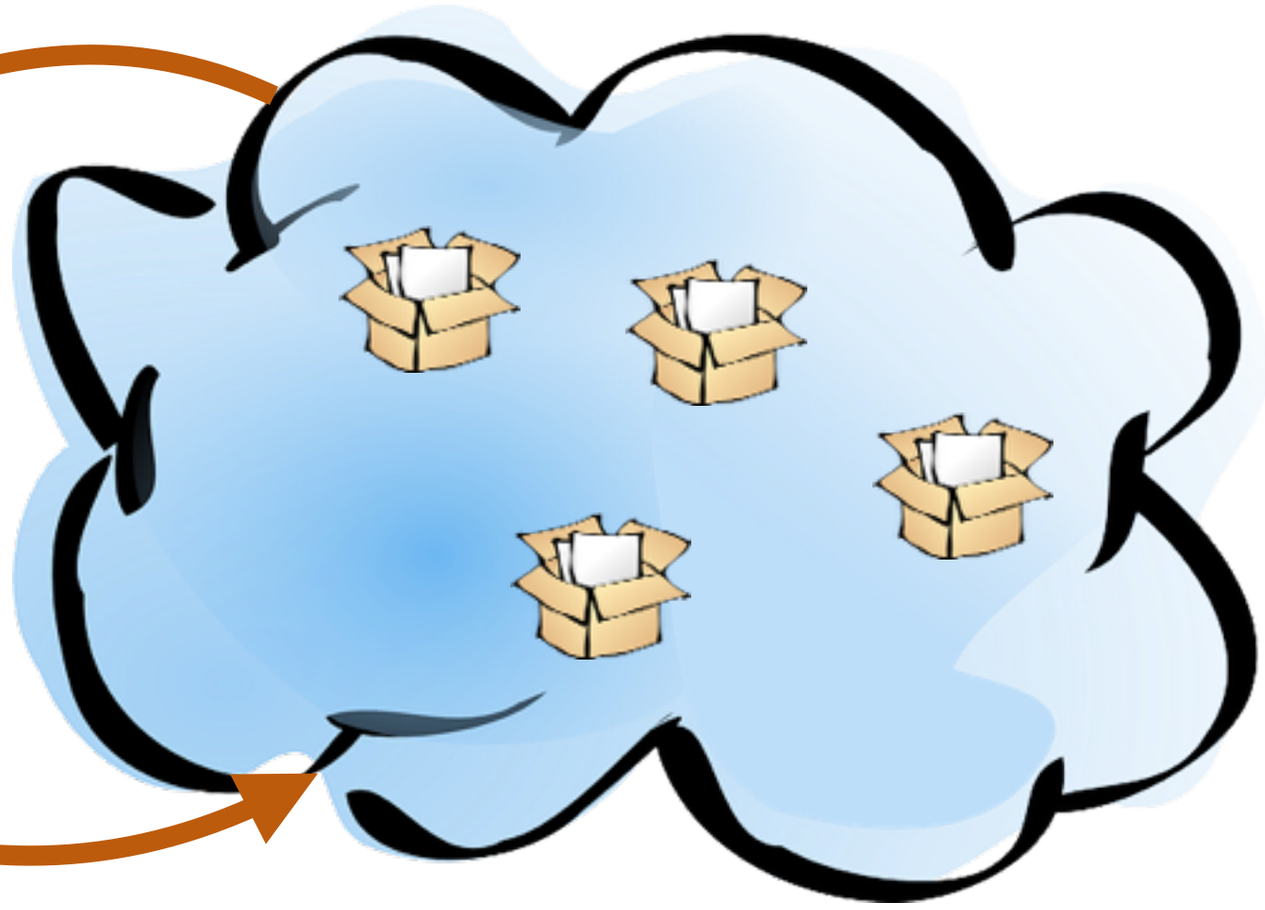
Programmation impérative

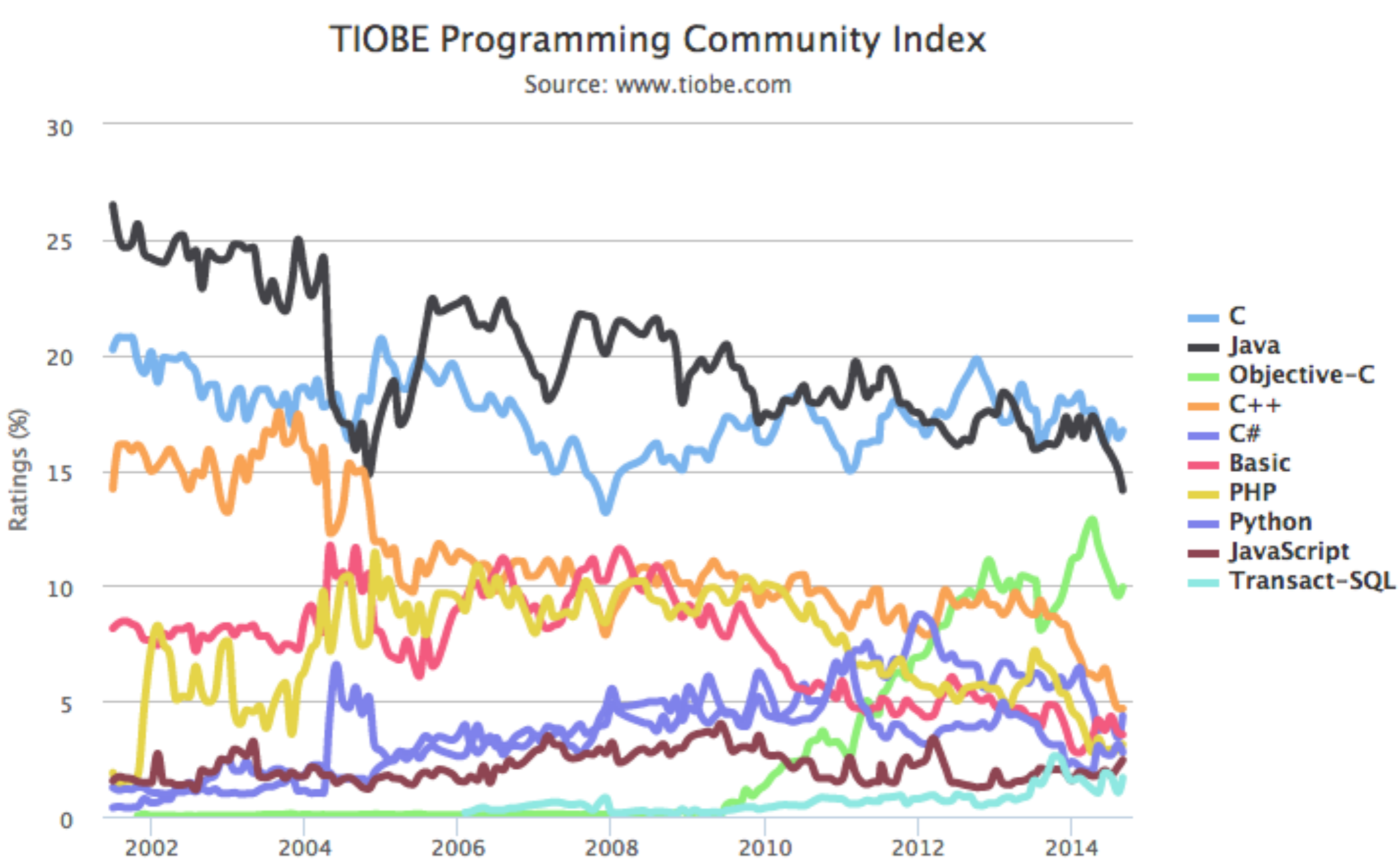
Langage C

Floréal Morandat
fmorandat@enseirb-matmeca.fr

Programmation impérative

fait-ci
fait-ça
lit un truc
écrit une chose
affiche machin
…





- Toujours autant utilisé
- Privilégié pour sa proximité avec le système
- Portable (... enfin, il faut le dire vite)
- Utilisé pour les systèmes contraints

Caractéristiques du C

- Langage compilé
- Typage statique
- (Assez) fortement typé
- «Haut» niveau (mais le plus bas)
- Gestion de la mémoire explicite
- Pas de surcoût à l'exécution
  - Il ne fait rien d'autre que ce que vous lui dites!

