Langage C

Eric Lecolinet

Télécom ParisTech

Octobre 2022

Brève historique (très partielle)

1972 : Langage C
AT&T Bell Labs

1983 : Objective C
NeXt puis Apple
Extension objet du **C**
Syntaxe inhabituelle inspirée de Smalltalk

1985 : C++
AT&T Bell Labs
Extension object du **C** par *Bjarne Stroustrup*

1991 : Python
Guido van Rossum
Simplicité et rapidité d'écriture
Interprété, typage **dynamique** (= à l'exécution)

1995 : Java
Sun, puis Oracle
Purement objet. Inspiré de C++ mais aussi Smalltalk, ADA, etc.

2001: C#
Microsoft
Originellement proche de Java. Inspiré de **C++**, Delphi, etc.

2011: C++11
Consortium ISO
Révision majeure Suivie de : C++14, C++17, C++20, C++23

Et aussi : Kotlin, Scala, Swift, Rust, Javascript, Ruby, Go, Ada, etc.

Sept. 2022

TIOBE
the software quality company

About us | Join TIOBE | News | Coding Standards | TIOBE Index | Contact

Products | Quality Models | Markets | Book a demo

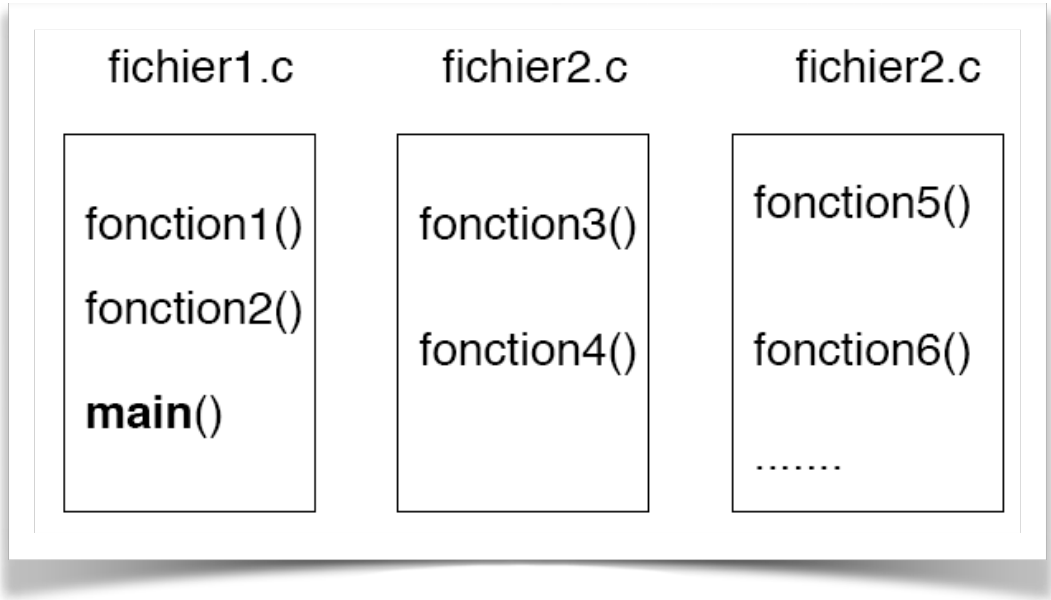
Sep 2022	Sep 2021	Change	Programming Language	Ratings	Change
1	2	↑	Python	15.74%	+4.07%
2	1	↓	C	13.96%	+2.13%
3	3		Java	11.72%	+0.60%
4	4		C++	9.76%	+2.63%
5	5		C#	4.88%	-0.89%
6	6		Visual Basic	4.39%	-0.22%
7	7		JavaScript	2.82%	+0.27%
8	8		Assembly language	2.49%	+0.07%
9	10	↑	SQL	2.01%	+0.21%
10	9	↓	PHP	1.68%	-0.17%
11	24	⇈	Objective-C	1.49%	+0.86%
12	14	↑	Go	1.16%	+0.03%
13	20	⇈	Delphi/Object Pascal	1.09%	+0.32%
14	16	↑	MATLAB	1.06%	+0.04%
15	17	↑	Fortran	1.03%	+0.02%
16	15	↓	Swift	0.98%	-0.09%

Aperçu d'un programme C

Un programme C comprend :

- un ou plusieurs **fichiers** C
- définissant des **fonctions**
- éventuellement des **variables globales**

- une seule fonction **main**()
  = la 1ère fonction appelée



- une **difficulté :**
  faire en sorte que les **appels** et des **définitions** des fonctions **correspondent**

Variables

Définition des variables

```
int i, j, k;          trois entiers
int tab[10];          un tableau de 10 entiers
int * p;              un pointeur d'entiers
```

ATTENTION la valeur est (généralement) indéterminée !

=> TOUJOURS initialiser les variables !!!

```
int i = 0, res = 0, truc = 0;
int tab[10] = {0};
int * p = NULL;
```

Les variables sont typées

Types de base

int	entier (short <= int <= long)
short	entier court (>= 16 bits)
long	entier long (>= 32 bits)
bool (ou **_Bool**)	booléen (inclure **stdbool.h**)
char	caractère ASCII / petit entier
float	flottant simple précision
double	flottant double précision
long double	encore plus précis

Par défaut les **entiers** sont **signés**

Pour préciser le **signe** :

- **signed int**
- **unsigned long**

Les **char** sont **signés** ou... **pas** !
selon la **plateforme !**

- **unsigned char** : de 0 à 255
- **signed char** : de -128 à 127

Types de base

Les tailles des types de base dépendent des plate-formes !

- spécifiées dans **limits.h** et **float.h** (dans: /usr/include sous Linux)

> **Example** :
>
> - **short** = 16 bits / **int** = **long** = 32 bits
> - **float** = 32 bits / **double** = 64 bits

Types de base normalisés

- **même taille** sur toutes les plateformes (définis dans **stdint.h**)

```
int8_t
int16_t
int32_t
int64_t
intmax_t
uint8_t
uint16_t
uint32_t
uint64_t
uintmax_t
etc.
```

Constantes

1234 037 (octal) 0x1f (hexadécimal)

1.234 1. 1e-2 => type **double**

'a' (caractère) => type **int** **'a', 'b', 'c'... sont des int pas des char !**

"abcd" (littéral) => type **char *** **litttéral = suite de caractères**

Constantes

Enumérations

```
enum  Jour  {LUNDI = 1, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
enum  Jour j = JEUDI;
if (j == SAMEDI) printf("en week end\n")
```

- commencent à 0 si on ne précise pas
- solution simple et efficace pour spécifier des **constantes**

Variables à valeur constante

```
const  float pi = 3.14;    // la valeur ne peut pas changer
```

Macros

Substitution textuelle avant la compilation

- par le **préprocesseur C**
- peuvent avoir des **paramètres** (voir plus loin)
- potentiellement **dangereuses** :

  => préférer les **enum** ou les **variables const** quand c'est possible

```
#define  TAILLE  1024
#define  MESSAGE  "hello world"

enum  {TAILLE = 1024};
const char*  MESSAGE = "hello world";
```

Tableaux

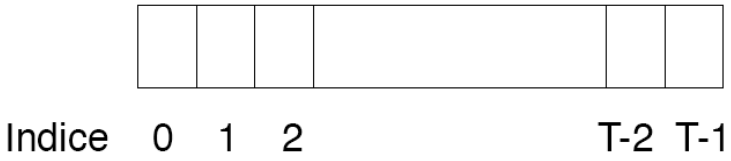
int vect[10]; tableau de 10 entiers

double mat[5][10]; tableau bi-dimensionnel de 5 lignes et 10 colonnes

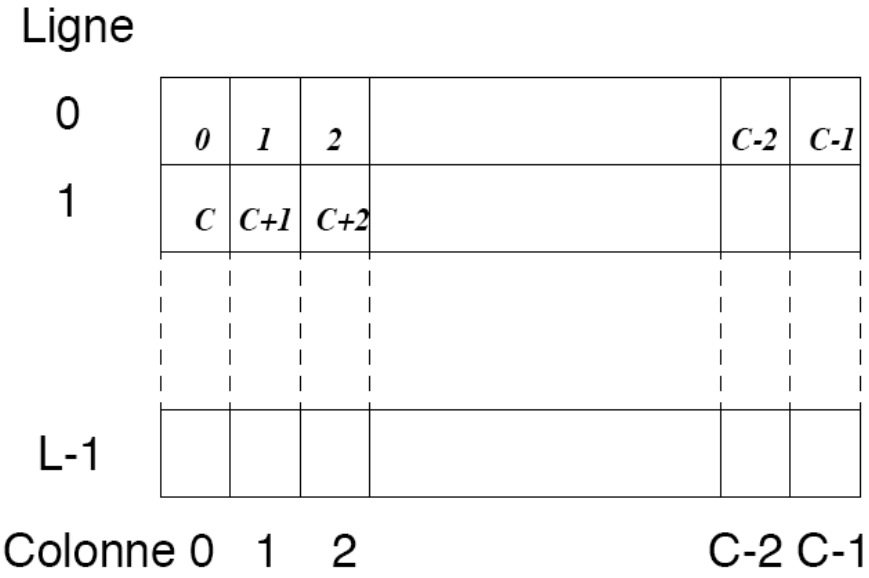
float cube[5][10][20]; tableau tri-dimensionnel

- les indices vont de **0** à **taille-1**
- les éléments sont **contigus** (placés en séquence)

int vect[T];



double mat[Lignes][Colonnes];



Tableaux

Initialisation

```
int vect[10] = {1, 2, 3};
int vect[] = {1, 2, 3};      // taille implicite
float mat[5][10] = {{1, 2, 3, 4}, {5, 6}, {7, 8, 9}};      // initialisation ligne à ligne
float mat[][10] = {{1, 2, 3, 4}, {5, 6}, {7, 8, 9}};      // tableau de 3 lignes (taille implicite)
```

La taille **la plus à gauche** peut être **implicite**

Paramétrer la taille des tableaux !

```
enum {TAILLE = 1024};        // ou:   #define  TAILLE  1024
char tableau[TAILLE] = {0};
```

sinon le programme sera **difficilement modifiable !**

Chaînes de caractères

Chaîne de caractères (strings)

- **tableaux** ou **littéraux**
- toujours terminées par le **caractère nul** (qui vaut 0)

```
const char* message = "hello";      // "hello" est un littéral
```

- en mémoire: 'h' 'e' 'l' 'l' 'o' '\0' (caractère **nul** ajouté **automatiquement**)
- les littéraux sont **constants** (ne **pas** modifier leur contenu !)

