

PROGRAMMATION IMPÉRATIVE (PG101/PG108/PG109/IF112)

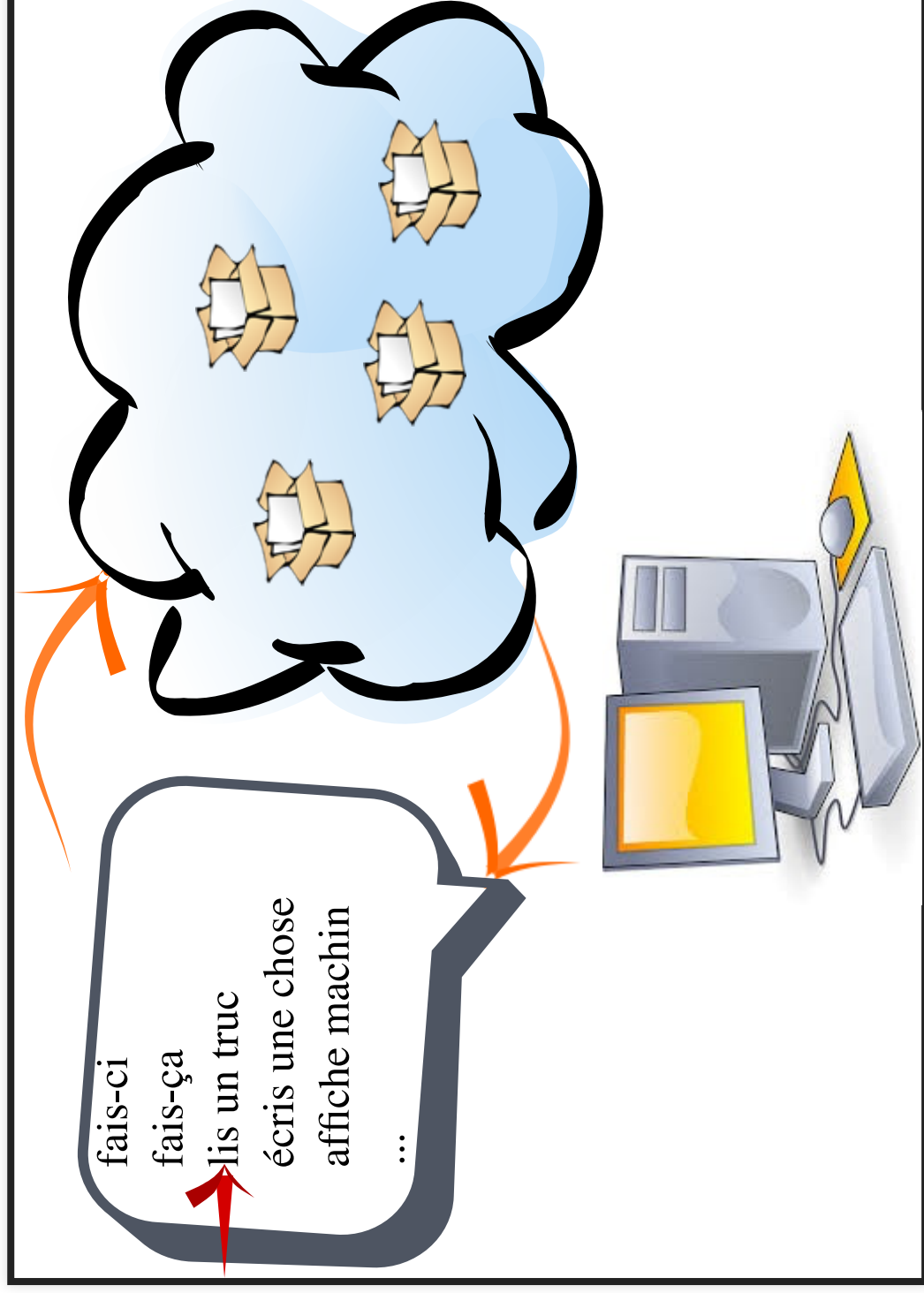
LANGUAGE C

Floréal Morandat

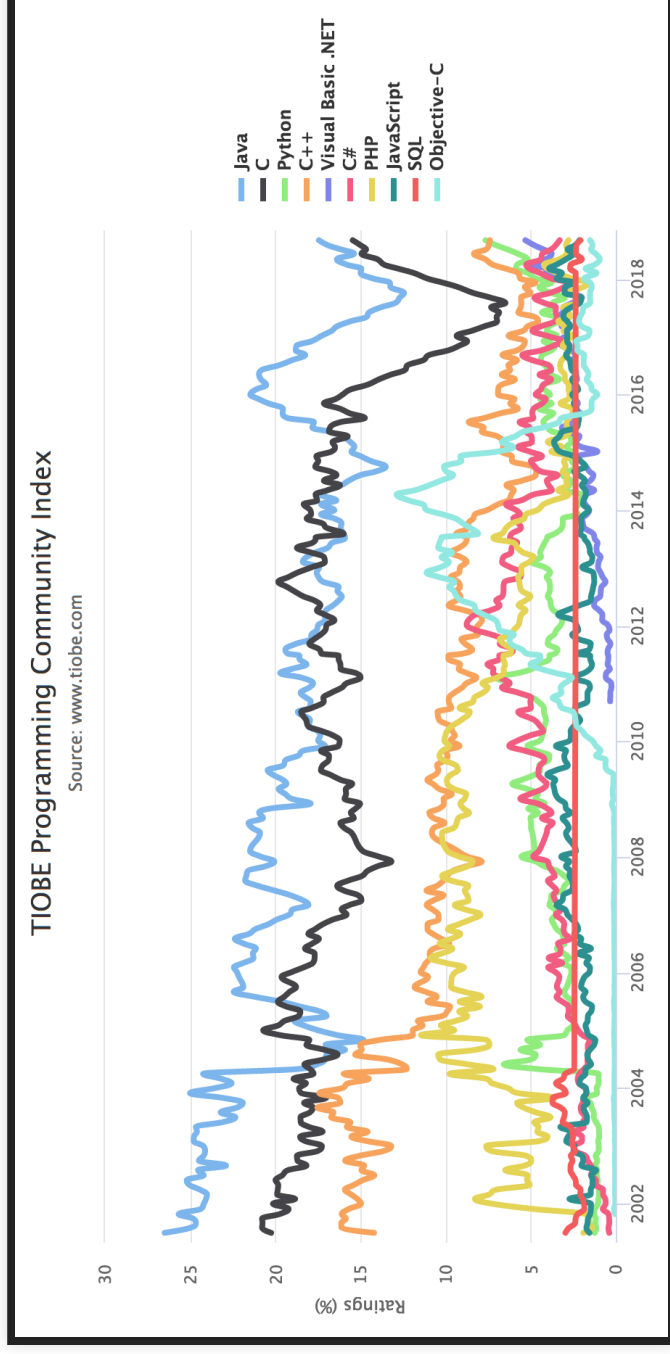
<http://www.labri.fr/~fmoranda/pg101/>



PROGRAMMATION IMPÉRATIVE



POURQUOI C ?



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



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)



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!



MON PREMIER PROGRAMME

example-01.c

```
#include <stdio.h>

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

Compiler et exécuter

```
# Pour compiler example-01.c en example-01
cc -std=c99 -Wall -o example-01 example-01.c
# Pour exécuter
./example-01
```



OPTIONS DE COMPILATION

<code>-o <output></code>	nom du programme généré
<code>-Wall</code>	Montre plus d'erreurs
<code>-std=c99</code>	active la norme C99
<code>-Werror</code>	attention => erreurs

AIDE

L'aide est en général dans la section 3 du manuel :

```
# Affiche l'aide de printf (3)
man 3 printf
```



S'IL VOUS PLAÎT

- Utilisez un VRAI éditeur de texte (emacs, vim, atom, ...)
 - 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 !)
-
- **Non respect = Non lecture**



LA TRILOGIE

Type

Toute valeur à un type

- Forme/taille de la case mémoire
- Pour le moment : **int**

Expression

Évaluable (produit une valeur) => type

Instruction

Exécutable => pas de type



EXPRESSIONS

- Évaluable $\Rightarrow$ Valeur
 - Typé
-
- Constante
 - Variable
 - Expression composée (principalement arithmétique)
 - Appel de fonction



TABLe DE PRÉCÉDENCE

Opération	Description	Assoc
()	Groupe	
* / %	Multiplier, diviser, modulo (reste)	G
+ -	Addition, soustraction (binaire)	G
> >=	Plus grand (ou égal)	G
< <=	Plus petit (ou égal)	
== !=	Égalité, différence (non égalité)	G
=	Affectation	D



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**



SYNTAXE

- Déclaration de variable :
`type nom`
`type nom = expr`
- RQ: Pour le moment le type est `int` (entiers)
- Utilisation :
`nom`
- Valable après la déclaration
Jusqu'à l'accolade fermante



INSTRUCTIONS

- Instruction simple
Se termine par un ;

- Bloc d'instructions

```
{ instructions }
```

RQ: ne termine pas par un ;



EXEMPLES

- Expression
- Déclaration de variable
- Structure conditionnelle (test)
- Structure répétitive (boucle)
