



Introduction au langage C

L. Lemarchand
UBO

<http://labsticc.univ-brest.fr/~lemarch/FR/Cours/>
Sur moodle pour les Tds et Tps, clé casd



Introduction



D. Ritchie, Labos Bell en 1972

- Langage de haut niveau (à l'époque)
 - impératif
 - modulaire
 - générique
 - bibliothèques standard
 - tables de hash, tris
- Programmation système

- Compilation et exécution
- Variables
- Types et calculs
- Boucles
- Entrées / sorties
- Tableaux
- Fonctions
- Structures



Un langage impératif

- Instructions séquentielles

- Recette de cuisine

- Evolution pas à pas de la mémoire

← PC

```
A = 0
B = 20
ETQ1: B = A + 5
A = B
regT = A - B
AllerTNonNull ETQ1:
C = B
Aller PRINTF:
```

54	67	30	00	29
A	B	C		



Un langage impératif

54	67	30	00	29
A	B	C		

- Les variables représentent la mémoire, leur contenu évolue au fur et à mesure que leur valeur (le contenu de la mémoire à l'endroit que symbolise la variable) est modifié.



Un langage impératif

- Heureusement les instructions sont moins détaillées ! (instructions de plus haut niveau)
- Séquentialité : évolution pas à pas (PC augmente)
 - Explicite : ;
 - Implicite : priorité entre opérateurs, ordre d'évaluation, ...

```
A = 0
B = 20
ETQ1: B = A +
A = B
regT = A - B
AllerTNonNull
C = B
Aller PRINTF:
```



Exemple

- Hello en C

```
#include <stdio.h>
main ( )
{
    printf("hello\n");
}
```

- et en Pascal

```
program hello;
begin
    writeln ('hello');
end.
```



Exemple

- Hello en C

```
#include <stdio.h>
main ( )
{
    printf("hello\n");
}
```

informations sur la
bibliothèque standard



Exemple

- Hello en C

```
#include <stdio.h>
main ( )
{
    printf("hello\n");
}
```

définition d'une fonction
sans arguments

les instructions de la fonction sont
placées entre accolades



Exemple

- Hello en C

```
#include <stdio.h>
main ( )
{
    printf("hello\n");
}
```

main contient une instruction, terminée par ;

appel à printf, avec une chaîne de caractères en argument



Le même

```
#include <stdio.h>
main ( )
{
    putchar('h');
    printf("ello");
    putchar('\n');
}
```

main contient une liste d'instructions, séparées par ;

appels à putchar, avec un caractère en argument à chaque fois



Compilation

- Passer d'un code source à un exécutable

```
main()
{
    ...
    ...
}
```



```
1001101001101
....
(fichier binaire)
```

fichier source **ls.c**
extension .c

fichier exécutable **ls**
pas d'extension

La commande Unix **/bin/ls**



Compilation

Avec un shell de commande :

```
iup212% vi hello.c
```

```
iup212% cc hello.c
```

```
iup212% a.out
```

```
hello
```

```
iup212%
```



Compilation

Avec un shell de commande :

```
iup212% vi hello.c
```

édition du source

```
iup212% cc hello.c
```

```
iup212% a.out
```

```
hello
```

```
iup212%
```



Compilation

Avec un shell de commande :

```
iup212% vi hello.c
```

```
iup212% cc hello.c
```

```
iup212% a.out
```

```
hello
```

```
iup212%
```

compilation :
par défaut, le fichier
exécutable créé
s'appelle a.out



Compilation

Avec un shell de commande :

```
iup212% vi hello.c
```

```
iup212% cc hello.c
```

```
iup212% a.out
```

```
hello
```

```
iup212%
```

exécution de a.out :
hello s'affiche sur la
sortie standard



L'exécutable

- Plusieurs possibilités
 - `iup212% cc hello.c`
 - `iup212% cc hello.c -o affiche`
 - `iup212% cc -o affiche hello.c`
- `hello.c` est un *argument* de `cc`
- `-o affiche` est une *option* (avec son propre argument) de `cc`



La chaine de compilation

- pré-processeur cpp (`cc -E fic.c → fic.i`)
manipulations textuelles
- le compilateur c0+c1 (`cc -S fic.c → fic.s`)
une passe → code assembleur
- l'optimiseur de code c2
dont optimisation de sauts
- l'assembleur as (`cc -c fic.c → fic.o`)
code relogeable
- l'éditeur de liens ld
résol. de liens, ajout de crt0.a et libc.a



