Cours ESIR2 IN BINP

Méthodes de programmation pour l'imagerie numérique

Éric MARCHAND ESIR

BINP

Objectif du cours :

- Ceci n'est pas un cours sur le traitement d'image, ...

  ... c'est un cours de méthodes de programmation pour l'image
- Capacité à **mettre en œuvre** les algorithmes étudiés

Plan du cours :

- Introduction : spécificité des images
- Rappel de cours : quelques langages, C, C++,
- Représentation des images
- Manipulation des images

Spécificité des images

Élément de base : le pixel

— noir et blanc, niveaux de gris, couleur, ... type

Structure de donnée élémentaire : l'image

— simple : un tableau bidimentionnel de pixels... représentation en mémoire

— mais de grande taille (par exemple : 512×512) efficacité

Algorithmes :

— itératifs, récursifs

— souvent coûteux en temps de calcul

— utilisation d'autre structure de donnée : arbres, listes, ...

Problème : trouver un langage adapté

Langage de programmation

Qualités à retenir

- Simplicité :
- Expressivité :
  - Langage de haut niveau (au minimum le typage)
  - Langage objet
- Efficacité :
  - Compilation ou interprétation efficace
  - Gestion dynamique de la mémoire
- Portabilité :
  - vers différents systèmes (Unix [Linux, Solaris, IRIX, ...],Mc OS [9, X], ..., Windows)
  - sur différentes machines (PC, SUN, MAC, SGI, DEC Alpha, ...)
- Réutilisabilité : création de bibliothèque

Quelques langages

Les anciens...

— Cobol	gestion
— Fortran	analyse numérique
— Pascal	enseignement
— Basic	bidouilleurs du début des années 80...
— Lisp	IA
— C	programmation système
— Smalltalk	premier langage objet

Quelques langages (2)

Les langages récents...

- Langage de programmation orientée objets
  - JAVA — WWW
  - Eiffel — enseignement
  - Objective C, C++ — développement lourd
- Autres :
  - Prolog — IA, Système expert
  - ML (CAML), OCAML — “LiSP typé”
  - ~~etc~~... Python — prototypage
  - Matlab — Prototypage, calcul

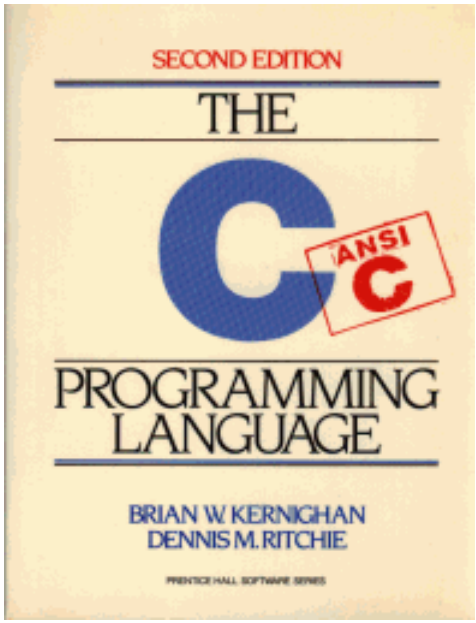
Quel langage pour l'image ?

	Fortran	Pascal	Lisp	JAVA	C	C++	Python	Matlab
Simplicité	- / +	+++	+	++ / -	+	++ / -	+ + +	+ +
Expressivité	-	+	- -	++	-	++	+	-
Efficacité	- / +++	-	-	- -	+++	++	- -	-
Portabilité	+	-	-	++	+++	+++	- -	-
Réutilisabilité	- / ++	+	+	++	+	+++	+ +	+ +
"Propre"	+ +	+ + +	-	+ +	-	+ + +	- - -	- -

Choix : C ou C++

↳ typé fortement

Le langage C



État civil :

Date de naissance : 1975

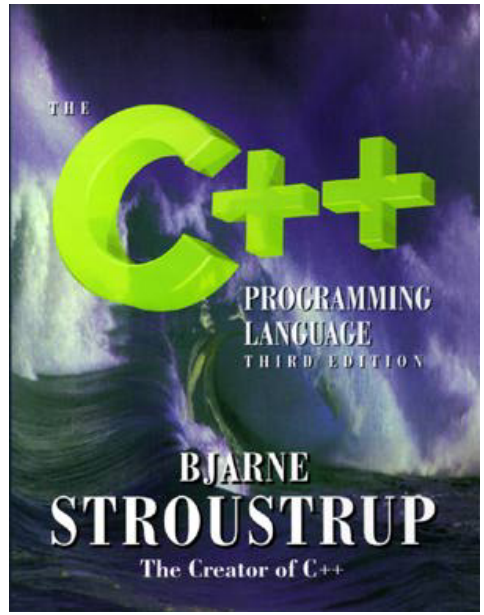
Lieu : Bell Labs

Parents : Brian W. Kernighan et Dennis M. Ritchie.

Caractéristiques :

- langage normalisé (ANSI C)
- langage impératif
- compilé — compilateur `cc` sur tout Unix
- proche du système, très efficace — peu adaté au calcul numérique
- faiblement typé — pb corrigé dans la version ANSI
- stable

Le langage C++



État civil :

Lieu : AT&T Labs

Date de naissance : 1983 chez ATT, Octobre 1985

Parents : Bjarne Stroustrup

Caractéristiques : celle du C plus

- Ansi C $\subset$ C++
- langage normalisé (ISO C++), impératif, compilé (eg, compilateur $g++$), fortement typé.
- abstraction des données
- langage orienté objets
- programmation générique

Qualité et défaut de C/C++

Avantages

- Langage très utilisé, très bien documenté
- Portabilité (OS, hardware, libraries, ...)
- Gestion dynamique de la mémoire (très efficace)
- Langage très stricte (C++)
- Langage objet (C++)
- Compilateurs très efficaces

Défauts

- Langage de bas niveau (C)
- Langage permissif (C)
- Gestion dynamique de la mémoire (très dangereux : absence de "*garbage collector*"... mais si pratique...)

Rappels de C

- Programmes de base (“Hello world”)
- Types élémentaires
- Constructeur élémentaire ([], `struct`, ...)
- Structure de contrôle (`if`, `while`)
- Allocation de la mémoire
  - Notion de pointeur
  - Mémoire dynamique
- Entrée/sortie

Extension au C++

- Programmes de base (“Hello world”)
- Constructeur élémentaire (`class`...)
- Allocation de la mémoire en C++, E/S en C++
- Notion de template

Quelques rappels de C

C est un langage très limité,

proche du système,

peu de types de bases,

peu de structure de contrôle,

peu de contrôles de type,

...

mais on va faire avec... car

...

puissant,

rapide,

simple.

Quelques rappels de C

- Types, opérateurs et expressions
- La structure des programmes
  - instructions et blocs
  - affectations
  - expressions conditionnelles
  - structures de boucle
- Fonctions et pointeurs
- I/O

Types élémentaires et leurs tailles

Très peu de types de base en C

`char`	caractère	signé	[-127..128]	8 bits
`unsigned char`	caractère	non signé	[0..255]	8bits
`short`	entier	signé	[-32767..32768]	16 bits
`unsigned short`	entier	non signé	[0..65536]	16 bits
`int`	entier	signé	[-2e32+1, 2e32]	32 bits
`long`	entier	signé	[-2x+1, 2x]	32 bits
`float`	réel	signé	[]	32 bits
`double`	réel	signé	[]	64 bits

Remarque 1 : absence de booléen ! FAUX = 0, VRAI = x ($x \neq 0$)

Remarque 2 : le “type” `void` signifie sans type...

Déclarations, variables

Les variables doivent être déclarées avant d'être utilisées :

```
int lower ;
int upper ;
float f ; float g ;
unsigned char c ;
```

```
int upper, lower ;
float f, g ;
unsigned char c ;
```

Initialisation lors de la déclaration

```
int lower  = 0 ;
int upper = 10 ;
float f = 1.0e-10 ;
char c = 'e' ;
```

Operateurs

Opérateurs de calcul

— Opérateur de calcul binaire : `+, -, *, /, %`

```
(( a * b) + c ) - d
```

Attention aux priorités !

— Opérateurs unaires : `+, -, ++, --`

`-a`

`a--` a-1 (décrémentation)

`a++` a+1 (incrémentation)

Opérateurs de relation et logiques

— Opérateur de relation : `>, <, >=, <=`

— Opérateur d'égalité/inégalité : `==, =!`

— Autres opérateur logiques : `&&` (et), `||` (ou), `!` (non)

La structure des programmes

— instructions, séquentialité

— structure conditionnelles

— structure de boucles

Instructions et blocs

Une expression de la forme `x=0` ou `i++` devient une instruction quand elle est suivi d'un point virgule "**;**"

```
instruction_A ;
instruction_B ;
```

"**;**" est un terminateur.

Les accolades { et } sont utilisées pour regrouper les instructions dans une instruction composée

```
{
    instruction_A ;
    instruction_B ;
}
```

} est **une seule** instruction

Affectations : opérateur et expressions

Affectation simple :

```
a = 2 ;
a = b + 10 ;
i = i + 2 ;
j = b++ ;
```

Autres opérateurs:

```
i += 2 ;
```

équivalent à

```
i = i + 2 ;
```

Expressions conditionnelles : La structure if-else

Conditionnelle :

```
if (expression)
      instruction_si_expression_VRAI ;
else
      instruction_si_expression_FAUX ;
```

Exemple :

```
if (a > b)     {
   z = a ;
   y = b ;
}
else
   z = b ;
```

Remarque : le `else` est facultatif

Expressions conditionnelles : l'instruction `switch`

`switch` est équivalent à `if ... else if ... else if ....`

```
char c = getchar() ;
swicth (c) {
  case '0' : i += 1  ; break ;
  case '1' :
  case '2' : i += 2 ;  break ;
  default  : i += 4 ; break ;
}
```

Remarque : le `default` est facultatif

Les boucles `while` et `for`

Instruction `while`

```
while (expression) instruction ;
```

`instruction` est exécuté tant que `expression` est non nulle (vrai)

Instruction `for`

```
for (expression1 ; expression 2 ; expression 3)
  instruction ;
```

équivalent à

```
expression1 ;
while (expression2) {
  instruction ;
  expression3 ;
}
```

Boucle `for` : exemples

Calcul du factoriel 10

```
int i ;
int f = 1 ;
for (i=1 ; i <= 10 ; i++) f = f * i ;
```

équivalent à

```
int i = 1 ;
int f = 1 ;
while (i<=10) {
    f = f * i ;
    i++ ;
}
```

Remarque : `for (;;) instruction ;` est une boucle infinie

Boucle `do ... while ;`

`while` et `for` testent la condition de fin de rebouclage au début de la boucle.

`do-while` la teste à la fin.

```
do
    instruction ;
while (expression) ;
```

l'`instruction` est exécutée puis l' `expression` testée, si elle est vrai `instruction` est réexécutée, sinon on sort de la boucle.

Pointeurs

Définition : Pointeurs

Un pointeur est une variable destinée à contenir une adresse mémoire.

Pointeur = zone dangereuse

“C’est un moyen extraordinaire de créer des programmes incompréensibles” dixit K&R

Leur utilisation est indispensable mais... à manier avec beaucoup de rigueur

Pointeurs

Éléments de syntaxe :

```
int x = 1 ; int y = 2 ;
```

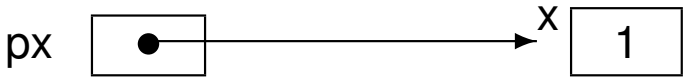
`x` et `y` sont des variables de type `int`


```
int *px = NULL ;
```

`px` est un pointeur sur un entier


```
px = &x ;
```

`&x` désigne l'adresse de la variable `x` en mémoire




```
y = *px ;
```

`*px` désigne le contenue de la variable d'adresse px

Complétez le tableau suivant pour chaque instruction du programme ci-dessus.

```
void main()     {
    int A = 1;
    int B = 2;
    int C = 3;
    int *P1, *P2;
    P1=&A;
    P2=&C;
    *P1=(*P2)++;
    P1=P2;
    P2=&B;
    *P1-=*P2;
    ++*P2;
    *P1*=*P2;
    A=++*P2**P1;
    P1=&A;
    *P2=*P1/=*P2;
}
```

	A	B	C	P1	P2
Init	1	2	3	/	/
P1=&A;	1	2	3	&A	/
P2=&C;	1	2	3	&A	&C
*P1=(*P2)++;	3	2	4	A	C
P1=P2;	3	2	4	C	C
P2=&B;	3	2	4	C	&B
*P1-=*P2;	3	2	2	C	B
++*P2;	3	3	2	C	B
P1=*P2;	3	3	6	C	B
A=++*P2**P1;	24	4	6	C	B
P1=&A;	24	4	6	A	B
*P2=*P1/=*P2;	6	6	6	A	B

Fonctions

Objectif :

- Diviser les tâches importantes de façon à écrire de petits programmes
- Modularité de votre code
- “Dissimuler” des détails de calcul

Syntaxe

```
type nom(liste_d_arguments)
{
  instruction ;
}
```

exemple :

```
int fact(int n) {
  if (n==1)
    return 1 ;
  else
    return n*fact(n-1) ;
}
```

Les arguments des fonctions et zone de validité

Remarque importante : les arguments sont transmis par valeur. La fonction reçoit une copie provisoire, qui lui est propre, de chaque argument.

```
int f(int y) {
int z=1 ; y = y+1 (2) ; z += y ; return z ; (3)
}

void main() {
int y = 2 ;
int x = 0 ; (1) x = f(y) ; (4)
}
```

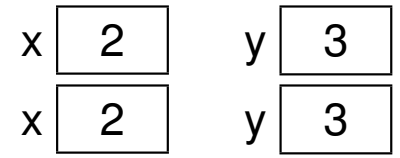
Donner les valeurs de `x` et `y z` en (1), (2), (3) et (4)

Pointeurs et arguments de fonction (1)

Puisque les arguments sont transmis par valeur, il n'existe pas de moyen directe de modifier un des paramètres d'une fonction.

