

Langage C

Eric Lecolinet

Télécom ParisTech

Septembre 2018

Brève historique

Début 70 : Langage C

Début 80 : C++

Extension du **C** par *Bjarne Stroustrup* aux **Bell Labs**
C++11 : révision majeure en 2011/2014

Début 80 : Objective C

Autre extension du **C** de **NeXt** puis **Apple**
Puissant, syntaxe inhabituelle inspirée de **Smalltalk**

Fin 80... : Python

Vise la simplicité/rapidité d'écriture, créé par *G. van Rossum*
Interprété, typage dynamique

Mi-90 : Java

Simplification du **C++** de **Sun Microsystems** puis **Oracle**
Egalement inspiré de **Smalltalk**, **ADA** ...

Début 2000: C#

A l'origine, le « Java de **Microsoft** »
Egalement inspiré de **Delphi**, **C++** ...

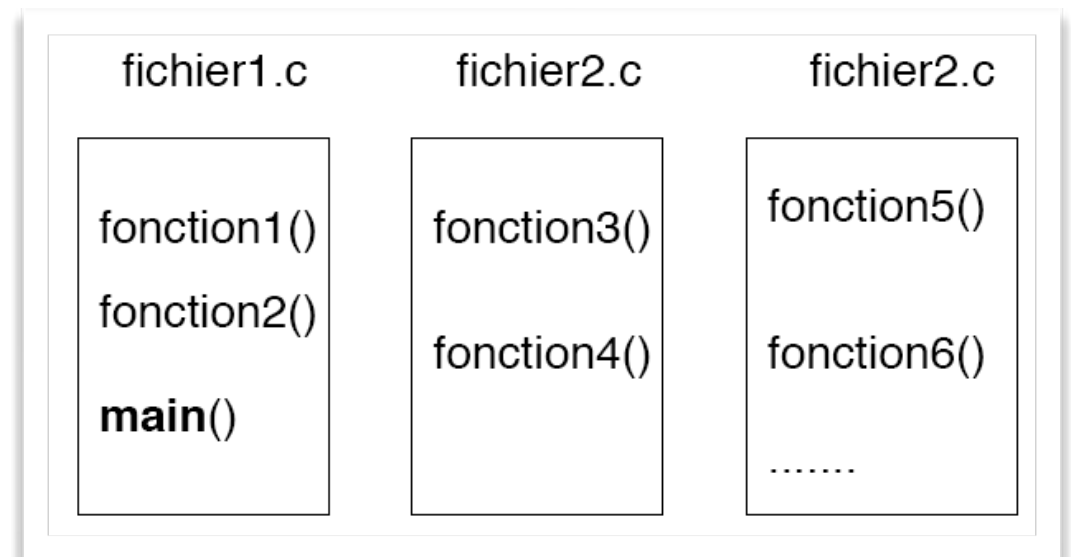
Mi-2000: Swift

Le successeur d'**Objective C**, par **Apple**

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**



Aperçu d'un programme C

Exemple : un programme composé de 2 fichiers

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

```
void foo(const char* name) {  
    printf("hello %s \n", name);  
}
```

```
int main() {  
    const char* name = "hubert";  
    foo(name);  
    dupondt();  
}
```

// dans un autre fichier

```
void dupondt() {  
    foo("dupondt");  
}
```

Problème clé : assurer la correspondance entre les appels et les définitions des fonctions

fichier1.c

```
fonction1()  
fonction2()  
main()
```

fichier2.c

```
fonction3()  
fonction4()
```

Types de base

Types de base

char : caractère ou « petit nombre » (1 octet)

int : entier (short <= int <= long)

short : entier court (>= 16 bits)

long : entier long (>= 32 bits)

bool : booléen (depuis C99, inclure stdbool.h)

float : flottant simple précision

double : flottant double précision

long double : encore plus précis !

Attention: les tailles varient selon la **plate-forme** !

— cf. **limits.h** et **float.h** headers standards (dans: `/usr/include` sous Linux)

— exemple : short = 16 / int = long = 32 / float = 32 / double = 64 bits

Types et signe

signed int

unsigned long

Les types sont **signés** par défaut **sauf char** (ça dépend de la plateforme !)

unsigned char : de 0 à 255

signed char : de -128 à 127

char : signé ou pas signé => **source d'erreurs** si on change de plateforme

Variables

Définition des variables

`int i, res, truc;` trois entiers dont la **valeur est indéfinie !!!**

TOUJOURS initialiser les variables !!!

`int i = 0, res = 0, truc = 0;`

`char * p = NULL;`

Remarques

- ne **pas** confondre `=` (affectation) avec `==` (égalité)
- avant **C99** : variables déclarées **avant** les instructions (!= Java, C++...)

Constantes

Types des constantes

1234 037 (octal) 0x1f (hexadécimal)

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

'a' (caractère) => type **int**

"abcd" (littéral) => type **char ***

Caractères spéciaux:

- '\0' = le caractère nul; valeur = 0
- '\n' = new line, '\t' = tabulation, etc.
- '\xxx' = caractère défini en octal
ex : '\007' = caractère BELL

Caractères: 'a'

- sont des **int** (et non des **char** !)
- en ASCII: 'A' = 65, 'O' (la lettre) = 78, '0' (le chiffre) = 48, '\0' (NUL char) = 0

Littéraux: "abcd"

```
char* message = "hello word";
```

- chaînes de caractères **constantes** (ne **pas** modifier leur contenu !)
- un '\0' (valeur = 0) est **automatiquement** rajouté à la fin par le **compilateur**

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		DEL

Source: www.LookupTables.com

Variables constantes, énumérations

Variables à valeur constante

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

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 par défaut
- moyen simple et efficace de spécifier des **constantes** entières

Macros

```
#define TAILLE 1024
#define BELL '\007'
#define MESSAGE "hello word"
```

