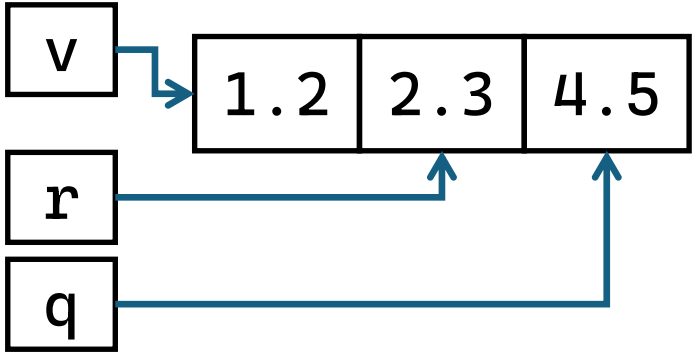


En général, le compilateur déduit la signification du contexte.
Mais il peut arriver, c'est rare, que l'instruction soit ambiguë. Dans ce cas, il faut l'aider, en particulier en ajoutant des (): `double r = (*p)*(*q);`

Opérations avec des pointeurs

- Un pointeur contient une adresse qui est un type particulier d'entier. On peut donc imaginer de faire des opérations arithmétiques ou autres sur les pointeurs.
- Seules certaines opérations ont un sens:
  - Addition d'un entier positif à un pointeur, le résultat est un pointeur, on dit qu'on décale le pointeur vers une adresse supérieure.
  - Soustraction d'un entier positif à une adresse, on dit qu'on décale un pointeur vers une adresse inférieure

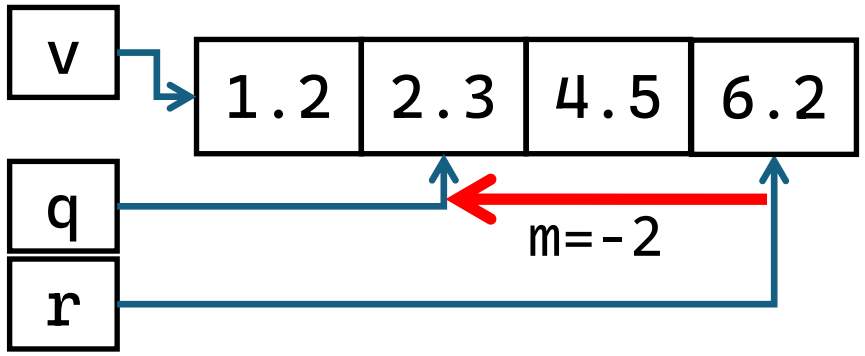
```
double v[] = {1.2, 2.3, 4.5};
double *q = v + 2;
double *r = q - 1;
```



Operations avec des pointeurs (2)

❑ Différence entre 2 pointeurs

```
double v[] = {1.2, 2.3, 4.5, 6.2};
double *q = &v[1]; /* ou q = v+1 */
double *r = &v[3]; /* ou r = v+3 */
int m = q - r;
```



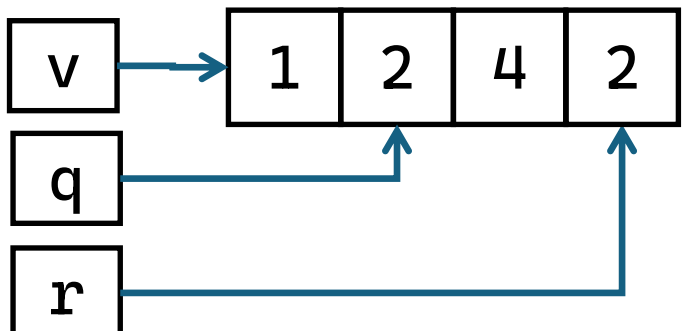
➢ Par contre, les opérations d'addition de 2 pointeurs, de multiplication et de division entre des pointeurs et des entiers n'ont pas de sens

Opérations avec des pointeurs (3)

❑ Comparaison de deux pointeurs

```
int v[] = {1, 2, 4, 2};
int *q = &v[1];
int *r = &v[3];
r == q; /* est faux */
*r == *q ; /* est vrai */
r > q;  /* est vrai */
```