- 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 **{ }**



EXEMPLE

example-02.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a = 5;
    int b = 3;
    int res;

    if (a > b) // Ces () sont OBLIGATOIRES
        res = a;
    else
        res = b;

    printf("Le max est: %d\n", res);
    return res;
}
```



STRUCTURE RÉPÉTITIVE

```
while (expression)  
    instruction
```

- Tant que **expression** est vraie
faire **instruction**
- **!!! Attention, l'éternité c'est long vers la fin !!!**
- Conseil: utilisez les **{ }**



EXEMPLE

example-03.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a = 5;
    int b = 3;
    int res = 0;

    while (b > 0) {
        res = res + a;
        b = b - 1;
    }

    printf("Le produit est: %d\n", res);
    return res;
}
```



FONCTIONS

- Déclaration de fonction :
`type nom( ) { }`
ou
`type nom(type nom_param1, ...) { }`
- 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
- Utilisation, dit *Appel*, de fonction :
`nom( )` ou `nom(arg1, ...)`



EXEMPLE

example-04.c

```
#include <stdio.h>

int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}

int main(int argc, char* argv[])
{
    int a = 5;
    int b = 3;
    printf("Le max de %d et %d est: %d\n", a, b, max(a, b));
    return 0;
}
```



TABLEAUX

- Notation []
- Commencent à 0
- **Doivent avoir une longueur à l'initialisation**
JAMAIS pour les paramètres
- Pas de primitive longueur !!
- Bornes non vérifiées !!
- Pas de copies
- Ne peuvent pas être retournés

```
int tableau[10];
```

```
int tableau[10] = { 1, 2, 42 };
```



Exemple : max

```
#include <stdio.h>

int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}

int max_tableau(int taille, int tableau[])
{
    int resultat = tableau[0];
    int i = 1;
    while (i < taille) {
        resultat = max(resultat, tableau[i]);
        i = i + 1;
    }
    return resultat;
}

int main(int argc, char *argv[])
{
    int tab[4] = { 10, 40, 42, 20 };
    int res = max_tableau(4, tab);
    printf("Max du tableau: %d\n", res);
    return 0;
}
```



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
- Conseil: Laissez vous des notes:
`// TODO penser à écrire cette fonction`



FONCTIONS ET PROTOTYPES

- deux fonctions paire 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);  
}
```