Codage des strings

- en **ASCII**
- ou en **UNICODE**

Code ASCII:

- 'A' = 65,
- 'O' (la lettre) = 78
- '0' (le chiffre) = 48
- '\0' (**NUL**) = 0
- '\n' : passage à la ligne
- '\t' : tabulation
- '\007' : caractère en octal

Code ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Unicode

- **but** : coder les symboles de **toutes les langues**
- **autrefois** : codage dépendant de la langue et du constructeur
- **normalisation : UNICODE**
  - **UTF-8** : chaque symbole fait **de 1 à 4 octets**
    - **Web**, Linux, MacOSX, Windows (récemment)
  - **UTF-16** : chaque symbole fait **2 ou 4 octets**
    - Java, Windows (autrefois)

Structures

Agrégats de données, ancêtres des **classes** (classes sans méthodes)

```
struct POINT {
    double x, y;
};

void foo() {
    struct POINT p1, tab[10];          // le mot clé struct est nécessaire !
    struct POINT p2 = {4.1, 7.3};      // initialisation comme tableaux
    p1 = p2;                           // = copie champ à champ
    p1.x = 1.;                         // . accède à un champ
    tab[7] = p2;
    tab[7].x = p2.x;
}
```

Structures et typedef

```
struct POINT {
   double x, y;
};

typedef struct POINT Point;      // typedef définit un nouveau type

void foo() {
     Point p1, tab[10];      // pas de mot clé struct
     Point p2 = {4.1, 7.3};
     p1 = p2;
     p1.x = 1.;
     tab[7] = p2;
     tab[7].x = p2.x;
}
```

on peut aussi écrire :

```
typedef struct POINT {
     double x, y;
} Point;
```

```
typedef struct {
     double x, y;
} Point;
```

Structures imbriquées

```
#include <stdio.h>

typedef struct {
    typedef struct {
        double x, y;
    } Point;                    // la struct Point est imbriquée dans la struct Rect

    Point  p1, p2
} Rect;

void foo() {
    Point a = {1., 2.}, b = {3., 4.}
    Rect r1;
    Rect r2 = {{1., 2.}, {3., 4.}};
    r1.p1 = a;
    r1.p2 = b;
    printf(" rect1 = (%f, %f) x (%f, %f) \n", r1.p1.x, r1.p1.y, r1.p2.x, r1.p2.y);
}
```

Opérateurs arithmétiques

Ordre de priorité

- unaire -x

*** / %**

+ - binaire x-y

```
l = i + j * k;          // attention à la précédence des opérateurs !
m = (i + j) * k;
```

Division

/ est la **division entière ou réelle** suivant le cas : 2 / 4 vaut 0 mais 2**.** / 4 vaut 0.5 **!!!**

% est le **reste** de la division entière

Transtypage implicite

Par défaut : conversion **implicite** vers le **type le plus grand**

Attention aux erreurs !

```
int i = 2, j = 4;
char c = '\n';
float x = 4., y1, y2, y3;

i = c;
c = i;
x = i;
i = x;
y1 = i + x;
y2 = i / x;
y3 = i / j;
```

Que vaut y3 ?

Transtypage implicite

Par défaut : conversion **implicite** vers le **type le plus grand**

```
int i = 2, j = 4;
char c = '\n';
float x = 4., y1, y2, y3;

i = c;
c = i;          /* entier tronqué */
x = i;
i = x;          /* réel tronqué */
y1 = i + x;
y2 = i / x;
y3 = i / j;
```

!!! y3 vaut 0 car i et j sont des int

Transtypage explicite

L'opérateur de cast fait une conversion de type explicite

```
int i = 2, j = 4;
float y, z;

y = i / j;              // y = ?
z = (float) i / j;      // z = ?
```

Transtypage explicite

L'opérateur de cast fait une conversion de type explicite

```
int i = 2, j = 4;
float y, z;

y = i / j;              // y = 0.0
z = (float) i / j;      // z = 0.5
```

Attention ne marche que pour les opérations basiques

- contre-exemple **FAUX :**

```
char * s = "33";
int i = 12;

s = (char*) i;    // compile mais faux !!!
i = (int) s;      // compile mais faux !!!
```

Incrémentation

Opérateurs

- **i++** et **++i** signifient **i = i + 1**
- **i += n** signifie **i = i + n,** même chose avec: **i--, --i, i -= n**

i++ versus ++i

```
i = 5; a = ++i;      /* i = ? , a = ? */
i = 5; a = i++;      /* i = ? , a = ? */
```

Incrémentation

Opérateurs

- **i++** et **++i** signifient: **i = i + 1**
- **i += n** signifie: **i = i + n,** même chose avec: **i--, --i, i -= n**

i++ versus ++i

```
i = 5; a = ++i;      /* i = 6 , a = 6 */
i = 5; a = i++;      /* i = 6 , a = 5 */
```

Attention aux ambiguïtés !

- l'abus de ++ / -- peut rendre une expression difficile à lire ou **indéfinie !**

Opérateurs logiques et relationnels

Priorité

! (négation unaire)

> >= < <=

== !=

&& (ET logique)

|| (OU logique)

moins prioritaires que les opérateurs **arithmétiques** (sauf négation unaire)

x < lim-1 équivaut à : x < (lim-1)

Evalués de la **gauche vers la droite : arrêt** dès que la valeur de vérité est trouvée :

```
if (k < TABSIZE && tab[k] != 0) ...

if (i <= 3 && (c = getchar()) != 'y')) etc...
```

Expressions conditionnelles

```
if (a > b) z = a;
else z = b;
```

```
if (k < tabsize && tab[k] != 0)  {        // rappel: arrêt dès que possible
  doit(tab[k]);
}
```

- ne pas oublier les **( )** autour du test et les **;**
- le **else** se rapporte au **if** le plus proche

mettre des **{ }** et **indenter** le code pour éviter les **ambiguïtés :**

peu lisible:

```
if (n > 0)
    if (a > b)
        z = a;
    else z = b;
```

```
if (n > 0) {
    if (a > b) z = a;
    else z = b;
}
```

```
if (n > 0) {
    if (a > b)  z = a;
}
else z = b;
```

Expressions conditionnelles

```
if (test1)  action1;
else if (test2)  action2;
else if (test3)  action3;
else action_par_défaut;
```

- structure de tests en "rateau"
- ne pas mettre des **{ }** inutiles

```
switch (expression) {
case const1:
   action1a;
   action1b;
   action1c;
   break;

case const2:
case const3:
   action2;
   break;

default:
   action3;
   break;
}
```

- **const1**, **const2** .... doivent être des **constantes** à valeur entière
- **break** ou **return** **sortent** du **switch**

Booléens

```
#include <stdbool.h>        // il faut inclure ce header
bool is_valid = false;      // true ou false

if (is_valid)  printf("Welcome dear customer \n");

 if (! is_valid)  call_the_police();      // ! est l'opérateur de négation
```

Avant **C99** et aujourd'hui encore : souvent **simulés par des macros :**

```
#define  FALSE 0
#define  TRUE  1
```

Opérateur ?

Principe

L'expression: test **?** expr1 **:** expr2 renvoie :

- expr1 si test est != 0
- expr2 sinon

Exemple

z = a > b **?** a : b ; équivaut à : **if** (a > b) z = a; **else** z = b;

Boucles

```
for (init; test; incr) {
    actions;
}
```

Equivaut à :

```
init;
while (test) {
    actions;
    incr;
}
```

- continuent tant que **test** != 0
- **test** effectué **avant** d'entrer dans la boucle
- **init**, **test** et **incr** sont des expressions quelconques

NOTE : préférer **for** à **while** : évite d'oublier des clauses !

```
#define TAB_SIZE  20        // définir tailles des tabeaux par des constantes
int tab[TAB_SIZE];
int i = 0;

for (i = 0; i < TAB_SIZE; i++) {        // les tableaux vont de 0 à TAB_SIZE-1
    tab[i] = 0;
}
```

Boucles

break, continue, goto

- **break** sort d'une **boucle** ou d'un **switch**
- **continue** passe à l'itération suivante
- **goto** : éviter sauf cas particuliers

```
for (init; test; incr) {
    actions;
}
```

Boucle do while

```
do {
    actions;
} while (test);
```

actions est exécuté au moins une fois

Boucles

Que fait ce fragment de code et que faut-il rajouter ?

```
char line[] = "il pleut parfois le lundi";
int i = 0;

for (i = 0;  ???; i++) {
    if (line[i] == ' ') break;
}
```

Boucles

Que fait ce fragment de code et que faut-il rajouter ?

```
char line[] = "il pleut parfois le lundi";
int i = 0;

for (i = 0;  line[i] != 0; i++) {
    if (line[i] == ' ') break;
}
```

Goto

A éviter – un cas où c'est utile :

```
bool foo() {
    for (....) {
        ...
        if (niveau_gasoil == 0) goto BIG_PROBLEM;
        ...
        for (....) {
            ....
            if (Touché(iceberg)) goto BIG_PROBLEM;
        }
    }
    return true;

    BIG_PROBLEM :      // tous les cas d'erreurs
        printf("Fatal error: The Titanic has sunk!\n")
        return false;
}
```

Manipulation de bits

Opérateurs

& ET

| OU inclusif

^ OU exclusif

<< décalage à gauche

>> décalage à droite

~ complément à un

Attention: ne pas confondre :

& avec **&&** (et logique)

| avec **| |** (ou logique)

```
int n = 0xff, m = 0;
m = n & 0x10;
m = n << 2;          // équivalent à:  m = n * 4
```

Priorité des opérateurs

Precedence	Operator	Description	Associativity

1	`++ --`
	`()`
	`[]`
	`.`
	`->`
	`(type){list}`
		Suffix/postfix increment and decrement
		Function call
		Array subscripting
		Structure and union member access
		Structure and union member access through pointer
		Compound literal(C99)
			Left-to-right

2	`++ --`
	`+ -`
	`! ~`
	`(type)`
	`*`
	`&`
	`sizeof`
	`_Alignof`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		Type cast
		Indirection (dereference)
		Address-of
		Size-of
		Alignment requirement(C11)
			Right-to-left

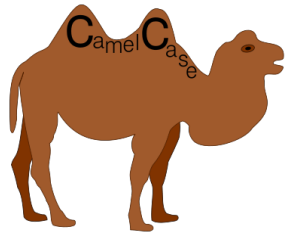
3	`* / %`	Multiplication, division, and remainder	Left-to-right
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=` `> >=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13[note 1]	`?:`	Ternary conditional[note 2]	Right-to-Left

14	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		Simple assignment
		Assignment by sum and difference
		Assignment by product, quotient, and remainder
		Assignment by bitwise left shift and right shift
		Assignment by bitwise AND, XOR, and OR	

15	`,`	Comma	Left-to-right

Style et commentaires

```
typedef struct {
    unsigned int width_, height_;
    float * pixels_;
} Image;

Image * createImage(unsigned int width,  unsigned int height);

unsigned int getWidth(Image *);
```



Règles

- être **cohérent**
- **indenter** (utiliser un IDE qui le fait automatiquement : **TAB** ou **Ctrl-I** en général)
- **aérer** et **passer à la ligne** (éviter plus de 80 colonnes)
- **camelCase** et mettre le **nom des variables** (pour la doc)
- **commenter** quand c'est utile