- **substitution textuelle** par le **préprocesseur C avant** la compilation
- les **macros** permettent de paramétrer le programme
- **éviter les macros si possible** : préférer les **énumérations** ou **variables const**

Macros paramétrées

```
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
int i = 1, j = 2;
int k = MIN(i, j);
```

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

Initialisation et taille implicite

`int vect[10] = {1, 2, 3};` tableau de 10 entiers
`int vect[] = {1, 2, 3};` tableau de **3 entiers (taille implicite)**

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

- indices allant de **0** à **taille-1**
- éléments **contigus** (placés en séquence)
- la taille **la plus à gauche** peut être implicite

Tableaux

Paramétrer la taille des tableaux

```
#define TAILLE 1024
```

```
char tableau[TAILLE]; // paramétrer : ne pas écrire char tableau[1024] !!!
```

- en **C99** les **initialiseurs** peuvent être des variables sous certaines conditions

Structures

```
#define NAME_SIZE 25
```

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

```
struct PERSON {  
    char firstname[NAME_SIZE], lastname[NAME_SIZE];  
    unsigned int age;  
};
```

- agrégat de données
- ancêtres des classes (classes sans méthodes)

Structures

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

```
void foo() {
```

```
    struct POINT p1, tab[10];
```

```
    struct POINT p2 = {4.1, 7.3};
```

```
    p1 = p2;
```

```
    p1.x = 1.;
```

```
    tab[7] = p2;
```

```
    tab[7].x = p2.x;
```

```
}
```

// le mot clé **struct** est nécessaire

// **initialisation** similaire aux tableaux

// **= copie** les structures **champ par champ**

// **. accède** à un **membre** de la structure

Typedef

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

```
typedef struct POINT Point;
```

```
void foo() {  
    // plus besoin de répéter struct  
    Point p1, tab[10], p2 = {4.1, 7.3};  
    .....  
}
```

typedef définit un nouveau **type** « **Point** »

Forme contractée

```
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 p1, p2;    // Point est imbriquée dans Rect  
} 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 = (%d, %d) x (%d, %d) \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 précedence des opérateurs !

$m = (i + j) * k;$

/ est la **division entière ou réelle** suivant le cas :

!!! attention : $2 / 4$ vaut 0 mais $2. / 4$ vaut 0.5 **!!!**

% = reste de la division entière

Transtypage implicite

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

Attention : les conversions **implicites** peuvent être **dangereuses** !

```
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**

Attention : les conversions implicites peuvent être **dangereuses !**

```
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
```

Transtypage explicite

L'opérateur de **cast** fait une conversion de type **explicite**

!!! Attention : ne convertit pas les données ! (seulement le type)

(Mauvais) exemple : transformer une chaîne en nombre (et inversement)

```
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 !

- un seul `++` ou `--` par ligne!

Opérateurs logiques et relationnels

Ordre de priorité

! (négation unaire)

> >= < <=

== !=

&& (ET logique)

|| (OU logique)

Ces opérateurs sont

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

$x < \text{lim}-1$ équivaut à : $x < (\text{lim}-1)$

— évalués **de la gauche vers la droite avec arrêt** dès que la valeur de vérité est trouvée:

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

```
if (i <= 3 && (c = getchar()) != 'y')) ... //NB: c doit etre un int (pas un char)
```

Booléens

Depuis C99 :

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

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

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

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: m = n * 4
```

Attention: ne pas confondre :

— & avec && (et logique)

— | avec || (ou logique)

— **parenthèses nécessaires** dans expressions composées (cf. priorités)

Priorité des opérateurs

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(<i>type</i>){ <i>list</i> }	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(<i>type</i>)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof _Alignof	Size-of Alignment requirement(C11)	
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

Tests et boucles

Expressions conditionnelles

- ne pas oublier les `()` autour du *test*
- `;` à la fin des deux instructions
- **else** optionnel

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

```
if (k < tabsize && tab[k] != 0)  
    doit(tab[k]);
```

Rappel : arrêt de l'évaluation dès que :

- condition fausse pour `&&`
- condition vraie pour `||`

```
if (test)  
    action1;
```

```
if (test)  
    action1;  
else  
    action2;
```

```
if (test) action1;  
else action2;
```

Actions imbriquées

```
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;
```

- le **else** se rapporte au **if** le plus proche
- mettre des **{ }** pour éviter les ambiguïtés
- **!!! indenter le code !!!**
=> utiliser un **éditeur de texte approprié**

else if cascadés

```
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
- alternative: le **switch**

switch

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

Remarques

- const1, const2 doivent être des **constantes** à valeur entière
- **break** ou **return** pour sortir du **switch** (sinon on continue jusqu'à la fin)

```
switch (expression) {
case const1:
    action1a;
    action1b;
    action1c;
    break;
case const2:
case const3:
    action2;
    break;
default:
    action3;
    break;
}
```

Opérateur ?

Principe

L'expression: `test ? expr1 : expr2`

renvoie :

- `expr1` si `test` est vrai (cad. `!= 0`)
- `expr2` sinon

Exemple

```
z = a > b ? a : b ;
```

équivalent à :

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

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

```
float x = 55., y = 77.;
```

```
.....
```

```
float res = MAX(x, y);
```

```
// penser à mettre des ( )
```

```
// attention aux effets de bord !
```

Boucles while et for

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

Equivaut à :

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

- **init**, **test** et **incr** sont des expressions quelconques
- continue tant que **test** != 0
- **test** effectué **avant** d'entrer dans la boucle
- préférer **for** à **while** : évite d'oublier des clauses !

Exemples

// Toujours définir les tailles par des CONSTANTES

```
#define TAB_SIZE 20
```

```
int tab[TAB_SIZE];
```

```
int i = 0;
```

// Les tableaux commencent à l'indice 0 => balayer tab de 0 à TAB_SIZE-1

