- Langage C avancé : Séance 6

- Outils d’aide à la compilation
- Types de données utilisateurs : vecteurs et matrices

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

CEA - DES/ISAS/DM2S/STMF/LDEI

Outils d’aide à la compilation

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Comme exemple, on utilisera le code source réparti dans les différents fichiers du répertoire **Exemple1_Make**:

`main.c`: contient le programme principal qui appelle les fonctions f et g et inclut les fichiers A.h et B.h

`A.c`: contient le code source de la fonction f et inclut le fichier A.h

`B.c`: contient le code source de la fonction g et inclut le fichier B.h

`A.h`: contient le prototype de la fonction f (sert à garantir que la fonction f est utilisée correctement)

`B.h`: contient le prototype de la fonction g (sert à garantir que la fonction g est utilisée correctement)

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

On compile les fichiers de code source, pour produire un exécutable exec1 (on a le choix du nom de l'exécutable)

- soit en une étape:

```
gcc main.c A.c B.c –o exec1
```

- soit en plusieurs étapes:

```
gcc –c main.c                    # crée le fichier main.o
gcc -c A.c                       # crée le fichier A.o
gcc –c B.c                       # crée le fichier B.o
gcc main.o A.o B.o –o exec1 # crée l'exécutable exec1
```

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Dès que le nombre de fichiers est important, on préfère procéder en plusieurs étapes pour éviter de recompiler le code source complet, si on ne modifie qu'une partie des fichiers.

Par exemple, si on modifie seulement l'intérieur de la fonction f (dans le fichier A.c):

```
gcc -c main.c      (inutile)
gcc -c A.c
gcc -c B.c         (inutile)
gcc main.o A.o B.o -o exec1
```

Dépendances entre les fichiers

Dans cet exemple, on a donc les règles suivantes:

- exec1 dépend de main.o, A.o et B.o
- main.o dépend directement de main.c et, indirectement, de A.h et B.h (inclus dans main.c)
- A.o dépend directement de A.c et, indirectement, de A.h (inclus dans A.c)
- B.o dépend directement de B.c et, indirectement, de B.h (inclus dans B.c)

Pour chaque règle, on devra effectuer une ou plusieurs commandes pour mettre à jour le résultat de la règle (souvent appelée « cible » ou target en anglais)

Dépendances entre les fichiers

Par exemple, si A.c a été modifié:

- Il faut recréer A.o, qui dépend de A.c, avec la commande

  ```
  gcc –c A.c
  ```

- Il faut recréer exec1 qui dépend de A.o, qui vient de changer, avec la commande

  - `gcc main.o A.o B.o –o exec1`

Dépendances entre les fichiers

Par exemple, si B.h a été modifié:

- Il faut recréer B.o, qui dépend de B.c (et B.c inclut B.h), avec la commande

```
gcc –c B.c
```

- Il faut recréer main.o, qui dépend de main.c (et main.c inclut B.h), avec la commande

```
gcc –c main.c
```

- Il faut recréer exec1 qui dépend de main.o et B.o, qui viennent de changer, avec la commande

```
gcc main.o A.o B.o –o exec1
```

Outil Make

Outil d’aide à la compilation : make

Très rapidement, pour recompiler « intelligemment » le code source en fonction des modifications, on a besoin d’utiliser un outil. Cet outil déclenche les compilations nécessaires (et elles seules) pour mettre à jour les binaires en fonction des modifications du code source.

L’outil utilisé ici est la commande `make` (il en existe d’autres: par exemple nmake sous windows)

On crée un fichier dont le nom est `Makefile` et où sont décrites les règles de construction, la commande make utilise ce fichier. Si on donne un autre nom au fichier de règles, il faut taper

```
make -f nom_du_fichier_de_règles
```

Syntaxe du fichier Makefile

Le fichier Makefile contient une liste de règles de la forme (chaque commande est précédée d'une tabulation):

```
Cible: Liste_de_dépendances
	Commande1
	Commande2
	…
```

Make détermine qu'il faut exécuter commande1, commande2,… si au moins une dépendance est plus récente que la cible

Quand la cible d'une règle ne correspond à aucun fichier et que les commandes ne créent pas la cible, la règle est toujours exécutée

Syntaxe « make » (2)

Quand on tape « make », l'outil examine les dépendances de la première cible.