Doxygen (génération de la documentation)

```
/** @brief Image de pixels.
  * @see createImage(), setPixel(), getPixel().
 */
typedef struct {
    unsigned int width_, height_;
    float * pixels_;
} Image;

/** @brief crée une image.
  * Les pixels sont initialisés à 0.
 */
Image * createImage(unsigned int width,
                    unsigned int height);

/// retourne la largeur de l'image.
unsigned int getWidth(Image *);

unsigned int getHeight(Image *);
///< retourne la hauteur de l'image.
```

```
/** @brief change la valeur d'un pixel.
  * _value_ doit être positive.
  * @see getPixel()
  */
void setPixel(Image *,
              unsigned int x,
              unsigned int y,
              float value);

float getPixel(Image *,
               unsigned int x,
               unsigned int y);
/**< @brief retourne la valeur d'un pixel.
  *  @see setPixel().
  */
```

Fonctions

Objectif

Découper un programme en **petites** entités :

- indépendantes
- réutilisables
- plus lisibles

Librairies et compilation séparée

Les fonctions peuvent être définies :

- dans **plusieurs** fichiers
- dans des **librairies**
-

ChercheVal V1 (un seul fichier “main.c”)

```c
#include <stdio.h>

int ChercheVal(float tab[ ], int tabCount, float val) {
    int k;
    for (k = 0; k < tabCount; k++) {
        if (tab[k] == val) return k;
    }
    return -1;
}

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};

    int indice = ChercheVal(donnees, 10, 4.);

    /* que manque t’il ici ? */
    printf("indice = %d , valeur = %f \n", indice, donnees[indice] );

    return 0;
}
```

ChercheVal V1 (un seul fichier “main.c”)

```
#include <stdio.h>

int ChercheVal(float tab[ ], int tabCount, float val) {
    int k;
    for (k = 0; k < tabCount; k++) {
        if (tab[k] == val) return k;
    }
    return -1;
}

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};

    int indice = ChercheVal(donnees, 10, 4.);

    /* que manque t'il ici ? */
    printf("indice = %d , valeur = %f \n", indice, donnees[indice] );

    return 0;
}
```

Remarques

- noter l’initialisation des tableaux
- il manque un test

Return et void

```
int ChercheVal(float tab[ ], int tabCount, float val) {
    int k;
    for (k = 0; k < tabCount; k++) {
        if (tab[k] == val) return k;
    }
    return -1;
}
```

ne pas oublier **return sinon la fonction renvoie n'importe quoi !**

void si la fonction ne **retourne rien** ou n'a **aucun paramètre**

```
void foo(void) {
  .....
}
```

Remarque

syntaxe obsolète :

```
void foo(i, j)
   int i, j;
{
   .....
}
```

- aucune vérification des arguments
- vieilles versions du langage C

Passage des arguments

Passage par valeur

- les **valeurs** passées en **argument** sont **recopiées** dans les **paramètres** des fonctions
- y compris les **struct** (recopiées **champ à champ**)

```
int ChercheVal(float tab[ ], int tabCount, float val) {
     .......
}

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    indice = ChercheVal(données, 10, 4.);
    .......
}
```

Passage des arguments

Passage par valeur

- les **valeurs** passées en **argument** sont **recopiées** dans les **paramètres**
- **mais pour les tableaux c'est leur adresse qui est recopiée !**
    - plus précisément l'adresse de leur 1er élément

```
int ChercheVal(float tab[ ], int tabCount, float val) {
     .......
}

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    indice = ChercheVal(donnees, 10, 4.);
    .......
}
```

Passage des arguments

Passage par valeur de l'adresse des tableaux

- *tab* est en réalité un **pointeur** (et non un **tableau**) !
- => foo() ne peut pas savoir le **nombre d'éléments**
- => le passer **en argument**

ATTENTION : sizeof ne peut **pas** marcher dans ce cas !!!

```
int ChercheVal(float tab[ ], int tabCount, float val) {
    .......
}

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    indice = ChercheVal(donnees, 10, 4.);
    .......
}
```

Passage des arguments

Question : quelles sont les valeurs affichées ?

```
void swap(int i, int j) {
    int aux = i;
    i = j;
    j = aux;
}

int main() {
    int i = 5, j = 7;
    swap( i, j );
    printf( " i = %d , j = %d \n", i , j );
}
```

Solution ?

Argc, argv

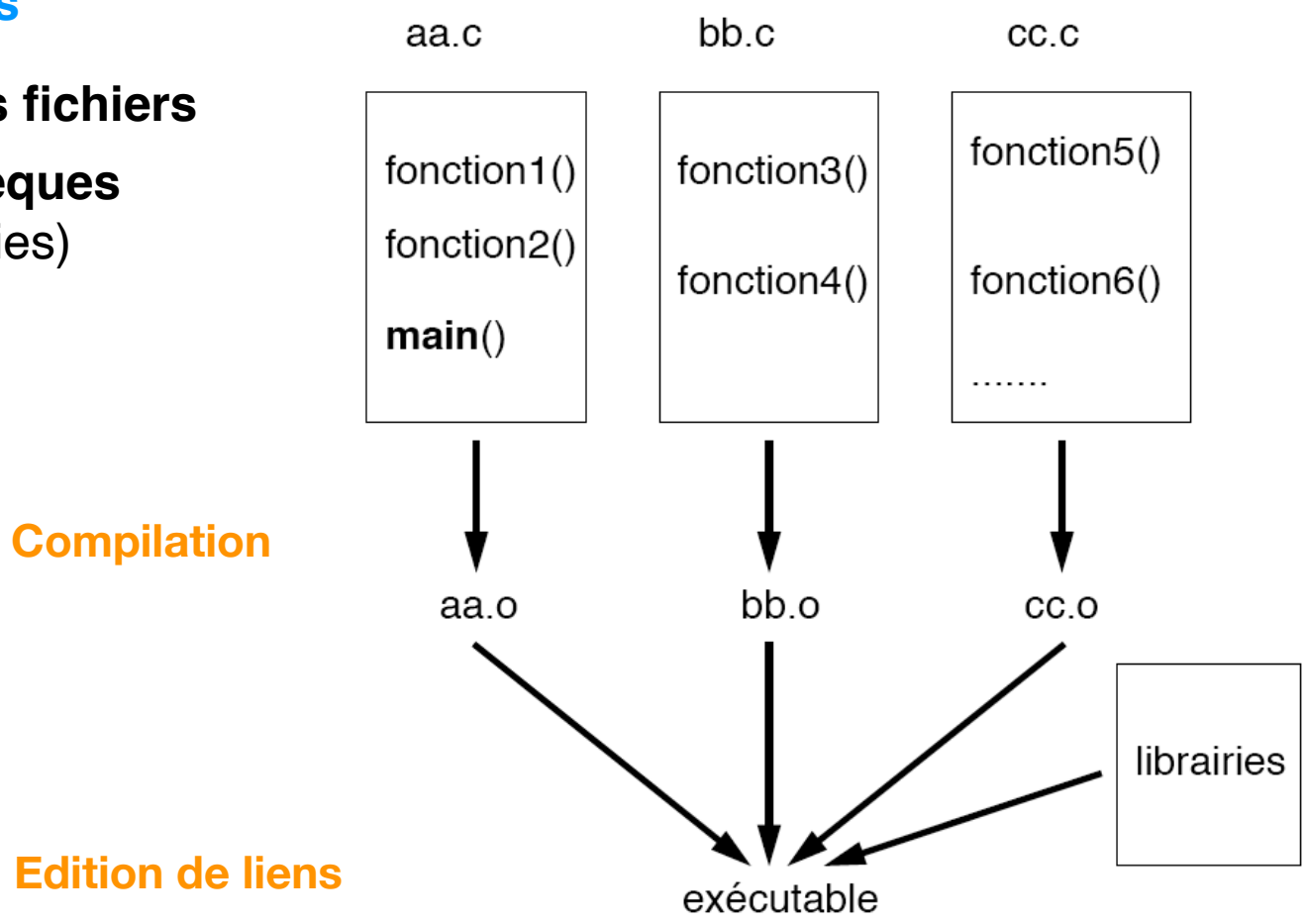
```
int main(int argc, char *argv[])
{
     if (argc <=1 ) {
          printf("Erreur: il faut au moins un argument !\n");
         return 1;
     }

     printf("Nom du programme: %s \n", argv[0]);
     printf("Premier argument: %s \n", argv[1]);
     .....
     return 0;
}
```

- **argc** : nombre d'arguments de la ligne de commande
- **argv** : tableau des arguments (chaque élément est un char*)
- **return** : retourne un code d'erreur (0 signifie OK)

Compilation séparée

Fonctions réparties

- dans **plusieurs fichiers**
- et les **bibliothèques** système (libraries)



Exemple : version incorrecte !

fichier cherche.c

```
int ChercheVal(float tab[ ], int tabCount, float val) {
    int k;
    for (k = 0; k < tabCount; k++) {
        if (tab[k] == val) return k;
    }
    return -1;
}
```

fichier main.c

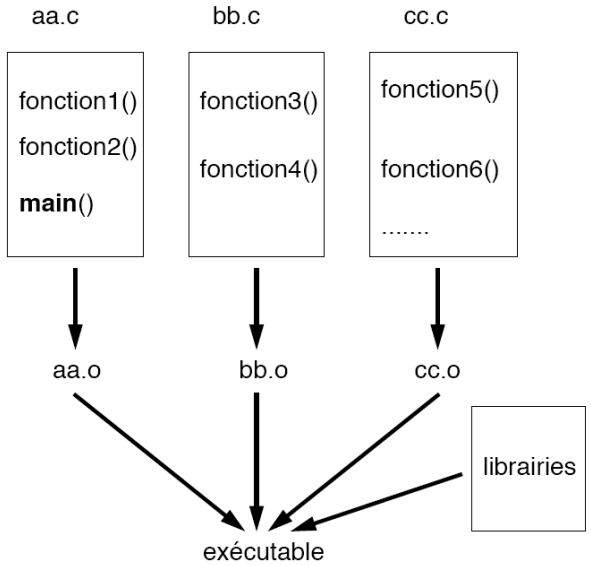
```
#include <stdio.h>

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    int indice = ChercheVal(donnees, 10, 4.);

    if (indice >= 0) {
        printf("indice = %d , valeur = %f \n", indice, donnees[indice] );
    }
    return 0;
}
```

compilation: gcc -o myprog cherche.c main.c

Quel est le problème ?