```
for (i = 0; i < TAB_SIZE; i++) {
```

```
    tab[i] = 0;
```

```
}
```

// formes équivalentes :

```
i = 0;
```

```
while (i < TAB_SIZE) tab[i++] = 0;
```

```
i = -1;
```

```
while (++i < TAB_SIZE) tab[i] = 0;
```

Divers

Boucle do while

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

- `actions` exécutées au moins 1 fois
- continue tant que `test != 0`

Break et continue

- `break` permet de sortir d'une boucle ou d'un `switch`
- `continue` passe à l'itération suivante

Goto et Labels

- **à éviter** sauf cas particuliers

Exemples

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;  
}
```

Exemples

Un cas où **goto** peut être 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;
}
```

Interlude

```
#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

```
#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
```

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**

Définition de fonction

```
type nom_fonction(paramètres) {  
    déclarations;  
  
    instructions;  
    return type;  
}
```

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;  
}
```

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;
}

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

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

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

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** des fonctions sont **recopiées** dans leurs **paramètres**
- y compris les **struct** (entièrement 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(donnees, 10, 4.);  
    .....  
}
```

Passage des arguments

Passage par valeur

- les valeurs passées en **argument** des fonctions sont **recopiées** dans leurs **paramètres**
- **!!! mais pas les tableaux** : c'est leur **adresse** qui est copiée !!!

```
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.);
    .....
}
```

Attention :

- *tab* ne « connaît » pas son nombre d'éléments, ***tabCount* est indispensable**
- *tab* est en fait un **pointeur**

Passage des arguments

Question : quelles sont les valeurs affichées ?

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

Doxygen (génération de la documentation)

Système de documentation automatique

- similaire à **JavaDoc**, mais plus général (supporte langages variés)

```
/// retourne la largeur.  
unsigned int getWidth();
```

```
unsigned int getHeight();  
///< retourne la hauteur.
```

```
/// change la position.  
/** voir aussi setX() et setY().  
*/  
void setPos(int x, int y);
```

```
void setX(int);  
/**< @brief change l'abscisse.  
* voir aussi setY() et setPos().  
*/
```

doxygen, doxywizard

cf. : www.doxygen.org/

et : <http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html>

Style et commentaires

```
/// retourne la largeur.  
unsigned int getWidth();
```

```
unsigned int getHeight();  
///< retourne la hauteur.
```

```
/// change la position.  
/** voir aussi setX() et setY().  
*/  
void setPos(int x, int y);
```

```
void setX(int);  
/**< @brief change l'abscisse.  
* voir aussi setY() et setPos().  
*/
```

Règles

- être **cohérent**
- **indenter** (utiliser un IDE qui le fait automatiquement : **TAB** ou **Ctrl-I** en général)
- **aérer**, éviter plus de 80 colonnes
- **commenter** quand c'est utile
- **camelCase**

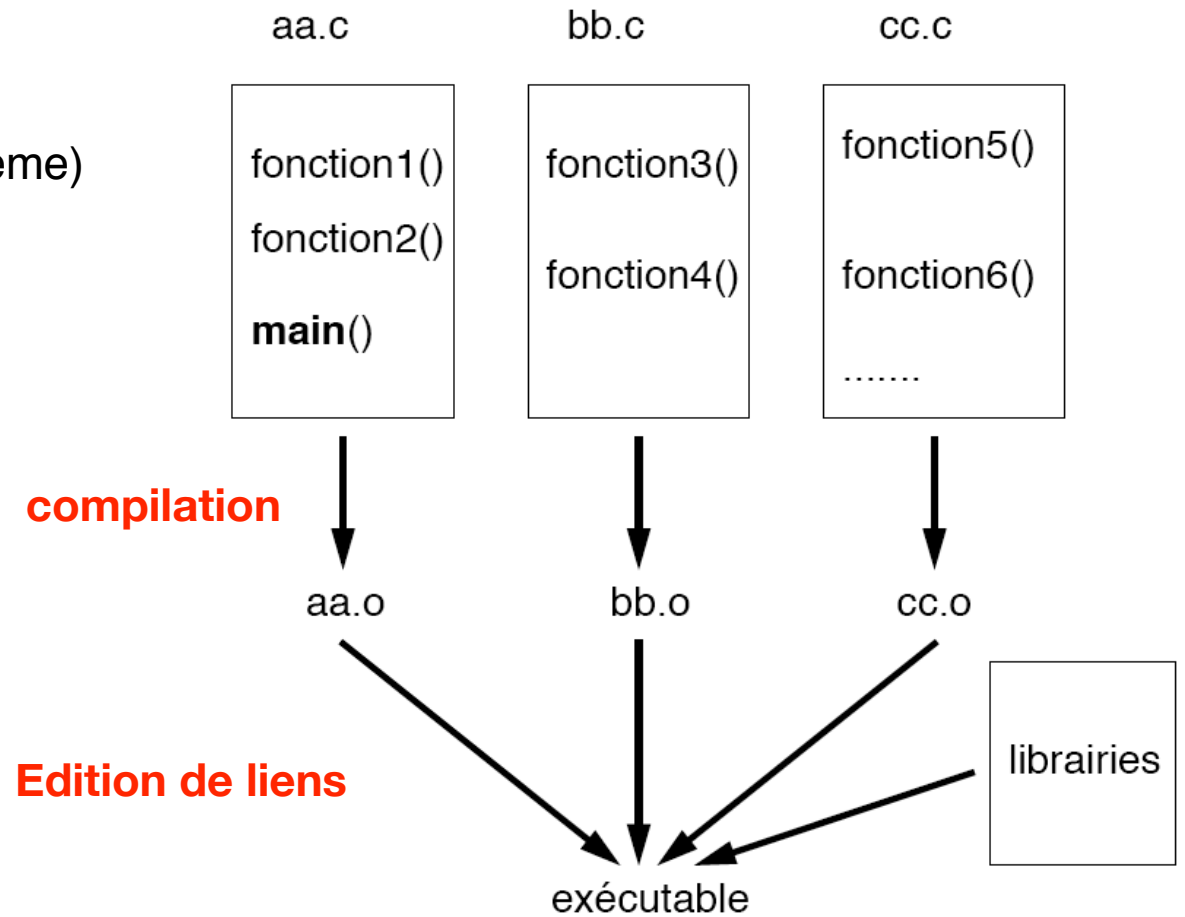
Compilation séparée

Fonctions réparties

- dans plusieurs fichiers
- et les bibliothèques (librairies système)