Exemple

```
void swap (int a, int b) {
int tmp = a;
a = a ;
b = tmp ;
}

void main() {
int x = 2 ; int y = 3 ;
swap (x,y) ;
}
```

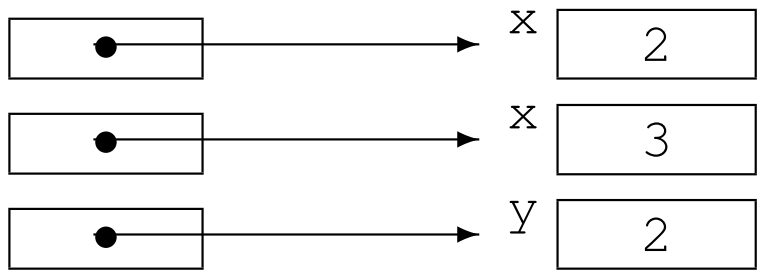


On a rien permuté... à cause de l'appel par valeur

Pointeurs et arguments de fonction (2)

Solution : ne pas passer la variable elle même, mais son adresse (pointeur)

```
void swap (int *a, int *b) {

int tmp =*a;
**a = *b ;
**b = tmp ;

}

void main() {
int x = 2 ; int y = 3 ;
swap (&x,&y) ;
}
```



x 2 y 3

x 3 y 2

`&x` et `&y` ne sont pas modifiées. Leur contenu, oui.

Les entrées/sorties en C

Par défaut, il n'y a pas d'entrée sortie en C !

Il faut utiliser les fonctions d'une bibliothèque particulière `stdio`.

```
#include<stdio.h>


void main() {
  int a ;
  printf("Hello, world \n") ;
  scanf("%d",&a) ;
  printf("a = %d \n",a) ;
}
```

Lecture de caractères isolés : `getchar`

Elle a pour rôle de lire un caractère en entrée.

Elle renvoie un caractère isolé lu sur l'entrée standart (généralement le clavier).

```
char c;
c = getchar();
```

Écriture de caractères isolés : `putchar`

Son rôle est de permettre l'affichage de carctères isolés

```
char c;
putchar(c);
```

Transmet la valeur actuelle de la variable c au périphérique standart de sortie pour affichage.

Entrée de données : scanf

scanf permet de saisir des valeurs numériques, des caractères simples ou des chaînes.

La syntaxe générale de la fonction scanf est la suivante:

```
scanf(format, arg1, arg2, arg3,... , argn)
```

```
#include <stdio.h>
main(){
  int reference;
  float prix;
  int nb = scanf ( "%d␣␣%f␣", &reference, &prix);
}
```

%d : type entier décimal (&reference)

%f : type numérique flottant (&prix)

scanf renvoie le nombre d'éléments lu

Écriture de données en sortie `printf`

```
printf(chaine, arg1, arg2, ..., argn) ;
```

La chaîne inclut des symboles spéciaux qui seront remplacés par le contenu des variables qui suivent la chaîne.

Ainsi le contenu de la variable est affichée suivant

- %d, %i : le format entier signé,
- %o : le format octal,
- %u : le format non signé,
- %x, %X : le format hexadécimal,
- %f : le format flottant : exemple %10.2f
- %c : le format caractère
- %s : le format chaîne de caractères

Le caractère \ permet d'afficher des caractères spéciaux :

- \n introduit un retour chariot

Exercice

Écrivez un programme qui lit la date du clavier et écrit les données ainsi que le nombre de données correctement recues sur l'écran.

Exemple:

```
Introduisez la date (jour mois annee): 11 11 1991

donnees recues : 3
jour  : 11
mois  : 11
annee : 1991
```

Exercice

Ecrire un programme qui lit un caractère au clavier et affiche le caractère ainsi que son code ASCII

Retour aux images...

Nous verrons au passage

- l'allocation dynamique de la mémoire
- la notion de structure `struct`
- l'accès aux fichiers
- etc.

Natures des images numériques

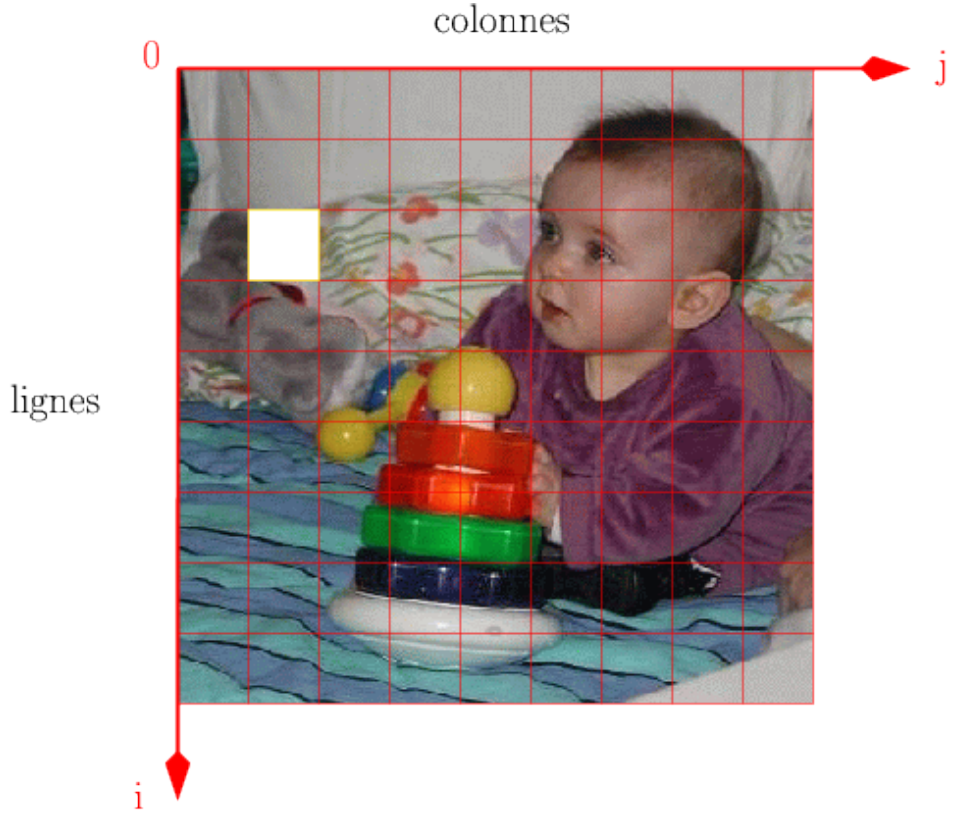


caractéristique de chaque pixel

- noir et blanc — $p \in \{ \text{blanc, noir} \}$
- niveaux de gris — $p \in [0..255]$
- couleur — $p = \{R,G,B\} / R,G,B \in [0..255]$

image = “matrice” de pixel

Représentation des images



En jaune, le pixel (2,1) :ligne 2, colonne 1 ; numérotation à partir de 0

Une image est définie par :

- Sa taille : nrows, ncols
- La “matrice” image contenant nrows $\times$ ncols pixels

Représentation des images en mémoire

La notion de tableau en C

La notion de tableau est très liée à celle de **pointeur**.

La déclaration

`int a[10] ;` définit un tableau `a` de taille 10.

Ce sont 10 objets, consécutifs en mémoire, baptisés `a[0], a[1],..., a[9]`

en fait `a` est un pointeur sur un entier. Si on a :

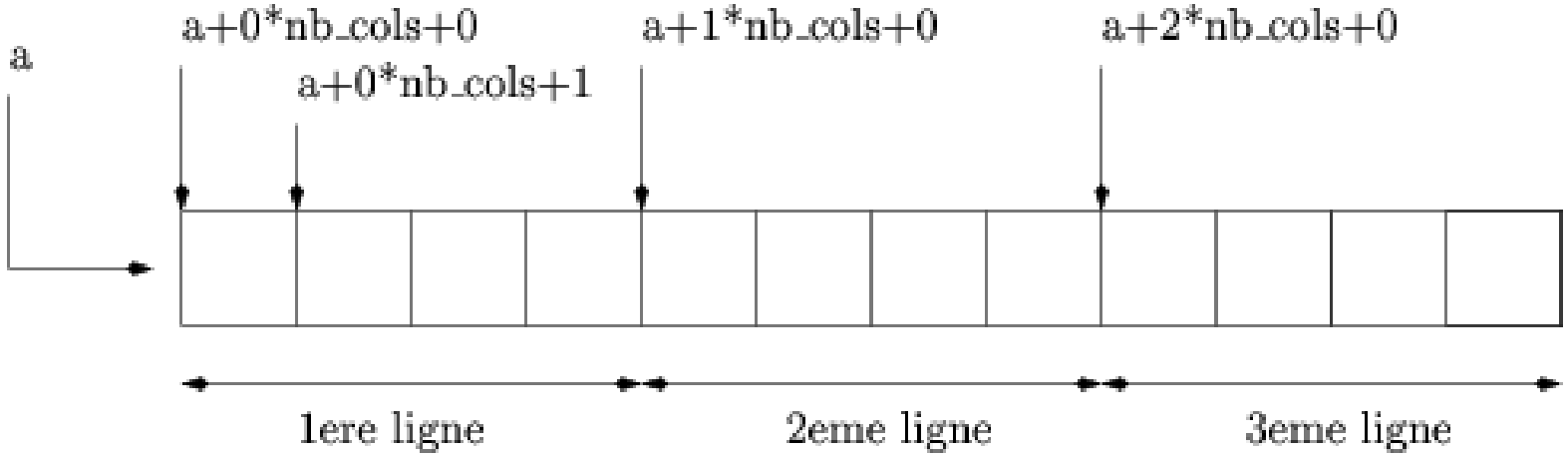
`int *pa ;` Alors l'affectation `pa = a ;` est valide

Remarque :

- `a` est égale à `&(a[0])`
- La notation `a[i]` est équivalente à `*(a+i)`
- `a+i` est un pointeur sur la ième case du tableau

Représentation par un tableau à une dimension

```
#define nrows 256
#define ncols 256
unsigned char I[nrows * ncols] ;
```



Accès à ième ligne jème colonne : `*(a + i*ncols + j)`

Représentation des images en mémoire
première méthode : tableau à une dimension

Image en niveau de gris : chaque pixel $\in$ [0..255]

$\Rightarrow$ codage sur un `unsigned char`

```
#define nrows 256
#define ncols 256
unsigned char I[nrows * ncols] ;
for (i=0 ; i < nrows ; i++)
  for (j=0 ; j < ncols ; j++)
    I[i* ncols + j] = 123 ;
```

Ce type de mise en œuvre pose plusieurs problèmes importants:

- Besoin de connaître *a priori* la taille de l'image
- Accès peu intuitif

C et les tableaux bidimensionnels

`int a[2][3] ;` définit un tableau à deux dimensions (2 lignes, 3 colonnes)

Accès : On y accède de la façon suivante `a[i][j]`

En mémoire : tableau de tableaux

`a` est un pointeur sur des tableaux de taille **fixée** : `3` ici

Ainsi l'affectation `int (*pa)[3] ; pa = a ;` est légale

- `&a[i][0] = a[i] = *(a+i)` est l'adresse de la ième ligne
- et `*(a+i)+j` est l'adresse de la case i,j
- Finalement, `a[i][j]` est équivalent à `*(*(a+i)+j)`