Problème

```
int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    int indice = ChercheVal(donnees, 10, 4.);
    ....
}
```



Les fichiers C sont compilés indépendamment

Si on appelle une fonction **non déclarée** :

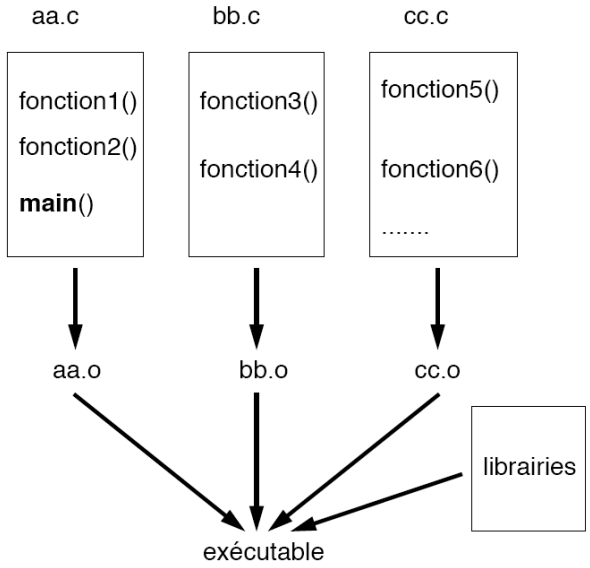
- **C** suppose qu'elle **existe** et **renvoie int** (*)
- **C ne vérifie pas** que les **arguments** correspondent aux **paramètres** !

source d'erreurs !

- *(*) selon les options des compilateurs (et leur age !)*

Problème

```
int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    int indice = ChercheVal(donnees, 10, 4.);
    ....
}
```



Trois cas possibles

— ChercheVal **existe** et a la **même signature** :
=> coup de bol, ça marche !

— ChercheVal **n'existe pas** :
=> erreur d'édition de lien « Symbol not found »

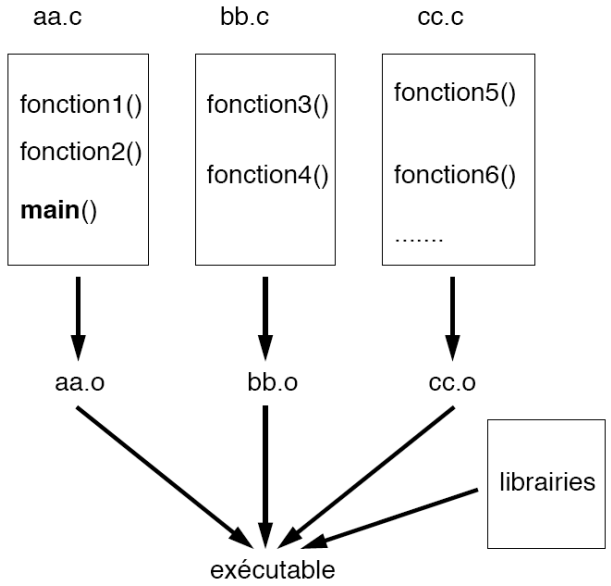
— ChercheVal **existe** mais n'a **pas la même signature** :
=> compile mais "comportement indéterminé" !!!

Solution : header partagés

Principe

- à chaque fichier **.c** qui **définit** des fonctions
- associer un fichier **.h** (« header ») qui **déclare** ces fonctions
- inclure ce header :
  - dans **les** fichiers .c qui **appellent** ces fonctions
  - **ET** dans **le** fichier .c qui **définit** ces fonctions

Cohérence vérifiée par transitivité !



Exemple : version correcte

fichier cherche.h

```
int ChercheVal(float tab[ ], int tabCount, float val) ;
```

fichier cherche.c

```
#include “cherche.h”

int ChercheVal(float tab[ ], int tabCount, float val) {
    int k;
    for (k = 0; k < taille; k++) {if (tab[k] == val) return k;}
    return -1;
}
```

fichier main.c

```
#include <stdio.h>
#include “cherche.h”

int main() {
    float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
    int indice = ChercheVal(donnees, 10, 4.);
    if (indice >= 0) printf("indice = %d , valeur = %f \n", indice, donnees[indice] );
    return 0;
}
```

Déclarations et headers

Déclaration de fonction

```
int ChercheVal(float tab[ ], int taille, float val) ;        // ; à la fin
```

déclare la **signature** (valeur de retour et paramètres) de cette fonction

Recherche des headers

```
#include "cherche.h"    // cherche dans le même répertoire (ou on compile)
#include <stdio.h>      // cherche dans répertoires système (e.g. /usr/include)
```

pour chercher dans des **répertoires supplémentaires** : **option -I**

```
gcc  -I/usr/local/qt/include  truc.c  -o truc
```

Appel de fonctions des bibliothèques

Ce qui précède vaut aussi pour les fonctions des bibliothèques

```c
#include <stdio.h>

int main() {
    double x = cos(0.5);
    printf("Résultat: %f \n", x);
}
```

Appel de fonctions des bibliothèques

Ce qui précède vaut aussi pour les fonctions des bibliothèques

```
#include <stdio.h>
#include <math.h>    // !!!!!!

int main() {
    double x = cos(0.5);
    printf("Résultat: %f \n", x);
}
```

compile mais résultat indéterminé sans le #include !

Headers des bibliothèques standard

Les plus courants

- `<stdio.h>` : entrées/sorties (`printf`, `scanf`)
- `<stdlib.h>` : fonctions générales (`malloc`, `rand`)
- `<stddef.h>` : définitions générales (déclaration de la constante `NULL`)
- `<string.h>` : chaînes de caractères (`strcmp`, `strlen`)
- `<math.h>` : fonctions mathématiques (`sqrt`, `cos`)

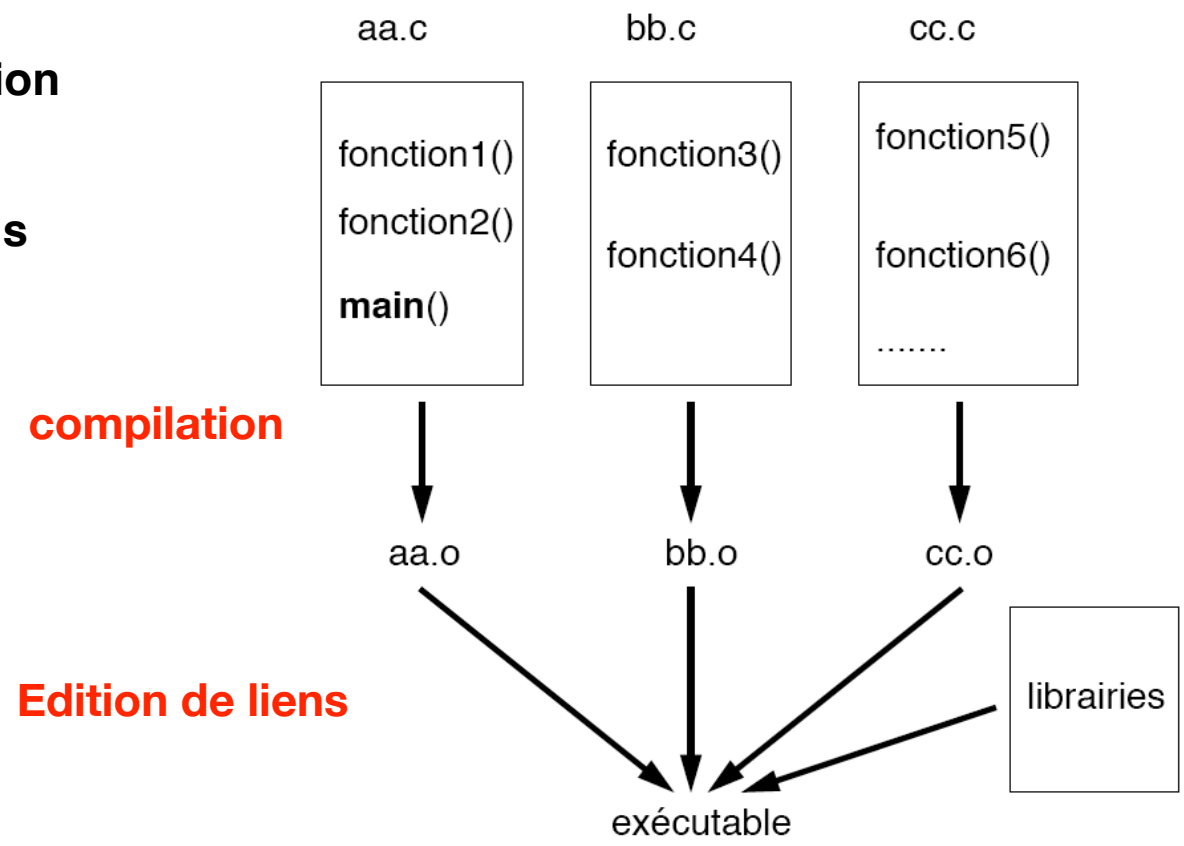
Egalement

- `<assert.h>` : assertions (`assert`)
- `<ctype.h>` : caractères (`isalnum`, `tolower`)
- `<errno.h>` : gestion des erreurs (déclaration de la variable `errno`)
- `<signal.h>` : gestion des signaux (`signal` et `raise`)
- `<time.h>` : manipulation du temps (`time`, `ctime`)
- et bien d'autres liés au système (cf. /usr/include sous Unix)

Headers et binaires des bibliothèques standards

Bibliothèques standard

- à chaque **header**
  - qu'il faut **inclure à la compilation**
- correspond un **binaire**
  - qu'il faut **lier à l'édition de liens**



Emplacements standards

- headers dans: /usr/include
- binaires dans: /usr/lib
- ou dans : /usr/local/... etc.

Makefile

Makefile = fichier de règles indiquant:

- les **fichiers** à compiler
- les **compilateurs** et leurs options
- les **bibliothèques**...

Typiquement : **un Makefile par répertoire**

La commande make

- lit le **Makefile**
- appelle **automatiquement** les outils adéquats (compilateurs, éditeur de lien ...)
- vérifie les **dates** et ne recompile **que** ce qui doit l'être

Règle d'or

- ne **pas** taper les commandes de compilation à la main (toujours fausses !)
- utiliser un **Makefile** ou un IDE

Exemple de Makefile

```
# compilateur C et ses options
CC= gcc
CFLAGS= -g -Wall -std=c99 -I/usr/local/qt/include

# librairies utilisées
LDLIBS = -L/usr/local/qt/lib -lqt

# fichiers objet de l'application et nom de l'exécutable
OBJS= tri.o donnees.o
EXEC= tri

# règle de production de l'application (la 2e ligne commence par une tabulation!)
$(EXEC) : $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS) $(LDLIBS)

# nettoyage
clean:
	-$(RM) $(EXEC) $(OBJS)
```

make : lance la 1ere règle (qui compile tout)

make clean : lance la règle "clean"

Ce makefile est incomplet :

Il tient compte des .c mais pas des .h !