Librairies standard

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



Bibliothèques statiques et dynamiques

Librairies statiques

- extension .a (sous Unix)
- simples "archives" de fichiers .o
- le code est **inséré dans l'exécutable** à la compilation

Librairies dynamiques (DLLs)

- extension .so, dylib, etc.
- le code n'est **pas inséré** dans l'exécutable
- il est chargé **dynamiquement** à l'exécution

Bibliothèques dynamiques

Avantages des librairies dynamiques

- les programmes prennent **beaucoup** moins de place
- ils sont plus **rapides** (moins de **swapping** car le binaire est partagé)

Inconvénients

- l'utilisateur **doit avoir cette librairie** sur son système
 - => attention aux **licences** (parfois différentes si lib. statique ou dynamique)
 - => attention 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 déboguer
- **-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 bibliothèques dans ce répertoire (*)
- **-l*library*** : chercher les fonctions dans cette bibliothèque

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

Makefile

Makefile = fichier de règles indiquant:

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

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

bibliothèques 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"

Compléments

Le **Makefile** précédent est incomplet :

- pas de règles pour créer les `.o` : c'est fait implicitement
- mais sans tenir compte des `.h` !

Il faut:

- soit mettre les règles à la main
- soit utiliser **makedepend**, configure ou un outil plus évolué

Exemple

=> cf Makefile du TP

Compléments

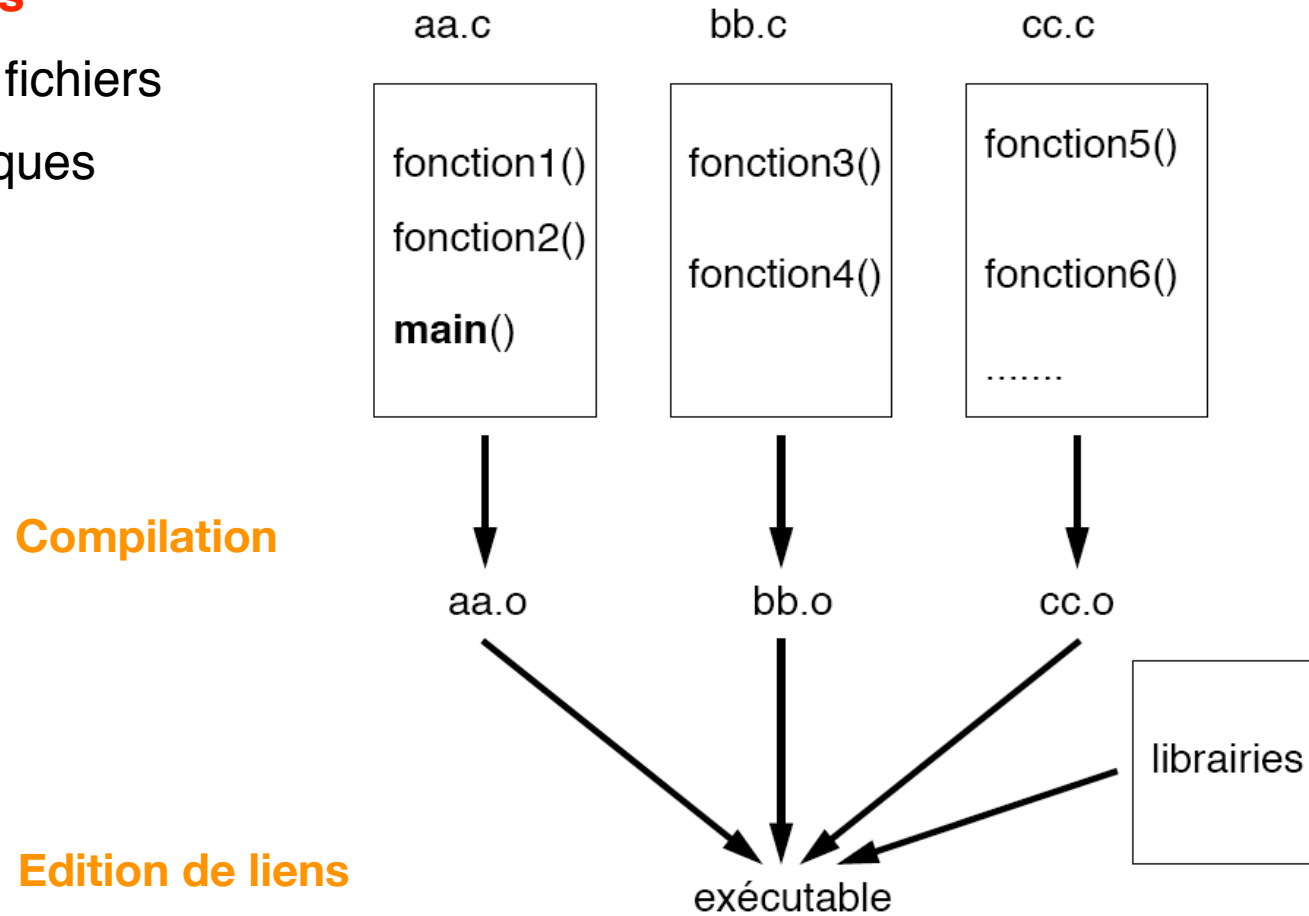
Autres outils utiles

- emacs, xemacs, etc.
- gdb, xgdb, ddd
- grep, nm
- ar
- tar, gtar, gzip, gunzip
- eclipse, kdevelop ...