Si une dépendance n'est pas un fichier existant ou est plus récente que la cible, make cherche dans le fichier Makefile si une autre règle dont la cible est la dépendance.

> Si aucune règle n'est disponible, make affiche un message d'erreur

Quand on tape « make nom_de_cible », l'outil examine les dépendances de la cible « nom_de_cible »

Syntaxe « make »

Dans le code source pris comme exemple, le fichier Makefile peut s'écrire :

```
exec1: main.o A.o B.o
	gcc main.o A.o B.o -o exec1
main.o: main.c A.h B.h
	gcc -c main.c
A.o: A.c A.h
	gcc -c A.c
B.o: B.c B.h
	gcc -c B.c
```

Voir le répertoire Exemple1_Make

Fonctionnalités supplémentaires

Plusieurs fonctionnalités permettent de simplifier ou d’écrire un makefile plus générique

- Variables utilisateur
- Commandes
- Variables spéciales
- Modèles de règle

Variables utilisateur

Si un mot est utilisé plusieurs fois dans plusieurs règles et/ou plusieurs commandes, on peut définir une variable (exemple : la variable Code contient le nom de l'exécutable construit par make) :

```
Code = exec1


${Code}: main.o A.o B.o
	gcc main.o A.o B.o -o ${Code}
clean:
	rm -f ${Code} *.o
```

Fonctions standards dans un makefile

Make propose des fonctions utilitaires, deux exemples

wildcard : crée une liste de tous les fichiers suivant un modèle

exemple : liste des fichiers C dans le répertoire du fichier Makefile

```
sources = $(wildcard *.c)
```

patsubst : à partir d'une liste de mots, crée une seconde liste où les mots de la première liste sont transformés avec une règle de substitution

exemple : on crée une liste de fichiers binaires (terminés par .o) à partir de la liste dans sources :

```
binaires = $(patsubst %.o, %.c, ${sources})
```

Fonctions standards dans un makefile (2)

On peut ensuite compiler un code qui contiendra tous les binaires correspondant aux fichiers sources C :

```
Exec1 : ${binaires}
    gcc ${binaires} -o Exec1
```

Variables spéciales

Ces variables sont automatiquement définies pour chaque règle et doivent être utilisée seulement dans les commandes associées à cette règle, elles ne sont pas modifiables.

$@ contient la cible d'une règle

$< contient l'ensemble des dépendances

$^ contient la première dépendance

Exemples représentent

```
Exec1 : main.o A.o B.o
    gcc $< -o $@
```

```
Exec1 : main.o A.o B.o
    gcc main.o A.o B.o -o Exec1
```

```
A.o : A.c A.h
    gcc -c $^ -o $@
```

```
A.o : A.c A.h
    gcc -c A.c -o A.o
```

Modèles de règle

Plusieurs règles peuvent être remplacées par un modèle de règle si les cibles et dépendances ont une forme similaire

Exemples :

```
%.pdf: %.tex
    pdflatex @<
```

La règle s'applique pour tous les noms de fichier se terminant en .tex et construisent le fichier dont le nom se termine en .pdf

```
%.o : %.c
    gcc -c $^ -o $@
```

Dans le répertoire Exemple1_Make, il y a plusieurs versions de fichiers Makefile équivalents pour compiler un code.

Dans le répertoire Exemple2_Make, se trouve un Makefile qui compile un code dont les sources se trouvent dans plusieurs répertoires.

Pour plus de fonctionnalités de make, voir le manuel de référence (de la version GNU de make) :

https://www.gnu.org/software/make/manual/make.pdf

ou

https://www.gnu.org/software/make/manual/html_node/index.html

Outils de plus haut niveau

L’écriture des fichiers Makefile n’est pas simple dans le cas de codes source complexes.
D’autre part, dans les exemples précédents, on a mis « en dur » dans les Makefiles des informations système, par exemple le nom du compilateur.

Des outils supplémentaires ont été proposés pour utiliser le système de compilation sur plusieurs machines, systèmes et avec des compilateurs différents.

Si on se trouve dans un environnement machine / système / compilateur spécifique, ces outils génèrent des fichiers Makefile adaptés.

Outils autools : automake, autoconf, ...