CHAÎNES DE CARACTÈRES

- Tableaux de char (i.e., `char chaine[ ]`)
 - donc longueur inconnue !
- Généralement terminées par un 0 (zero terminated string)
- Constantes: "`Ceci est une chaine constante`"
- Le format (pour `printf`): `%s`
- Par convention, on utilisera la notation pointeur dans les signatures de fonctions:
`void foo(char *chaine);`

FONCTION DE MANIPULATIONS DE CHÂÎNES

- Ces fonctions sont dans : `#include <string.h>`
- longueur d'une chaîne :

```
unsigned int strlen(char* str);
```

- Comparer deux chaînes :

```
int strcmp(char* s1, char* s2);
```

Retourne: <0 plus petit, 0 pareil, >0 plus grand

- Copier une chaîne :

```
char *strcpy(char *dst, char *src);
```

- Convertir une chaîne en nombre :

```
int atoi(char* str);
```

il faudra `#include <stdlib.h>`



STRUCTURE RÉPÉTITIVES

Boucle classique

```
i = 0 ; // Initialiser
while (i < 10) { // Tester
    instructions // Faire
    i = i + 1; // Incréments
}
```

Boucle compacte

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

- Erreur classique : Oublier d'incrémenter
- Uniquement du sucre syntaxique
- Toutes les infos sur la même ligne
- Conseil: ne les utilisez que quand ils sont clairs



EXEMPLE: FACT

```
int pow(int n, int e)
{
    int r = 1;
    int i = 0;
    while (i < e) {
        r = r * n;
        i = i + 1;
    }
    return r;
}
```

Peut s'écrire:

```
int pow(int n, int e)
{
    int r = 1;
    for (int i = 1; i < e; i = i + 1)
        r = r * n;
    return r;
}
```



LA LIGNE DE COMMANDE

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

- **argc**: nombres d'arguments
- **argv**: tableau des valeurs des arguments
- Retourne le code d'erreur du programme (0 tout va bien)

Le premier argument est toujours le nom du programme.

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

int main(int argc, char *argv[])
{
    int i = 1, sum = 0;
    while (i < argc) {
        sum = sum + atoi(argv[i]);
        i = i + 1;
    }
    printf("La somme du programme %s est : %d\n", argv[0], sum);
    return EXIT_SUCCESS;
}
```

Exercice: réécrire ce programme (et le tester) avec un **for**



REPRÉSENTATION DES NOMBRES

- Bases
 - Constantes décimales : 42
 - Constantes hexadécimales : 0x42 (66)
 - Constantes octales : 042 (34)
 - constantes caractères : '*' (42)
- Négatifs, règle du complément à deux :
 - $x == \sim x + 1$

- Opérateurs bits-à-bits (bitwise)

non	et	ou	ou exclusif	décalage à gauche	décalage à droite
$\sim$	$\&$	$ $	$\wedge$	$\ll$	$\gg$



TYPES ENTIERS

- taille plate-forme dépendante
- **int** : 16 bits mini. (souvent 32)
- **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**)
- Exemple :
unsigned long long foo = 0xffffffff



ORDRE DE GRANDEURS

	Bits	Non signé	Signé
~	n	$[0; 2^{n-1}]$	$[-2^{n-1}; 2^{n-1}-1]$
char	8	$[0; 255]$	$[-128; 127]$
short	16	$[0; 65\,535]$	$[-32\,768; 32\,767]$
long	32	$[0; 4\,294\,967\,295]$	$[-2\,147\,483\,648; 2\,147\,483\,647]$
long	64	$[0; 18\,446\,744\,073\,709\,551\,616]$	$[-9\,223\,372\,036\,858\,374\,607\,295; 9\,223\,372\,036\,858\,374\,607\,295]$



PRINTF

- Chaîne de formats (%) ordonnés
- Attention au nombre d'arguments!
- Rappel: documentation `man 3 printf`

```
printf("Hello %s à %zu lettres", "world", strlen("world"));
```

Format (ex: %u)

d	u	x		O		S	C
Décimal signé	décimal non signé (Unsigned)	hexadécimal		Octal	~	String	Caractère

Taille (ex: %llu)

hh	h	l	ll				z
char	short	Long	Long Long				size_t



TAILLE DES 'OBJETS'

sizeof(...) : donne la taille d'un objet en octet
(retourne un **size_t** $\approx$ **unsigned** au moins **int**)

```
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: %zu\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 EXIT_SUCCESS;
}
```

LOGIQUE

- Pas de booleans
- (pseudo-)Opérateurs

=> pas forcements évalués

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

- Attention: $\sim x$!= !x
- Attention: $a \ \&\& \ b$!= a & b



TABLE DE PRÉCÉDENCE (2)

Op. Description Assoc

~ !	Négation bit-à-bit et logique plus, moins unaire	D
+ -		
* / %	Multiplier, diviser, modulo (reste)	G
+ -	Addition, soustraction (binaire)	G
> >=	Plus grand (ou égal)	G
< <=	Plus petit (ou égal)	
== !=	Égalité, différence (non égalité)	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 un type entier
- Branches constantes
- **Attention aux breaks**

Un exemple

```
if (choix == 'O' || choix == 'o') {  
    printf("Oui");  
} else if (choix == 'N' || choix == 'n')  
    printf("Non merci");  
} else {  
    printf("No lo po compris");  
}
```

```
switch (choix) {  
    case 'O':  
    case 'o':  
        printf("Oui");  
        break;  
    case 'N':  
    case 'n':  
        printf("Non merci");  
        break;  
    default:  
        printf("No lo po compris");  
        break;  
}
```



FLOT DE CONTRÔLE

- **return**

retourne à la fonction appelante avec une valeur qui doit être omise si la fonction est void.

JUSTE POUR ÊTRE COMPLET (MAIS PAS À SAVOIR)

- **break**

sort de la boucle la plus intérieure (*innermost loop*)

- **continue**

retourner au début de la boucle la plus intérieure (*innermost loop*)



QUELQUES OPÉRATEURS

(UTILES ET DANGEREUX)

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