Histoire

- Inventé en 1972 (Dennis Ritchie, Bell Labs)
- Spécification : C K&R (1978)
- Normalisation ANSI : Ansi C / C89 (1989-1990)
- **Normalisation ISO : C99 (1999)**
- Mise-à-jour : C11 (ou C1x) (2011)

Mon premier programme

example-01.c

```c
#include <stdio.h>

/*
  This program prints ``Hello world''
*/
int main(int argc, char **argv) {
  printf("Hello world\n");
  return 0;
}
```

```
$ cc -std=c99 -Wall –o example-01 example-01.c
$ ./example-01
Hello world
```

Options de compilation

- -o <output> (nom du programme généré)
- -Wall (montre plus d'erreurs)
- -std=c99 (moins restrictif que -ansi)
- -Werror (attention => erreurs)

S'il vous plaît ...

- Utilisez un VRAI éditeur de texte (emacs, vim)
- Mettez de la couleur
- Utilisez l'indentation automatique
- Restez cohérent
- Utilisez des noms évocateurs
- Respecter l'espacement
- Testez régulièrement
- Commentez (et en anglais c'est mieux !)

Variables

- Espace mémoire nommés
  - Typés

- Format
  - première lettre : [a-zA-Z_] (svp pas de _)
  - les autres : [a-zA-Z0-9_]
  - Maximum : 64 caractères
  - ATTENTION : a != A

Variables

- Déclaration de variable
  `type nom`
  `type nom = expr`
- *Pour le moment type est 'int' (entiers)*

- Utilisation
  nom

- Valable après la déclaration
  Jusqu'à l'accolade fermante
- Conseil : Mettez les au **début** du bloc.

Expressions

- Évaluable => Valeur
- Typé

- Constante
- Variable
- Expression composée
- Appel de fonction

Expressions

		Assoc
x( )	Appel de fonction	
( )	Groupe	
* / %	Multiplier, diviser, modulo (reste de la div)	G
+ -	Addition et soustraction (binaire)	G
> >= < <=	Plus grand (ou égal) Plus petit (ou égal)	G
== !=	Égalité, différent	G
=	Affectation	D

Instructions

- Instruction simple
  - Se termine par un ;
- Bloc d'instructions
  - { instructions }
  - pas de ;

Exemple d'instructions

- Expression
- Déclaration de variable
- Structure conditionnelle (test)
- Structure répétitives (boucles)
- Retourner une valeur
- Rien

Structure conditionnelle

```
if (expression)
    instruction
```

```
if (expression)
    instruction
else
    instruction
```

- Si l'expression est vraie,
  alors on exécute l'instruction
  sinon on exécute l'autre instruction
- !!! Attention à l'imbrication !!!
- `else` se réfère au `if` le plus proche
- Conseil: utilisez les {}

Structures répétitives

```
while (expression)
    instruction
```

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

- Tant que expression est vraie
  faire instruction
- !!! Attention, l'éternité c'est long vers la fin !!!
- Faire instruction tant que expression est vraie
- !!! Attention la deuxième forme ne se finit pas par une instruction !!!

Fonctions

- Déclaration de fonction
  ```
  type nom() { }
  ```
  ou
  ```
  type nom(type nom1, ...) { }
  ```
- *Pour le moment type est 'int' (entiers) ou 'void' néant*
- *Si la fonction a un type de retour elle DOIT se terminer par un `return`*
- *Les paramètres sont copiés: passage par valeur (pour le moment sauf les tableaux)*
- *le type tableau NE PEUT PAS servir de type de retour*
- Utilisation
  ```
  nom()
  ```
  ou
  ```
  nom(arg1, ...)
  ```

Tableaux

- Notation []
- Commencent à 0
- Doivent avoir une longueur à l'initialisation
  (nb: facultatif pour les paramètres)
  - Pas de primitive longueur !!!
    Bornes non vérifiées !!!
  - Pas de copies

```
int tableau[10];    int tableau[] = { 1, 2 };
```

example-02.c

```c
int max(int a, int b) {
  if (a > b) // Ces () sont OBLIGATOIRES
    return a;
  else
    return b;
}
```

example-02.c

```c
int max(int a, int b) {
  if (a > b) // Ces () sont OBLIGATOIRES
    return a;
  else
    return b;
}
int maxTableau(int nbElements, int tableau[]) {
  int resultat = tableau[0];
  int i = 1;
  while (i < nbElements) { // celles-ci aussi
    resultat = max(resultat, tableau[i]);
    i = i + 1;
  }
  return resultat;
}
```

example-02.c

```c
int max(int a, int b) {
  if (a > b) // Ces () sont OBLIGATOIRES
    return a;
  else
    return b;
}
int maxTableau(int nbElements, int tableau[]) {
  int resultat = tableau[0];
  int i = 1;
  while (i < nbElements) { // celles-ci aussi
    resultat = max(resultat, tableau[i]);
    i = i + 1;
  }
  return resultat;
}
int main(int argc, char **argv) {
  int tab[] = {10, 20, 42, 40};
  int res = maxTableau(4, tab);
  printf("Max du tableau: %d\n", res);
  return 0;
}
```

example-02.c

```c
#include <stdio.h>
int max(int a, int b) {
  if (a > b) // Ces () sont OBLIGATOIRES
    return a;
  else
    return b;
}
int maxTableau(int nbElements, int tableau[]) {
  int resultat = tableau[0];
  int i = 1;
  while (i < nbElements) { // celles-ci aussi
    resultat = max(resultat, tableau[i]);
    i = i + 1;
  }
  return resultat;
}
int main(int argc, char **argv) {
  int tab[] = {10, 20, 42, 40};
  int res = maxTableau(4, tab);
  printf("Max du tableau: %d\n", res);
  return 0;
}
```

Exemple : fibo

```c
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i = 2;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  while (i < n) {
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number
    i = i + 1;
  }

  return tab[n - 1] + tab[n - 2];
}
```

Exemple : fibo

```c
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i = 2;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  while (i < n) {
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number
    i = i + 1;
  }

  return tab[n - 1] + tab[n - 2];
}
```

Exercice :
Éviter la duplication

Commentaires

- Pas de paraphrase de code
- Commentaires multi-lignes
  - commencent par /*
  - finissent au PREMIER */
- Commentaires mono-lignes (depuis c99)
  - Commencent par //
  - finissent à la fin de la ligne

Fonctions (2)

- Soit deux fonctions pair et impaire qui doivent mutuellement se voir.

```
int pair(int n) {
  if (n == 0) return 1;
  return impair(n - 1);
}

int impair(int n) {
  if (n == 0) return 0;
  return pair(n - 1);
}
```

Fonctions (2)

- Soit deux fonctions pair et impaire qui doivent mutuellement se voir.
- Solution: prototype (ou déclaration anticipée)

```
int pair(int n) {
  if (n == 0) return 1;
  return impair(n - 1);
}

int impair(int n) {
  if (n == 0) return 0;
  return pair(n - 1);
}
```