Attention :

`int (*pa)[3] ;` est différent de `int *a[3] ;` qui est un tableau de 3 pointeurs sur des entiers

Représentation des images en mémoire
Seconde méthode : tableau à deux dimensions

```
#define nrows 256
#define ncols 256
unsigned char I[nrows][ncols] ;
for (i=0 ; i < nrows ; i++)
  for (j=0 ; j < ncols ; j++)
     I[i][j] = 123 ;
```

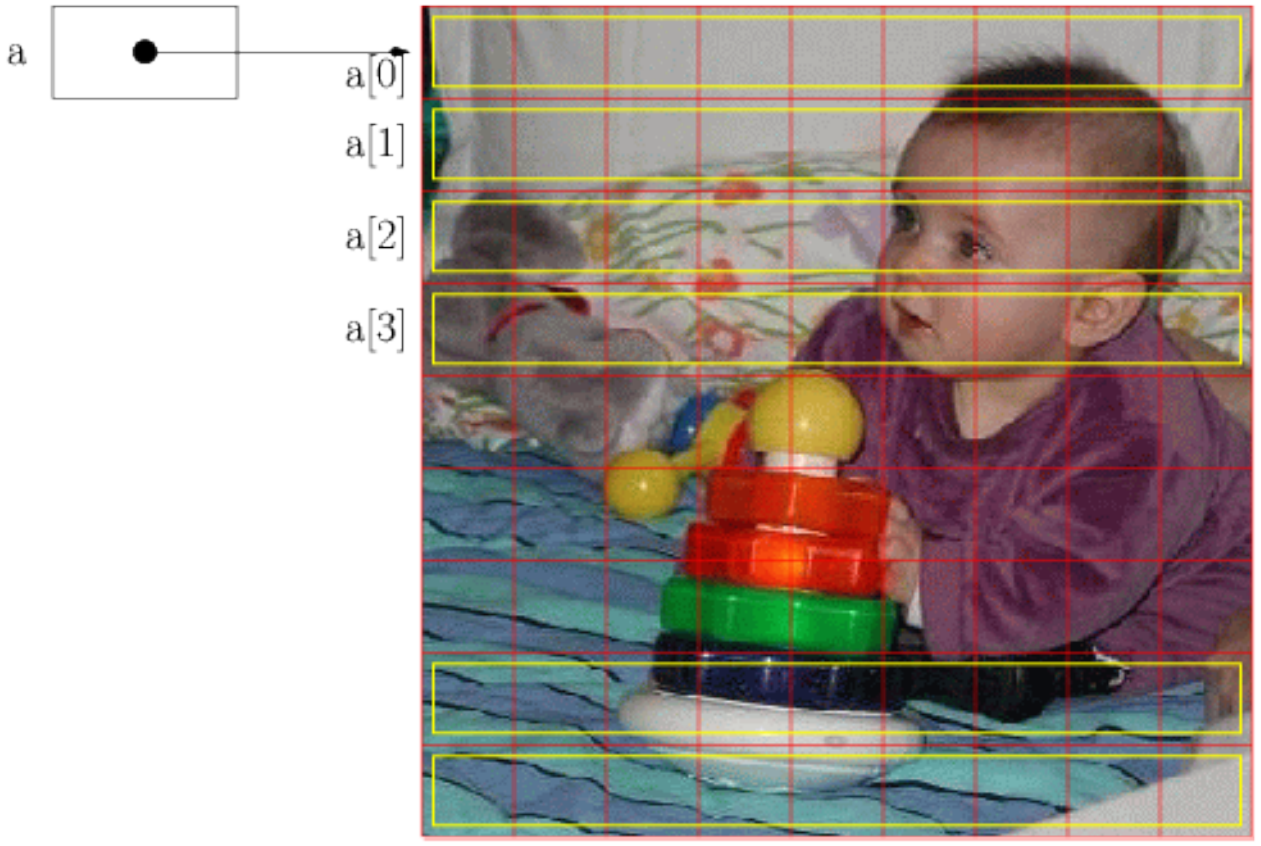
Ce type de mise en œuvre pose un problème majeur :

— Besoin de connaître *a priori* la taille de l'image

Mais

— Beaucoup plus lisible

Représentation des images par des tableau à deux dimensions



Problème avec l'allocation statique

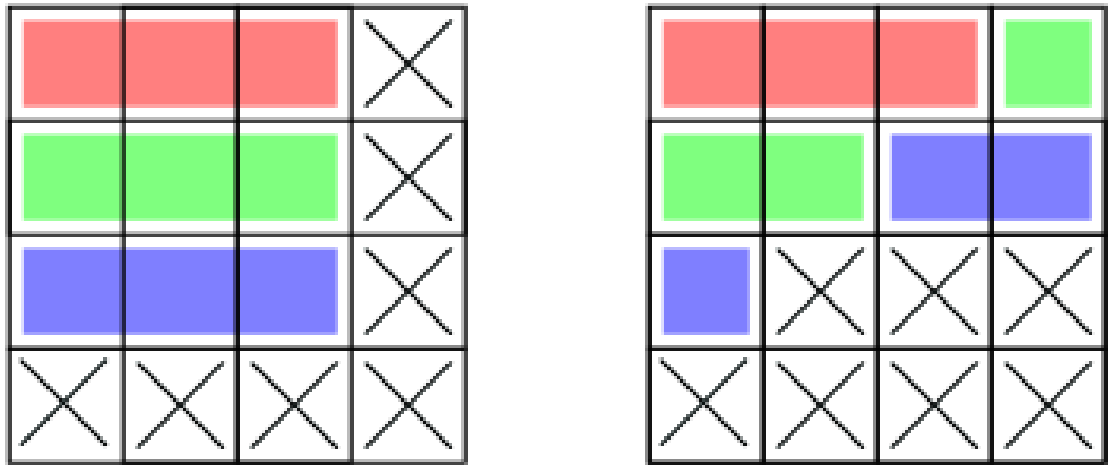
Problème principal des méthodes précédentes :

L'allocation statique de la taille des tableaux.

Peu pratique (comment connaitre a priori la taille d'une image lue sur disque ?)

$\Rightarrow$ Surdimensionné $\Rightarrow$ Gaspillage potentiel de mémoire

Représentation en mémoire d'une image 3 x 3 si on a déclaré un 4x4

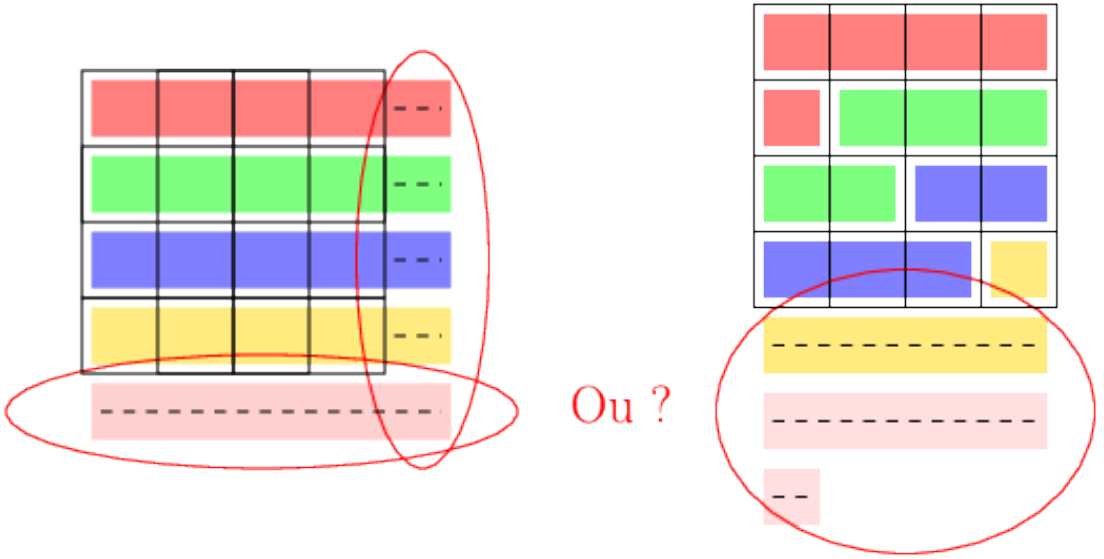


`unsigned char a[4][4]` ou `unsigned char a[16]`

Problème avec l'allocation statique

On ne résout pas vraiment le problème en augmentant la taille...

Représentation en mémoire d'une image 5 x 5 si on a déclaré un 4x4



`unsigned char a[4][4]` ou `unsigned char a[16]`

Allocation dynamique de la mémoire

Structure de donnée statique : créée à la compilation

$\Rightarrow$ impossible de les adapter aux données traitées

Structure de donnée dynamique : crées lors de l'exécution des programmes

$\Rightarrow$ l'utilisateur alloue une zone mémoire de taille quelconque

Petit rappel de système, et de C...

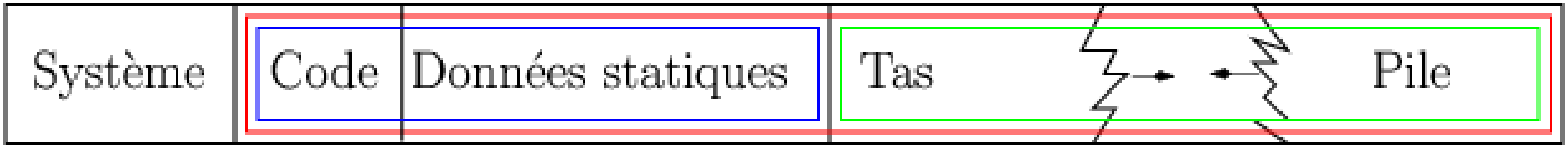
Zônes mémoires

— Système

— Mémoire statique

  — Code + constante — lecture seule

  — Données statiques — taille connue à la compilation



— Mémoire dynamique

  — Tas : Allocation dynamique par l'utilisateur

  — Pile : système, paramètres de fonctions, etc.

Allocation de mémoire dynamique

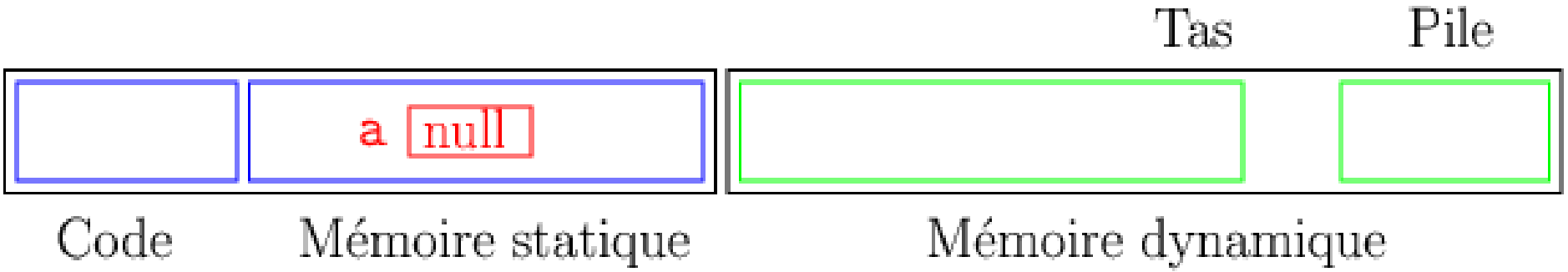
```
int *a ;
```

`a` est un pointeur sur des entiers

`a` est une variable statique

```
a = NULL ;
```

`a` est un pointeur qui ne pointe (encore) sur rien



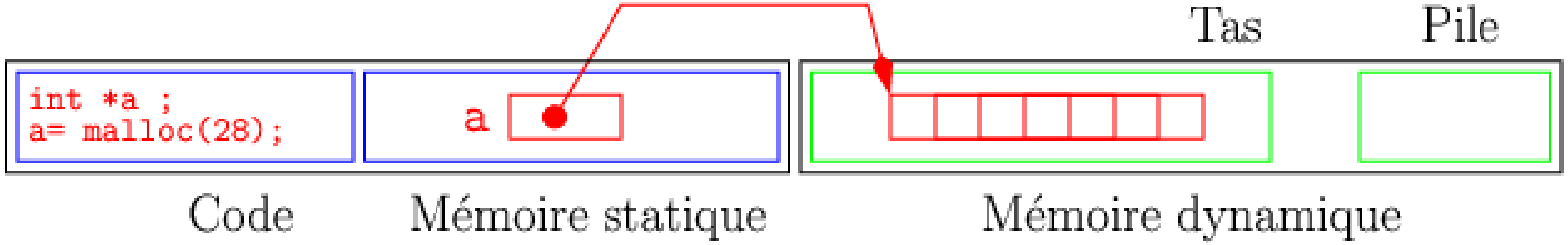
Objectif : allouer une zone mémoire sur laquelle `a` pointera

Allocation de mémoire dynamique : malloc(...)