- rajouter les règles à la main
- ou utiliser un outil comme **makedepend** (voir TP)

Bibliothèques statiques et dynamiques

Librairies statiques

- extension .a (sous Unix)
- archives de fichiers .o
- code **inséré dans l'exécutable**

Bibliothèques statiques et dynamiques

Librairies statiques

- extension .a (sous Unix)
- archives de fichiers .o
- code **inséré dans l'exécutable**

Librairies dynamiques (DLLs)

- extension .so, dylib, etc.
- code chargé **dynamiquement** à l'exécution

Avantages

- les programmes prennent **moins de place**
- ils sont **plus rapide**s (moins de swapping car binaire partagé)

Inconvénients

- l'utilisateur **doit avoir cette librairie** installée sur sa machine
- => attention aux **licences** et aux **versions**

Sous Unix

Les DLLs sont recherchées dans l'**ordre** indiqué par la variable**:**

LD_LIBRARY_PATH (ou équivalent) du shell Unix

Compilateur et éditeur de liens

Principales options du compilateur C

- **-g** pour deboguer
- **-O/-O1/-O2**... pour optimiser
- -**Wall** pour afficher plus d'infos avec gcc (toujours mettre cette option !)
- **-I***directory* : chercher les headers dans ce répertoire (*)

Principales options de l'éditeur de liens

- **-L***directory* : chercher les librairies dans ce répertoire (*)
- **-l***library* : chercher les fonctions dans cette librairie

(*) on peut mettre plusieurs **-I***directory* et **-L***directory* mais **l'ordre** importe

Doxygen (génération de la documentation)

```
/** @brief Image de pixels.
  * @see createImage(), setPixel(), getPixel().
 */
typedef struct {
    unsigned int width_, height_;
    float * pixels_;
} Image;

/** @brief crée une image.
  * Les pixels sont initialisés à 0.
 */
Image * createImage(unsigned int width,
                    unsigned int height);

/// retourne la largeur de l'image.
unsigned int getWidth(Image *);

unsigned int getHeight(Image *);
///< retourne la hauteur de l'image.
```

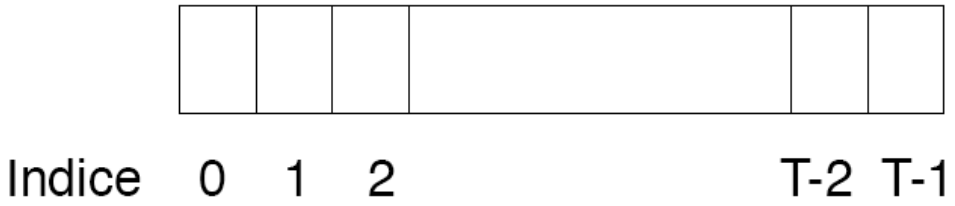
```
/** @brief change la valeur d'un pixel.
  * _value_ doit être positive.
  * @see getPixel()
  */
void setPixel(Image *,
              unsigned int x,
              unsigned int y,
              float value);

float getPixel(Image *,
               unsigned int x,
               unsigned int y);
/**< @brief retourne la valeur d'un pixel.
  *  @see setPixel().
  */
```

Adresses, tableaux, pointeurs

Adresses et tableaux

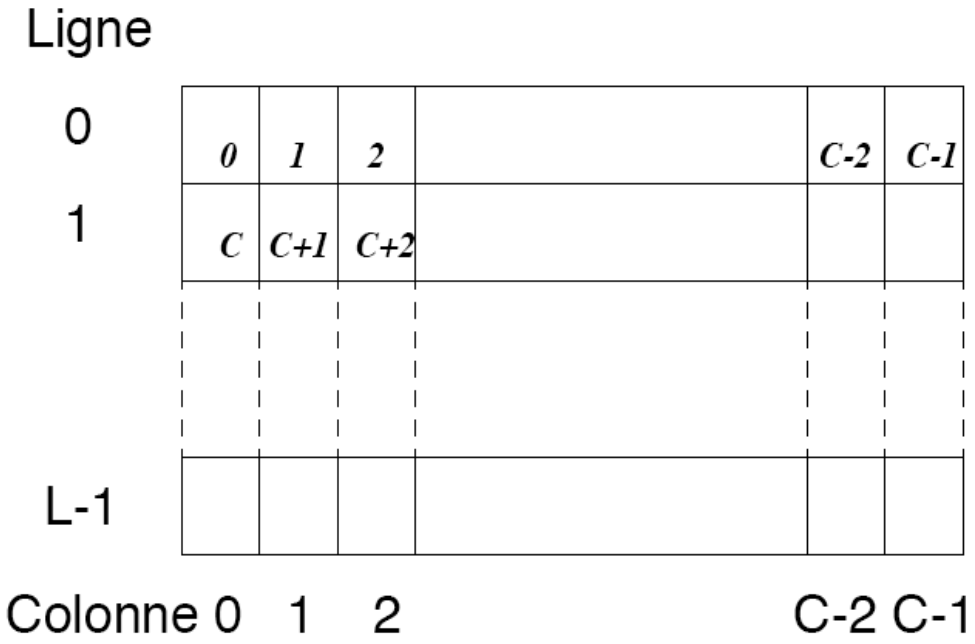
Vecteur

type vect[T];



Matrice

type mat[L][C];



Adresses et déréférencement

Operateurs

- **&**c : **adresse** d'une entité c
- ***a** : **valeur** pointée par une adresse a (opérateur de **déréférencement**)

Adresse du 1er élément. de vect : **&**vect[0] == **&**(vect[0]) == **vect**

Adresse du ième elt. de vect: **&**vect[i] == **&**(vect[i]) == vect + i

Valeur du ième elt. de vect : vect[i] == *(vect + i)

Adresse de l'elt. (x,y) de mat : **&**mat[y][x] == mat + y * nb_col + x

Valeur de l'elt. (x,y) de mat : mat[y][x] == *(mat + y * nb_col + x)

Pointeurs

- **variables** contenant une **adresse**
- les **pointeurs** sont (généralement) **typés**

Exemple

```
int i = 0, *p =NULL;        // p = pointeur d'entier

p = &i;
*p = 5;
printf("%d %d \n", *p, i);

i = 7;
printf("%d %d \n", *p, i);
```

Arithmétique des pointeurs

```
char tabchar[10];
char* pc = tabchar;

int tabint[10];
int* pi = tabint;

pc = pc + 5      //  5 caractères plus loin (= 5 octets)
pi  = pi  + 5;   //  5 entiers plus loin (= 20 octets si int sur 4 octets)
```

L'**arithmétique des pointeurs** n'a de sens que s'ils sont **typés** !

Pointeurs et tableaux

Les **pointeurs** et les **tableaux** sont intimement liés :

```
float *p = NULL, *q = NULL, tab[] = {1, 2, 3, 4, 5};
p = &tab[2];
printf("%f %f \n", *p, tab[2]);

p = tab + 2;
printf("%f %f \n", *p, tab[2]);

*p = 0.0;
printf("%f %f \n", *p, tab[2]);

q = p++;
*(q+2) = 666.666;
printf("%f %f \n", *p, *q)
```

Valeur des éléments de tab ?

Pointeurs et tableaux

un **pointeur** est une **variable** (et peut donc changer de valeur)

```
float *p = NULL, tab[] = {1, 2, 3, 4, 5};
p = tab;      CORRECT
p++;          CORRECT
```

un **tableau n'est pas** une **variable** !

```
tab = p;      NE COMPILE PAS !!!
tab++;        NE COMPILE PAS !!!
```

mais dans les **2 cas** :

```
float val = tab[i];      CORRECT      tab[i] == *(tab + i)
float val = p[i];        CORRECT       p[i]  == *(p + i)
```

Interlude

```c
#include <stdio.h>

int main() {
  int tab[] = {1,2,3};

  int a1 = tab[1];
  int a2 = 1[tab];
  int a3 = *(tab+1);
  int a4 = *(1+tab);

  printf("%d %d %d %d \n", a1, a2, a3, a4);
}
```

Interlude

```c
#include <stdio.h>

int main() {
  int tab[] = {1,2,3};

  int a1 = tab[1];
  int a2 = 1[tab];
  int a3 = *(tab+1);
  int a4 = *(1+tab);

  printf("%d %d %d %d \n", a1, a2, a3, a4);
}
```

```
$ make tt
$ ./tt
2 2 2 2
```

Passage des arguments (le retour !)

Passage par valeur de *l'adresse* des tableaux

```
void foo(float tab[ ], unsigned int tab_count) {
    ......
}
```

équivaut à:

```
void foo(float* tab, unsigned int tab_count) {
    ......
}
```

- ***tab*** est en réalité un **pointeur** !

Passage des arguments par pointeur

Les pointeurs (ou les tableaux) **permettent de récupérer une valeur** via une **indirection**

```
void foo() {
    int i = 0;
    float x = 0;
    char str[10];
    scanf("%d %f %s", &i, &x, str);
}
```

- les arguments de **scanf**() doivent être des **adresses**
- ce qui lui permet de modifier les **valeurs** des pointés
- noter qu'on passe **str** et non **&str** en argument !

Exemple

```
void swap1(int i, int j)
{
    int aux;
    aux = i; i = j; j = aux;
}

int main()
{
    int a = 5, b = 0;
    swap1(a, b);
    printf(" a = %d, b = %d \n", a, b);
}
```

Exemple

```
void swap1(int i, int j)
{
    int aux;
    aux = i; i = j; j = aux;
}

int main()
{
    int a = 5, b = 0;
    swap1(a, b);
    printf(" a = %d, b = %d \n", a, b);
}
```

a et b inchangés

```
void swap2(int* pi, int* pj)
{
    int aux;
    aux = *pi; *pi = *pj; *pj = aux;
}

int main()
{
    int a = 5, b = 0;
    swap2(&a, &b);
    printf(" a = %d, b = %d \n", a, b);
}
```

swap2 échange les valeurs de a et b

car pi et pj de swap2() pointent sur les variables a et b de main()

Chaînes de caractères (strings)

En C, une chaîne de caractères est :

- une suite de caractères **terminée par un 0** (la valeur entière nulle)

Deux manières de les définir :

- par un **tableau** : **char** s**[ ]** = "abcd";
- par un **pointeur** pointant sur un **littéral** : **char*** p = "abcd";

Dans les deux cas le 0 final est rajouté **automatiquement**

Question :

- **sizeof**(s) = ?
- **sizeof**(p) = ?

Chaînes de caractères (strings)

En C, une chaîne de caractères est :

- une suite de caractères **terminée par un 0** (la valeur entière nulle)

Deux manières de les définir :

- par un **tableau** : **char** s**[ ]** = "abcd";
- par un pointeur pointant sur un **littéral** : **char*** p = "abcd";

Attention !

- **sizeof**(s) = 5 : nombre de caractères + 0 final
- **sizeof**(p) = taille du **pointeur** (ex : 4 pour processeur 32 bits)

Rappel

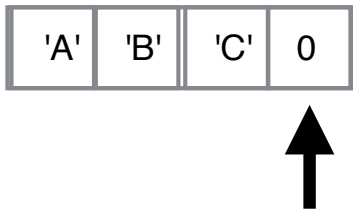
```
void foo(float tab[ ], unsigned int tab_count) {      // sizeof(tab) = taille du pointeur !
    ....
}
```

Exemple

Calcul de la longueur d'une chaîne de caractères

```
int strlen(const char* s) {        /* Version 1 */     int strlen(const char s[ ]) { ...}
    int n = 0;
    for ( ; *s != 0; s++) n++;
    return n;
}
```

const : valeur qui ne peut **pas** être **modifiée**



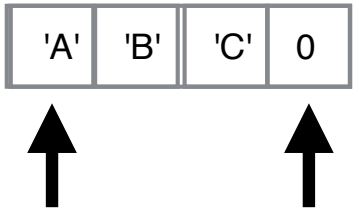
exemple: strlen("ABC")

Exemple

Calcul de la longueur d'une chaîne de caractères

```
int strlen(const char* s) {        /* Version 1 */
    for (int n = 0; *s != 0; s++) n++;
    return n;
}

int strlen(const char* s) {        /* Version 2 */
    const char* p = s;
    while (*p) p++;          // équivaut à : while (*p != 0) p++;
    return p - s;
}
```

p - s : nombre de char entre **s** et **p**



Remarque sur les TPs

A rendre via eCampus :

- **questions 6, 7 et 8**
- si vous êtes bloqué(e) avant, **passez au plus vite à la question 6**

Ne pas bloquer sur les questions !