Fonctions (2)

- Soit deux fonctions pair et impaire qui doivent mutuellement se voir.
- Solution: prototype (ou déclaration anticipée)

```
int pair(int n);
int impair(int n);

int pair(int n) {
  if (n == 0) return 1;
  return impair(n - 1);
}

int impair(int n) {
  if (n == 0) return 0;
  return pair(n - 1);
}
```

Les chaînes de caractères

- Tableaux de char (i.e., `char chaine[]`)
  - Longueur inconnue
  - Généralement terminées par un 0
  - Constantes : `"Ceci est une chaine constante"`
  - Le format (pour printf/scanf) : %s

- On utilisera la notation pointeurs :

```
char * chaine = "test";
```

Manipuler des chaînes

#include <string.h>

- Longueur d'une chaîne :
  `unsigned int strlen(char* str);`
  - Parcours complet de la chaîne : O(n)
- Comparer deux chaînes :
  `int strcmp(char* s1, char* s2);`
- Copier une chaîne :
  `char* strcpy(char* dest, char* src);`
- Convertir une chaîne en nombre :

#include <stdlib.h>

```
int atoi(char* str); long atol(char* str);
long strtol(char* str, char**end, int base);
```

Structures répétitives 2

- Boucle classique :
  - Initialiser
  - Tester
  - Faire
  - Incrémenter

```
i = 0;
while (i < 10){
    instruction
    i = i + 1;
}
```

Structures répétitives 2

- Boucle classique :
  - Initialiser
  - Tester
  - Faire
  - Incrémenter

```
i = 0;
while (i < 10){
    instruction
    i = i + 1;
}
```

```
for (i = 0; i < 10; i = i + 1)
    instruction
```

Exemple : fibo

```c
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i = 2;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  while (i < n) {
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number
    i = i + 1;
  }

  return tab[n - 1] + tab[n - 2];
}
```

Exemple : fibo

```
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i = 2;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  for (i = 2; i < n ; i = i + 1)
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number



  return tab[n - 1] + tab[n - 2];
}
```

Exemple : fibo

```
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  for (i = 2; i < n ; i = i + 1)
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number



  return tab[n - 1] + tab[n - 2];
}
```

Exemple : fibo

```
/*
 Compute fibonacci
 Returns 0 if n is negative
*/
int fibo(int n) {
  int tab[n];       // store intermediate value
  int i;

  if (n < 1)        // Range check (i.e., negative and bounds)
    return 0 ;
  if (n == 1)
    return 1;

  tab[0] = 0;       // Initial values
  tab[1] = 1;

  for (i = 2; i < n ; i = i + 1)
    tab[i] = tab[i - 1] + tab[i - 2]; // Compute next number



  return tab[n - 1] + tab[n - 2];
}
```

Exercice :
Ne pas utiliser un tableau

La ligne de commande

Nombres d'arguments

Les arguments

```
int main(int argc, char *argv[]) { ... }
```

!!! Le premier est **toujours** le nom du programme !!!

La ligne de commande

Nombres d'arguments

Les arguments

```
int main(int argc, char *argv[]) { ... }
```

!!! Le premier est **toujours** le nom du programme !!!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i, sum = 0;

  for (i = 1; i < argc; i = i + 1)
    sum = sum + atoi(argv[i]);

  printf("La somme du programme %s est : %d\n", argv[0], sum);
}
```

Types entiers

- **tailles plate-forme dépendante**
- int : 16 bits mini.
- char : 8 bits
- short (int) : 16 bits mini.
- long (int) : 32 bits mini. (suffixe L)
- long long (int) : 64 bits depuis C99 (suffixe LL)
- unsigned (int) : non signé (suffixe U)
- constantes décimales : 42
- constantes hexadécimales : 0x42 (66)
- constantes octales : 042 (34)
- constantes caractères : '*' (42)

Représentation

- Base
- Complément à deux : `-x == ~x + 1`
- Opérateurs bit-à-bits (bitwise)

non	et	ou	ou exclusif	décalage à gauche	décalage à droite
~	&	|	^	<<	>>

Maximums

	Bits n	Non signé [0 ; 2^n - 1]	Signé [-2^(n - 1); 2^(n - 1) - 1]
char	8	[0; 255]	[-128; 127]
short	16	[0; 65535]	[-32768; 32767]
long	32	[0; 4294967295]	[-2147483648; 2147483647]
long long	64	[0; 18446744073709551616]	[-9223372036854775808; 9223372036854775807]

Maximums

(1 << n) - 1

	Bits n	Non signé [0 ; 2^n - 1]	Signé [-2^(n - 1); 2^(n - 1) - 1]
char	8	[0; 255]	[-128; 127]
short	16	[0; 65535]	[-32768; 32767]
long	32	[0; 4294967295]	[-2147483648; 2147483647]
long long	64	[0; 18446744073709551616]	[-9223372036854775808; 9223372036854775807]

Taille des ‘objets’

- sizeof(...) : donne la taille d’un objet en octets

Taille des ‘objets’

- sizeof(...) : donne la taille d’un objet en octets

```
int main(int argc, char **argv) {
  short s = 42;
  int i = 42;
  long l = 42;
  long long ll = 42;

  printf("s: %zu\ni: %zu\nl: %zu\nll: %lu\n",
    sizeof(s), sizeof(i), sizeof(l), sizeof(ll));




  return 0;
}
```

Taille des ‘objets’

- sizeof(...) : donne la taille d’un objet en octets

```
int main(int argc, char **argv) {
  short s = 42;
  int i = 42;
  long l = 42;
  long long ll = 42;

  printf("s: %zu\ni: %zu\nl: %zu\nll: %lu\n",
    sizeof(s), sizeof(i), sizeof(l), sizeof(ll));

  printf("s: %zu\ni: %zu\nl: %zu\nll: %zu\n",
    sizeof(short), sizeof(int), sizeof(long), sizeof(long long));

  return 0;
}
```

printf

- Chaine de formats (%)
- Ordonnés
- !!! ATTENTION: nombre d'arguments!!!
- `man 3 printf`

d	u	x	o
Décimal signé	décimal non signé (Unsigned)	heXadécimal	Octal

hh	h	l	ll	z
char	sHort	Long	Long Long	taille

scanf

- Saisie de données
- Formats identique à printf
- !!! ATTENTION : à la syntaxe !!!

scanf

- Saisie de données
- Formats identique à printf
- !!! ATTENTION : à la syntaxe !!!