La fonction malloc(...)

```
int nb_elt = 7 ;

a = (int *)malloc(nb_elt * sizeof(int)) ;
```

— alloue une zone mémoire de 28 octets (7 * `sizeof(int)`)



— cette zone mémoire est située dans le tas

— les 28 octets sont contigus

— `malloc()` renvoie l'adresse du premier élément

— **Typage** : `void *malloc(int size);`

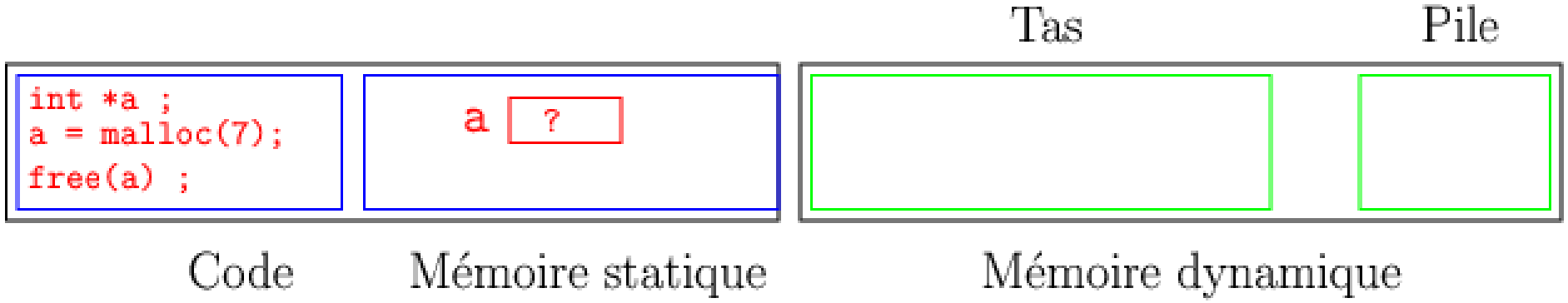
⇒ il faut donc convertir le résultat en fonction du type du pointeur `(int *)`

Liberation de la mémoire : `free(...)`

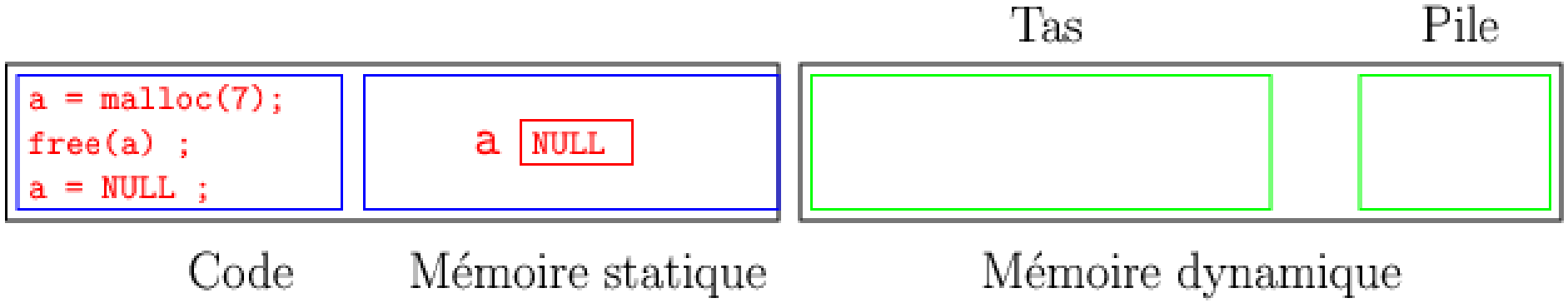
`void free(void *ptr)` : `free()` libère la zone mémoire pointée par `ptr`.

Le pointeur `ptr` doit avoir été initialisé grâce à un `malloc()`

```
if (a != NULL) free(a) ;
```



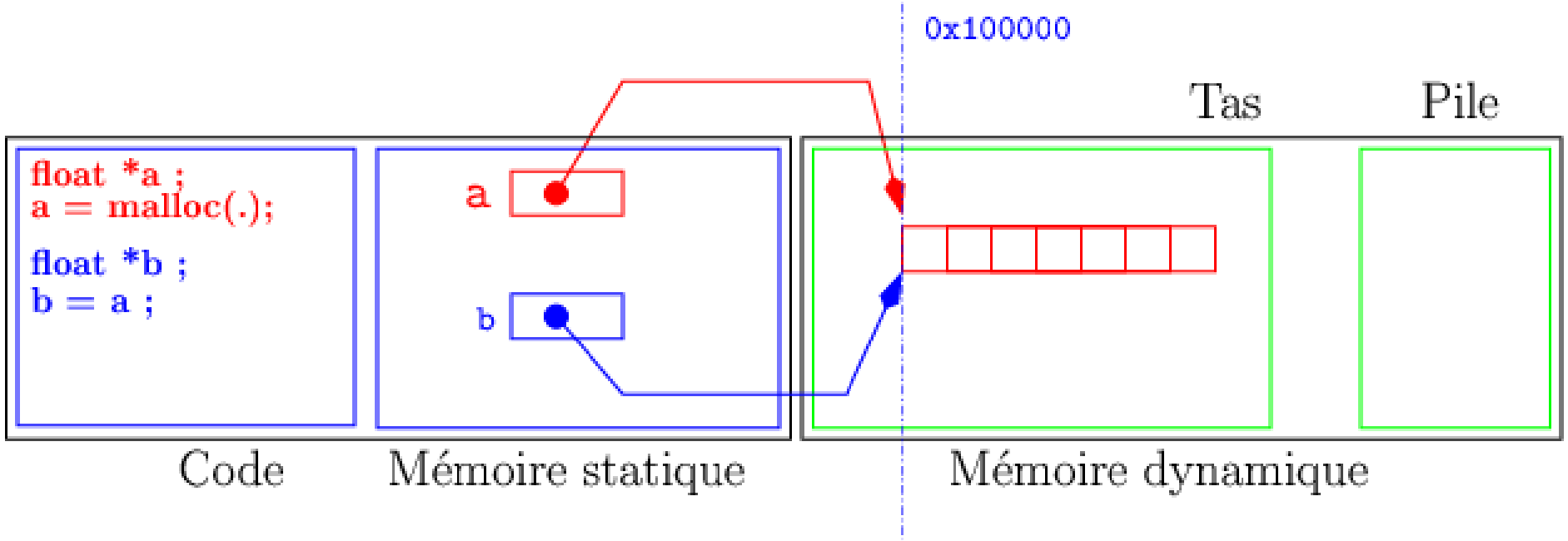
Mettre le pointeur `a` à `NULL`



Problème des références multiples

Il est possible que deux pointeurs fassent référence au même objet !

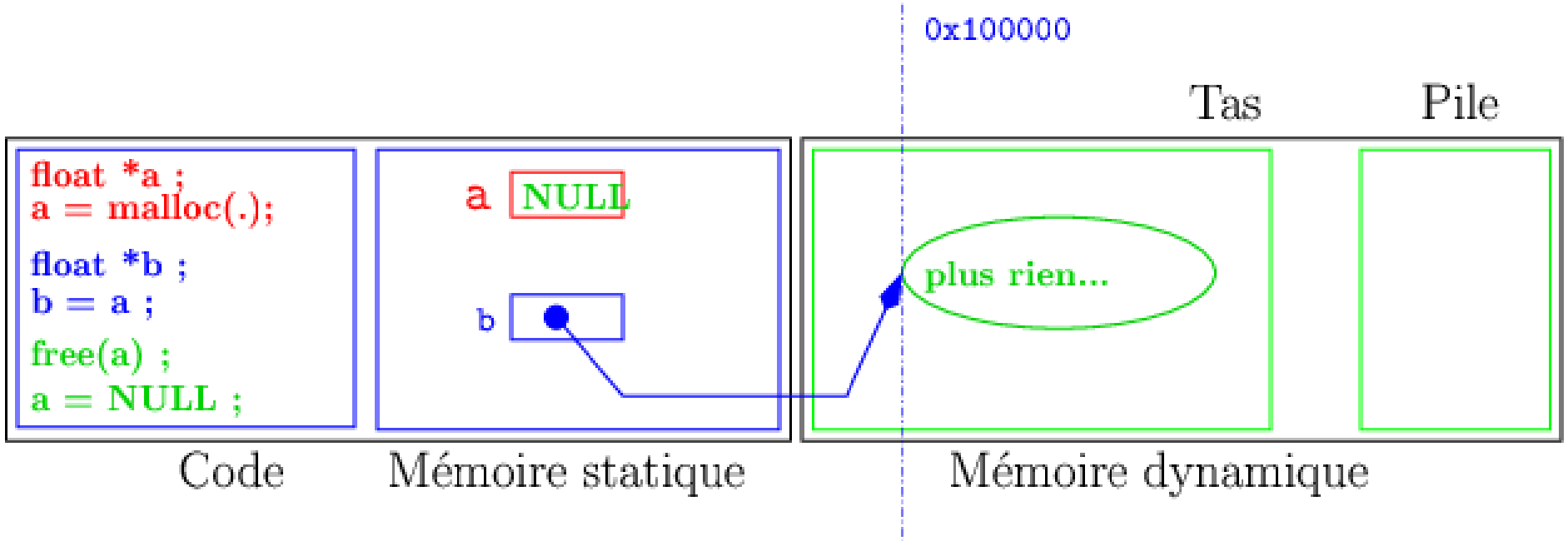
```
float *a ;
a = (float *)malloc(10*sizeof(float)) ;
float *b ;
b = a
```



a = 0x100000 et b = 0x100000

Problème des références multiples (2)

```
free(a) ; a = NULL ;
```



a = NULL et b = 0x100000 encore !!!

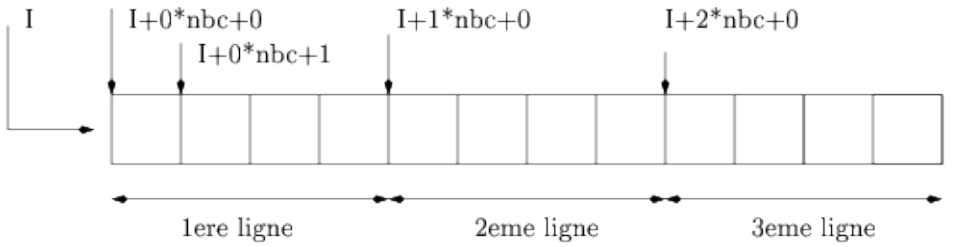
`b` pointe toujours vers une zone mémoire qui a été désallouée (qui n'existe plus)

un accès à `b[1]` ⇒ `Segmentation fault`

Représentation des images en mémoire
Troisième méthode : allocation dynamique et tableau

Allocation dynamique d'un tableau unidimensionel

```
uchar *bitmap ;
bitmap = (uchar *)malloc(ncols*nrows*sizeof(uchar)) ;
for (i=0 ; i < nrows ; i++)
  for (j=0 ; j < ncols ; j++)
    bitmap[i* ncols + j] = 123 ;
...
free(bitmap) ;
```



L'accès reste peu intuitif

Cela reste la solution la plus classique...

Représentation des images en mémoire
allocation dynamique et tableau (2)

La définition d'un tableau unidimensionel est presque indispensable.