La comparaison de pointeurs consiste à comparer les adresses contenues dans les pointeurs



Affichage des pointeurs / adresses

Un pointeur contient une adresse (un entier de type size_t).
La fonction printf peut l'afficher comme un entier, mais utilise aussi un format spécifique (%p) :

```
double x ;
double *p = &x ; /* p contient l'adresse de x */
printf("adresse de x = %ld\n", p) ;
printf("adresse de x = %p\n", p) ;
```

Ces lignes compilent mais le compilateur affiche des messages d'avertissement.
Ajouter ce qui manque pour ne plus avoir de messages du compilateur

Opérations avec des pointeurs (4)

Il vaut mieux ne pas comparer des pointeurs sur des types différents

Certains compilateurs l’acceptent (et affichent peut-être un message d’avertissement), d’autres non.

Exercice: somme des composantes d'un vecteur

Le code ci-dessous initialise un vecteur de n doubles et calcule la somme des ses composantes.

Modifiez-le en utilisant uniquement des pointeurs pour accéder aux composantes du vecteur.

```
for (i=0; i<n; i++)
  v[i] = sin(i*1.0);

somme = v[0];

for (i=1; i<n; i++)
  somme += v[i];
```

Variables locales et globales

Variable : portée

Quand on définit une variable, sa portée (partie du code où la variable est utilisable) dépend de la façon et de l'endroit où la variable est déclarée : (par ordre d'étendue)

- variable locale à un bloc d'instructions
- variable locale à une fonction
- variable locale à un fichier
- variable globale

Variable locale à un bloc d'instructions ou à une fonction

```
 1: void calcul() {
 2:  int i;
 3:  double x = 0.0;
 5:  for (i=0; i<10; i++) {
 6:   float x = 2.5*i;
 7:   printf("%d x %g\n", i, x);
 8:  }
 9:  x = 0;
10: }
11:
12: int main() {
13:  calcul() ;
14:  return 0 ;
15: }
```

2 variables x existent dans ce code :

- une variable locale à la fonction calcul (entre les lignes 4 et 10),
- une variable locale au bloc d'instructions des lignes 6 et 7.

La variable locale à la fonction calcul existe mais n'est pas utilisable aux lignes 6 et 7 (masquée par la seconde variable x)

Aucune variable x n'existe dans le programme principal.

Variable locale à un fichier

```
 1: static double x;
 2: void calcul() {
 3:   for (int i=0; i<10; i++) {
 4:     x = 2.5*i;
 5:     printf("%d x %g\n", i, x);
 6:   }
 7:   x = 0;
 8: }
 9: static int k;
10: int main() {
11:   calcul() ;
12:   k=5;
13:   printf("%d x %g\n", k, x);
14:   return 0 ;
15: }
```

Les variables x et k sont définies dans le fichier (avec le mot clef static), en dehors des fonctions de ce fichier.

La variable x existe dans le fichier à partir de la ligne 1 et la variable k à partir de la ligne 19.

Variable globale

```
1)  double x;
2)  void calcul() {
3)    int i;
4)    for (i=0; i<10; i++) {
5)      x = 2.5*i;
6)      printf("%d x %g\n", i, x);
7)    }
8)    x = 0;
9)  }
10) static int k;
11) int main() {
12)   calcul() ;
13)   k=5;
14)   printf("%d x %g\n", k, x);
15)   return 0 ;
16) }
```

La variable x est globale, elle existe dans tous les fichiers.
Mais elle est visible (et utilisable) dans les fichiers où elle est déclarée (voir exemples).

Si une variable globale est déclarée dans plusieurs fichiers, elle doit être déclarée avec le mot clef "extern" dans tous les fichiers sauf un (voir exemple 2)

Initialisation des variables

Les variables locales à un bloc d'instructions ou à une fonction ne sont pas automatiquement initialisées.

Les variables locales à un fichier ou globales sont automatiquement initialisées à zéro.