```
int main(int argc, char **argv) {
  int a, r;
  float f;
  char s[50];

  r = scanf("%d %g %s", &a, &b, s);
  printf("lu: %d, a=%d, b=%g et s=%s\n", r, a, b, s);

  return 0;
}
```

Logique

- Pas de booléens
- Opérateurs

Et (logique)	Ou (logique)	Non (logique)
&&	||	!

- Attention: ~x != !x
- Attention: a && b != a & b

Table de précédence 2

		Assoc
~ ! + -	Négation bit-à-bit et logique Conservation / Changement de signe (unaire)	D
* / %	Multiplier, diviser, modulo	G
+ -	Addition et soustraction (binaire)	G
> >= < <=	Plus grand (ou égal) Plus petit (ou égal)	G
== !=	Égalité, différent	G
&	Et bit-à-bit (and)	G
^	Ou exclusif bit-à-bit (xor)	G
|	Ou bit-à-bit (or)	G
&&	Et logique	G
||	Ou logique	G
=	Affectation	D

Structure conditionnelle 2

- switch (cond) { }
- Seulement type entier
- Branches (constantes)

```
if (choix == ‘o’) {
  printf("Oui, s’il vous plait");
} else if (choix == ‘n’) {
  printf("Non merci");
} else {
    printf("Je ne comprend pas.");
}
```

Structure conditionnelle 2

- switch (cond) { }
- Seulement type entier
- Branches (constantes)

```
switch (choix) {
  case ‘o’:
    printf("Oui, s’il vous plait");
    break;
  case ‘n’:
    printf("Non merci");
    break;
  default:
    printf("Je ne comprend pas.");
    break;
}
```

Structure conditionnelle 2

- switch (cond) { }
- Seulement type entier
- Branches (constantes)

```
switch (choix) {
   case ‘o’:
   case ‘O’:
     printf("Oui, s’il vous plait");
     break;
   case ‘n’:
   case ‘N’:
     printf("Non merci");
     break;
   default:
     printf("Je ne comprend pas.");
     break;
}
```

Flot de contrôle

- return
  Retourner à la fonction appelante
  (éventuellement avec une valeur)

Juste pour être complet (mais pas à savoir)

- break
  Sortir de la boucle la plus intérieure (innermost loop)

- continue
  Retourner au début de la boucle (innermost loop)

Les nombres flottants

	Signe	Exposant	Mantisse (fraction)
float	1	8	23
double	1	11	52

- IEC 60559 ≈ IEEE 754
- (-1)^signe * (1+man) * 2^(exp-2^|exp-1|-1)
- 2 zéros ! 2 infinités et quelques NaN
- Attention : a / b * b != a
  - if(a ~~==~~ b) if(fabs(a - b) < une_valeur)

Les nombres flottants

- Par défaut : double
  flottantes : suivie d'un f ... ex: truncf()

- Puissance : pow et sqrt
- Arrondis : floor et ceil
- Troncature : trunc
- Valeur absolue/signe : fabs, sign
- isfinite(), isinf(), isnan(), and isnormal()

Affichage (des flottants)

l	f	e	g
modificateur pour les doubles	notation décimale	notation scientifique	choisi la notation la plus adaptée
	`0.0 .0 0.`	`1e0 1.0e0 . 0e1`	

```
float f = 1.0 / 3.0;
printf("%f %e %g\n", f, f, f);
f = 1 / 3;
printf("%f %e %g\n", f, f, f);
f = 1.0 / 3;
printf("%f %e %g\n", f, f, f);
```

Affichage (des flottants)

l	f	e	g
modificateur pour les doubles	notation décimale	notation scientifique	choisi la notation la plus adaptée
	0.0 .0 0. 0	1e0 1.0e0 . 0e1	

Un peu plus de formatage

```
float f = 1.0 / 3.0;
printf("%f %e %g\n", f, f, f);
f = 1 / 3;
printf("%f %e %g\n", f, f, f);
f = 1.0 / 3;
printf("%f %e %g\n", f, f, f);
```

```
double f = 1.0 / 3.0;
printf("%5.2lf\n", f);
```

Nombre de caractères minimum à afficher

Nombre maximum de décimales à afficher

Affichage (des flottants)

l	f	e	g
modificateur pour les doubles	notation décimale	notation scientifique	choisi la notation la plus adaptée
	`0.0 .0 0.`	`1e0 1.0e0 .0e1`	

Un peu plus de formatage

```
float f = 1.0 / 3.0;
printf("%f %e %g\n", f, f, f);
f = 1 / 3;
printf("%f %e %g\n", f, f, f);
f = 1.0 / 3;
printf("%f %e %g\n", f, f, f);
```

```
double f = 1.0 / 3.0;
printf("%5.2lf\n", f);
```

Nombre de caractères minimum à afficher

Nombre maximum de décimales à afficher

```
int i = 1;
printf("%5.2d\n", i);
```

Affichage (des flottants)

l	f	e	g
modificateur pour les doubles	notation décimale	notation scientifique	choisi la notation la plus adaptée
	0.0 .0 0. 0	1e0 1.0e0 . 0e1	

```
float f = 1.0 / 3.0;
printf("%f %e %g\n", f, f, f);
f = 1 / 3;
printf("%f %e %g\n", f, f, f);
f = 1.0 / 3;
printf("%f %e %g\n", f, f, f);
```

Un peu plus de formatage

```
double f = 1.0 / 3.0;
printf("%5.2lf\n", f);
```

Nombre de caractères minimum à afficher

Nombre maximum de décimales à afficher

```
int i = 1;
printf("%5.2d\n", i);
```

Nombre de caractères minimum à afficher

Nombre minimum de chiffres à afficher

Types simples

- Entiers
  - (unsigned) char
  - (unsigned) short
  - (unsigned) int
  - (unsigned) long
  - (unsigned) long long
- Flottants
  - float
  - double
  - long double
- Spécial
  - void (néant)

Structures

- Types utilisateurs
  - composition de types nommés (champ)
  - taille ? sizeof()
  - tableau taille connue

Structures

- Types utilisateurs
  - composition de types nommés (champ)
  - taille ? sizeof()
    - tableau taille connue

```
struct Point {
    float x;
    float y;
};
```

Structures

- Types utilisateurs
  - composition de types nommés (champ)
  - taille ? sizeof()
    - tableau taille connue

```
struct Point {
    float x;
    float y;
};


void affichePoint(struct Point p) {
  printf("(%g;%g)\n", p.x, p.y);
}
```

Structures

- Types utilisateurs
  - composition de types nommés (champ)
  - taille ? sizeof()
  - tableau taille connue

```c
struct Point {
    float x;
    float y;
};
```

```c
struct Point p;
p.x = 1;
p.y = 2;
struct Point p1 = { 1, 2 };
struct Point p2 = { .y=2, .x=1 }; // depuis C99
```

```c
void affichePoint(struct Point p) {
  printf("(%g;%g)\n", p.x, p.y);
}
```

Énumérations

- Entiers nommés
- Numérotation automatique
  (mais on peut l'aider)

```
enum Couleur {
 TREFLE, CARREAU, COEUR, PIQUE;
};

enum Valeur {
 DEUX=2, TROIS, QUATRE, CINQ,
 SIX, SEPT, HUIT, NEUF, DIX,
 VALET, DAME, ROI, AS;
};
```

Énumérations

- Entiers nommés
- Numérotation automatique
    (mais on peut l'aider)

```
enum Couleur {
 TREFLE, CARREAU, COEUR, PIQUE;
};

enum Valeur {
 DEUX=2, TROIS, QUATRE, CINQ,
 SIX, SEPT, HUIT, NEUF, DIX,
 VALET, DAME, ROI, AS;
};
```

```
struct Carte {
 enum Valeur valeur;
 enum Couleur couleur;
};
```

Énumérations

- Entiers nommés
- Numérotation automatique
- (mais on peut l'aider)

```
enum Couleur {
 TREFLE, CARREAU, COEUR, PIQUE;
};

enum Valeur {
 DEUX=2, TROIS, QUATRE, CINQ,
 SIX, SEPT, HUIT, NEUF, DIX,
 VALET, DAME, ROI, AS;
};
```

```
struct Carte {
 enum Valeur valeur;
 enum Couleur couleur;
};

int plusfort(struct Carte c1,
             struct Carte c2) {
  return c1.valeur > c2. valeur
        || c1.valeur == c2.valeur
        && c1.couleur > c2.couleur;
};
```

Le pré-processeur

- Avant de compiler le programme
- Commence par un #

```
#define TAILLE_MAX 200
#include <fichier>
#include "fichier"
```

Quelques opérateurs (utiles et dangereux)

+=	*=	>>=	&=
-=	/=	<<=	|=
	%=		^=