Quelques erreurs de compilation

- Variable non définie

```
reglisse[1586]% cc paesnoc.c -o paesnoc
paesnoc.c: In function 'paes_update_grid':
paesnoc.c:720:7: error: 'a' undeclared (first use in this function)
  for (a = 0; a < paes->objs; a++) {
      ^
```

- Fonctions non définies

```
reglisse[1588]% !cc
cc paesnoc.c -o paesnoc
/tmp/ccQyfSvk.o: dans la fonction « paes_init »:
paesnoc.c:(.text+0x355): référence indéfinie vers « paes_update_grid »
```

```
cc paesnoc.c -o paesnoc
/tmp/cc8A3kQ1.o: dans la fonction « paes_find_loc »:
paesnoc.c:(.text+0x2e56): référence indéfinie vers « pow »
```



Quelques outils

- Debugger **gdb**
 - `cc -g monProg.c -o monProg`
 - `gdb monProg`
 - `gdb> run`
 - ...
- Vérification mémoire **valgrind**
 - `valgrind monProg`
- Processus de compilation complexe **make**
 - `make monProg`
 -
- IDE (eclipse, code::blocks sous windows)



Un autre exemple

- Celcius = $(5/9)(\text{Fahrenheit} - 32)$
- Afficher la table de conversion :

0	-17
---	-----

20	-6
----	----

40	4
----	---

60	15
----	----

80	26
----	----



Programme C

```
#include <stdio.h>
main ()
{
    int fahr, cel;
    int mini, maxi,
        intervalle;

    mini = 0;
    maxi = 40;
    intervalle = 20;
```

```
    fahr = mini;
    while (fahr <= maxi) {
        cel = 5*(fahr-32)/9;
        printf("%d\t%d\n", fahr, cel);
        fahr = fahr + intervalle;
    }
}
```



Les variables

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

```
main ()
```

```
{
```

```
    int fahr, cel;
```

```
    int mini, maxi,  
        intervalle;
```

variables visibles
(utilisables dans
des instructions)

```
}
```

Utilisation de variables :

- une **variable** contient une *valeur*
- elle a un **type**
- elle doit être **déclarée** avant d'être utilisée

- en écriture (**affectation**)

```
mini = 0;
```

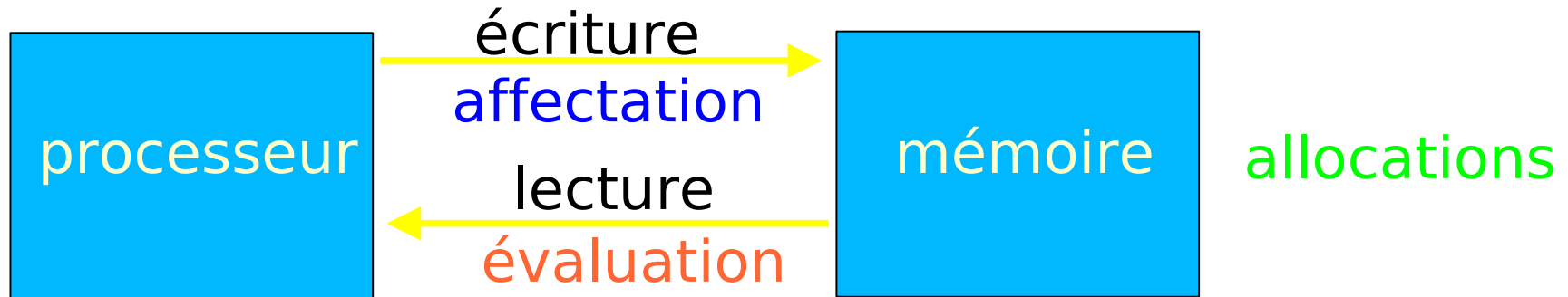
- en lecture dans des expressions

```
fahr = mini;    fahr <= maxi
```