Outils autotools

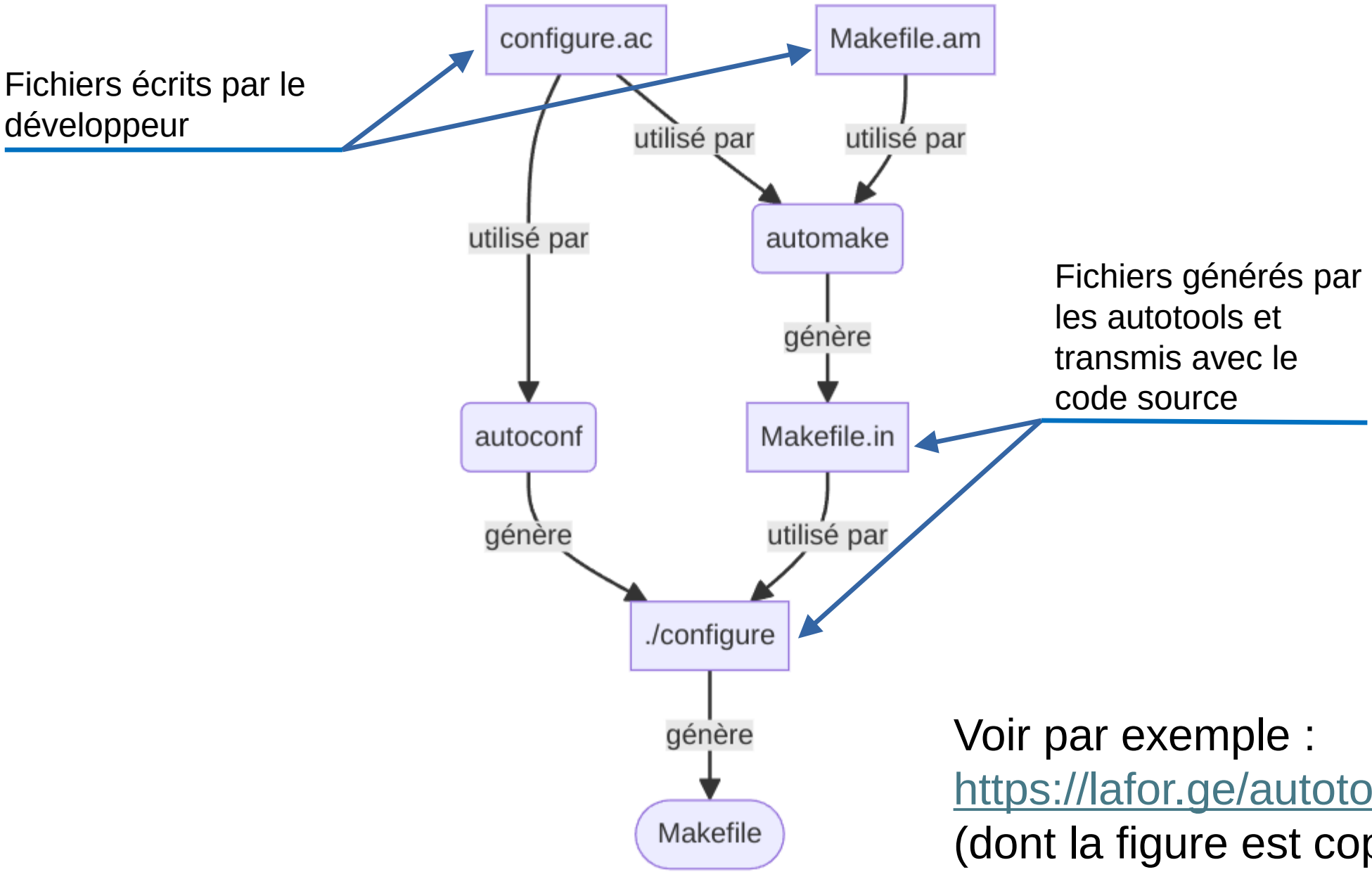
La famille d’outils m4, autoconf, automake, … ont été conçus pour générer les Makefiles et des outils de configuration.

Le processus complet se fait en 2 parties:

- *une première partie à faire par ceux/celles qui écrivent le code source*
- *une seconde partie à faire par ceux/celles qui compilent le code source et installent le code compilé.*

Un fichier principal de configuration « configure » est créé par les autotools, il teste le compilateur et les autres outils pour savoir s’ils fonctionnent, quelles sont leur options, ...