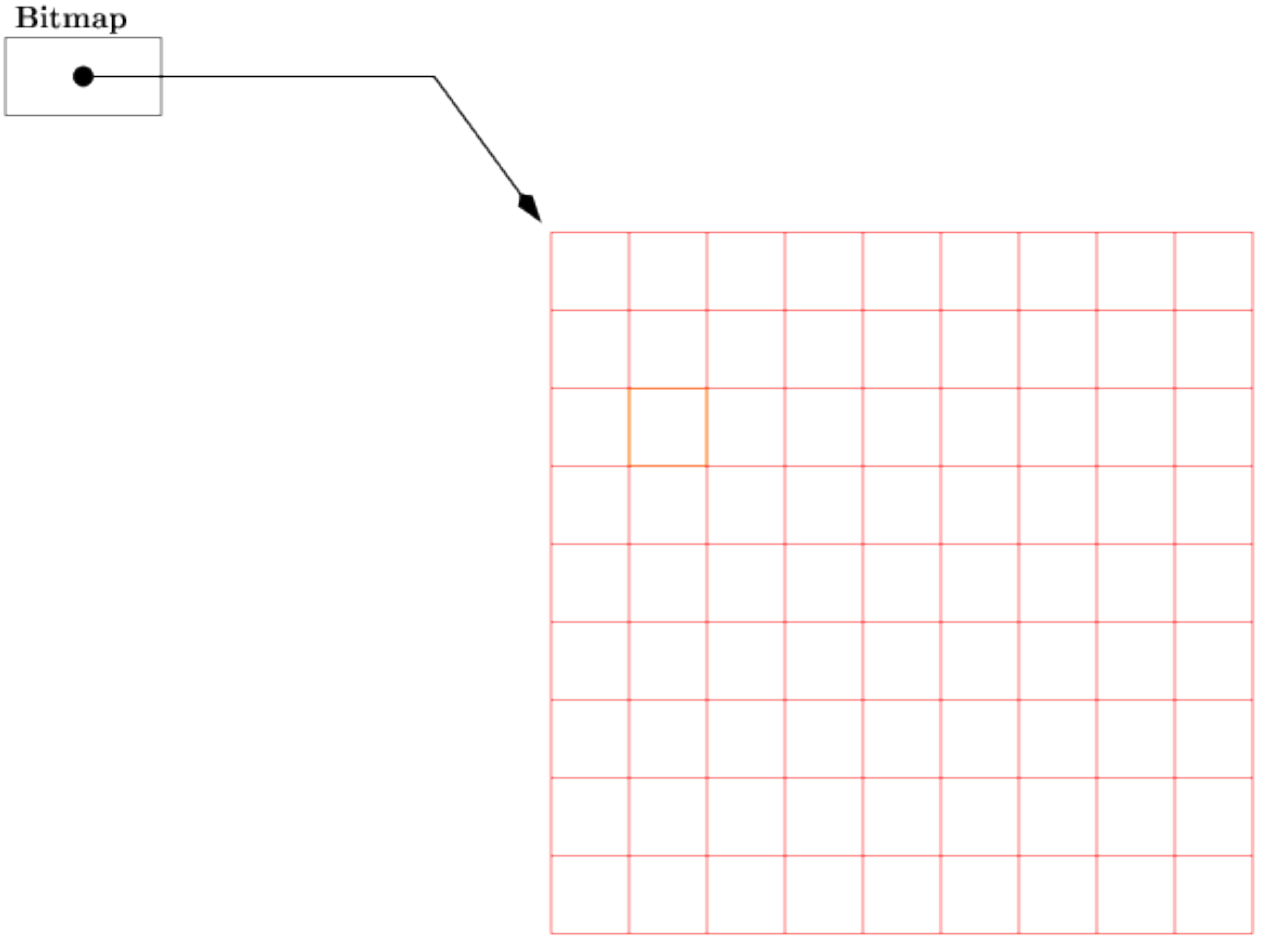
```
uchar *bitmap ;
bitmap = (uchar *)malloc(ncols*nrows*sizeof(uchar)) ;
```

Mais on peut faciliter l'accès aux données

```
uchar **I ;
I = (uchar **)malloc(nrows*sizeof(uchar *)) ;
for ( i =0  ; i < nrows ; i++)
    I[i] = bitmap + i*ncols ;
...
I[10][12] = 123 ;
...
free(I) ; free(bitmap) ;
```

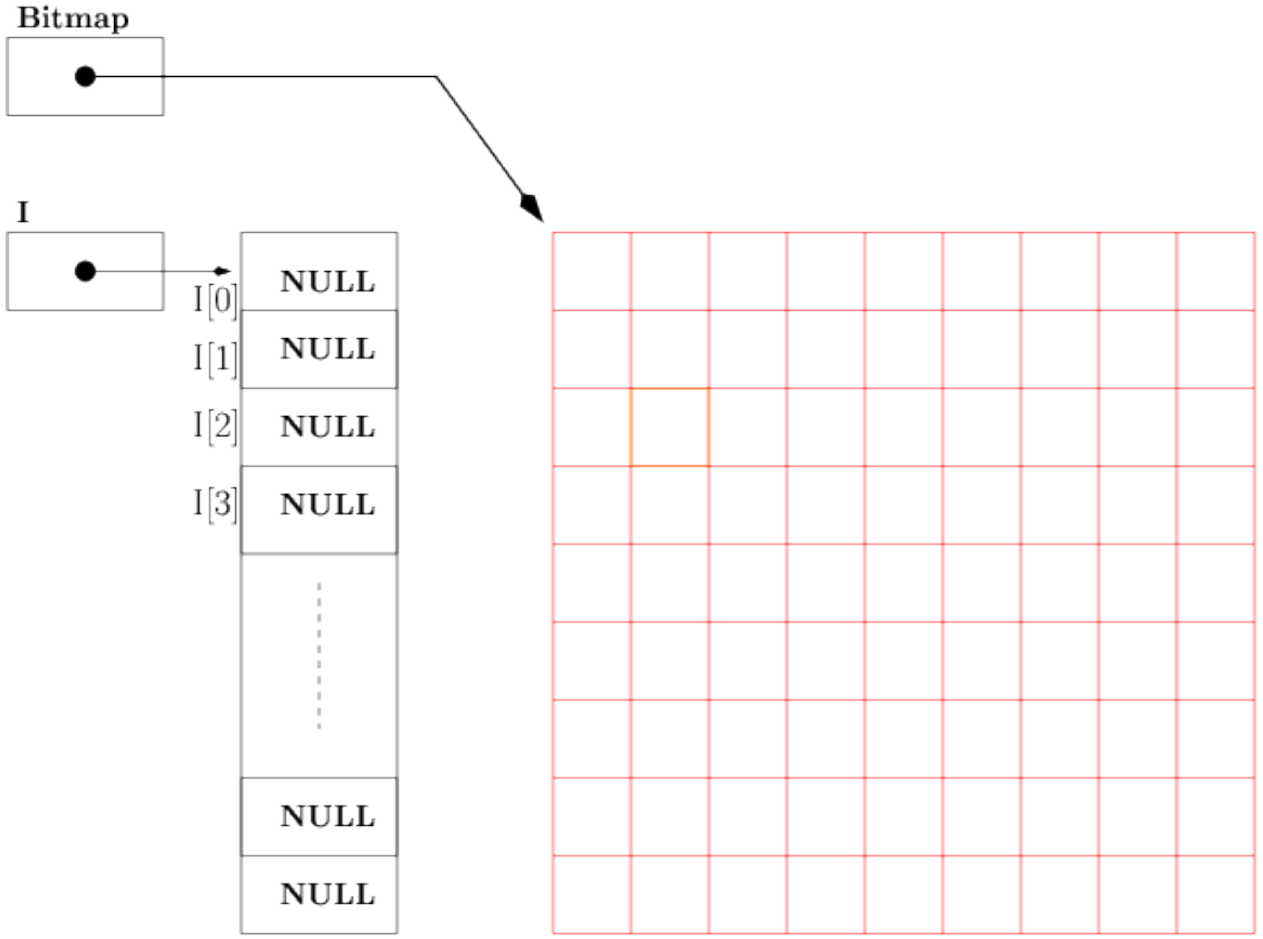
Représentation des images en mémoire
allocation dynamique et tableau (2)

```
uchar *bitmap ;
bitmap = (uchar *)malloc(ncols*nrows*sizeof(uchar)) ;
```



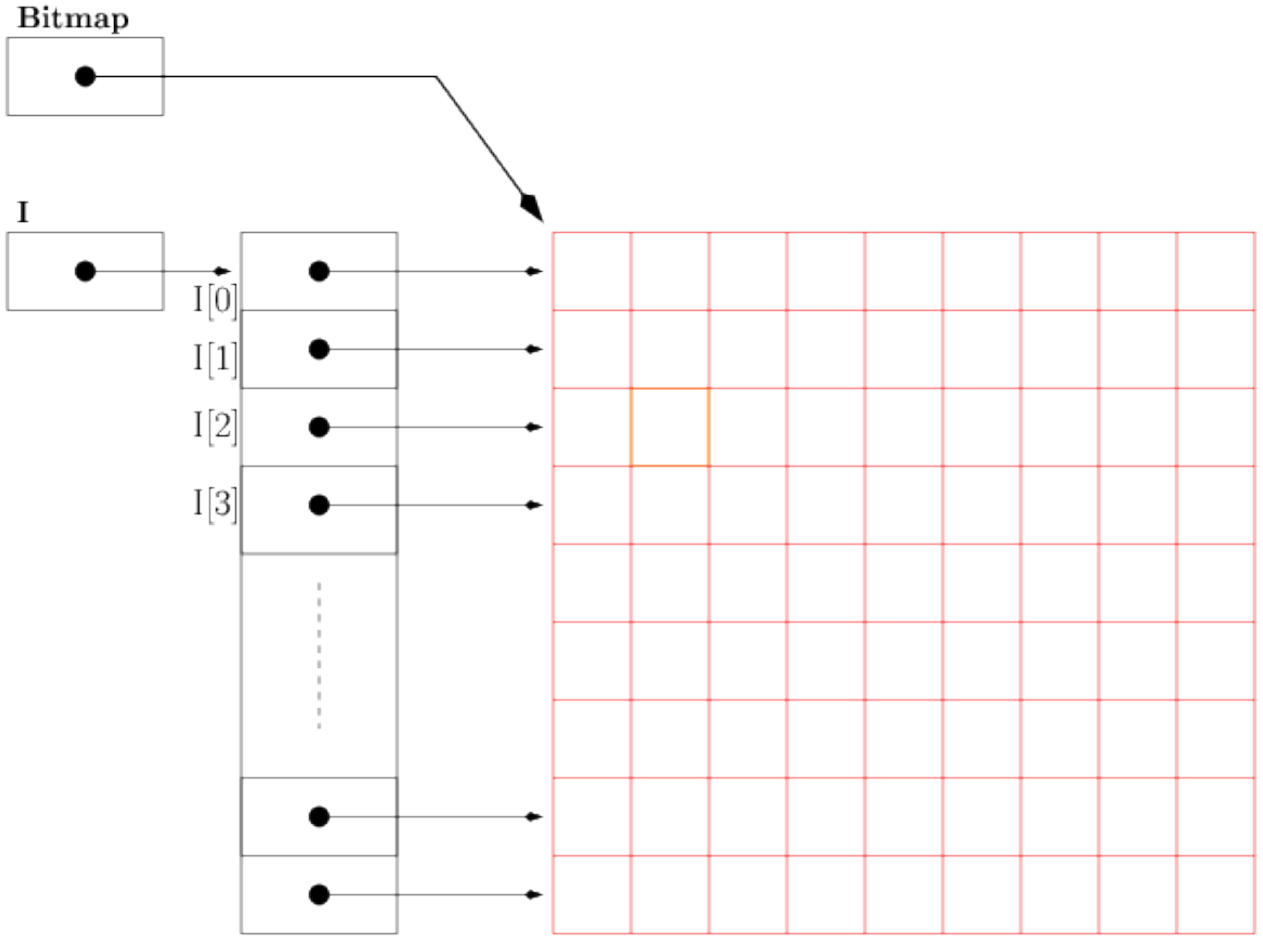
Représentation des images en mémoire
allocation dynamique et tableau (2)

```
uchar **I ;
I = (uchar **)malloc(nrows*sizeof(uchar *)) ;
```