```
a = a + 3
a += 3
```

	Incrémentation	Décrémentation
Pre-	++ a	--a
Post-	a++	a--

```
int a = 2
printf("%d\n", a++);
printf("%d\n", ++a);
```

Exemple : Pile

- Operations
  - Empiler (push)
  - Dépiler (pop)
  - Regarder (peek)
  - Nombre d'éléments (size)
  - Est Vide / Est Pleine (empty/full)
  - Initialiser

Exemple : Pile

- Operations
  - Empiler (push)
  - Dépiler (pop)
  - Regarder (peek)
  - Nombre d'éléments (size)
  - Est Vide / Est Pleine (empty/full)
  - Initialiser
- Données
  - Elements (items)
  - Sommet (top)
  - ~~Taille (size)~~

Pointeurs (Références)

- Contient une adresse mémoire
- Permet d'accéder à cette adresse (R/W)
- Les pointeurs sont typé
- Permet un passage par référence
- Permet de «retourner» plus d'une valeur
- Convention: NULL (0L) => Absence de valeur
- Toujours la même taille (en général `sizeof(int)`)

Notations

- Déclaration:
  `type* var;`

- Initialisation:
  ```
  var = NULL;
  var = un_autre_pointeur;
  var = &une_autre_variable;
  ```

- Utilisation
  `type a = *var;`

Notations

- Déclaration:
  `type* var;`

Se lit : *var est de type type
ou : var est une référence de type type

- Initialisation:
  ```
  var = NULL;
  var = un_autre_pointeur;
  var = &une_autre_variable;
  ```

- Utilisation
  `type a = *var;`

Notations

- Déclaration:
  `type* var;`

Se lit : *var est de type type
ou : var est une référence de type type

- Initialisation:
  ```
  var = NULL;
  var = un_autre_pointeur;
  var = &une_autre_variable;
  ```

l'«autre» doit avoir le même type

- Utilisation
  `type a = *var;`

Notations

- Déclaration:
  `type* var;`

Se lit : *var est de type type
ou : var est une référence de type type

- Initialisation:
  ```
  var = NULL;
  var = un_autre_pointeur;
  var = &une_autre_variable;
  ```

l'«autre» doit avoir le même type

- Utilisation
  `type a = *var;`

En bref	
*	Suivre / Déréférencer
&	Prendre l'adresse

Exemple

- Inverser deux valeurs

```
void main(){
  int a = 42;
  int b = 24;

  printf("avant swap %d %d\n", a, b);
  swap(a, b);
  printf("apres swap %d %d\n", a, b);
}
```

Exemple

- Inverser deux valeurs

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

void main(){
  int a = 42;
  int b = 24;

  printf("avant swap %d %d\n", a, b);
  swap(a, b);
  printf("apres swap %d %d\n", a, b);
}
```

Exemple

- Inverser deux valeurs

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

void main(){
  int a = 42;
  int b = 24;

  printf("avant swap %d %d\n", a, b);
  swap(&a, &b);
  printf("apres swap %d %d\n", a, b);
}
```

Exemple

- Inverser deux valeurs

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

void main(){
  int a = 42;
  int b = 24;

  printf("avant swap %d(%p) %d(%p)\n", a, &a, b, &b);
  swap(&a, &b);
  printf("apres swap %d(%p) %d(%p)\n", a, &a, b, &b);
}
```

Pointeurs & structures

```
struct Point {
    float x;
    float y;
};
```

```
void swap(struct Point *p) {
  float tmp = (*p).x;
  (*p).x = (*p).y;
  (*p).y = tmp;
}
```

Pointeurs & structures

```
struct Point {
    float x;
    float y;
};
```

```
void swap(struct Point *p) {
  float tmp = (*p).x;
  (*p).x = (*p).y;
  (*p).y = tmp;
}
```

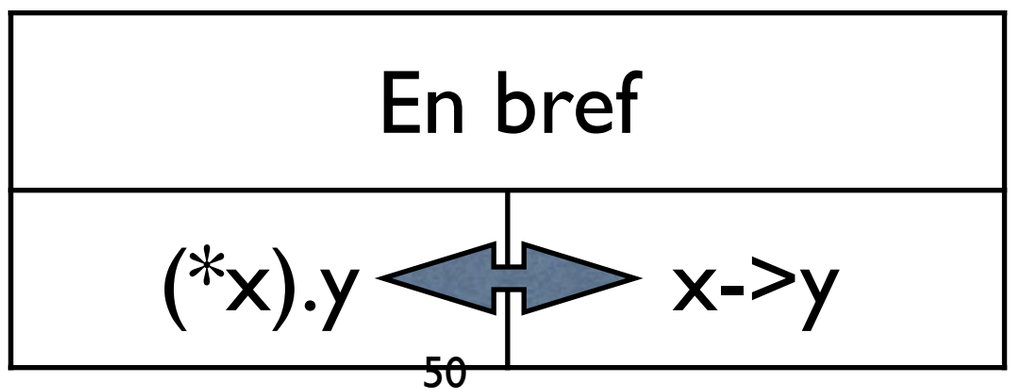
```
void swap(struct Point *p) {
  float tmp = p->x;
  p->x = p->y;
  p->y = tmp;
}
```

Pointeurs & structures

```
struct Point {
    float x;
    float y;
};
```

```
void swap(struct Point *p) {
  float tmp = (*p).x;
  (*p).x = (*p).y;
  (*p).y = tmp;
}
```

```
void swap(struct Point *p) {
  float tmp = p->x;
  p->x = p->y;
  p->y = tmp;
}
```



Pointeur et tableaux

- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

Pointeur et tableaux

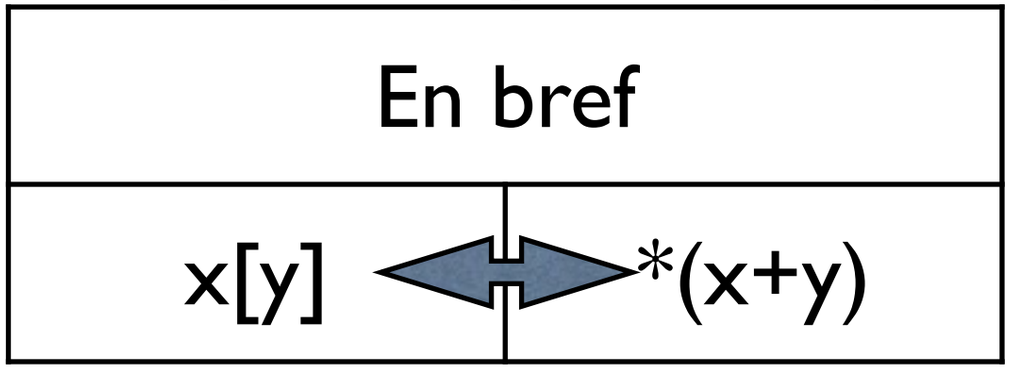
- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

```
int a[5] = {1, 2, 3, 4, 5};
if(a[3] == *(a + 3))
  printf(«toujours affiché\n»)
```