Outils autotools



Voir par exemple :
https://lafor.ge/autotools/
(dont la figure est copiée ici)

Exemple autotools

Dans le répertoire Exemple3_autotools se trouvent le code source et les fichiers Makefile.am et configure.ac (et des fichiers qui décrivent le code, les auteurs, etc.) :

```
aclocal
automake --add-missing
autoconf
```

Le code source + les fichiers/répertoires créés par les autotools sont transmis à ceux/celles qui installent utilisent le code et tapent les commandes :

```
./configure --prefix <chemin d’installation>
make
make install
```

Outils autotools

Les autotools sont utilisés par un grand nombre de codes C (et C++) existants. Il est donc utile de connaitre ces outils.

Pour plus d'information, les manuels de ces outils sont disponibles :

https://www.gnu.org/software/autoconf/manual/autoconf.pdf

https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.72/html_node/index.html

On conseille cependant d'utiliser l'outil présenté aux pages suivantes, plus récent et plus puissant.

Outil CMake

cmake est un outil qui génère des procédures de configuration et de compilation.
Pour utiliser cmake, il faut écrire un fichier CMakeLists.txt dans tous les répertoires qui contiennent des fichiers sources.

cmake est un « vrai » langage de programmation avec :

- des boucles (foreach … endforeach),
- des instructions conditionnelles (if … else … endif)
- des variables définies par cmake ou par l'utilisateur,
- des fonctions,
- des générateurs (pour créer des makefiles ou d'autres systèmes de compilation, par exemple des projets visual studio (sous Windows))

Cmake Exemple 1

Dans le répertoire Exemple4_Cmake, le premier exemple concerne un code compilé code1 à partir d'un seul fichier C : main.c dans le sous- répertoire sources.

Créer un fichier CMakeLists.txt dans le répertoire Exemple4_Cmake/sources (qui contient main.c) avec:

```
cmake_minimum_required(VERSION 3.10)
project(Exemple1 C)
add_executable(code1 main.c)
```

Dans le répertoire Exemple4_Cmake, taper les commandes :

```
cmake -S sources -B builds
cmake --builds builds
```

Cmake Exemple 1

Examiner le contenu des répertoires sources et builds après avoir tapé les commandes cmake.

- **Le répertoire sources n'est pas modifié** (aucun autre fichier que CmakeLists.txt et le(s) fichier(s) source).
- Le répertoire builds contient le fichier code1 (code compilé) et aussi tous les fichiers intermédiaires.

L'utilisateur du code a besoin du fichier code1 mais pas des fichiers intermédiaires.

On va légèrement modifier le fichier CmakeLists.txt et le commandes de compilation pour mettre le code compilé final à un autre endroit que les fichiers intermédiaires

L’utilisateur du code a besoin du fichier code1 mais pas des fichiers intermédiaires.

On va légèrement modifier le fichier CmakeLists.txt et le commandes de compilation pour mettre le code compilé final à un autre endroit que les fichiers intermédiaires

Cmake Exemple 1

Ajouter la ligne qui commence par install ci-dessous dans le fichier CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(Exemple1 C)
add_executable(code1 main.c)
install(TARGET code1 DESTINATION bin)
```

Supprimer le répertoire builds et taper les commandes :

```
cmake -S sources -B builds -DCMAKE_INSTALL_PREFIX=distrib
cmake --build builds
cmake --install builds
```

Cmake Exemple 1

Examiner le contenu des répertoires sources, builds et distrib après avoir tapé les commandes cmake.

On a une séparation claire des fichiers:

- le répertoire sources contient les fichiers source non modifiés + les fichiers CMakeLists.txt
- le répertoire builds contient les fichiers intermédiaires produits par la compilation

  on peut le supprimer sans risquer de perdre des fichiers source ou le code final compilé et on peut le régénérer facilement
- le répertoire distrib contient le code final compilé

  on peut transmettre ce répertoire aux utilisateurs du code, il contient tout ce qui est nécessaire pour utiliser le code

Cmake Exemple 2

Dans le répertoire Exemple5_Cmake, le second exemple concerne un code compilé code2 (résolution de système linéaire en utilisant des structures vecteurs et matrices) à partir de plusieurs fichiers dans des répertoires différents.

De plus, il y a des fichiers qui ne contiennent pas de code source C qu'il faut installer en même temps que le code (fichiers d'exemples)

Cmake Exemple 2

Les différents répertoires contiennent chacuns un fichier CMakeLists.txt, dont un est le fichier cmake principal:

```
sources:
    CMakeLists.txt  (fichier principal)
    types:
        CMakeLists.txt (fichier secondaire)
    data:
        CMakeLists.txt (fichier secondaire)
```

Le fichier cmake principal appelle les fichiers secondaires qui définissent des cibles. Ces cibles sont utilisées comme dépendances dans le fichier principal.

Examiner les différents fichiers CMakeLists.txt

Structures vecteur et matrice

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.

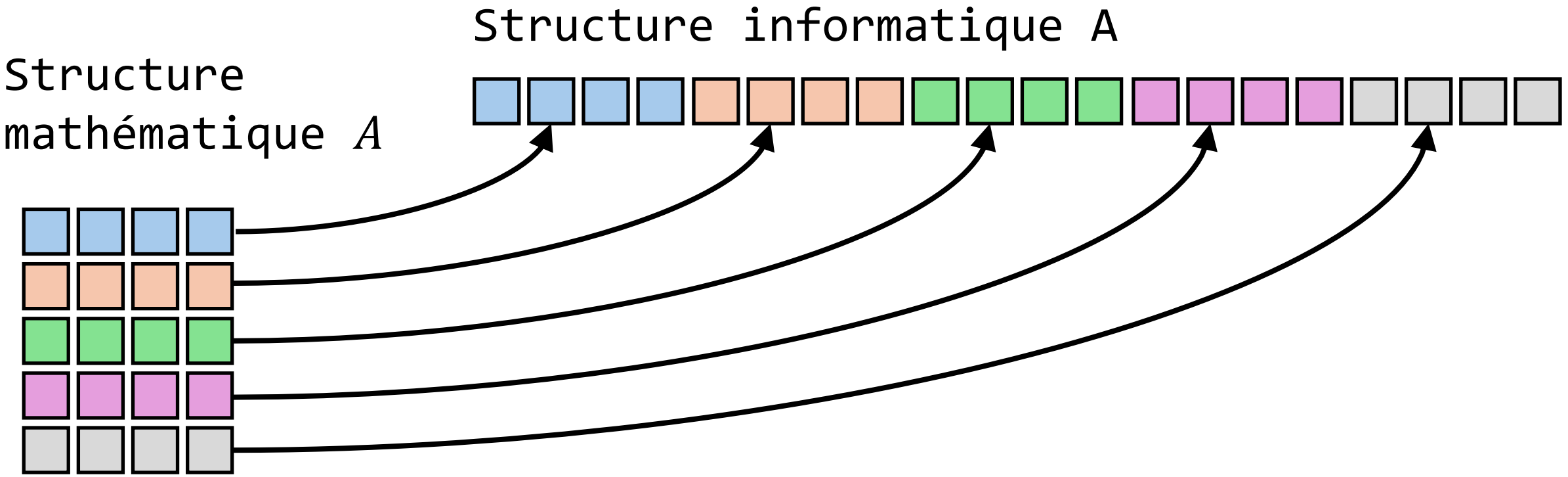
Pour les matrices, par contre, plusieurs choix sont possibles.

Rangement des coefficients de matrices à n lignes et m colonnes

Première façon, lignes par lignes

Structure informatique A

Structure mathématique A

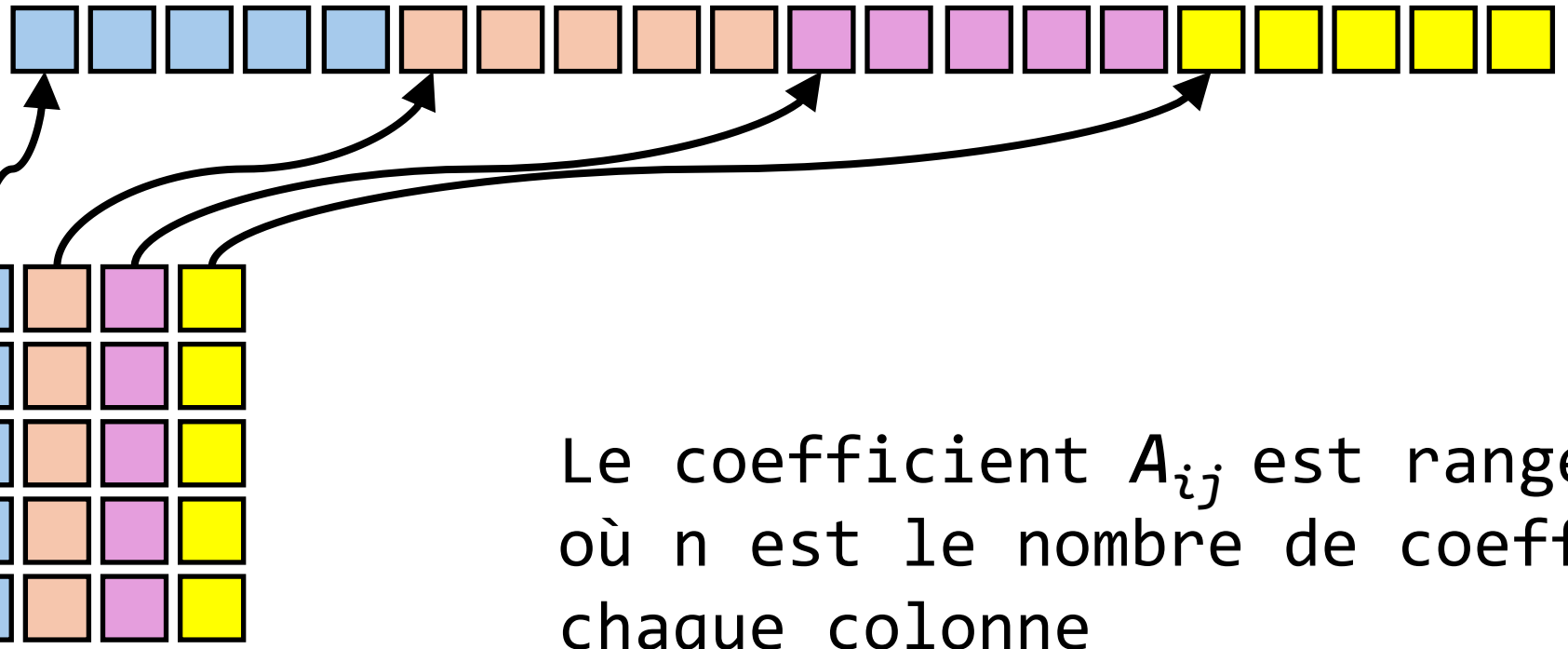


Le coefficient A_{ij} est rangé en A[i*m+j] où m est le nombre de coefficients dans chaque ligne

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

Deuxième façon, colonnes par colonnes

Structure informatique A



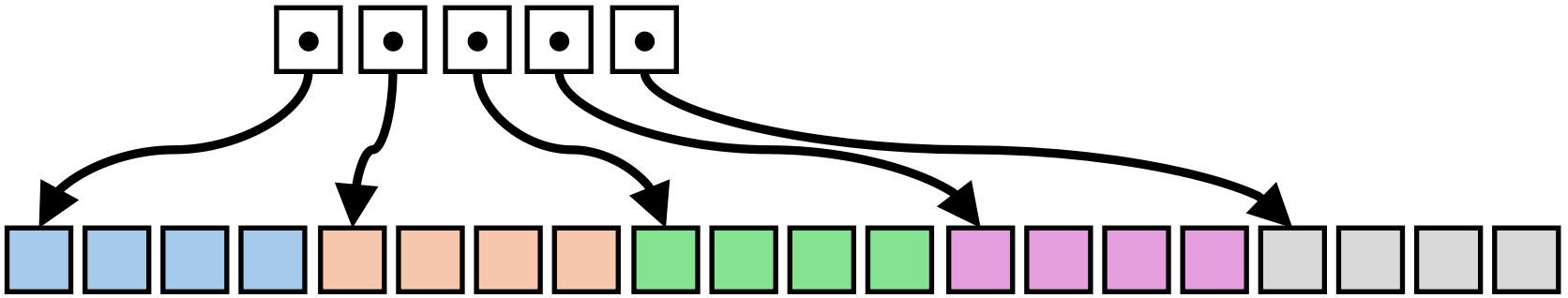
Structure mathématique A

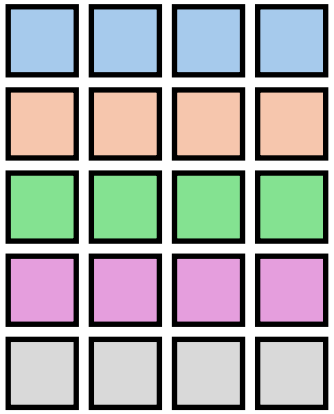
Le coefficient A_{ij} est rangé en A[i+j*n] où n est le nombre de coefficients dans chaque colonne

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

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

Structure informatique A