```
a = a + 3;  
// Est équivalent à  
a += 3;
```

	Incrém.	Décrémén.
Pre-	++a	--a
Post-	a++	a--

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



EXPRESSION CONDITIONNELLE

L'OPERATEUR TERNAIRE

- En fonction d'une **expression** (condition), évalue l'une ou l'autre des deux **expressions**
- **ATTENTION à la lisibilité (n'imbriquez pas trop)**
- Conseil parenthésiez tout en cas de doute

Syntaxe

```
cond ? expr-true : expr-false
```

Exemple

```
int max = (a > b) ? a : b
```



LES NOMBRES FLOTTANTS

type	Signe	Exposant	Mantisse
float	1	8	23
double	1	11	52

- IEC 60559 $\approx$ IEEE 754

$$-1^{signe} * (1 + man) * 2^{exp-2^{|exp-1|-1}}$$

- 2 zéros, 2 infinités et quelques *Not a Number* (NaN)
- Attention : $a * b / b \neq a$
- ~~$\text{if}(a == b)$~~ $\text{if}(\text{fabs}(a - b) < \text{epsilon})$



FONCTIONS SUR LES FLOTTANTS

- Par défaut : **double**
version flottante : suivie d'un **f** ; ex: **truncf**
- Puissance : **pow**, **sqrt**
- Arrondis : **floor**, **ceil**
- Troncature : **trunc**
- Valeur absolue: **fabs** (note le **f** au début)
- **isfinite()**, **isinf()**, **isnan()**, **isnormal()**



AFFICHAGE (DES FLOTTANTS)

modificateur	%f	%e	%g
double: 1	notation décimale	notation scientifique	notation la plus adaptée
long double: 11	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 );
```

```
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 minimum de chiffres à afficher
```



TYPES SIMPLES (SCALAIRES)

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



POINTEURS (RÉFÉRENCES)

- Contient une adresse mémoire (flèche)
 - Permet d'accéder à cette adresse (R/W)
 - Les pointeurs sont typés
 - Permet un passage par référence
-
- Convention : **NULL** (**0L**) => Absence de valeur
 - Conseil : Utilisez la métaphore de la flèche, dessinez les !
 - Toujours la même taille (dépendant de la plate-forme)



NOTATIONS

- Déclaration :
`type* var;`
- Initialisation :
`var = NULL;`
`var = autre_ptr;`
`var = &autre_var;`
- Utilisation:
`type a = *var;`
`var = &a;`



*	Suivre / Déréférencer
&	Prendre l'adresse

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

L'*autre* doit avoir le même type

INVERSER DEUX VALEURS

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

```
// Cette version ne marche pas ...
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main(int argc, char *argv[])
{
    int a = 42;
    int b = 24;

    printf("avant swap %d(%p) %d(%p)\n", a, &a, b, &b);
    // Aucune appel du type swap(a, b) ne peut faire l'affaire
    // On doit donc obligatoirement passer par des pointeurs
    swap(&a, &b);
    printf("apres swap %d(%p) %d(%p)\n", a, &a, b, &b);

    return EXIT_SUCCESS;
}
```



POINTEURS ET TABLEAUX

- Les tableaux n'existent pas ! (du point de vue des types)
- Tableau = Pointeur sur une zone contiguë + réservation
(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");
```

En bref

$x[y] \Leftrightarrow *(x + y)$



ARITHMÉTIQUE DES POINTEURS

- Tous les opérateurs arithmétiques d'addition, de soustraction et de comparaison.
 - Addition/soustraction de pointeurs et scalaire :
- Valeur incrémentée de **sizeof (type)**
- Soustraction de deux pointeurs:
- Nombre d'éléments les séparant
(donc divisé par **sizeof (type)**)

```
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;
}
```



POINTEURS CONSTANTS

- Constantes : `const char a = 5;` (depuis C99)
-

- Pointeurs constants:

Le terme juste après `const` est constant :

`char const *foo; ==> *foo = 0` est illegal

`char* const foo ==> foo = bar` est illegal

Par abus de syntaxe :

`const char *foo; <==> char const *foo;`

void * LE TYPE POLYMORPHE

- Tous les pointeurs ont la même taille
- **void *var**; est valide est représente un pointeur quelconque
- Compatible avec tous les types pointeurs
- ***var** est ILLEGAL (on ne sait pas ce qu'il y a au bout)
- AUCUNE arithmétique possible

```
void *memcpy(void *dst, const void *src, size_t bytes)
{
    const char *s = src;
    char *d = dst;
    while (bytes --)
        *d++ = *s++;
    return dst;
}
```



SAISIE DE DONNÉES

- **scanf**
- Format identique à printf (mais pas tous disponibles)
- **Attention à la syntaxe !**
- Le **%s** ne lit jamais d'espace
- Utile pour lire des fichiers formatés
- Pour des saisie utilisateur, préférez **fgets** (et convertisez)

```
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 EXIT_SUCCESS;
}
```

```
int main(int argc, char *argv[])
{
    int r;
    char buffer[51] = {};
    fgets(buffer, 50, stdin);
    printf("J'ai lu: '%s'\n", buffer);
    return EXIT_SUCCESS;
}
```



STRUCTURES

- Types définis par l'utilisateur
- Composition de types nommés (champ)
- Taille doit être calculable par **sizeof ()**
 - Tableau taille connue
 - Pas récursives
 - Peuvent être retournés / copiés.