Conseil

Eviter autant que possible les variables locales par fichier et globales. Eventuellement, s'en servir pour des données constantes au cours de l'exécution.

Plusieurs raisons:

- Effets de bord : retracer quelle instruction/fonction a modifié une variable globale peut être difficile.
- Utilisation complexe dans un code parallèle (voir une des prochaines séances).

Vecteurs et matrices

Vecteurs utilisant la mémoire dynamique

Définir un vecteur avec de la mémoire dynamique a plusieurs avantages:

- Adapter précisément la taille des vecteurs utilisés à chaque exécution du code (sinon, il faudrait définir des vecteurs de taille maximale);
- Permettre de définir des vecteurs aussi grands que possible (compte tenu des possibilités de la machine);
- Gérer finement l'utilisation de la mémoire (par exemple, réutiliser la zone mémoire d'un vecteur si on n'en a plus besoin).

Mais aussi un désavantage:

- Devoir gérer explicitement la réservation et la libération mémoire (une erreur fréquente est d'oublier de libérer la mémoire dynamique après utilisation, on verra des outils pour contrôler cela).

Vecteurs utilisant la mémoire dynamique (2)

Examiner un exemple d'utilisation dans exemples/Exemple5/version1
Et comparer avec exemples/Exemple5/version2 qui utilise des macros C

Rangement des coefficients de vecteurs et matrices

La mémoire des ordinateurs est organisée comme une structure à une dimension d’octets.

Il y a donc une façon « naturelle » de ranger les coefficients des vecteurs en mémoire : dans l’ordre des indices croissants.

Pour les matrices, par contre, plusieurs choix sont possibles.

Matrices denses

Les matrices denses sont des matrices où tous les coefficients sont peut-être non nuls.

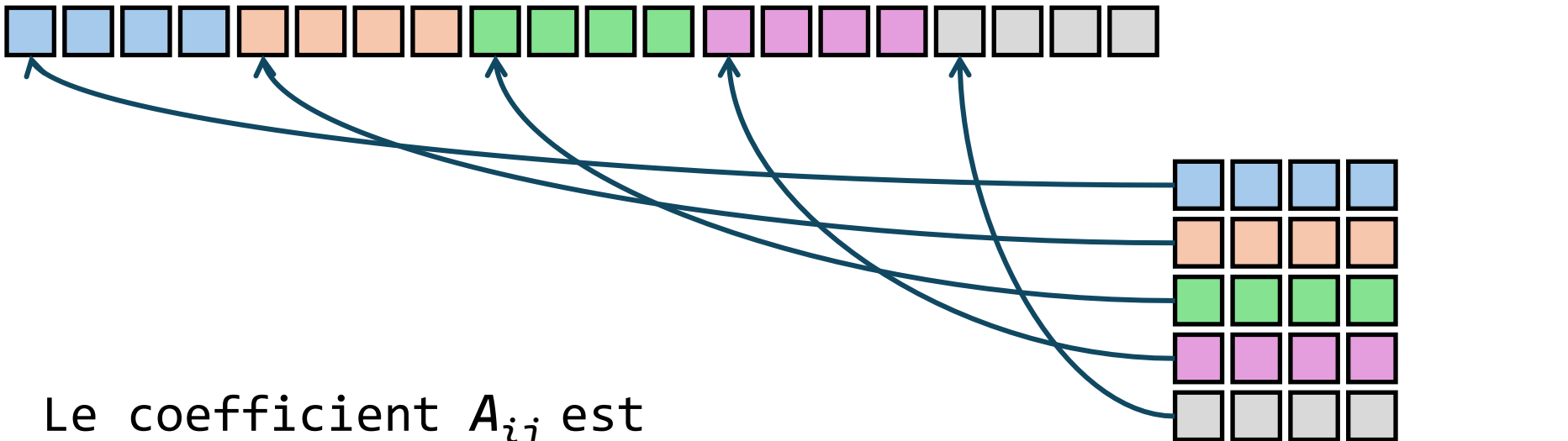
Les structures informatiques pour les matrices denses représentent donc tous les coefficients en zone mémoire.