Structure mathématique A

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

Rangement des coefficients de matrices à n lignes et m colonnes (4)

Le choix du rangement des coefficients en mémoire dépend de plusieurs facteurs, entre autres:

- le langage de programmation
- le sens de parcours des coefficients par l'algorithme choisi par l'utilisateur

Par exemple, si A est une matrice, u et v deux vecteurs, $^{T}u\ A\ v$ peut être calculé de deux façons:

- $(^{T}u\ A)\ v$ se fait plus rapidement si A est rangée colonne par colonne
- $^{T}u\ (A\ v)$ se fait plus rapidement si A est rangée ligne par ligne

Vecteurs, matrices

On a vu qu'un vecteur et une matrice sont caractérisées par le type et le nombre de leurs composantes. **En C, il n'y pas de type vecteur et matrice en tant que tels** (on doit utiliser des pointeurs et mettre les informations de taille à part dans des variables entières)

Par exemple, pour écrire une fonction qui calcule le produit matrice vecteur, on doit fournir :

- Une matrice de taille n x m (matrice de départ)
- Un vecteur de taille p (vecteur de départ)
- Un autre vecteur de taille q (qui va recevoir le résultat du calcul)

Avec les contraintes que p = m et q = n (sinon le calcul du produit n'a pas de sens)

Il est possible de définir la fonction produit matrice vecteur de la façon ci-dessus:

```
void ProduitMatriceVecteur(double * M, int nM, int mM,
                           double * V, int nV,
                           double * W, int nW)
{ … }
```

Et appeler la fonction comme ceci pour calculer y = Ax:

```
ProduitMatriceVecteur(A, n, m, x, p, y, q);
```

L’utilisateur doit faire très attention de transmettre les bonnes valeurs en arguments, en particulier les tailles.

Vecteurs, matrices

Une solution est de définir des types vecteur et matrice (c'est le choix de la plupart des langages modernes)
En C, on peut utiliser des structures pour cela.

Exercice:

Dans le répertoire SystemeLineaire/types sont définies des structures vides Vecteur et Matrice dans les fichiers matrice.h et vecteur.h, les compléter pour y ranger leurs informations de taille et leurs composantes.

Exercice:

Dans les fichiers vecteur.c et matrice.c, les fonctions alloueMatrice et alloueVecteur doivent réserver de la mémoire pour des matrices et vecteurs connaissant leurs tailles. Elle sont actuellement vides, les compléter.

De même pour les fonctions libereVecteur et libereMatrice qui rendent au système, la mémoire utilisée par un vecteur ou une matrice.

Exercice:

Dans le répertoire SystemeLineaire/data se trouve deux fichiers qui contiennent chacun une matrice, après une première ligne qui donne la taille de la matrice et le nombre de coefficients non nuls, sous la forme d'une liste de triples:

numéro de ligne,
numéro de colonne,
valeur du coefficient

Le fichier matrice.c définit la fonction lectureMatrice qui lit une matrice depuis un fichier. Cette fonction est incomplète, la compléter

Fonctions utilitaires

Pour rendre la lecture du code plus claire, on définit souvent des fonctions qui accèdent au coefficient A_{ij}.
Par exemple pour le stockage ligne par ligne :

Fonction qui récupère une valeur depuis un coefficient

```
inline double get(Matrice M, int i, int j)
{ return M.coef[i*M.m + j]; }
```

Fonction qui range une valeur dans un coefficient

```
inline set(Matrice M, int i, int j, double v)
{ M.coef[i*M.m + j] = v; }
```

Macros

On peut utiliser des macros du préprocesseur, toujours dans le cas du stockage ligne par ligne :

```
#define Coef(M,i,j) M.coef[(i)*M.m + j];
```

(ne pas oublier de mettre des parenthèses autour de i, sinon Coef(A,k+1,l) sera interprété comme A.coef[k + 1*A.m + l] qui est différent de A.coef[(k + 1)*A.m + l])