```
struct point {  
    int x;  
    int y;  
}; // Attention au `;
```

```
void affiche_point(struct point p)  
{  
    printf("%d;%d\n", p.x, p.y);  
}
```

```
int main(int argc, char *argv[])  
{  
    struct point p;  
    p.x = 1; p.y = 2;  
    struct point p1 = { 1, 2 };  
    struct point p2 = { .y=2, .x=1 }; // depuis C99  
    struct point p3 = p;  
    affiche_point(p);  
}
```



POINTEURS ET STRUCTURES

- Permet de faire des structures chaînés
- Priorité du `.` supérieure à `*`
 - Message d'erreur cryptique
 - Syntaxe dédiée

`( *x ) . y` $\Leftrightarrow$ `x->y`

- Types incomplets
 - On ne peut faire QUE des pointeurs
 - On ne peut pas les suivre

```
struct i_list {
    int value;
    struct i_list *next;
};
void append(struct i_list *node,
            struct i_list *other)
{
    node->next = other;
}
```

```
struct foo;

struct bar {
    struct foo *xyzyz;
};

void baz(struct foo *some_foo);
```



ÉNUMÉRATIONS

- Entiers nommés
- Numérotation automatique (mais on peut l'aider)
- Affichage, calcul et autre comme pour un entier

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

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

enum couleur atout = PIQUE;
```

```
struct carte {
    enum valeur valeur;
    enum couleur couleur;
};

struct carte black_jack =
{ VALET, PIQUE };

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



TRUCS ET ASTUCE

Pour les **enums** contigues:

- Pour tester les bornes
 - Définissez au moins un dernier élément dans l'énum
 - Définissez éventuellement un premier élément
- Pour afficher en toutes lettres utilisez
une fonction + un tableau + testez les bornes

```
enum kind {  
    ROCK, POP, JAZZ, HIP_HOP,  
    KIND_COUNT, FIRST_KIND=ROCK  
};  
  
static const char *txt[KIND_COUNT] = {  
    "Rock", "Pop", "Jazz", "Hip Hop"  
}; // RQ static sera expliqué plus tard  
  
const char *kind_txt(enum kind k)  
{  
    if (k >= FIRST_KIND && k < KIND_COUNT)  
        return txt[k];  
    return NULL; // éventuellement "UNKNOWN"  
}
```

PRÉ-PROCESSEUR

- Avant de compiler le programme
- Commence par un **#** et se termine à la fin de la ligne
 - Constantes (généralisés par la notion de Macro)

Défini par **#define**, substitué jusqu'à la fin du fichier

Attention aux priorités et évaluations multiples

- Inclusion de fichier (cf. slide suivant)
- Compilation conditionnelle (cf. slide sur-suivant)

```
#define FOO 42
#define BAR (FOO + 3)
#define BAZ FOO + 3

int a = FOO; // a == 42
int b = BAR * 2; // b == 90
int c = BAZ * 2; // Attention: c == 48
```

```
#define MAX(__x, __y) \
  ((__x) > (__y) ? (__x) : (__y))

int m = MAX(foo(), bar());
// Nombre d'appel de foo et bar ?
int f = foo(), b = bar();
int mm = MAX(f, b); // OK
```



FICHIERS D'EN-TÊTES

- Par convention se terminent par **.h**
- Inclus par **#include** suivi de :
 - **"fichier.h"** s'il s'agit d'un fichier perso.
(répertoire courant)
 - **<fichier.h>** s'il s'agit d'un fichier std.
(include path)
- Uniquement des déclarations :
 - Prototypes
 - Constates/Macros
 - Types (eventuellement incomplets)
 - etc
- **Attention aux récursions (cf. slide suivant)**



PRÉPROCESSEUR ET COMPILEUR SÉPARÉE

- Prototype non implémenté
=> fonction externe(**extern**)
- Éviter les dépendances circulaires (astuce des **#ifndef**)
- Exemple de compilation :

```
cc -c main.c
cc -c calcul.c
cc -o executable main.o calcul.o
```

```
// calcul.h
#ifndef __CALCUL_H
# define __CALCUL_H

#define TAILLE_MAX 42

struct bar; // Types incomplets

void foo(int x); // Fctn. ext.
void baz(struct bar *xyzy);
// ... autres prototypes
#endif
```

```
// main.c
#include "calcul.h"