Pointeur et tableaux

- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

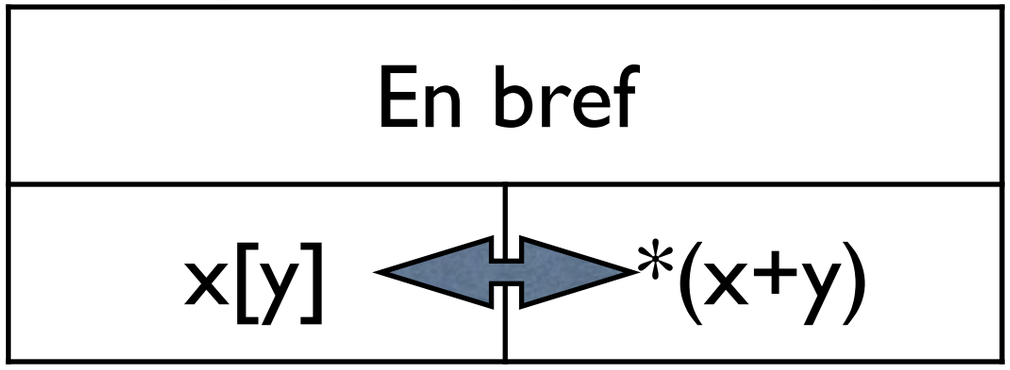
```
int a[5] = {1, 2, 3, 4, 5};
if(a[3] == *(a + 3))
  printf(«toujours affiché\n»)
```



Pointeur et tableaux

- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

```
int a[5] = {1, 2, 3, 4, 5};
if(a[3] == *(a + 3))
  printf(«toujours affiché\n»)
```

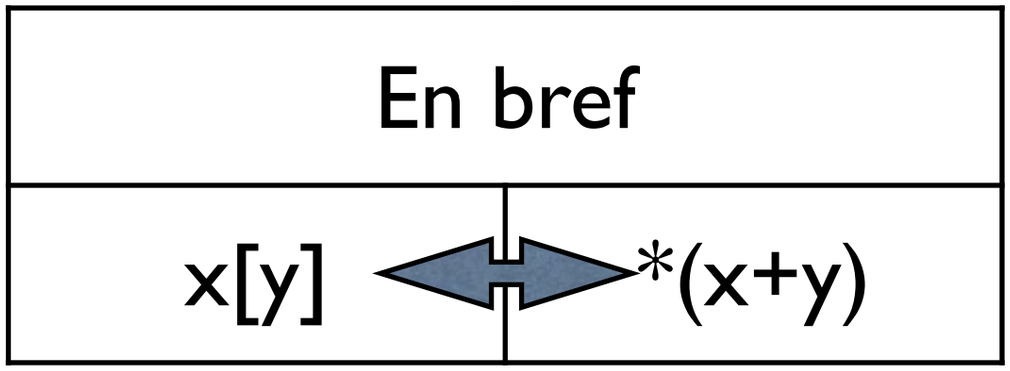



```
unsigned strlen(char* str) {
   unsigned len = 0;
   while(*str) {
      len ++;
      str ++;
   }
   return len;
}
```

Pointeur et tableaux

- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

```
int a[5] = {1, 2, 3, 4, 5};
if(a[3] == *(a + 3))
  printf(«toujours affiché\n»)
```



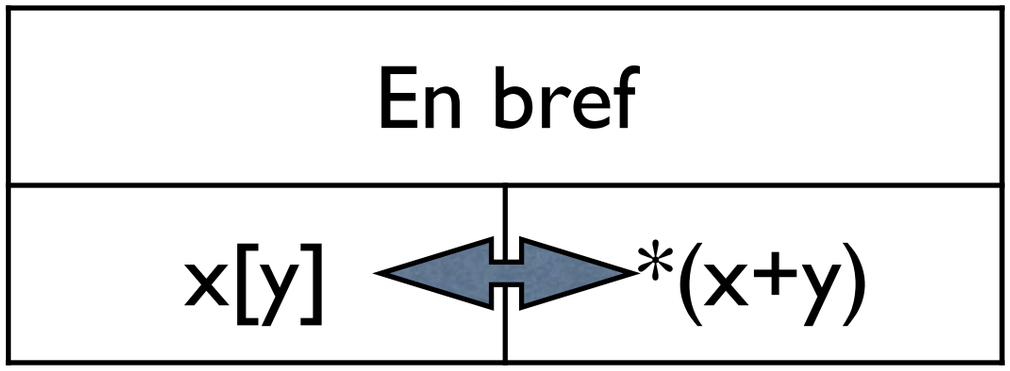

```
unsigned strlen(char* str) {
   unsigned len = 0;
   while(*str) {
      len ++;
      str ++;
   }
   return len;
}
```

```
unsigned strlen(char* str) {
   unsigned len = 0;
   while(*str ++)
      len ++;
   return len;
}
```

Pointeur et tableaux

- Les tableaux n'existent pas !
- Tableau = Pointeur sur une zone contiguë
- (pas de copie de tableau car copie du pointeur)

```
int a[5] = {1, 2, 3, 4, 5};
if(a[3] == *(a + 3))
  printf(«toujours affiché\n»)
```




```
unsigned strlen(char* str) {
   unsigned len = 0;
   while(*str) {
      len ++;
      str ++;
   }
   return len;
}
```

```
unsigned strlen(char* str) {
   unsigned len = 0;
   while(*str ++)
      len ++;
   return len;
}
```

```
unsigned strlen(char* str) {
   char *ptr = str;
   while(*ptr++)
      ;
   return ptr - str;
}
```

Arithmétique de pointeurs

- Tous les opérateurs arithmétiques d'addition, de soustraction et de comparaison.
- Valeur incrémenté de **sizeof(type)**

LE type polymorphe

- Tous les pointeurs ont la même taille
- `void *var;` est valide
- CEPENDANT `*var` est ILLEGAL
- compatible avec tous les types pointeurs

Gestion de la mémoire

- Deux opérations: *allouer* et *libérer*
  ```
  void* malloc(unsigned long nb);
  void free(void *ptr);
  ```
- La mémoire est non initialisée
- Tout bloc alloué doit être libéré et UNE SEULE fois

Gestion de la mémoire 2

- Autres fonctions utiles :
  - void *calloc(unsigned nbelem, unsigned size) (aloue initialisé)
  - void *realloc(void* ptr, int size) (étend une zone)
  - void *memset(void *buff, int c, unsigned len) (initialise)
  - void *memcpy(void *d, void *s, unsigned len) (copie)

Un peu plus sur les variables

- Variables locales/globales
- Modificateurs : static, extern, ...
- Constantes
- Déclaration, définition, initialisation

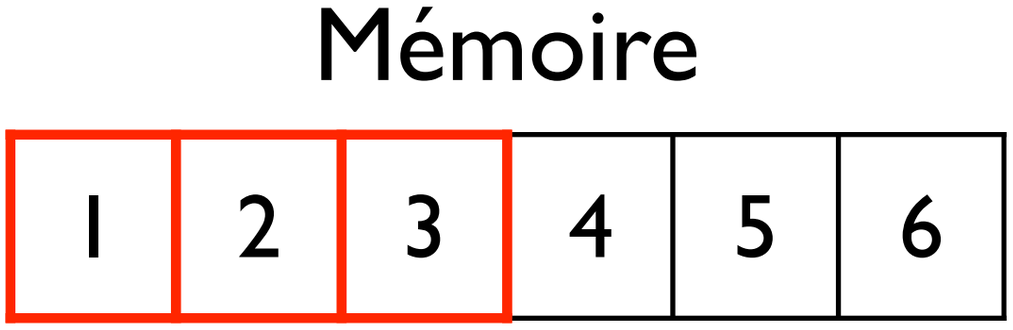
Gestion des erreurs

- Pas de mécanisme dédié
- variable: int errno
- fonctions:
  - void perror(const char *prefix)
  - char *strerror(int errnum)