Pour une matrice à N lignes et M colonnes, il faut donc (au moins) une zone mémoire de taille N x M x sizeof(coefficient) octets.

Rangement des coefficients de matrices à n lignes et m colonnes

Première façon, lignes par lignes (voir Exemple6/version1)

Structure informatique A



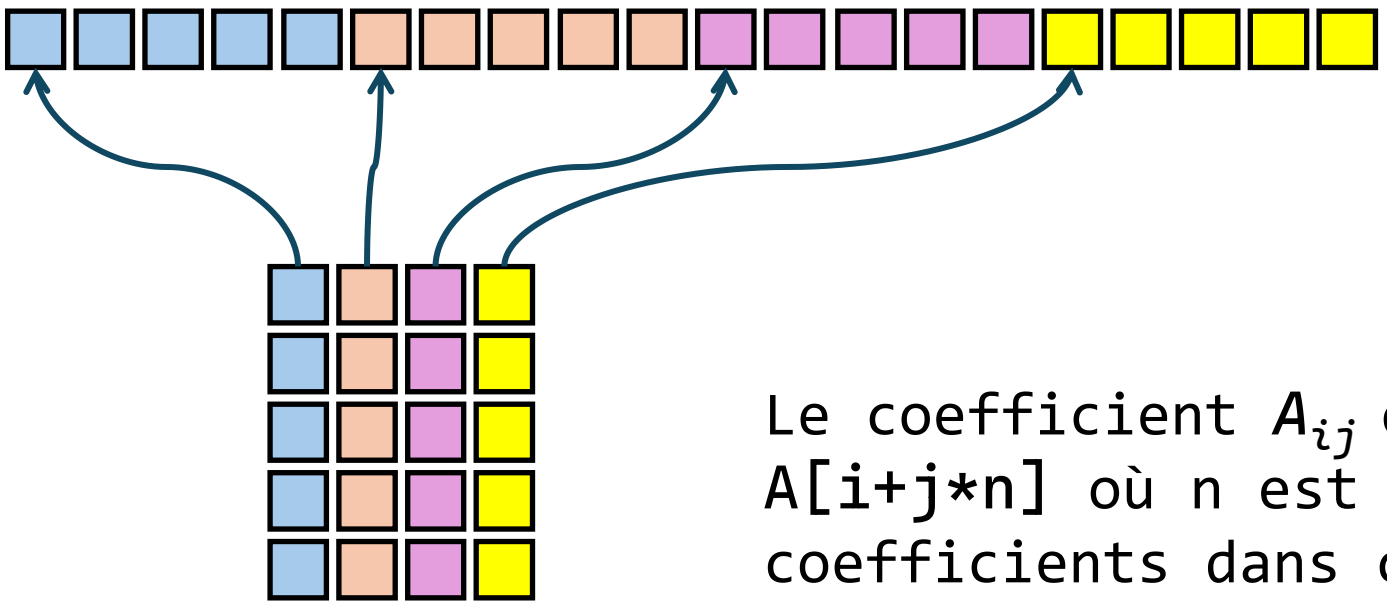
Le coefficient A_{ij} est rangé en `A[i*m+j]` où `m` est le nombre de coefficients dans chaque ligne

Structure mathématique A

Rangement des coefficients de matrices à n lignes et m colonnes (2)