// ...
foo(42);
```

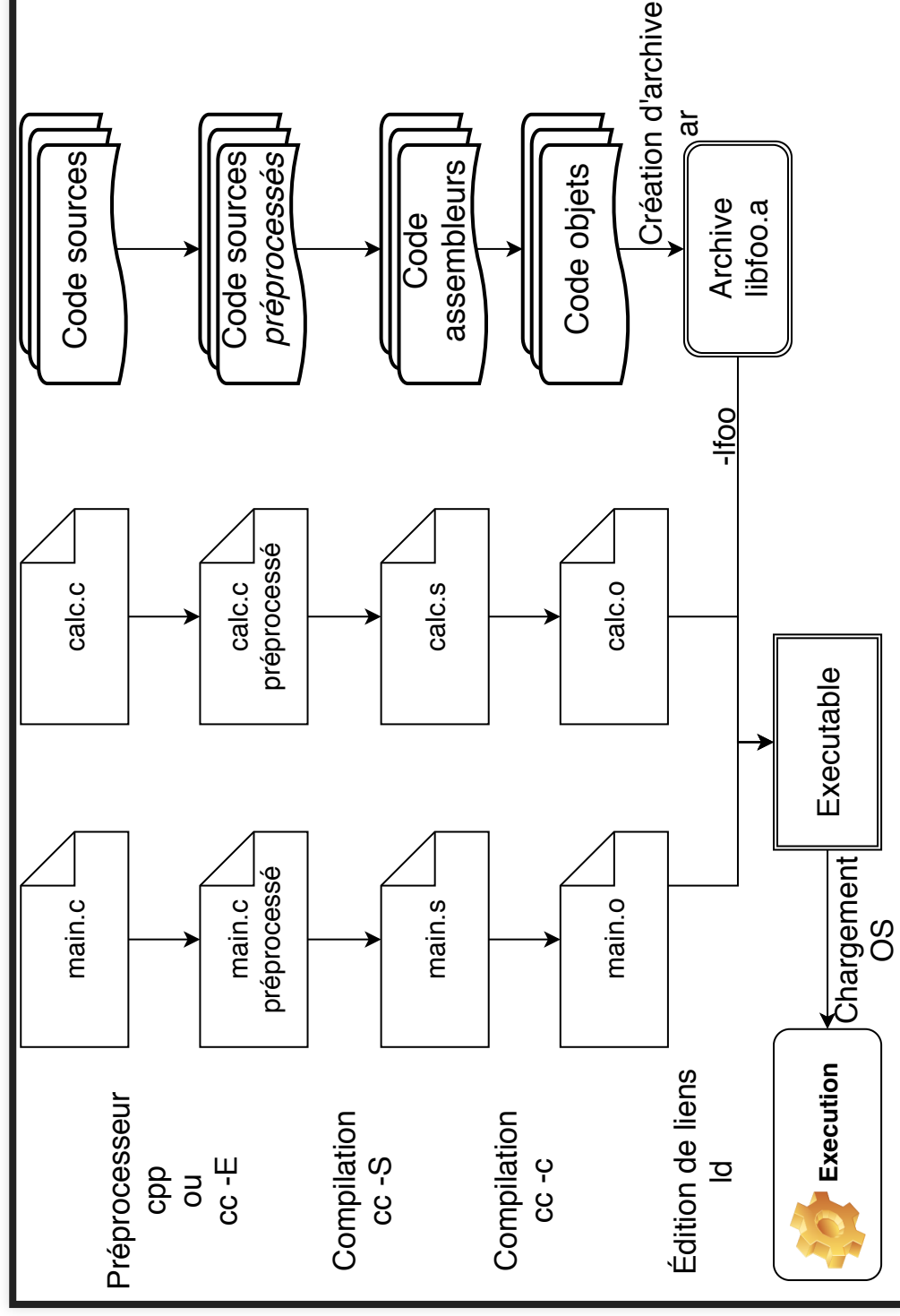
```
// calcul.c
#include "calcul.h"

void foo(int n) {
    // ...
}
```

VARIABLES ET COMPIATION SÉPARÉE

- Variables locales valable uniquement dans la fonction (ce qu'on fait depuis le début, et ce qu'il faut maximiser)
- Variables globales, valables tout le programme (rare)
- Tous les noms globaux sont exportés
 - **static** sur une variable globale l'empêche
 - **static** sur une variable locale
 - => La variable garde sa valeur entre les appels
 - **extern** sur une variable globale/fonction
 - => Il existe une var/fonction, sa définition est ailleurs

CHAÎNE DE COMPILATION



GESTION DE FICHIERS

- Type associé : FILE *

- Mode d'ouverture du fichier

- Limite du nombre fichier ouvert

Tout fichier ouvert doit être fermé

- Protocole

1. Ouvrir fichier

```
FILE* fopen(char* nom, char* mode);
```

2. Travailler sur le fichier (slides suivants)

3. Fermer le fichier

```
int fclose(FILE* fd);
```



MODES D'OUVERTURES

mode	Mode d'ouverture	Position
r	Lecture	Début
w	Écriture	Début
a	Écriture. Écritures toujours à la fin (pas de déplacement)	Fin
r+	Lecture/écriture. Pas de création	Début
w+	Lecture/écriture. Le fichier peut être créé	Début



FICHIERS TEXTES

Ces fonctions considèrent le contenu du fichier comme des chaînes de caractères.

- `printf("foo");` $\Leftrightarrow$ `fprintf(stdout, "foo");`
- `int fprintf(FILE *fd, char *fmt, ...);`
- `int fscanf(FILE *fd, char *format, ...);`
- essai de lire une ligne :
`int fgets(char *buffer, int size, FILE *fd);`
- lit un char. `int fgetc(FILE *fd);`
- écrit un char. `int fputc(int c, FILE *fd);`



FICHIERS DÉJÀ OUVERTS

- **stdin**
Entrée standard en lecture. Par défaut le clavier
- **stdout**
Sortie standard en écriture. Par défaut l'écran
- **stderr**
Sortie d'erreur en écriture. Par défaut l'écran

Rediriger:

```
./mon_programme < fichier_in > fichier_out 2> fichier_err
```



MANIPULATION GÉNÉRIQUES

(OU QUELQUONQUES)

- Lectures / Écritures:

```
unsigned fread(void *ptr, unsigned size,  
unsigned nitems, FILE *stream);  
unsigned fwrite(const void *ptr, unsigned size,  
unsigned nitems, FILE *stream);
```

- Position / déplacement:

```
long ftell(FILE *fd);  
void rewind(FILE *fd);  
int fseek(FILE *fd, long offset, int whence);
```



EXEMPLE COPIER UN FICHIER