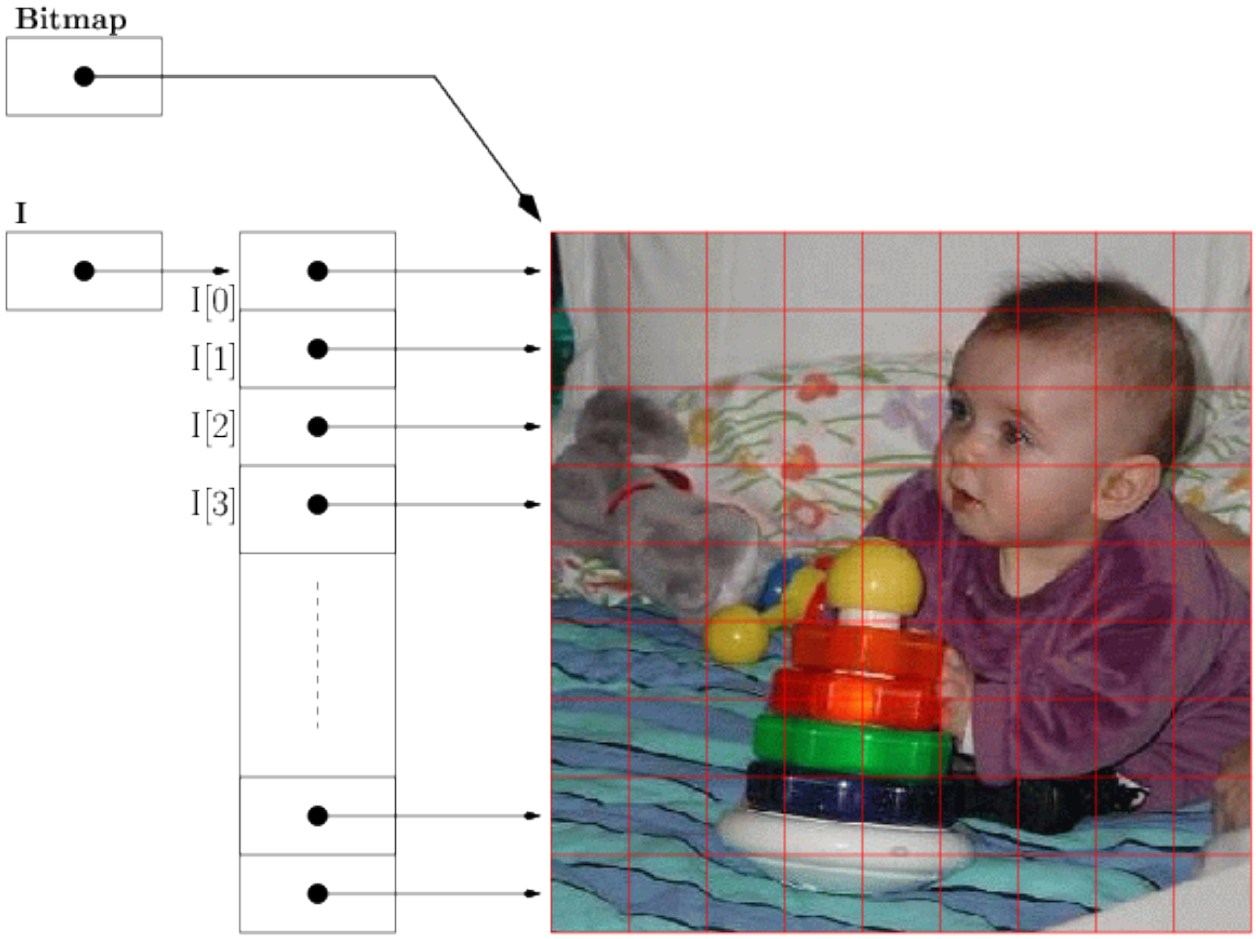
Représentation des images en mémoire
allocation dynamique et tableau (2)

```
for ( i =0  ; i < nrows ; i++)
    I[i] = bitmap + i*ncols ;
```



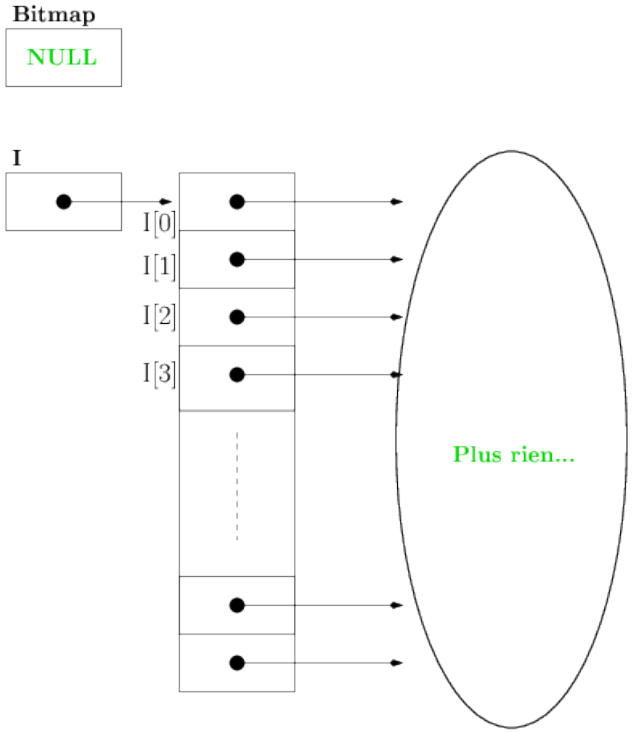
Représentation des images en mémoire
allocation dynamique et tableau (2)

```
for(i =0  ; i < nrows ; i++)  for(j =0  ; j < ncols ; j++)
    I[i][j] = random(...) ;
```

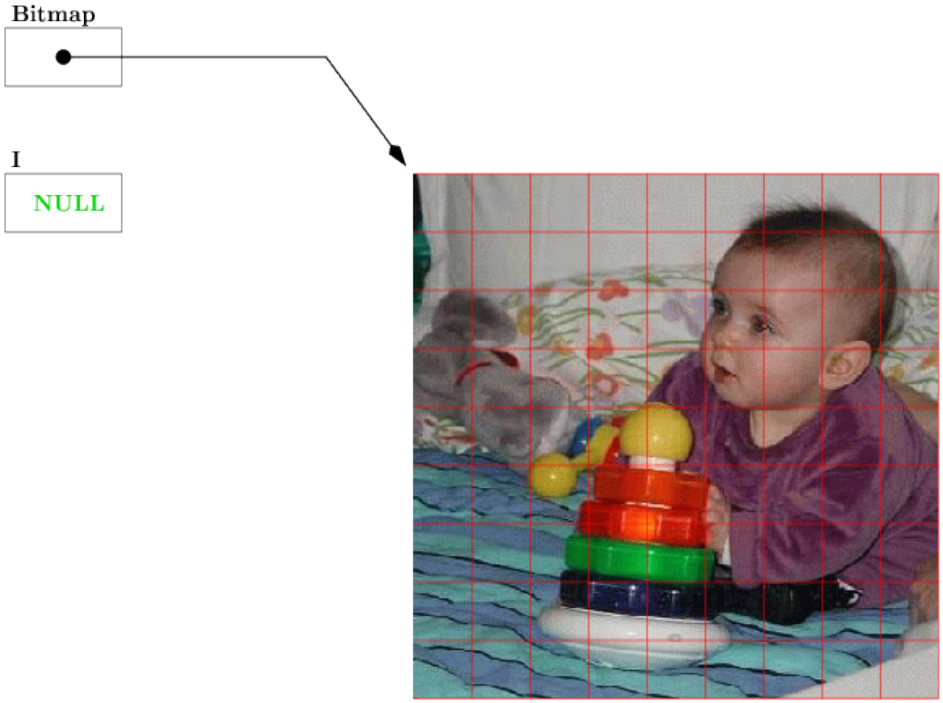


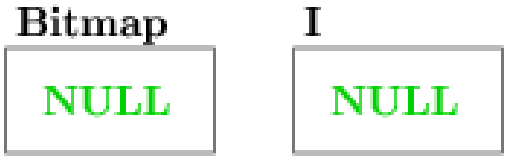
Représentation des images en mémoire
allocation dynamique et tableau (2)

```
free(bitmap) ; bitmap = NULL ;
```




```
free(I) ; I = NULL ;
```






```
free(I) ;
```

```
free(bitmap) ;
```

Autre approche

Et la structure de données suivante...

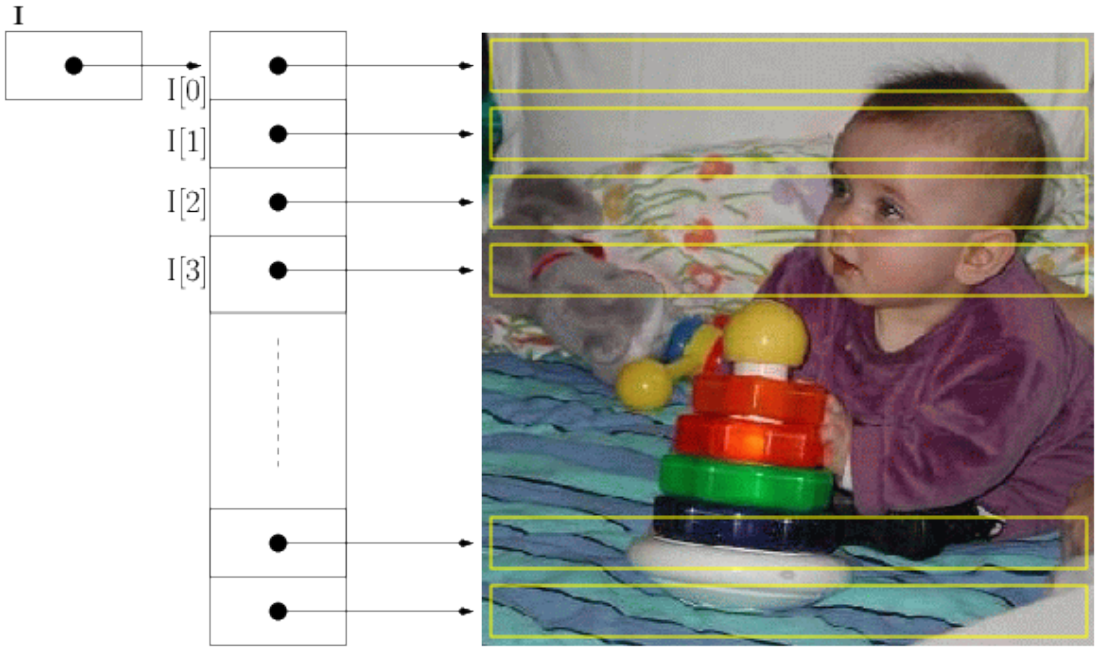
```
uchar **I ;
I = (uchar **)malloc(nrows*sizeof(uchar *)) ;
for ( i =0  ; i < nrows ; i++)
    I[i] = (uchar *)malloc(ncols*sizeof(uchar)) ;
```

Décrivez cette approche

Qu'en pensez vous ?

Autre approche : tableau de pointeurs sur des tableaux

```
uchar **I ;
I = (uchar **)malloc(nrows*sizeof(uchar *)) ;
for ( i =0  ; i < nrows ; i++)
    I[i] = (uchar *)malloc(ncols*sizeof(uchar)) ;
```



Autre approche : tableau de pointeurs sur des tableaux

Avantage :

- I[i][j] permet l'accès aux données
- allocation dynamique

Problème :

- Perte de la contiguïté des donnée en mémoire
  - Plus d'utilisation possible de `∀i, bitmap[i] = x ;`
  - Recopie plus longue (pas de `memcpy`)
  - Les calculs d'adresse sont plus complexes
- Désallocation de la mémoire plus coûteuse

$\Rightarrow$ À ne pas retenir...

Representation des images par des objets structurés

Representation des images par des objets structurés

Objectif : Regrouper toute les informations liées à une image dans le même objet

Notion de *structure* :

Une structure rassemble une ou plusieurs variables, qui peuvent être de types différents, et que l'on regroupe sous un seul nom pour le manipuler plus facilement.

Principes fondamentaux des structures

Définition d'une structure

Considérons un point 2D représenté par des coordonnées x et y entières.