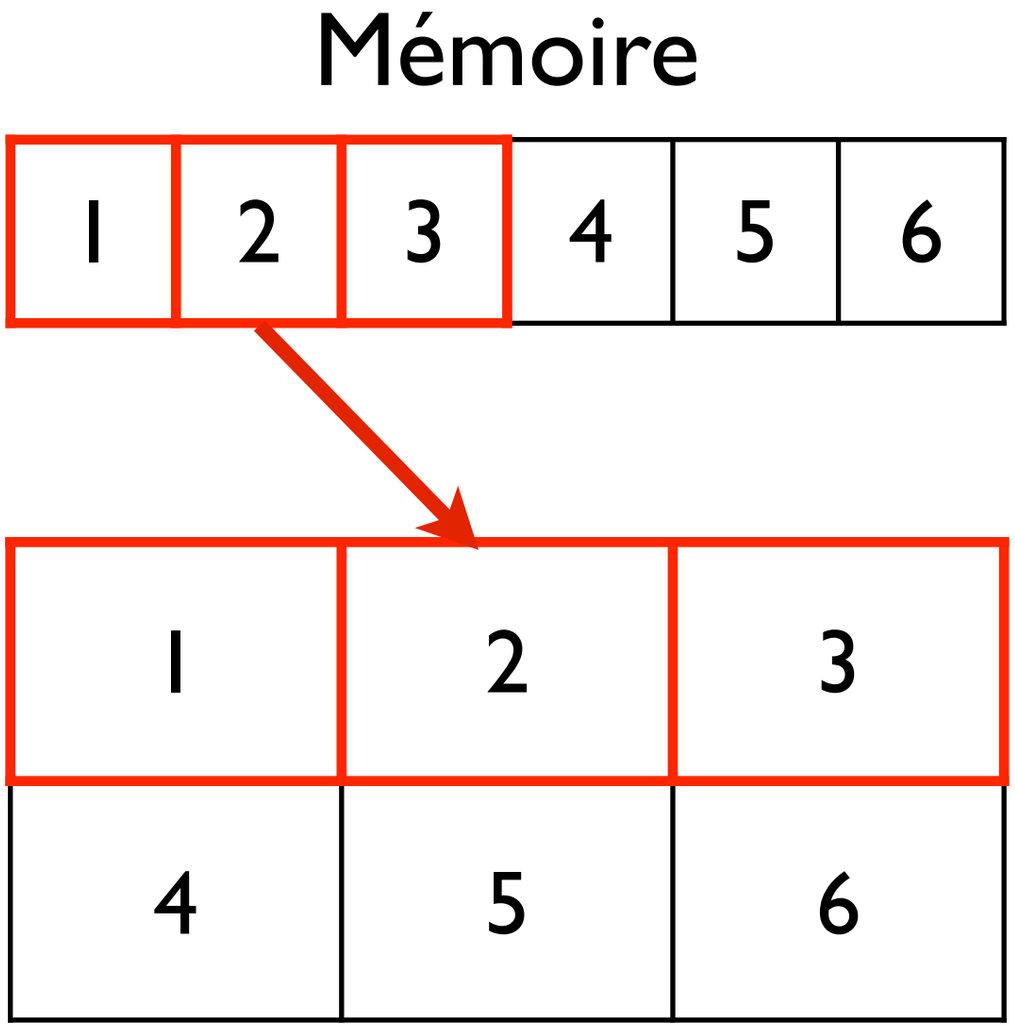
Fichier d'entête

- Par convention '.h'
- Inclus par #include
- Uniquement des déclarations
- !!Attention!! aux récursions