```
#define BSIZE 100
void copy_file(FILE *dst, FILE *src) {
    int r;
    char buffer[BSIZE];
    while((r = fread(buffer, 1, BSIZE)) > 0) {
        if(fwrite(buffer, 1, r, dst) <= 0)
            break;
    }
}
```

- Exercice : Faire l'équivalent de la commande **cp**
- Il vous faudra gérer plus proprement les erreurs.



ORDRE DES OCTETS

ENDIANNESS (BOUTISME)

- A partir du moment où il y a échange de donnée, l'ordre des octets compte
- Deux grandes familles
 - **Big-Endian**: Octet de poids fort en premier (ordre réseau, Motorola 68k, Java, jpeg, ...)
 - **Little-Endian**: Octet de poids faible en premier (intel, FAT, ext2, ...)
- Il faut donc reconstruire les entiers dans le bon ordre



GESTION DE LA MÉMOIRE

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



FONCTIONS UTILES

- Aloue initialisé
`void *calloc(size_t item_count, size_t item_size)`
- Réduit ou étend une zone
`void *realloc(void* ptr, int size)`
- Initialise une zone à value
`void *memset(void *buff, int value, size_t len)`
- Copie une zone
`void *memcpy(void *d, const void *s, size_t len)`



GESTION DES ERREURS

- Pas de mécanisme dédié
- Variable globale : `int errno; (errno.h)`
Elle contient le dernier code d'erreur
- `void perror(char* prefix);`
Affiche le prefixe suivi du dernier message d'erreur
- Toutes les fonctions systèmes
 - Gestion de fichier
 - Gestion de la memoire
 - etc.



ERREURS DURANT LE DÉVELOPPEMENT

- Pour vous aider lors du développement (**assert.h**):
assert(expr);
arrête le programme si expr est faux
- Considérez que ces lignes peuvent être ignorées
(il suffit de compiler avec **-DNDEBUG**)

```
if ((fd = fopen(file, "r")) == NULL) {  
    perror("Error opening file");  
    exit(1);  
}
```

```
fd = fopen(file, "r");  
assert(fd); // c'est mal
```

GOTO

- Ne devrait jamais être utilisé
=> code spaghetti
- Sauf pour rattraper des erreurs

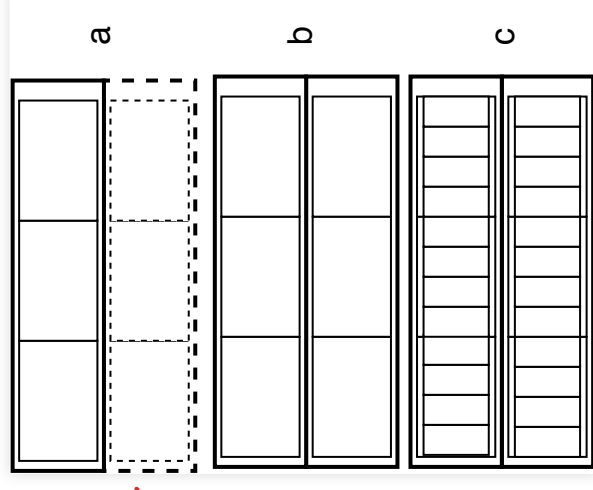
```
void do_something() {  
    int *buffer = malloc()  
    if (!buffer) goto end;  
    // do things  
    if (some_error_happens) goto free_memory;  
    // do more things  
    free_memory:  
    free(buffer);  
end:  
}
```



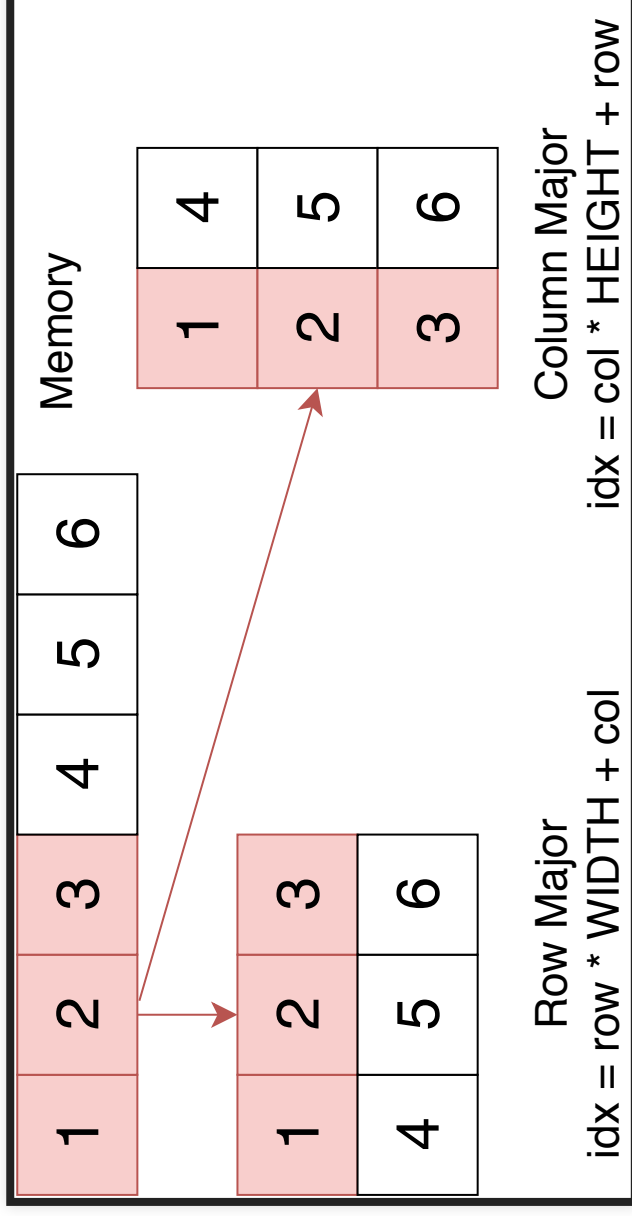
TABLEAUX MULTI-DIMENSIONNELS EN C

- Homogènes (tous les éléments ont le même type)
- Dense (Pas de trous, toutes les lignes ont la même taille)
- Toutes les dimensions sont fixés à la déclaration
- La première dimension est libre dans les paramètres
- `void f(int a[]) <=> void f(int *a)`
- `void f(int a[][20]) <!=> void f(int`
- *Row-Major* cf. slide suivante
- Peu utilisé cf. slides suivante