Ces deux composantes peuvent être mise dans une structure ainsi :

```
struct Tpoint {
  int x ;
  int y ;
} ;
```

`x` et `y` s'appellent des **membres**

Déclaration d'une variable

```
struct Tpoint p ;
```

définie une variable de type point

Principe fondamentaux des structures

Initialisation

On peut initialiser une structure en faisant suivre sa définition par une liste de variables initiales.

```
struct Tpoint p = {320, 200} ;
```

Reférence à un membre

Dans une expression, on fait réference à un membre d'une structure en utilisant une construction de la forme :

nom-de-structure . membre

Exemple :

```
pt.x = 320 ;
int a ; a = pt.y ;
```

Operations autorisées sur les structures

— la copie ou l'affectation : la structure est alors un tout !

```
pt1 = pt2 ;
```

est équivalent à

```
pt1.x = pt2.x ;
pt1.y = pt2.y ;
```

— La recupération de son addresse avec l'opérateur `&` : `&pt`

— l'accès à ses membres : `pt.x`

Création de type : `typedef`

Il peut être utile de définir un nouveau type

- code plus lisible
- simplification des notations (ex : on supprime le `struct`)

Exemple 1 :

```
typedef uchar unsigned char ;
uchar v  ;
```

Exemple 2 :

```
struct point {
  int x ;
  int y ;
}  ;
typedef Tpoint point ;
```

```
typedef struct point {
        int x ;
        int y ;
    } Tpoint ;
```

Structures et fonctions

1. fonction rendant une structure

```
Tpoint fct(...) ;
```

2. passage de structure par valeur

```
void fct(Tpoint pt) ;
```

3. passage de l'addresse d'une structure

```
void fct(Tpoint *pt) ;
```

Structures et fonctions
(1) fonction rendant une structure

```
Tpoint CreatePoint(int x, int y)
{
  Tpoint tmp ;
  tmp.x = x ;      tmp.y = y ;
  return tmp ;
}
```

- Pas de conflit entre le nom de l'argument et celui du membre
- le `return` réalise en fait une copie de la structure
- Utilisation

```
Tpoint pt = CreatePoint(2,2) ;
```

Structures et fonctions
(2) passage de structure par valeur

```
Tpoint ComputeMiddlePoint(TPoint pt1, Tpoint pt2)
{
  Tpoint tmp ;
  tmp.x = (pt1.x+pt2.x)/2 ;
  tmp.y = (pt1.y+pt2.y)/2  ;
  return tmp ;
}
```

Structures et fonctions
(3) passage de l'adresse d'une structure

Pointeur de structure Un pointeur de structure est semblable a un pointeur de variable

```
Tpoint *p ;
Tpoint origine ;
p = &origine ;
```

On y accède par : `(*p).x`

Remarque importante : les parenthèses sont nécessaires car la priorité de l'opérateur "." est supérieure à celle de l'opérateur *

`*p.x` est équivalent à `*(p.x)` ce qui est incorrect (`x` n'est pas une adresse)

Notation simplifiée : `p->`*`membre-de-structure`*

Ainsi `(*p).x` est équivalent à `p->x`

Structures et fonctions
(3) passage par adresse d'une structure

```
void CreatePoint(Tpoint *p, int x, int y)
{
  p->x = x ;      p->y = y ;
}
```

Créer un programme qui initialise un point à (0,0)

Structure image

Proposer une structure pour stocker les images

Fonctions associées

Il est important de définir un certain nombre de fonctions permettant de simplifier l'accès aux images.

Proposer des fonctions asociées à ces tâches

- Initialisation
- Destruction
- Accès aux données
- Copie
- etc.

Copie d'image

Que ce passe t'il dans le programme suivant

```
{
  uchar a ;
  TImage I1 ;
  TImage I2 ;

  InitImage(&I1, 10, 10) ;
  I2 = I1 ;
  FreeImage(&I1) ;
  a = I2.row[2][2] ;
}
```

Copie d'image

Proposer une fonction qui réalise la copie de deux images

```
{
  uchar a ;
  TImage I1 ;
  TImage I2 ;

  InitImage(&I1, 10, 10) ;

  CopyImage( ...  )   ;              // I2 <--- I1 ;
}
```

Autre type d’image

Définir les structures de données nécessaires à la définition d’image couleur

Stockage des images
Le format PNM

```
[Mars] > man pnm

pnm(5)                                                    pnm(5)

NAME
       pnm - portable anymap file format

DESCRIPTION
       The  pnm  programs  operate  on portable
       bitmaps, graymaps, and pixmaps, produced
       by the pbm, pgm, and  ppm  segments.
       There is no file format associated with
       pnm itself.
```

Noir et blanc : PBM

Each PBM image consists of the following:

- The "magic number" is "P1"
- Whitespace (blanks, TABs, CRs, LFs).
- The width in pixels of the image, formatted as ASCII characters in decimal.
- Whitespace.
- The height in pixels of the image, again in ASCII decimal.
- Newline or other single whitespace character.
- Each pixel in the raster is represented by a byte containing ASCII ’1’ or ’0’, representing black and white respectively. There are no fill bits at the end of a row.
- White space in the raster section is ignored.
- You can put any junk you want after the raster, if it starts with a white space character.
- No line should be longer than 70 characters.

Noir et Blanc

— Créer un type image Noir et Blanc

— Implémenter un lecteur d'image PBM

C++ : Quelques éléments

- De C à C++
  - Fonctions, passage de paramètres
  - Allocation dynamique de la mémoire
  - Fonctions : *Overloading*
- Construction de classes
  - Classes = structures + fonctions
  - Constructeurs / destructeurs
  - *Overloading*
- Templates

Le C++ et les fonctions
Passage de paramètres par référence

Rappel, en C, seul le passage de paramètres par valeur existait :

```
void f(int a, int b, int* sum, int* prod)          {
        *sum = a + b;
        *prod = a * b;
}

void g()          {
        int s, p;
        f(37, 47, &s, &p);
}
```

Le C++ et les fonctions
Passage de paramètres par référence

En C++, on rajoute le passage par référence :

```
void f(int a, int b, int& sum, int& prod)      {
        sum = a + b;
        prod = a * b;
}

void g()        {
        int s, p;
        f(37, 47, s, p);
}
```

Le & devant la variable signifie que le contenu de cette variable est passée par adresse (ou référence) et est donc modifiable.

Le C++ et les fonctions
Polymorphisme

En C++, plusieurs fonctions peuvent avoir le même nom si leurs paramètres sont différents.

Par exemple deux fonctions pour charger des images de types différents :

```
void LoadImage(Timage &I) { ... } ;

void LoadImage(TimageColor &I) { ... } ;
```

Le choix de la fonction à exécuter se fait en fonction de ces paramètres

Allocation dynamique de mémoire

Rappel en C, l'allocation dynamique de mémoire se fait en utilisant le couple `malloc/free`

```
int* ip;

ip = (int*)malloc(sizeof(int) * 100);
...
free((void*)ip);
```

Allocation dynamique de mémoire
Les opérateurs `new` et `delete`

Ceci est toujours valide en C++, mais on leur préférera le couple `new/delete`

```
int *ip;
int *tabip
ip = new int;
tabip = new int[100] ;
...
delete ip;
delete []tabip ;
```

Avantage : On a plus besoin de s'occuper du nombre d'octets à allouer

Remarque : `new()`, `new[]()`, `delete()` et `delete[]()` sont des opérateurs et non pas des fonctions.

Les classes

Définition : Classe La notion de classe donne en fait la Définition d'un objet. Elle peut être assimilée à la définition d'un nouveau type. Une classe regroupe un ensemble de données, de type élémentaires ou non, en un nouveau type.

Le terme **membre** désigne un élément de la classe. Un membre d'une classe peut être une donnée, ou bien une fonction.

Exemple de prototype d'une classe image

```
class CImage
{
private:
  int ncols ;
  int nrows ;
  uchar **row ;
public:
  uchar *bitmap ;
```

```
public:
  int  Resize(int nbl, int nbc) ;

  inline  int GetRows() ;
  inline  int GetCols() ;

  uchar *operator[](int n) ;

  CImage(int nbl, int nbc) ;
  CImage(const CImage&);
  ~CImage() ;
  void operator=(const CImage &m);
}
```

.

Template

Les classes “templates” servent à créer des classes génériques où un ou plusieurs type ne sont pas spécifié à l’avance

Exemple : CImage a été défini ici pour des uchar

Doit on recréer une nouvelle classe image pour des images de float, de RGB, etc..

Non !

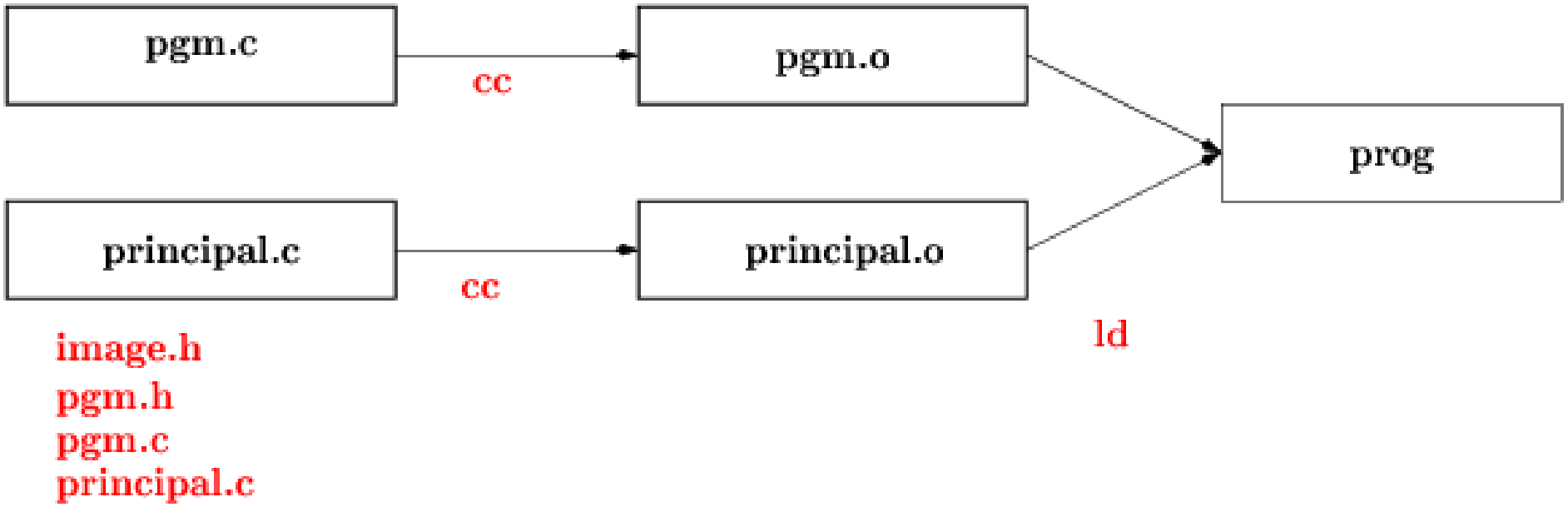
Exemple de prototype d'une classe image "template"

```
template<class Type>
class CImage
{
private:
  int ncols ;
  int nrows ;
  Type **row ;
public:
  Type *bitmap ;

.
```

```
public:
  int  Resize(int nbl, int nbc) ;

  inline  int GetRows() ;
  inline  int GetCols() ;

  Type *operator[](int n) ;

  CImage(int nbl, int nbc) ;
  CImage(const CImage<Type>&);
  ~CImage() ;
  void operator=(const CImage<Type> &m);
}
```

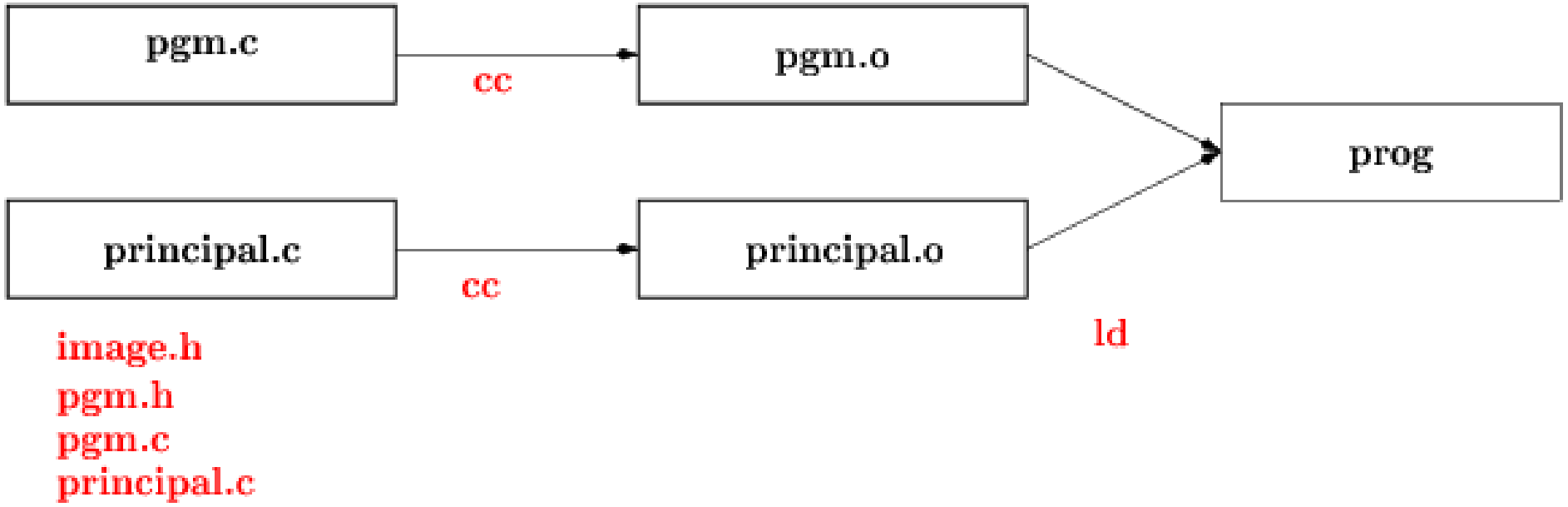
.png

Compilation du C/C++



— Cela soulève le problème de la compilation séparée

Compilation du C/C++



Solution naïve : un programme shell

```
cc -c pgm.c
cc -c principal.c
cc -o prog principal.o pgm.o -lm
```

Bof !