Deuxième façon, colonnes par colonnes

Structure informatique A

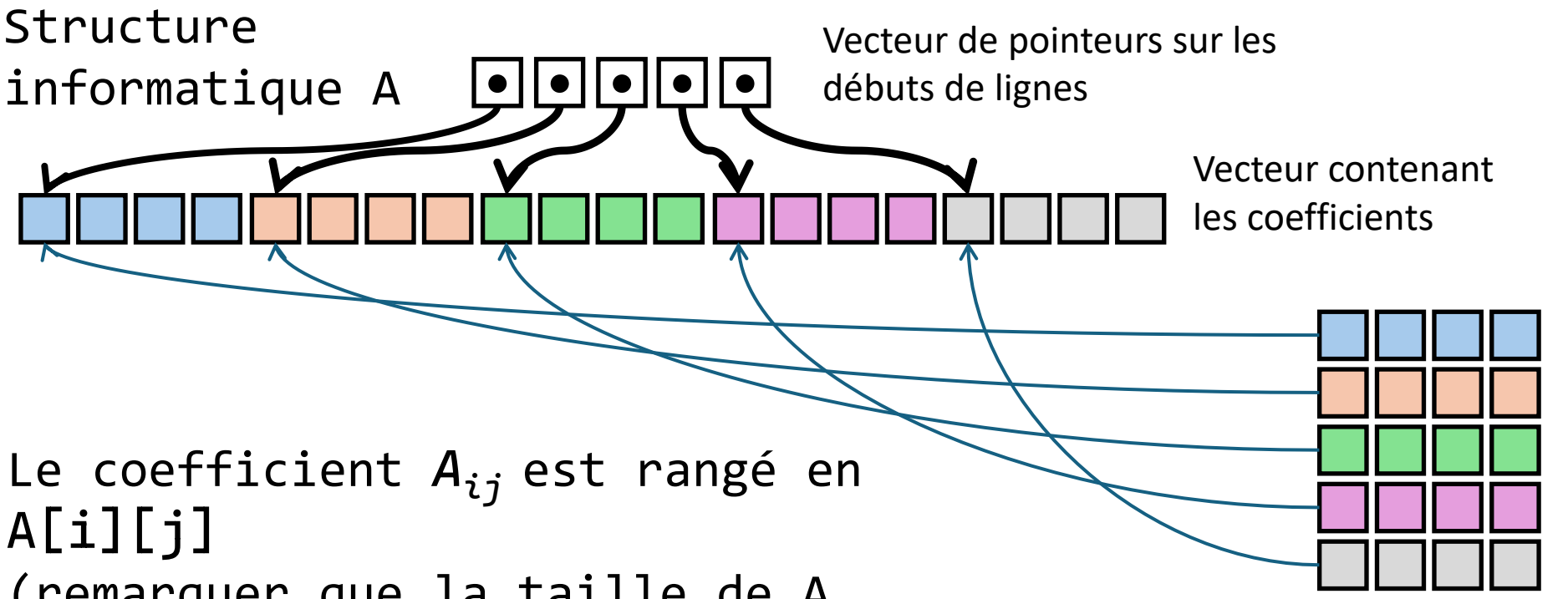


Structure mathématique A

Le coefficient A_{ij} est rangé en `A[i+j*n]` où n est le nombre de coefficients dans chaque colonne

Rangement des coefficients de matrices à n lignes et m colonnes (3)

Troisième façon, ligne par ligne en passant par un vecteur de pointeurs



Le coefficient A_{ij} est rangé en A[i][j]
(remarquer que la taille de A n'intervient pas dans la formule)

Différentes versions du calcul de ${}^tU\ A\ V$

Dans le répertoire Exemple6, se trouve 6 versions du calcul de ${}^tU\ A\ V$.
On compare les 3 modes de représentation des matrices vus précédemment et les 2 formules ${}^tU\ (A\ V)$ et $({}^tU\ A)\ V$

Exécuter ces versions et discutez leurs avantages et inconvénients.

Pour compiler et exécuter une version, on fournit un script run.sh dans chacuns des 6 sous-répertoires. Se placer dans Exemple6 et taper:

```
(cd versionY; ./run.sh)
```

(où Y est 1, 1b, 2, 2b, 3, 3b)

Matrices creuses

Les matrices creuses sont des matrices où, par définition, peu de coefficients sont différents de zéro.

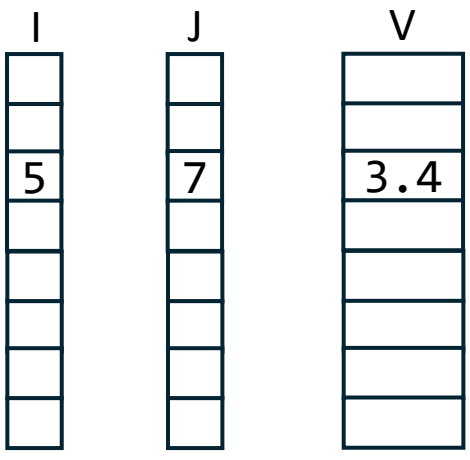
Pour des matrices de grande taille, il est intéressant de réserver de la place mémoire que pour ces coefficients.