Tableaux multi-dimensionnels



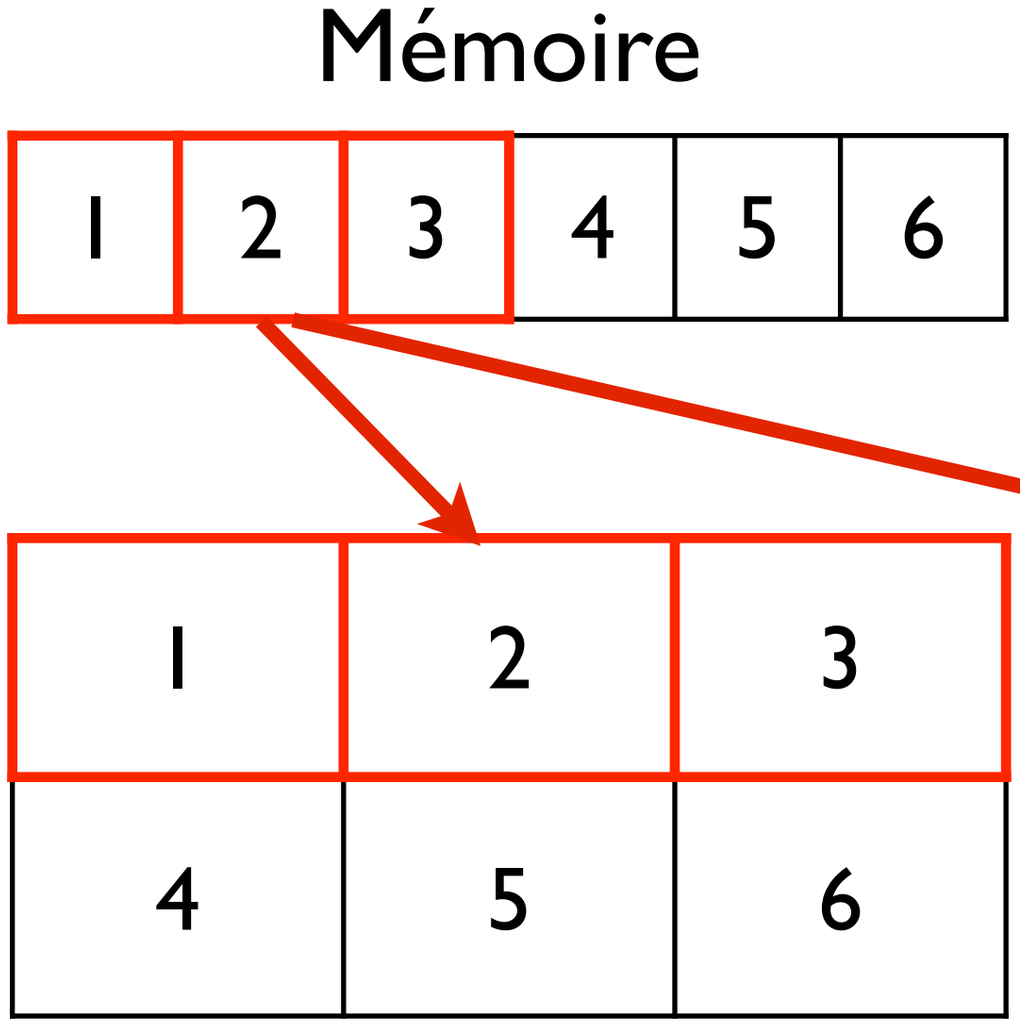
Tableaux multi-dimensionnels



Row-Major

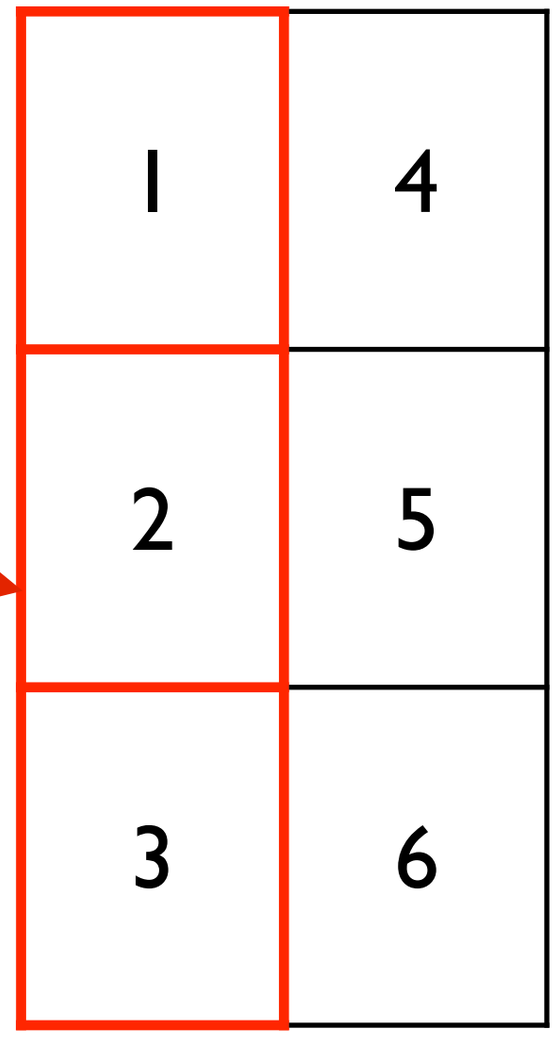
idx = lig*LARGEUR+col

Tableaux multi-dimensionnels



Row-Major

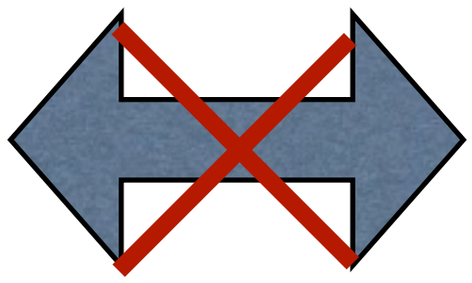
idx = lig*LARGEUR+col



Column-Major

idx = col*HAUTEUR+lig

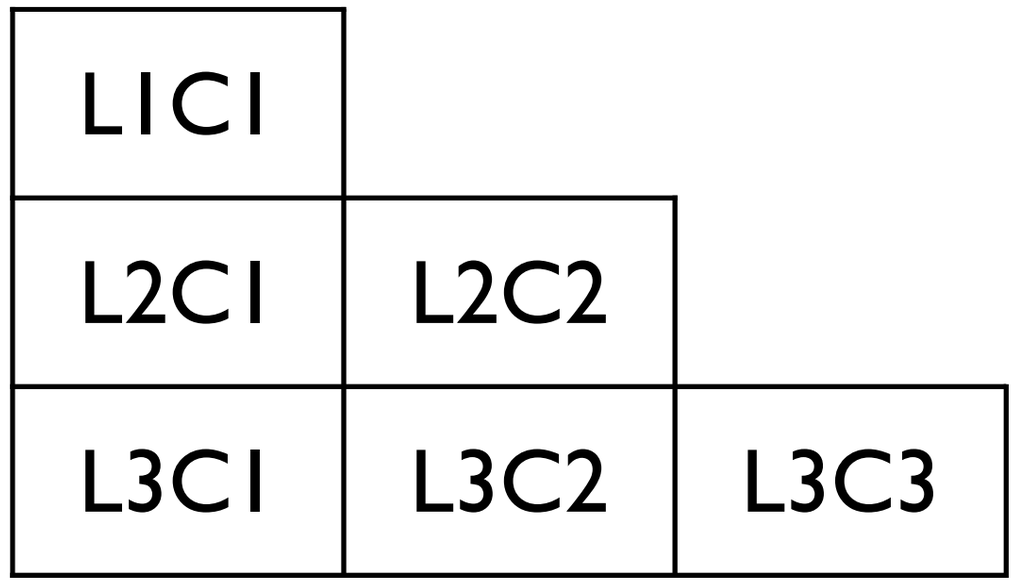
Tableaux multi-dimensionnel en C

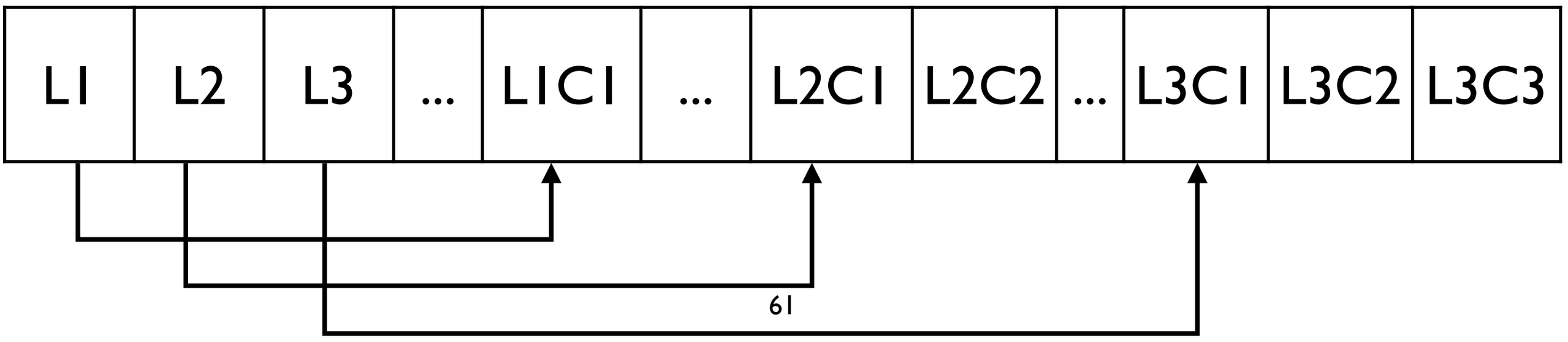
- Homogène (même type)
- Dense
- Row-major
- int a[] ≈ int *a  int a[][] ≠ int **a
- Pas plus d'une dimension libre ...
  et forcement la première (dû à la formule)
- int col = idx%HAUTEUR, lig=idx/HAUTEUR;

Tableaux non denses

```
int l1[] = {L1C1},
    l2[] = {L2C1, L2C2},
    l3[] = {L3C1, L3C2, L3C3};
int *tab[3] = {l1, l2, l3};

int **tab2 = malloc(3*sizeof(int*));
int *data = malloc(6*sizeof(int));
tab2[0] = data;
tab2[1] = data + 1;
tab2[2] = data + 1 + 2;
// Notez que les données ne sont tjs pas ici ...
```





Gestion de fichiers

- Cycle de vie :
  - ouverture (fopen)
    - operations (fprintf, fscanf, fgets, fgetc, ...)
    - fermeture (fclose)
- 3 flots spéciaux : stdin, stdout, stderr

Gestion de fichier 2

- FILE *fopen(const char *fname, const char *mode)
- int fclose(FILE *hdl)
- void rewind(FILE *hdl)
- int fseek(FILE *hdl, long offset, int whence)
- long ftell(FILE *hdl)
- unsigned fread(void *ptr, unsigned size, unsigned nitems, FILE *stream)
- unsigned fwrite(const void *ptr, unsigned size, unsigned nitems, FILE *stream);
- int feof(FILE *stream);

Mode d'ouverture

r	Ouverture en lecture	Début	R
w	Ouverture en écriture, le fichier est crée ou tronqué au besoin.	Début	W
a	Ouverture en écriture. Écritures toujours à la fin (pas de seek).	Fin	W
r+	Ouverture en lecture et écriture. Pas de création	Début	R/W
w+	Ouverture en lecture et écriture. Le fichier est crée ou tronqué au besoin.	Début	R/W
a+	Ouverture en lecture écriture. Écritures toujours à la fin (pas de seek).	Fin	R/W