Retour à la compilation séparée

Fonctions réparties

- dans plusieurs fichiers
- et les bibliothèques



ChercheVal V2 (version **incorrecte**)

```
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 cherche.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;  
}
```

fichier main.c

Quel est le problème ?

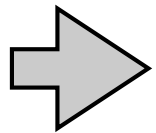
Problème

Les fichiers C sont compilés **indépendamment**

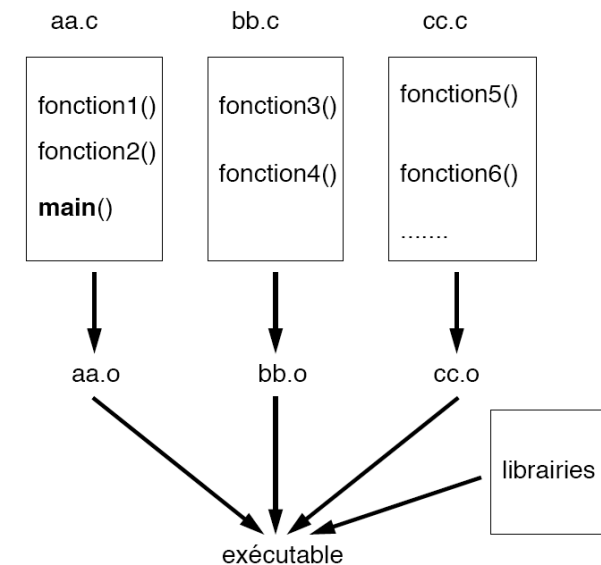
=> **pas de vérification de cohérence** entre la **définition** et l'**appel** !

Quand on appelle une fonction **inconnue**

– le compilateur C suppose qu'elle **existe** et renvoie un **int**



Formidable source d'erreurs !



Compilation indépendante des .c

Problème

Les fichiers C sont compilés indépendamment

=> pas de vérification de cohérence entre la définition et l'appel !

Quand on appelle une fonction inconnue

— le compilateur C suppose qu'elle existe et renvoie un int

Trois cas possibles

— fonction définie **nulle part** :

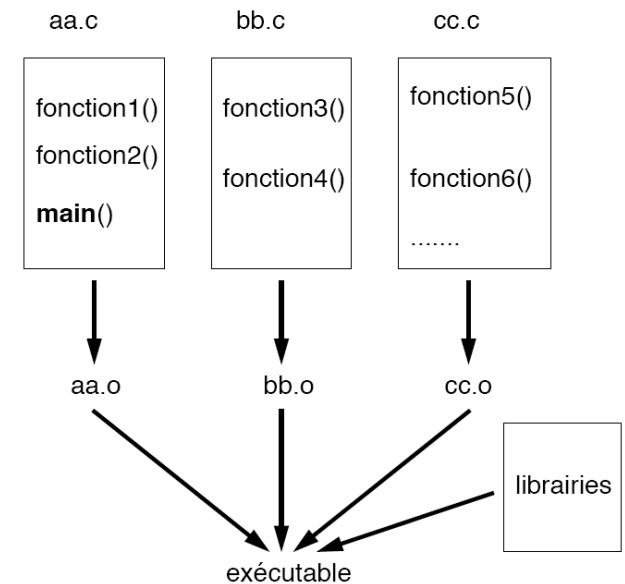
erreur d'édition de lien « Symbol not found »

— fonction définie avec la **même signature** :

exécutable créé et correct

— fonction définie avec une **autre signature** :

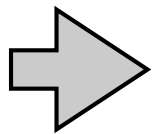
l'exécutable créé mais **comportement indéterminé**



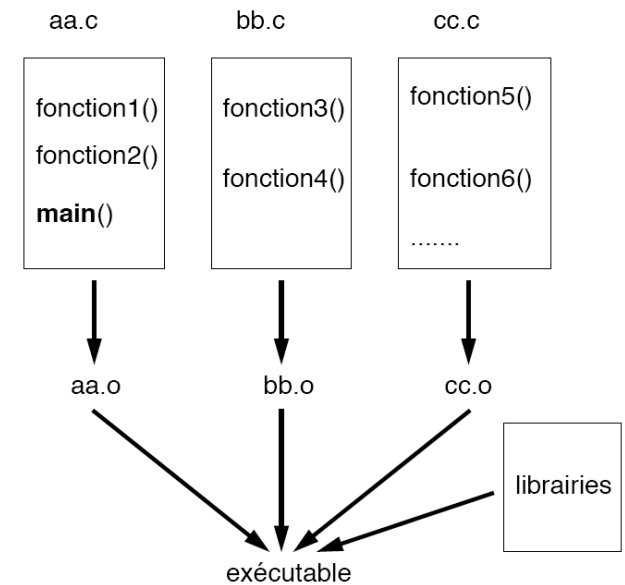
Solution : header partagés

Principe

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



Cohérence vérifiée par transitivité !