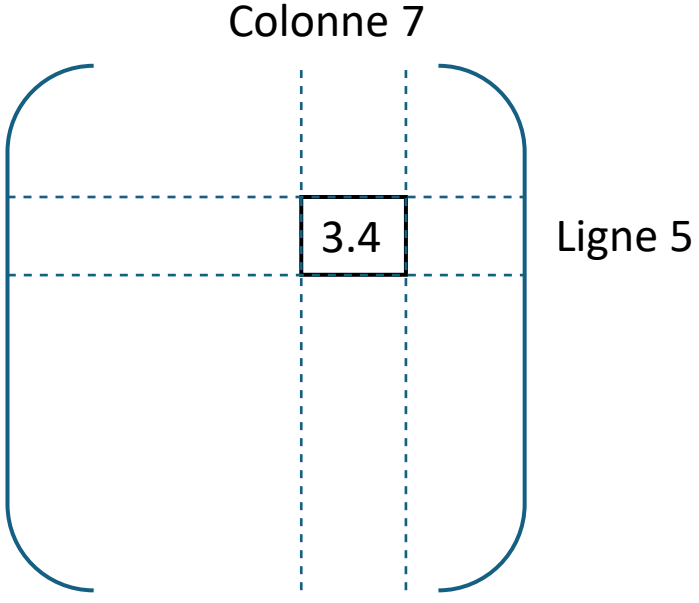
Ce type de matrices apparait dans beaucoup de situations, par exemple la discrétisation des équations aux dérivées partielles ou de la théorie des graphes avec des connexions entre voisins proches.

Exemple de représentation des matrices creuses

Une technique consiste à représenter en mémoire une matrice creuse avec 2 vecteurs d'entiers et un vecteur de type des coefficients de la matrice. La taille des 3 vecteurs est le nombre de coefficient non nuls de la matrice :

- Le 1er vecteur I contient les indices de ligne de chaque coefficient
- Le 2ème vecteur J contient les indices de colonne de chaque coefficient
- Le 3ème vecteur V contient la valeur de chaque coefficient.





Calcul de ${}^tU A V$ où A est une matrice creuse

Dans le répertoire Exemple7, on trouvera le code correspondant à la matrice creuse avec la représentation mémoire vue à la page précédente.

Il existe beaucoup de représentations de matrices creuses.
Celle qui est choisie dépend en général de la consommation mémoire (ici 3 vecteurs de taille du nombre de coefficients non nuls) et de la vitesse de calcul avec les algorithmes du code.

Exercice : transposition d'une matrice creuse

Dans le répertoire exercices/Exercice1, on trouvera le code correspondant au type MatriceCreuse avec la représentation mémoire vue à la page 43.

Le programme principal main.c crée une matrice creuse lue à partir d'un fichier m1.mtx.

Structure en C de la matrice :

I	J	V
1	1	2.8
1	4	1.0
2	2	-3.0
3	4	2.0
4	2	1.0
4	5	2.1
5	3	2.3

Format d'affichage habituel de cette matrice (ne pas afficher les traits) :

2.8	0.0	0.0	1.0	0.0
0.0	-3.0	0.0	0.0	0.0
0.0	0.0	0.0	2.0	0.0
0.0	1.0	0.0	0.0	2.1
0.0	0.0	2.3	0.0	0.0

Exercice : transposition d'une matrice creuse

Compléter les deux fonctions suivantes dans le fichier matrice_creuse.c :

- `printSparseMatrix` qui affiche à l'écran la matrice creuse sous forme habituelle
- `transposeSparseMatrix` qui remplace la matrice par sa transposée $A^T_{ij} \leftarrow A_{ji}$