- **passer à la suivante** (demander de l'aide aux encadrants au TPs suivant)
- essayer de suivre **l'emploi du temps**

Lire une chaîne de caractères depuis le terminal

```
char* s;              // OK ?
scanf("%s", s);

char s[20];           // OK ?
scanf("%s", s);

char s[1000];         // OK ?
scanf("%s", s);
```

Lire une chaîne de caractères depuis le terminal

```
char* s;              // FAUX !!!
scanf("%s", s);

char s[20];           // DANGEREUX !!!
scanf("%s", s);

char s[1000];         // DANGEREUX (mais un peu moins...)
scanf("%s", s);
```

Pourquoi ?

- parce que la mémoire n'est **pas allouée** (1er cas)
- ou possiblement **trop petite** (2e et 3e cas)
- **énorme source d'erreurs et de piratage**

Lire une chaîne de caractères depuis le terminal

Plus sûr :

```
char s[20];
scanf("%19s", s);
```

- lit **au plus** 19 caractères (19 = 20-1 : attention au 0 final !)
- s'arrête au **premier espace** (ou tabulation ou retour chariot)

Lire une chaîne de caractères depuis le terminal

Plus général :

```
char buffer[100];
fgets(buffer, sizeof(buffer), stdin));      // stdin = entrée standard
```

char* **fgets**(**char*** *str*, **int** *taille*, **FILE*** *stream*)

- lit la **ligne entière** depuis le fichier
- mais **au plus** *taille-1* **octets**
  - le reste sera lu au prochain appel de **fgets**()
- retourne **NULL** en **fin de fichier** ou en **cas d'erreur**

```
while (fgets(buffer, sizeof(buffer), file)) {
    ……
}
```

Entrées/sorties

```
char* fgets(const char* buffer, int buffer_size, FILE* stream)
int fputs(const char* buffer, FILE* stream)
```

- **fgets**() renvoie le buffer ou **NULL** en cas d'erreur ou en fin de fichier

```
int fscanf(FILE* stream, const char* format, ...)
int fprintf(FILE* stream, const char* format, …)
```

- **fscanf / fprintf** renvoient le **nombre d'éléments** lus ou imprimés

- **entrées/sorties console** : pseudo fichiers
  - **stdin**
  - **stdout**
  - **stderr**

Buffers de caractères

int **sscanf**(**const char*** str, **const char*** format, …)

int **sprintf**(**char*** str, **const char*** format, ...) // **danger :** débordement si *str* trop petit !

int **snprintf**(**char*** str, **size_t** taille, **const char*** format, …);

Exemple

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define CODE_SECRET "1234"

int main(int argc, char**argv)
{
  bool is_valid = false;
  char code[5];

  printf("Enter password: ");
  scanf("%s", code);

  if (strcmp(code, CODE_SECRET) == 0)
     is_valid = true;

  if (is_valid)
     printf("Welcome dear customer ;-)\n");
  else
     printf("Invalid password !!!\n");

  return 0;
}
```

Questions :

Que fait ce programme ?

Est-il sûr ?

Mémoire et piratage

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define CODE_SECRET "1234"

int main(int argc, char**argv)
{
  bool is_valid = false;
  char code[5];

  printf("Enter password: ");
  scanf("%s", code);

  if (strcmp(code, CODE_SECRET) == 0)
     is_valid = true;

  if (is_valid)
     printf("Welcome dear customer ;-)\n");
  else
     printf("Invalid password !!!\n");

  printf("Adresses: %p %p %p %p\n",
         code, &is_valid, &argc, argv);
  return 0;
}
```

Avec LLVM sous MacOSX 10.7.1 :

- **Enter password:** 111111
- **Welcome dear customer ;-)**

- Adresses:
- 0x7fff5fbff98a 0x7fff5fbff98f 0x7fff5fbff998 0x7fff5fbff900

Débordement mémoire :

— technique de **piratage informatique**

— typiquement pour changer l'adresse de retour dans la pile

Mémoire dynamique

But

- créer ou détruire **dynamiquement** des objets pendant l'exécution

Principe

- **allouer** une zone mémoire et faire pointer un **pointeur** dessus
- **libérer** cette zone mémoire quand elle n’est **plus utilisée**

Mémoire dynamique

Inclure le header **stdlib.h** : #include <stdlib.h>

void* **malloc**(**size_t** *size*) : **alloue** de la mémoire

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

size_t **sizeof**(*datatype*) : **taille en octets** d'un type ou d'une variable

```
#include <stdlib.h>

void foo() {
    int count = 0;
    scanf("%d", &count);

    int * tab = malloc(count * sizeof(int));
    for (int k = 0; k < count; ++k) tab[k] = 0;
    ....

    free(tab);          // free() libère la mémoire (elle alors peut être réutilisée)
    tab = NULL;         // NULL indique que tab pointe sur rien

    ....
}
```

Mémoire dynamique

void* **malloc**(**size_t** *size*) : **alloue** de la mémoire

void **free**(**void** * *ptr*) : **libère** la mémoire pointée

size_t **sizeof**(*datatype*) : **taille en octets**

void* **calloc**(**size_t** *count*, **size_t** *size*) :

- similaire à **malloc**() mais met la **mémoire à 0**

void* **realloc**(**void** * *ptr*, **size_t** *size*) :

- change la **taille** de la zone pointée par *ptr*
- appelle **malloc**() et **free**() si nécessaire

Remarque

malloc(), **calloc**(), **realloc**(), renvoient **NULL** si **pas assez de mémoire**

Mémoire dynamique

ATTENTION

- **toujours initialiser les pointeurs !!!!!!!!!**
- **toujours mettre leur valeur à NULL** quand ils ne **pointent sur rien !!!!!!!!!**

```
#include <stdlib.h>

void foo() {
    ....
    int * tab = malloc(count * sizeof(int));
    ....
    free(tab);
    tab = NULL;
    ....
    tab[0] = 1;          // FAUX car la mémoire à été désallouée
    free(tab);           // OK car tab est NULL
}
```

Exemple (solution « naive »)

```c
#include <stdio.h>      /* pour printf */
#include <stdlib.h>     /* pour malloc */

float* NewVect(int card) {
    float *v = malloc(card * sizeof(float));
    if (v == NULL) fprintf(stderr, "NewVect: No more memory\n");
    return v;
}

float* AddVect(float* v1, float* v2, int card) {
    if (v1 == NULL || v2 == NULL) {
        fprintf(stderr, "AddVect: Null argument\n");
        return NULL;
    }
    float* res= NewVect(card);
    if (res != NULL) {
        for (int k = 0; k < card; k++)  res[k] = v1[k] + v2[k];
    }
    return res;
}
```

```c
void foo() {
    float a[3] = {1., 2., 3.};
    float b[3] = {4., 5., 6.};
    float *x = NULL, *y = NULL;

    x = AddVect(a, b, 3);
    y = AddVect(x, a, 3);
    ……..
    free(x);   // ne pas oublier !
    free(y);
}
```

Améliorations

L’exemple précédent présente plusieurs inconvénients

- lequels ?
- que faudrait-il faire ?

Améliorations

L'exemple précédent présente plusieurs inconvénients

Les opérations effectuent un **malloc**() :

- 1) **coût** (si beaucoup d'opérations)
- 2) **il faut faire des free**() pour libérer la mémoire, ***mais ou ?***

  => **complique** le code et peut occasionner des **erreurs**
- 3) résultat (via return) **pas compatible avec les tableaux**

Améliorations

L'exemple précédent présente plusieurs inconvénients

Les opérations effectuent un malloc() :

- 1) **coût** (si beaucoup d'opérations)
- 2) il faut **faire des free()** pour libérer la mémoire, mais ou ?

  => complique le code et peut occasionner des erreurs
- 3) résultat (via return) **pas compatible avec les tableaux**

Solutions

- ne **pas allouer de mémoire** dans la fonction (sauf si c'est explicite)
- **3e paramètre** pour stocker pour le résultat

Exemple (amélioration)

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

bool AddVect(float* v1, float* v2, float* res, int card) {
    if (v1 == NULL || v2 == NULL || res == NULL))  {
        fprintf(stderr, "AddVect: Null argument\n");
        return false;
    }
    for (int k = 0; k < card; k++)  res[k] = v1[k] + v2[k];
    return true;
}
```

```c
void foo() {
    float a[3] = {1., 2., 3.};
    float b[3] = {4., 5., 6.};
    float c[3];
    float* d = NewVect(3);

    AddVect(a, b, c, 3);     // c = tableau
    AddVect(c, a, d, 3);     // d = pointeur
    ......
    FreeVect(d);
}
```

Compléments

Valgrind, memalloc, etc.

- testent la mémoire à **l'exécution**
- détectent :
  - la **mémoire non allouée** et les **débordements** mémoire
  - les **fuites mémoire**

Compléments

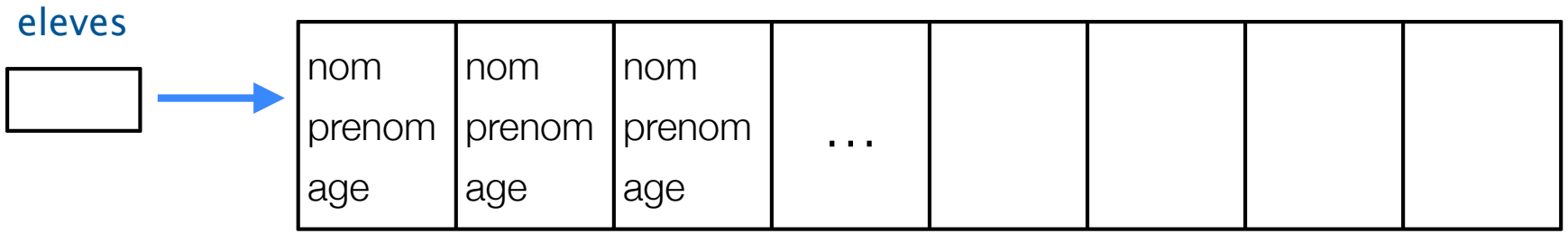
Performances

malloc() consomme un **temps non négligeable**

- non déterministe => difficile à prévoir
- même chose pour **new** en **C++** ou le **ramasse miettes** en **Java**
- parfois c'est ce qui **prend le plus de temps** dans un programme !

=> n'utiliser **malloc** que lorsque c'est utile !

Structures et pointeurs



(pseudo)tableau de structures