Gestion de la compilation séparée

Problème principal :

- des fichiers non modifiés peuvent être recompilés

perte de temps

Solutions :

- Ne recompiler que ce qui est nécessaire
  - les sources modifiés
  - les sources dont les dépendances ont été modifiées

Utiliser un programme capable de comparer les dates de modifications des fichiers : `make`

`make`

Utiliser un fichier : le `makefile` ou `Makefile`

Ce fichier décrit :

- les dépendances
- les commandes à exécuter pour construire les fichiers

Un makefile est composé de

- règles
- définitions de macro-variables

`make` et les règles

```
cible : dependance 1 dependance2 ...
<TAB>commande1
<TAB>commande2
```

Les commandes sont exécutées lorsque l'une des dépendances est plus récente que la cible.

Exemple :

```
pgm.o : pgm.c image.h
    cc -c pgm.c
```

Règles particulières

La cible n'est pas un fichier, et la commande ne créée pas de fichier

Exemple : Règle pour faire le "ménage"

```
clean:
    rm -f *.o
    rm -f *%
    rm -f *~

rmbin:
    rm -f *.o prog
```

Remarque : dans ce cas pas de dépendances

`make` exécutera toujours ces commandes si cette cible est demandé

Fonctionnement de la commande `make`

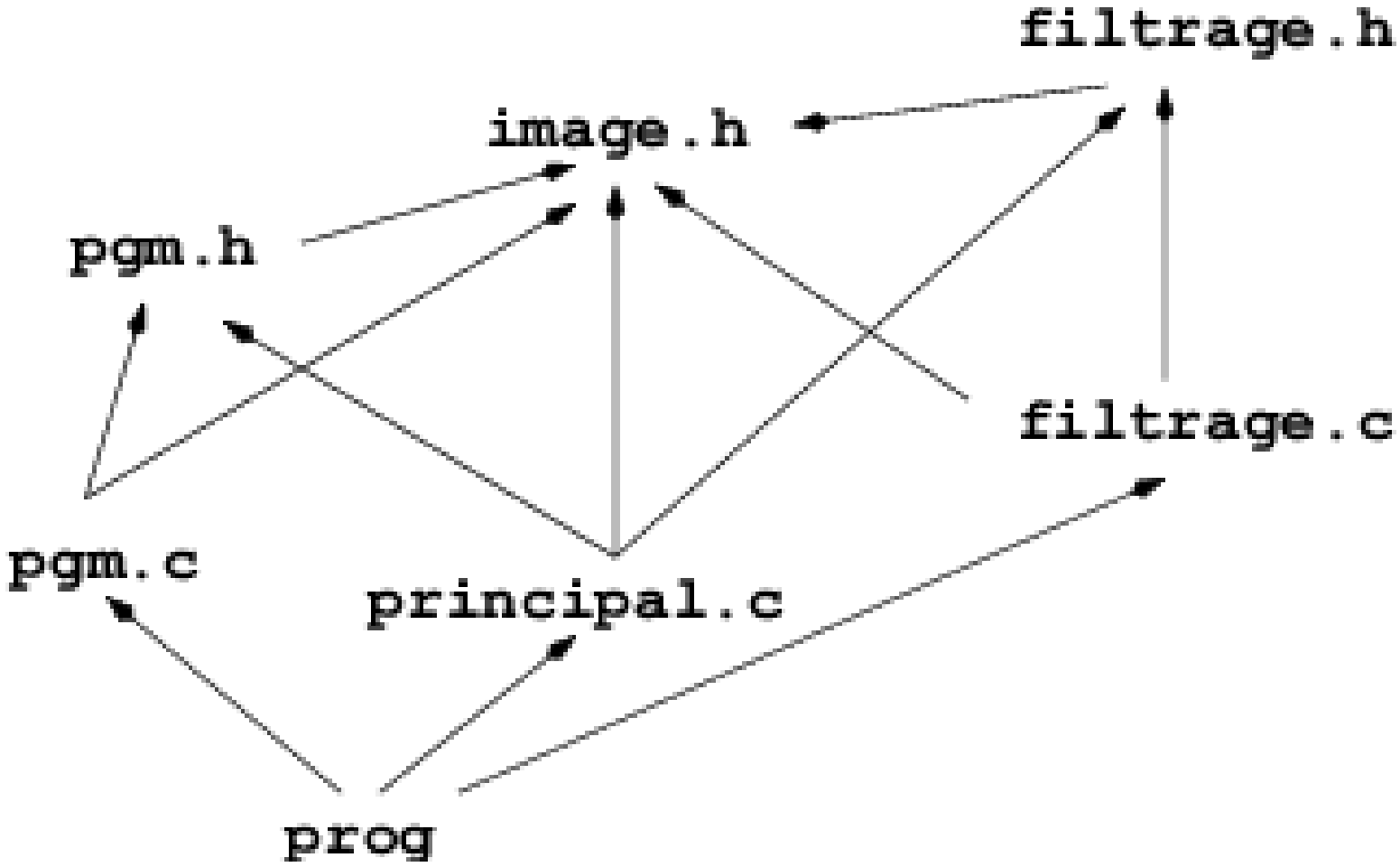
```
antekirtta [23:58]%  make cible
```

ou

```
antekirtta [23:59]% make
```

Dans ce dernier cas la première règle du makefile est exécutée

Etablir un makefile pour les sources suivant



A $\longrightarrow$ B : signifie A utilise B

Utilisation de macro commandes

On peut définir un certain nombre de variable :

```
CC = gcc
```
définition du compilateur

```
CFLAG = -Wall -g -O2
```
option de compilation

```
LIB = -lm -lc
```
librarie utilisé pour l'édition de lien

```
CWALL = $(CC) $(CFLAG)
```
compilation

Réécrire le makefile précédent

Librairies : la commande `ar`

Il peut être utile de regrouper les fichiers objets (`.o`) en libraries (archive).

Définition : (extrait du man)

The `ar` program creates, modifies, and extracts from archives. An archive is a single file holding a collec tion of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

`ar` is considered a binary utility because archives of this sort are most often used as libraries holding commonly needed subroutines.

Insertion d'un fichier dans une librairie

```
ar rc lib.a fic.o
```

place le fichier `fic.o` dans la librairie `lib.a`

- option `r` insère le fichier dans la librairie
- option `c` créé la librairie si besoin

Créé ou met à jour l'index de la librairie

```
ar s lib.a
```

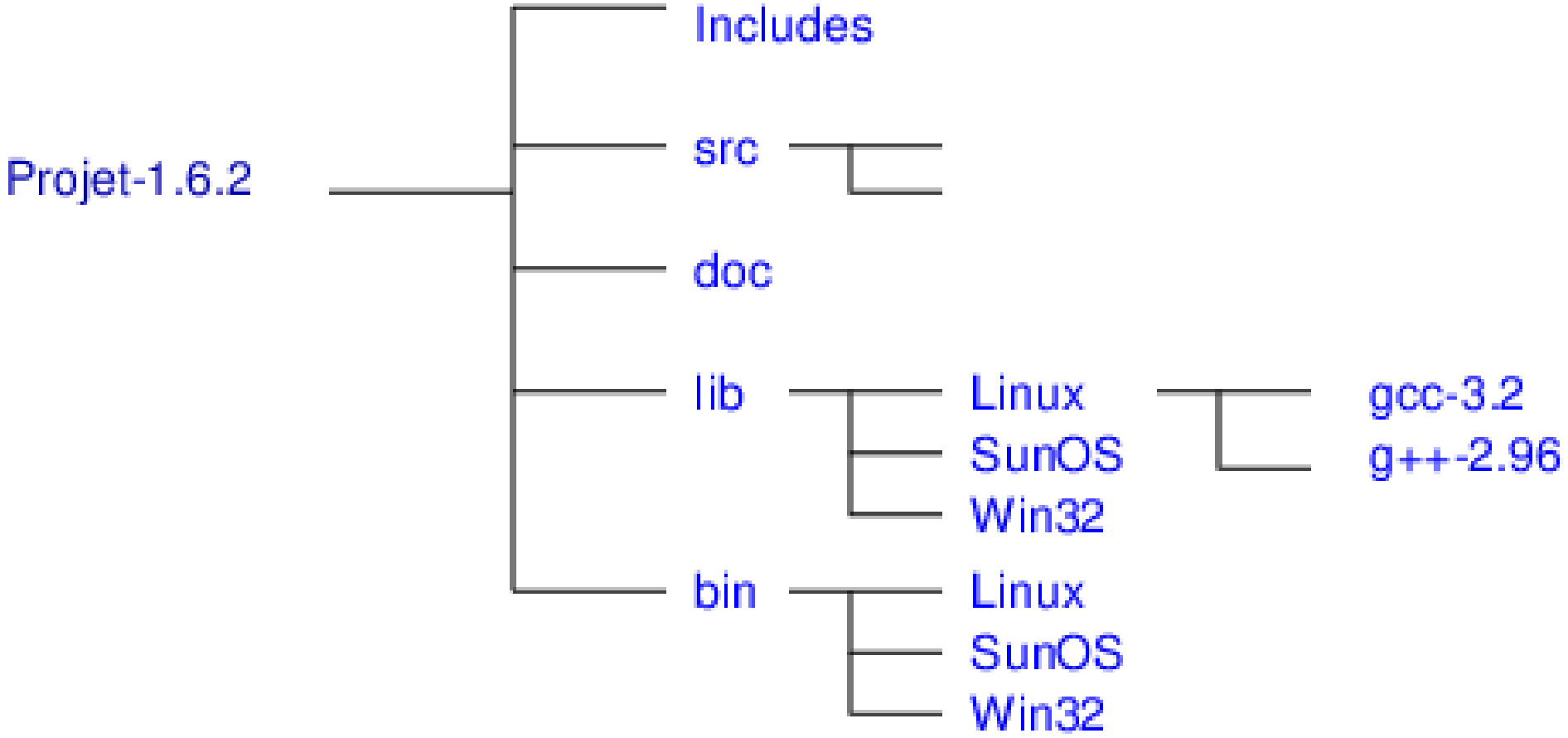
Création d'une librairie (2)

```
LIBIMAGE = libimage.a

$(LIBIMAGE) =\
        $(LIBIMAGE)(pgm.o) \
        $(LIBIMAGE)(filtrage.o)
        ar ts $(LIBIMAGE)

$(LIBIMAGE)(pgm.o) : pgm.c pgm.h image.h
    $(CWALL) -c pgm.c
    ar rc $@ $%              # execute ar rc libimage.a pgm.o
    rm -f $%                 # execute rm -f pgm.o



$(LIBIMAGE)(filtrage.o) : filtrage.c filtrage.h image.h
    $(CWALL) -c filtrage.c
    ar rc $@ $%
```

```
    rm -f $%

principal.o : \
    principal.c image.h pgm.h \
    filtrage.h
    $(CWALL) -c principal.c

prog : $(LIBIMAGE) principal.o
    $(CWALL) -o prog  \
    principal.o $(LIBIMAGE) $(LIB)
```

Gestion de projet : quelques principes (très) généraux

Arborescence : structurée, évolutive, multi-plate forme



Gestion de projet : quelques principes (très) généraux

Makefile : multi-plate forme

Utiliser les macro-variable

- OS utilisé

```
OS = $(shell uname -s)
```

- Type de compilateur

```
# compilateur
CC                = g++
#numero de version du compilateur
VERSION         =  $(shell $(CC) --version)
```

- Mise en place automatique des libraries et binaires

```
LIB = $(DIRPROJET)/lib/$(OS)/$(VERSION)
BIN = $(DIRPROJET)/bin/$(OS)/$(VERSION)
```

Bibliographie

Livres de réference

[1] Brian W. Kernighan and Dennis M. Ritchie. "*The C Programming Language, Second Edition*, Prentice Hall, Inc., 1988.

[2] Bjarne Stroustrup. "*The C++ programming language (third edition)*", Addison Wesley, 1997.

[3] S. Lippman. "*C++ primer)*", Addison Wesley, 1992.