ChercheVal V3 (version correcte)

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

fichier cherche.h

```
#include "cherche.h"
```

fichier cherche.c

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

```
#include <stdio.h>  
#include "cherche.h"
```

fichier main.c

```
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;  
}
```

Remarques

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 <stdio.h> // cherche dans le même répertoire
```

```
#include "cherche.h" // cherche dans répertoires standards (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

```
#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 standards des bibliothèques

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)

Variables locales

Variables locales (et paramètres)

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

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

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

Variables globales

```
#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, **accessibles dans tout le programme**
- existent pendant toute la durée du programme
- initialisées à 0

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 de variable 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

```
#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

- similaires aux **variables globales**
- sauf qu'elles sont **propres à un fichier** (et inconnues dans les autres fichiers)
=> beaucoup moins dangereuses que les variables globales

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
- sauf qu'elles sont **propres à une fonction** et inconnues des autres
- servent à **conserver une valeur** entre les appels fonctionnels

Variables: compléments

register, volatile

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

Structure de bloc : code entre { et }

- les blocs peuvent être **imbriqués** (et contenir des déclarations de variables)

```
{  
    int i = 0, taille = 100;  
    ....etc....  
    if (i > taille) return -1;  
    else {  
        int k = 0;  
        for (k = 0; k < taille; k++)  
    }  
}
```

Adresses mémoires et tableaux

Vecteur

type vect[T];



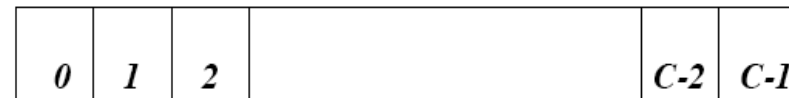
Indice 0 1 2 T-2 T-1

Matrice

type mat[L][C];

Ligne

0



1

L-1

Colonne 0 1 2 C-2 C-1

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 (normalement) **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);
```

Passage des arguments

Rappel: passage d'arguments dans les fonctions

- passage par **valeur** : les arguments sont **recopiés** dans les paramètres
- **mais pas le contenu des tableaux** : c'est leur **adresse** qui est recopiée !!!

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

équivalent à:

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

- **tab** est en réalité un **pointeur** => **taille du tableau inconnue !**

Passage des arguments

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

```
void foo() {  
    int i;  
    float x;  
    scanf("%d %f", &i, &x);  
}
```

- **scanf()** récupère les **adresses** des variables *i* et *j* de **foo()**
- ce qui lui permet de modifier leurs **valeurs** (= ce qu'elles contiennent en mémoire)

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()`

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` ?

Remarques

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
```

```
float val = p[i];     CORRECT
```

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 sur les pointeurs** n'a de sens que s'ils sont **typés** !

Pointeurs non typés (void*)

Type des pointeurs et void *

- **void * p** : pointeur **non typé**
- affectation entre pointeurs de types différents possible via des **casts**
- attention aux **erreurs** !

```
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 */
```

Attention

- les tailles **dépendent de la machine** (et des options de compilation)
- les **int, float...** doivent commencer **à certaines adresses** :
(exemple : tous les 4 octets pour certains processeur 32 bits)

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 n;
    for (n = 0; *s != '\0'; s++) n++;
    return n;
}
```

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

Exemple

Calcul de la longueur d'une chaîne de caractères

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

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

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

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**

Exemple

```
#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

```
#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

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 caractères** (le reste sera lu par le prochain appel à **fgets()**)
- retourne **NULL** en fin de fichier ou en cas d'erreur

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


Autres fonctions d'entrées/sorties

Depuis/dans un fichier

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

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

int **fputs**(**const char*** str, FILE* stream)

- renvoient le nombre d'éléments lus (ou imprimés)
- entrées/sorties console : pseudo fichiers: **stdin**, **stdout**, **stderr**

Depuis/dans une chaîne 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, ...);

voir compléments à la fin du poly ...

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

Utilisation

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

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

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

```
int * tab = (int *) malloc(tabCount * sizeof(int));
```

```
for (int k = 0; k < tabCount; ++k) tab[k] = 0;
```

```
.....
```

```
free(tab); // Attention : free() ne met pas tab à NULL
```

```
tab = NULL; // NULL indique que tab pointe sur rien
```

Mémoire dynamique

Utilisation

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

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

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

```
int * tab = (int *) malloc(tabCount * sizeof(int));
```

```
for (int k = 0; k < tabCount; ++k) tab[k] = 0;
```

```
.....
```

```
free(tab); // Attention : free() ne met pas tab à NULL
```

```
tab = NULL; // NULL indique que tab pointe sur rien
```

void* **calloc**(**size_t** *count*, **size_t** *size*) : **alloue** de la mémoire et la met à 0

void* **realloc**(**void** **ptr*, **size_t** *size*) : **change** la taille de la zone mémoire
appelle **malloc**(), **free**() et **memcpy**()

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

Mémoire dynamique

ATTENTION

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

```
.....  
free(tab);  
tab = NULL;  
..... // par mégarde on fait un 2e free() après d'autres calculs  
free(tab); // pas de problème car tab = NULL (sinon ca planterait !)
```

Mémoire dynamique

Performances

- **malloc()** consomme un **temps non négligeable**
de même que **new** (ou le ramasse miettes) en Java, C#, C++, etc.
- c'est même souvent **ce qui prend le plus de temps** dans un programme !
=> ne pas en faire inutilement !