- elle est visible dans son **{ bloc }** de déclaration



Variables et exécution



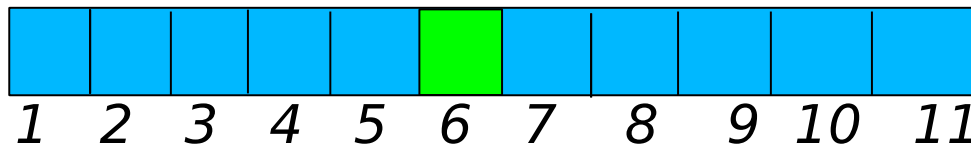
```
main ()
{
    int fahr, cel, mini;
    mini = 0;
    fahr = mini;
    ...
    cel = 5*(fahr-32)/9;
}
```

- Les variables symbolisent la mémoire
- Leur contenu évolue dans l'ordre d'exécution des instructions



Variables et mémoire

- **Mémoire** : tableau linéaire d'octets
- Allocation : réserver des **octets** pour stocker la valeur d'une variable



- *adresse* mémoire : index dans le tableau
 - **Problèmes** :
 - Combien d'octets pour une variable ?
 - Quand peut-on réutiliser la mémoire ?
- (libération, désallocation de la variable)**



Désallocation : en fin de visibilité

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

```
main ()
```

```
{
```

```
    int fahr, cel;
```

```
    int mini, maxi,  
        intervalle;
```

```
    variables visibles  
    (utilisables dans  
    des instructions)
```

```
}
```

Allocations en début
de **bloc** {

Désallocations en fin
de **bloc** }



Désallocation : un autre exemple

```
/* fahr, max et intervalle  
   définis au début de main() */
```

```
while (fahr <= maxi) {  
    int cel;  
    cel = 5*(fahr-32)/9;  
    printf("%d\t%d\n", fahr, cel);  
    fahr = fahr + intervalle;
```

```
}  
/* cel n'est plus utilisable ici */
```

← En passant : /*
commentaire */

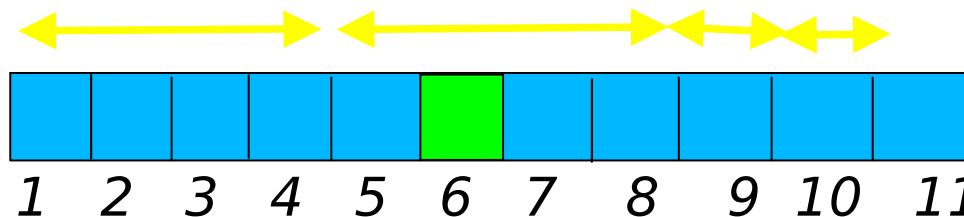
← Allocation en début
de bloc {

← Désallocation en fin
de bloc }

Taille de l'espace mémoire : typage

- **Type** d'une variable :
 - espace mémoire pour la stocker
 - représentation en mémoire
 - représentation pour les entrées/sorties
 - règles de calcul
- Le type est donné lors de la déclaration

```
int fahr = 46, cel;  
char car = 'a', a = '\n';
```



 un octet



Types courants

type	taille (octets)	constantes (exemple)	affichage (format)
char	1	'a' '\n'	%c
int	4	1 - 242	%d
float	4	4.58 - 12.0	%f
double	8	4.58 - 12.0	%lf
char *	4	"coucou"	%s

```
main() {  
    int fahr = 46, taille;  
    char car = 'a';
```

```
    taille = sizeof(int);  
    printf("%c%d %d\n",  
           car, fahr, taille);  
}
```



Types et calculs

```
#include <stdio.h>
main ()
{
    float fahr, cel;
    int mini, maxi,
        intervalle;

    mini = 0;
    maxi = 40;
    intervalle = 20;
```

```
    fahr = mini;
    while (fahr <= (float)maxi) {
        cel = 5.0*(fahr-32.0)/9.0;
        printf("%f\t%f\n", fahr, cel);
        fahr = fahr + intervalle;
    }
}
```

Faire un calcul plus précis



Conversions implicites et explicites

- Conversion implicite
- Conversion explicite (cast)
- Précision :
 - Calcul entier
5 / 2 vaut 2
 - Calcul flottant
5.0 / 2.0 vaut 2.5

```
fahr = mini;  
while (fahr <= (float)maxi) {  
    cel = 5.0*(fahr-32.0)/9.0;  
    printf("%f\t%f\n", fahr, cel);  
    fahr = fahr + intervalle;  
}
```