```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;

void foo() {
        Eleve * eleves = NULL;
        int elevesNbr = 0;

        eleves = lireEleves(&elevesNbr);
        afficherEleves(eleves, elevesNbr);
}
```

Structures et pointeurs

```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;

void foo() {
        Eleve * eleves = NULL;
        int elevesNbr = 0;

        eleves = lireEleves(&elevesNbr);
        afficherEleves(eleves, elevesNbr);
}
```

```
Eleve* lireEleves(int * elevesNbr) {
        printf("Entrer le nombre d'élèves : ");
        scanf("%d", elevesNbr);
        eleves = calloc(*elevesNbr, sizeof(Eleve));
        ......
        return eleves;
}
```

Structures et pointeurs

```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;

void foo() {
        Eleve * eleves = NULL;
        int elevesNbr = 0;

        eleves = lireEleves(&elevesNbr);
        afficherEleves(eleves, elevesNbr);
}
```

la notation: **p->a** équivaut à: **(*p).a**

```
Eleve* lireEleves(int * elevesNbr) {
        printf("Entrer le nombre d'élèves : ");
        scanf("%d", elevesNbr);
        eleves = calloc(*elevesNbr, sizeof(Eleve));
        ……
        return eleves;
}

void afficherEleves(Eleve* eleves, int elevesNbr) {
        if (eleves == NULL || elevesNbr <= 0) return;
        Eleve* p = NULL;
        for (p = eleves; p < eleves+elevesNbr; ++p) {
                printf("Eleve : %s %s\n", p->nom, p->prenom);
                printf("age : %d\n\n», p->age);
        }
}
```

Remarque

```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;
```

nom et *prenom* sont des **pointeurs**

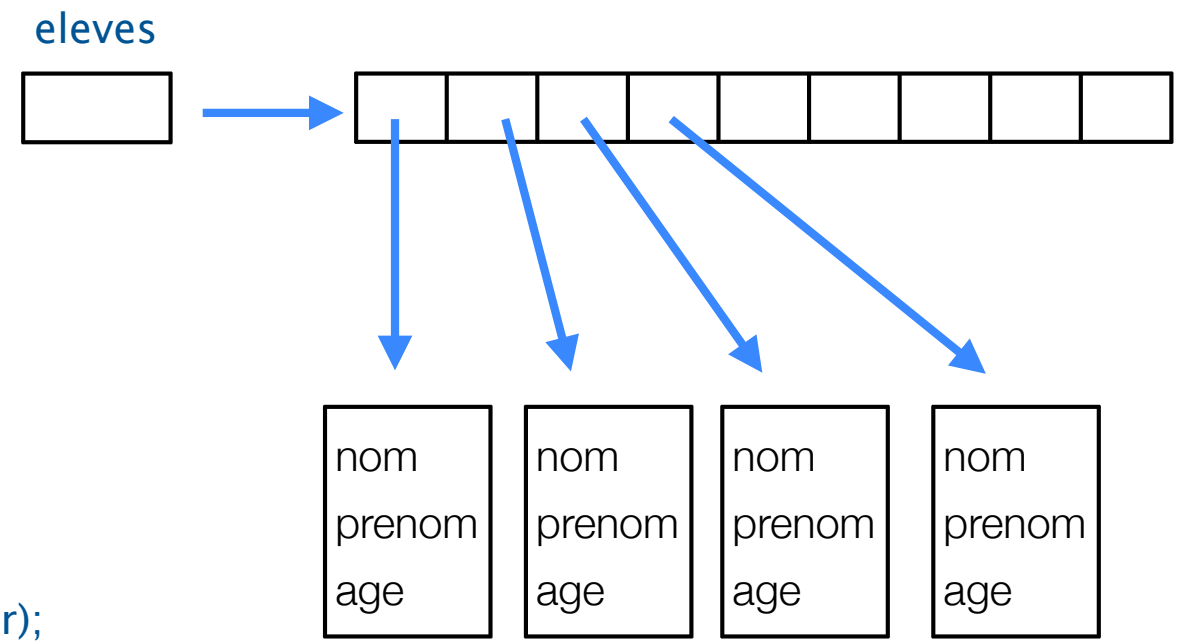
=> **allouer la mémoire** dynamiquement avec **malloc**() ou **strdup**()

```
char buffer[500];
if (fgets(buffer, sizeof(buffer), stdin) != NULL ) {
    p->nom = strdup(buffer);
}
```

char * strdup(const char * *str*) : fait un **malloc** et une **copie** (cf. manuel et « string.h »)

Tableaux de pointeurs

```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;

void foo() {
        Eleve ** eleves = NULL;
        int elevesNbr = 0;

        eleves = lireEleves(&elevesNbr);
        afficherEleves(eleves, elevesNbr);
}
```

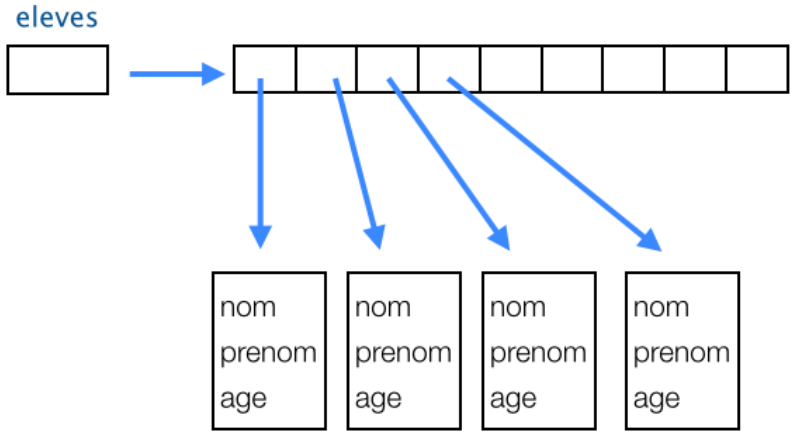


(pseudo)tableau de structures

Structures et pointeurs

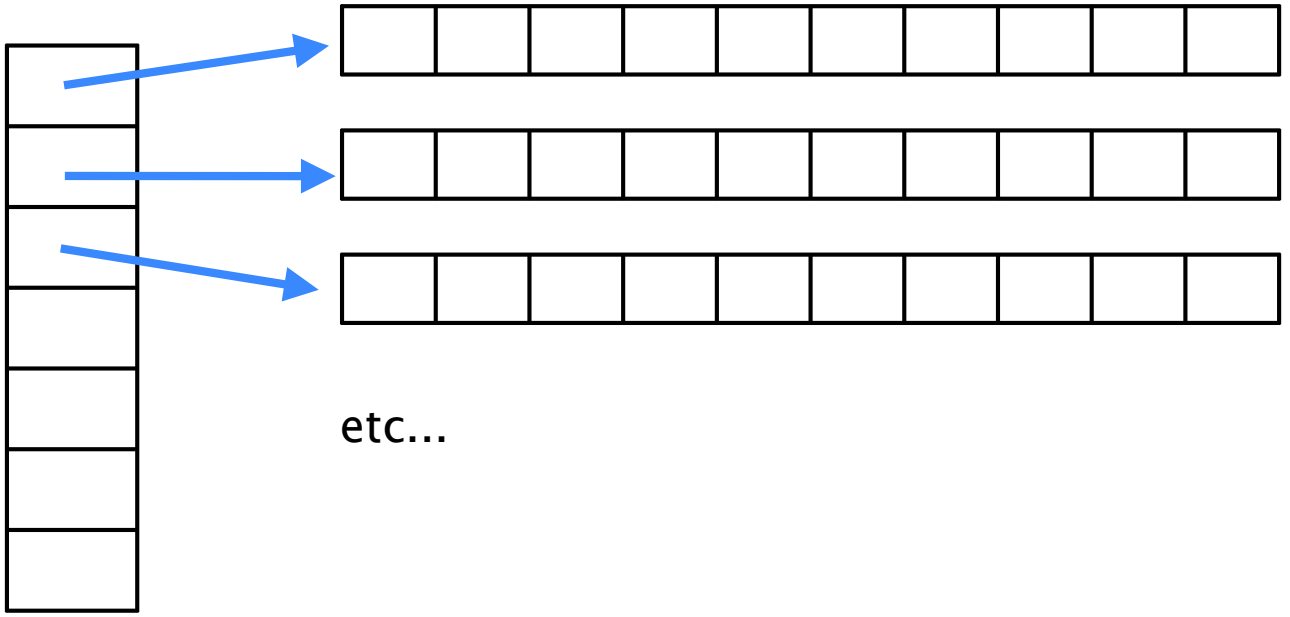
```
typedef struct {
        char * nom;
        char * prenom;
        int age;
} Eleve;

void foo() {
        Eleve ** eleves = NULL;
        int elevesNbr = 0;

        eleves = lireEleves(&elevesNbr);
        afficherEleves(eleves, elevesNbr);
}
```

```
Eleve ** lireEleves(int * elevesNbr) {
        printf("Entrer le nombre d'élèves : ");
        scanf("%d", elevesNbr);
        Eleve** eleves = calloc(*elevesNbr, sizeof(Eleve *));
        for (Eleve** p = eleves; p < eleves+elevesNbr; ++p) {
                *p = calloc(1, sizeof(Eleve));
        }
        ......
        return eleves;
}
```



Tableaux de tableaux

```
char * jour[ ] = {
    "Lundi",
    "Mardi",
    "Mercredi",
    "Jeudi",
    "Vendredi",
    "Samedi",
    "Dimanche",
};
```



c'est une manière de créer des **tableaux bi-dimensionnels**

Note : les lignes peuvent avoir des **longueurs différentes**

Variables

Variables locales

Variables locales (et paramètres)

- accessibles uniquement dans **cette** **fonction**
- dans la **pile** : **crées/détruites** quand on **entre/sort** de la fonction

```
int ChercheVal(float tab[ ], int tabCount, float val) {        // paramètres formels
    int k = 0;                                                 // variable locale
    for (k = 0; k < tabCount; k++) {
        if (tab[k] == val) return k;
    }
    return -1;
}
```

- *tab, taille, val* : **paramètres formels** (cas particulier de variables locales)
- *k* : **variable locale** («automatique» en jargon C)
    - valeur initiale **indéfinie** => **toujours initialiser les variables locales !**

Variables globales

```c
#include <stdio.h>

float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};          // variable globale
int tabCount = 10;                                                      // variable globale

int ChercheVal(float val) {
    int k = 0;
    for (k = 0; k < tabCount; k++) {if (donnees[k] == val) return k;}
    return -1;
}

int main() {
    int indice = ChercheVal(4.);
    if (indice >= 0) printf("indice = %d , valeur = %f \n", indice, donnees[indice] );
    return 0;
}
```

Variables globales

- déclarées en dehors des fonctions,
- existent **pendant toute la durée du programme**, initialisées à 0
- **DANGEREUSES car accessibles partout !**

Variables globales

Les variables globales sont dangereuses et doivent être proscrites

- comment comprendre et vérifier **qui modifie quoi ?**
- empêchent la **réutilisation du code** (tout dépend de tout)
- entraînent des **collisions de noms** (même nom dans plusieurs fichiers)

Dans les rares cas où elles sont utiles :

1) Les **déclarer** dans les **headers** :