Exemple (solution « naive »)

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

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

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

```
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

- lesquels ?
- que faut-il faire ?

Améliorations

L'exemple précédent présente plusieurs inconvénients

Les opérations effectuent un `malloc()` :

- 1) **coût non négligeable** (si beaucoup d'opérations)
- 2) **il faut faire des `free()`** pour libérer la mémoire
=> 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 non négligeable** (si beaucoup d'opérations)
- 2) **il faut faire des `free()`** pour libérer la mémoire
=> complique le code et peut occasionner des erreurs
- 3) résultat (via return) **pas compatible avec les tableaux**

Solution

Les opérations :

- ne doivent **pas allouer de mémoire**
- doivent avoir un **3e paramètre** pour stocker pour le résultat

Exemple

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

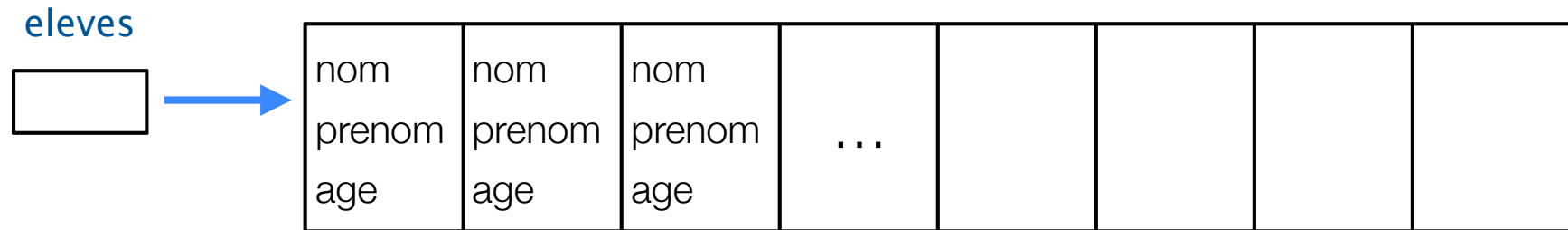
```
void VectError(const char* message) {
    fprintf(stderr, "%s \n", message);
    // à suivre...
}
```

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

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

    AddVect(a, b, x, 3);    // x = tableau
    AddVect(x, a, y, 3);    // y = pointeur
    .....
    free(y);
}
```

Structures et pointeurs



```
typedef struct {
```

```
    char * nom;
```

```
    char * prenom;
```

```
    unsigned int age;
```

```
} Eleve;
```

(pseudo)tableau de structures

```
void foo() {
```

```
    Eleve * eleves = NULL;
```

```
    int elevesNbr = 0;
```

```
    eleves = lireEleves(&elevesNbr);
```

```
    afficherEleves(eleves, elevesNbr);
```

```
}
```

Structures et pointeurs

```
typedef struct {
    char * nom;
    char * prenom;
    unsigned 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 = (Eleve*) calloc(*elevesNbr, sizeof(Eleve));
    .....
    return eleves;
}
```

Structures et pointeurs

```
typedef struct {
    char * nom;
    char * prenom;
    unsigned 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 = (Eleve*) calloc(*elevesNbr, sizeof(Eleve));
    .....
    return eleves;
}

void afficherEleves(Eleve* eleves, unsigned 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;  
    unsigned int age;  
} Eleve;
```

- *nom* et *prenom* sont des **pointeurs**
=> **allouer la mémoire** dynamiquement avec **malloc()** ou **strdup()**

```
char buffer[100];  
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;
```

```
    unsigned int age;
```

```
} Eleve;
```

```
void foo() {
```

```
    Eleve ** eleves = NULL;
```

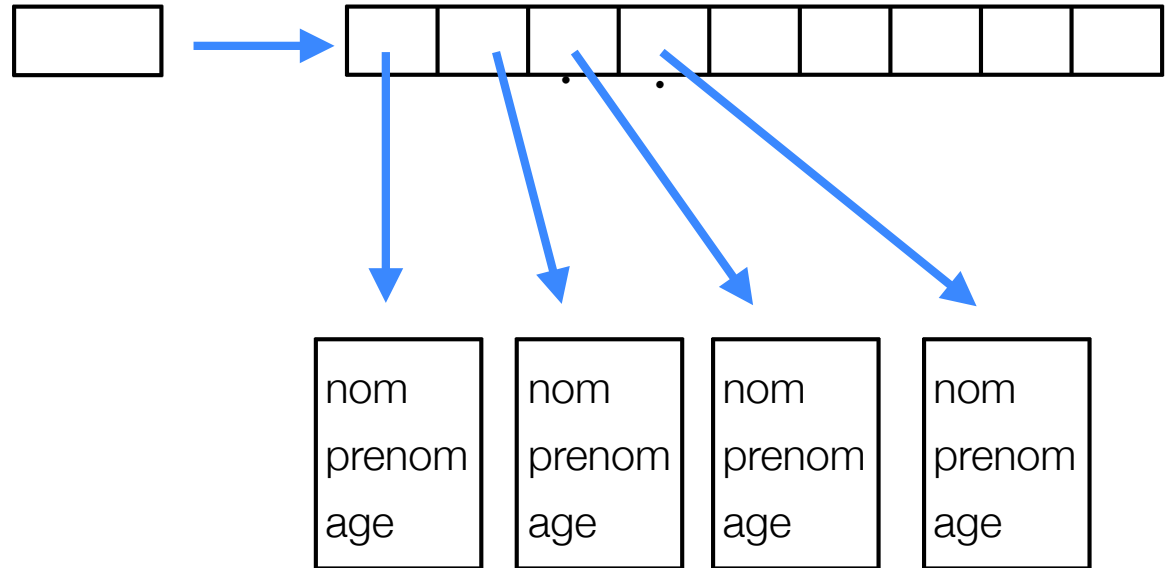
```
    int elevesNbr = 0;
```

```
    eleves = lireEleves(&elevesNbr);
```

```
    afficherEleves(eleves, elevesNbr);
```

```
}
```