```
void foo(int a[][3])
{
    int b[2][3]; // Se lit b est :
    // un tableau de deux tableaux
    // de trois entiers
    int c[2][3][4];
}
```



TABLEAUX MULTI-DIMENSIONNELS



- Conseil faites des tableaux 1D
- Linéarisez vous-même (faites une fonction)

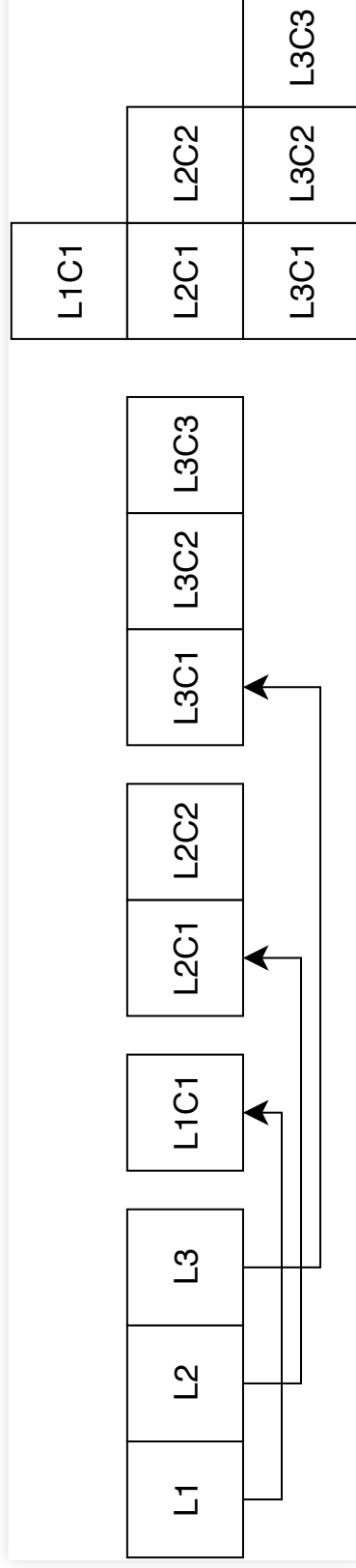
```
int idx(int row, int col, int width, int height)
{
    return row * width + col;
}
```



TABLEAUX NON DENSES

```
// Déclaration statique
int l1[1] = {L1C1},
l2[2] = {L2C1, L2C2},
l3[3] = {L3C1, L3C2, L3C3};
int *tab[3] = {l1, l2, l3};

// Avec allocation dynamique
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 tjrs pas ici ...
```



UNIONS

- Comme une structure où **au plus** un champs peut être utilisé en même temps.
- La taille (**sizeof**) est au moins égale à la taille du plus grand élément
- Souvent utilisé en conjonction avec une structure et une énumération

```
union some_value {  
    int i;  
    double d;  
};
```

```
union some_value v;  
v.i = 10;  
v.d = 20.0;  
// v.i invalide maintenant  
printf("d: %lf i:%d", v.d, v.i);
```



TYPES ANONYMES

Pour les **struct**/**enum**/**union**

- On peut définir des variables après les **{ }**
- Le nom est facultatif
- Utilisé pour les *sous*-structures

```
struct p_value {  
    enum {INT, DOUBLE} type;  
    union {  
        int i;  
        double d;  
    } value;  
};
```

```
struct p_value v =  
{ INT, {.i = 45} };  
switch (v.type) {  
    case INT:  
        printf( "%d", v.value.i);  
        break;  
    case DOUBLE:  
        printf( "%lf", v.value.d);  
        break;  
}
```



POINTEURS DE FONCTIONS

- Les fonctions sont chargées en mémoire, elles ont donc une adresse
- On peut faire un pointeur sur cette adresse (en utilisant simplement le nom de la fonction)
- On peut appeler la fonction via son pointeur
- **Attention à la syntaxe relativement tordue**

`type_de_retour (*) (types_des_parametres)`

- Pour nommer une variable (un paramètre), le nom est dans l'
(*) après l'*

```
int (*compare)(const char*, const char*) = strcmp;  
compare(s1, s2); // Compare en tenant compte de la case  
compare = strcasecmp;  
compare(s1, s2); // Compare sans tenir compte de la case
```



ALIAS DE TYPE

typedef

- Permet de renommer un type
- Malheureusement compatible avec le type dont il est l'alias
- Permet de faire disparaître la différence entre une structure et un scalaire
- Est en général **TRES** mal utilisé :
 - Pour s'économiser le mot **struct**
 - Cacher un pointeur (*)
- Aucun cas simple où l'utilisation est justifié

typedef type a aliaser nouveau_nom;



CAS ACCEPTABLES

VOIRE PROPRE

- Renommer un pointeur de fonction si ça améliore la lisibilité
- Masquer des détails de l'architecture
- S'assurer de la présence d'un **volatile** dans un type (utile pour communiquer avec du *hardware*)
- Fabriquer un type opaque (en général types systèmes)
on ne doit vraiment pas savoir s'il s'agit d'un scalaire ou d'une structure

```
typedef void (*print_function)(void *);  
void print_array(int len, void **array, print_function fun);
```

```
#if ULONG_MAX > UINT_MAX  
    typedef long my_int64;  
#elif ULONG_MAX > ULONG_MAX  
    typedef long long my_int64;  
#endif
```

```
typedef volatile int vint;
```

```
typedef int pid_t;  
pid_t get_pid();
```



CONCLUSION

```
int main(int argc, char *argv[])  
{  
    return EXIT_SUCCESS;  
}
```