```
extern float donnees[ ];      // déclare l'existence d'une variable globale
extern int taille;            // extern est indispensable !
```

2) Les **définir** dans **un seul** fichier **.c** !

Variables statiques

```c
#include <stdio.h>

static float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};      // variable statique
static int tabCount = 10;                                                  // variable statique

int ChercheVal(float val) {
    int k = 0;
    for (k = 0; k < tabCount; k++) {if (donnees[k] == val) return k;}
    return -1;
}

int main() {
    int indice = ChercheVal(4.);
   if (indice >= 0) printf("indice = %d , valeur = %f \n", indice, donnees[indice] );
    return 0;
}
```

Variables statiques de fichier

- similaires aux **variables globales**
- mais **propres à un fichier** : **inconnues** des autres fichiers

Variables statiques dans une fonction

```
void Affiche(float tab[ ], int taille) {
    static int n_ieme_fois = 1;                    // variable statique
    int k = 0;

    printf("J'affiche pour la %d e fois\n", n_ieme_fois++);

    for (k = 0; k < taille; k++) {
        printf(" tab[%d] = %f \n", k, tab[k]);
    }
}
```

Variables statiques de fonction

- similaires aux précédentes
- mais **propres à une fonction** (inconnues des autres)
- servent à **conserver une valeur** entre les appels fonctionnels

Variables: compléments

Structure de bloc : code entre **{** et **}**

- pour restreindre la **portée** des variables
- les blocs peuvent être **imbriqués**

```
{
    int i = 0, taille = 100;

    if (i > taille) return -1;
    else {
        int k = 0;
        for (k = 0; k < taille; k++)
    }
}
```

register, volatile

- **register :** pour optimiser l'exécution (inutile avec compilateurs actuels)
- **volatile :** pour éviter les optimisations (threads, entrées-sorties...)

Récursivité

Les **variables locales** et les **paramètres** sont stockés dans la **pile**

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

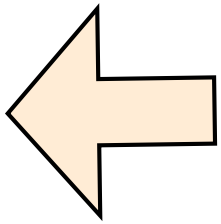
```
int fact(int n) {
   return (n <= 1) ? 1 : n * fact(n - 1);
}
```

```
void printd(int n) {
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    if (n / 10) printd(n / 10);
    putchar(n % 10 + '0');
}
```

Que fait printd ?

Parcours d'arbre

```c
#include <stdio.h>

void prefix(Node* n) {
    printf(" %s ", n->val);
    if (n->left) prefix(n->left);
    if (n->right) prefix(n->right);
}

void infix(Node* n) {
    if (n->left) infix(n->left);
    printf(" %s ", n->val);
    if (n->right) infix(n->right);
}

void postfix(Node* n) {
    if (n->left) postfix(n->left);
    if (n->right) postfix(n->right);
    printf(" %s ", n->val);
}
```



```c
typedef struct NODE {
    struct NODE *left;
    char* val;
    struct NODE* right;
} Node;
```

Self-référence à NODE !

Parcours d'arbre

```
int main() {
    Node a = {NULL, "a", NULL};       // feuilles
    Node b = {NULL, "b", NULL};
    Node c = {NULL, "c", NULL};
    Node d = {NULL, "d", NULL};
    Node plus = {&a, "+", &b};        // noeuds intermediaires
    Node div = {&c, "/", &d};
    Node star = {&plus, "*", &div};   // racine

    prefix(&star);
    printf("\n");
    infix(&star);
    printf("\n");
    postfix(&star);
    printf("\n");
    return 0;
}
```

Résultat ?

Compléments sur les macros :
Directives de compilation

fichier graphics.h

```
#ifndef graphics_h
#define graphics_h

#if defined(__APPLE__)
#  include <OpenGL/gl.h>
#  include <OpenGL/glu.h>
#elif defined(_WIN32)
#  include <windows.h>
#else
#  include <GL/gl.h>
#  include <GL/glu.h>
#endif
```

Variantes selon l'OS

fichier person.h

```
#ifndef person_h
#define person_h

#include <string>
#include <iostream>

typedef struct {
   char* name;
   int id;
} Person;

Person* createPerson(const char* name);
void setPersonId(Person*, int id);

#endif
```

Protection contre les inclusions multiples :

```
#include "person.h"
#include "person.h"
```

ne compilerait pas sans le #ifndef

Compléments sur les macros :
Macros paramétrées

Macros paramétrées

```
#define MAX(a,b)  ((a) > (b) ? (a) : (b))

float x = 55., y = 77.;
float res = MAX(x, y);      aka:  res = ((x) > (y) ? (x) : (y)) ;
```

Penser à mettre des ( )

```
#define MAX(a,b)  ((a) > (b) ? (a) : (b))

float res = MAX(x-1, y * 3);      // OK
float res = MAX(x < y,  10);      // ne compile pas sans les ()
```

Attention aux effets de bord !

```
float res = MAX(x++,  y++);     // compile mais faux !
```

Compléments sur les pointeurs :
Pointeurs non typés et casts

void * p : pointeur **non typé**

Afffectation entre pointeurs de **types différents** possible via des **casts**

```
int i;                  // un int occupe 4 octets

char* pc;               // un char occupe 1 octet

pc = (char*) &i;        // pc pointe sur le 1er octet de i

pc = &i;                ERREUR  types non compatibles !
```

Compléments sur les pointeurs :
Pointeurs non typés

Afffectation entre pointeurs de **types différents** possible via des **casts**

```
int i;                  // un int occupe 4 octets

char* pc;               // un char occupe 1 octet

pc = (char*) &i;        // pc pointe sur le 1er octet de i

pc = &i;                ERREUR  types non compatibles !
```

Attention

- les tailles **dépendent de la machine** (ou des compilateurs ou de leurs options) !
- les **int, float**, **double**, les **pointeurs** doivent commencer à **certaines adresses** :

  *ex : tous les **4 octets** pour **float**, **8 octets** pour pointeurs*

Manipulation de bits

Opérateurs

&	ET
|	OU inclusif
^	OU exclusif
<<	décalage à gauche
>>	décalage à droite
~	complément à un

```
int n = 0xff, m = 0;
m = n & 0x10;
m = n << 2;          // équivalent à ?
```

Attention

- ne pas confondre **&** avec **&&** (**et** logique), ni **|** avec **||** (**ou** logique)
- mettre des **parenthèses** dans les expressions composées (faible priorité)

Entrées/sorties

Fin de fichier

```
int feof(FILE * fichier)
```

Synchronisation

```
int fflush(FILE * fichier)
```

Déplacement dans un fichier

```
int fseek(FILE *, long offset, int from);    // from = SEEK_SET, SEEK_CUR, SEEK_END
long ftell(FILE *);
```

Ouvrir/fermer un fichier

Type : FILE

flux de données : entrées/sorties bufferisées

Ouvrir un fichier

FILE * **fopen**(**const char** * chemin, **const char** * **mode**)

Fermer un fichier

int **fclose**(**FILE** * fichier)

mode	lecture	écriture	crée le fichier	vide le fichier	position du flux
r	X				début
r+	X	X			début
w		X	X	X	début
w+	X	X	X	X	début
a		X	X		fin
a+	X	X	X		fin

Exemple

```
bool readFiles(const char * inputFileName, const char * outputFileName) {
        FILE * in = fopen(inputFileName, "r");          // r = read/lecture
        if (in == NULL) {
                fprintf(stderr, "Can't open input file %!s\n", inputFileName);
                return false;
        }

        FILE * out = fopen(outputFileName, "w");     // w = write/écriture
        if (out == NULL) {
                fprintf(stderr, "Can't open output file %s!\n", outputFilename);
                return false;
        }

        while ( ! feof(in) ) {
                ... etc ...
        }

        fclose(in);          // ne pas oublier fclose() !
        fclose(out);
        return true;
}
```

Entrées/sorties

```
char* fgets(const char* buffer, int buffer_size, FILE* stream)
int fputs(const char* buffer, FILE* stream)
```

- **fgets**() renvoie le buffer ou **NULL** en cas d'erreur ou en fin de fichier

```
int fscanf(FILE* stream, const char* format, ...)
int fprintf(FILE* stream, const char* format, …)
```

- **fscanf / fprintf** renvoient le **nombre d'éléments** lus ou imprimés

- **plus rapides mais pas portables :**

```
size_t  fread(void* ptr, size_t size, size_t count, FILE* stream)
size_t  fwrite(const void* ptr, size_t size, size_t count, FILE* stream)
```

Outils de développement

Débogueur

- exécution pas à pas
- trouver où plante un programme et examiner la mémoire

Valgrind, memalloc, etc.

- teste la mémoire à l'exécution, détecte :
  - la mémoire non allouée et les débordements mémoire
  - les fuites mémoire

Analyseur de performance (profiler)

- détecter quelles fonctions consomment beaucoup de temps CPU
- **on ne fait pas les optimisations à l'aveugle !**
  - ca ne sert généralement à rien et ca peut même être **contre-performant**

Pseudo-objet en C

C

```c
typedef struct {
      char* name;
      long id;
} User;

User* newUser(const char* name, int id);
void  deleteUser(User*);
void  setUserName(User*, const char* name);
void  printUser(const User*);
....

void foo() {
     User * u = newUser("Dupont");
     setUserName(u, "Durand");
     .....
     deleteUser(u);
     u = NULL;
}
```

C++

```c++
class User {
      char* name;     // il faudrait utiliser string
      long id;
public:
      User(const char* name, int id);
      virtual ~User();
      virtual void setName(const char* name);
      virtual void print() const;
      ....
};

void foo() {
     User * u = new User("Dupont");
     u->setName("Durand");
     ....
     delete u;
     u = nullptr;
}
```

Pseudo-héritage en C

```c
typedef struct User {
  int a;
  void (*print) (const struct User*);
} User;

typedef struct Player { // subclass
  User base;
  int b;
} Player;

void print(const User* u) {
  (u->print)(u);
}

void printUser(const User *u) {
  printf("printUser a=%d \n", u->a);
}

void printPlayer(const Player *u) {
  printf("printPlayer a=%d b=%d\n",
         u->base.a, u->b);
}
```

```c
User* newUser() {
  User* p = (User*) malloc(sizeof(User));
  p->a = 0;
  p->print = printUser;
  return p;
}

Player* newPlayer() {
  Player* p = (Player*) malloc(sizeof(Player));
  p->base.a = 0;
  p->base.print = printPlayer;  //cast nécessaire
  p->b = 0;
  return p;
}

int main() {
  Player* p = newPlayer();
  p->base.a = 1;
  p->b = 2;
  print(p);
}
```