eleves

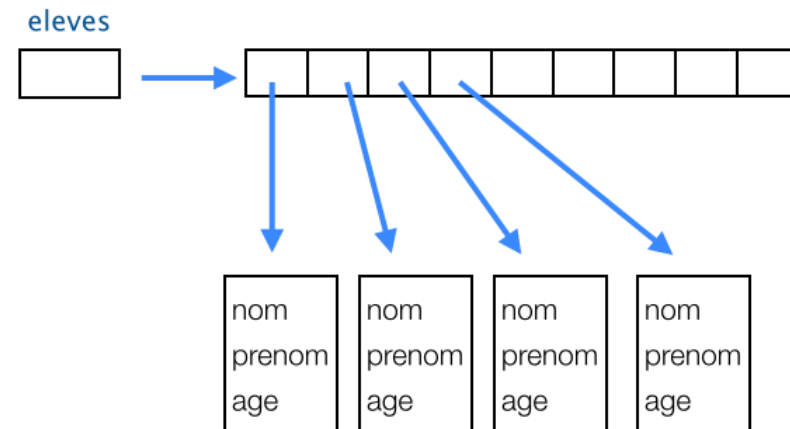


(pseudo)tableau de structures

Structures et pointeurs

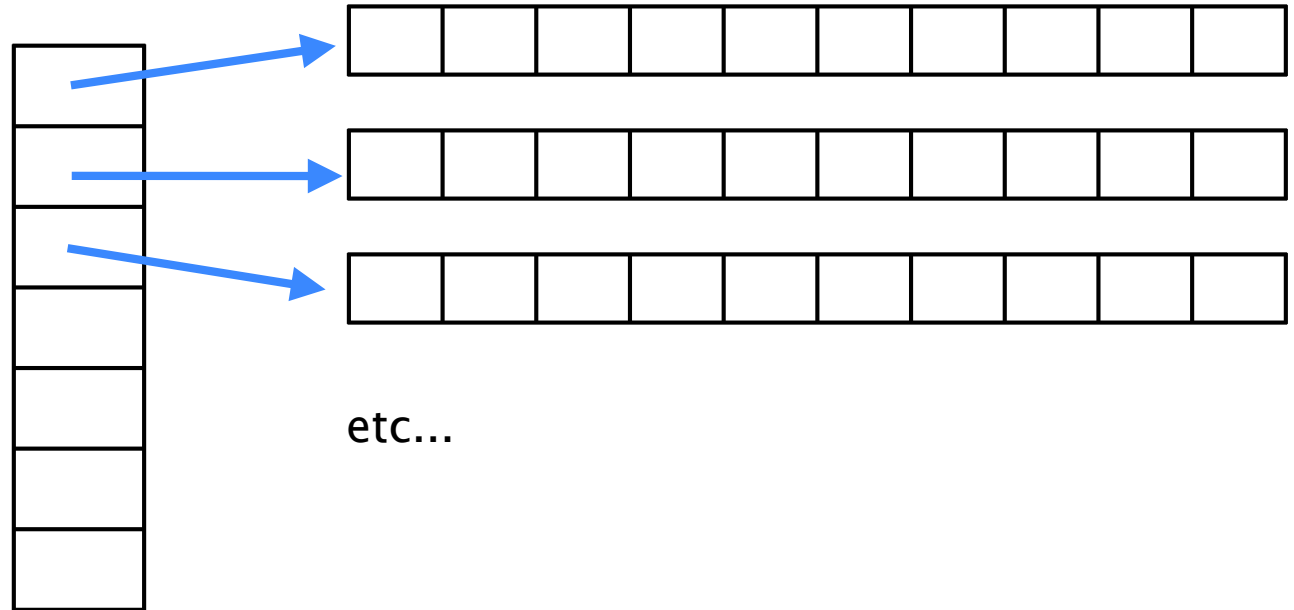
```
typedef struct {  
    char * nom;  
    char * prenom;  
    unsigned int age;  
} Eleve;  
  
void foo() {  
    Eleve ** eleves = NULL;  
    int elevesNbr = 0;  
  
    eleves = lireEleves(&elevesNbr);  
    afficherEleves(eleves, elevesNbr);  
}
```

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



Tableaux de tableaux

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



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

Autres exemples

- tableau d'objets que l'on veut trier
- tableau avec des lignes de longueurs différentes

Récurtivité

Les **variables locales** et les **paramètres** sont stockés dans la **pile**
(sauf contenu des tableaux passés en argument)

Exemples

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

```
int fact2(int n) {  
    return (n <= 1) ? 1 : n * fact2(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 ?

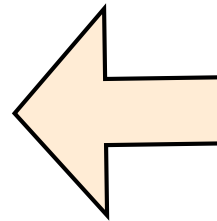
Parcours d'arbre

```
#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);  
}
```



```
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 ?

Entrées/sorties

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

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

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

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

Entrées/sorties au format texte

E/S formatées

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

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

- renvoient le nombre d'éléments lus ou imprimés
- entrées/sorties depuis la console :
 - pseudo-fichiers: **stdin**, **stdout**, **stderr**
 - fonctions **scanf()** et **printf()**

E/S par ligne

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

int **fputs**(const char* str, FILE* stream)

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

Entrées/sorties binaires

Depuis/dans un fichier

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)

- plus rapides que les E/S formatées mais pas portables
- il existe également des fonctions de plus bas niveau : **read()** et **write()**

Entrées/sorties: compléments

Fin de fichier

— **int** **fEOF**(FILE * fichier)

Synchronisation

— **int** **fflush**(FILE * fichier)

Déplacement dans le fichier

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

— **long** **ftell**(FILE * flux);

Directives de compilation

```
#ifndef Person_h
#define Person_h

#include <string>
#include <iostream>

#if defined(RAW_POINTERS)
#  define PTR(TYPE) TYPE*

#elif defined(INTRUSIVE_POINTERS)
#  include "intrusive_ptr.h"
#  define PTR(TYPE) intrusive_ptr<TYPE>

#else
#  error "Undefined pointer mode"
#endif

#endif /* Person_h */
```

Outils de développement

Débugueur

- 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 (profilier)

- 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

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

User* createUser(const char* name, int id);
void destroyUser(User*);
void setUserName(User*, const char* name);
void printUser(const User*);
....

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

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

```
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);  
}
```

```
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);  
}
```