Conversions vers le type
le plus grand



exercices

- 3 entiers : dans `main()` { }
- Declarer 3 entiers, initialiser le premier et afficher les 3. Afficher ensuite leur somme et leur produit. Ajouter la somme au premier d'entre eux. Reafficher le premier. Faire la division entiere du premier par le second et la division réelle du second par le troisième. Afficher les résultats.
- Afficher la taille mémoire du premier entier.
- Comment utiliser correctement `printf()` ? Essayez avec `Printf()`.
- Compilez et exécutez votre programme



exercices

- **Typage:** déclarer 3 entiers a, b, c dans main()
- Combien vaut a ?
- Déclarer et initialiser un char c et un double d
- Déclarer un tableau de char tc
- Déclarer un pointeur de double pd
- **Taille des variables**
- **OK ?**
 - **a = b * 3; a = c; c = a;**
 - **d = c; b = d;**
 - **tc = c; pd = tc;**



exercices

```
main() {  
    int a = 8 ;  
    if (a > 0) {  
        int b = 5 ;  
        printf("division : %d\n", b / a) ;  
    }  
    printf("produit : %d\n", b * a) ;  
}
```

OK ? Ou qu'est ce qui ne va pas ?



Evaluation

```
fahr = mini;  
while (fahr <= maxi) {  
    cel = 5.0*(fahr-32.0)/9.0;  
    printf("%f\t%f\n", fahr, cel);  
    fahr = fahr + intervalle;  
}
```

une variable
est
remplaçable
par une
expression du
même type

```
fahr = mini;  
while (fahr <= maxi) {  
    printf("%f\t%f\n", fahr, 5.0*(fahr-32.0)/9.0);  
    fahr = fahr + intervalle;  
}
```



Expressions conditionnelles

- Toute valeur non 0 est vraie et 0 est faux
- Conversion implicite en booléen → pas de type booléen
- Opérateurs booléens
 - && ET
 - || OU
 - ! NOT
 - == != égal différent de
 - <= >= sup/inf ou égal à
 - < > sup/inf à



Autres Expressions

- La séquence ,

```
a = 3 , 4
```

```
for (i=8, j=3; i<12; i++, j--) { }
```

- L'expression conditionnelle ?:

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

```
if (a>b)
```

```
    maxi = a;
```

```
else
```

```
    — maxi = b;
```

```
char *reussite = note >= 10 : "ok" : "ko"
```



Vrai ou faux ?

- A
- A = 5
- B = 0
- A == B
- A != B
- A >= B && !B
- A || B
- A > 3 && A <= 9
- A | B
- A << B



Instruction conditionnelle

```
if (condition)
    Bloc
[ else
    Bloc]
```

```
If (a>8)
    printf("%d\n", a*8);
else
    printf("%d\n", a*3);
```

```
if (a>8) {
    printf("%d\n", a*8);
    printf("grand\n") ;
}
else {
    printf("petit\n") ;
    printf("%d\n", a*3);
}
```



Boucle do while();

```
gain = 1000 ;  
do {  
    printf("jouez !\n") ;  
    gain = gain + jeu() ;  
} while (gain >= 0) ;
```

```
do  
    Bloc  
while (condition) ;
```




Structure de contrôle For

```
for (init ; condition de fin; incrémentation)  
    bloc
```

- init évaluée *une fois*, avant l'entrée dans la boucle
- condition de fin évaluée *avant chaque entrée* dans la boucle
- incrémentation évaluée *à chaque fin* de boucle

```
for (fahr = mini; fahr <= maxi; fahr = fahr + intervalle) {  
    cel = 5.0*(fahr-32.0)/9.0;  
    printf("%f\t%f\n", fahr, cel);  
}
```



Boucles avec compteurs: while et for

```
fahr = mini;  
while (fahr <= maxi) {  
    cel = 5.0*(fahr-32.0)/9.0;  
    printf("%f\t%f\n", fahr, cel);  
    fahr = fahr + intervalle;  
}
```

```
for (fahr = mini; fahr <= maxi; fahr = fahr + intervalle) {  
    cel = 5.0*(fahr-32.0)/9.0;  
    printf("%f\t%f\n", fahr, cel);  
}
```



Programme C version 3

```
#include <stdio.h>

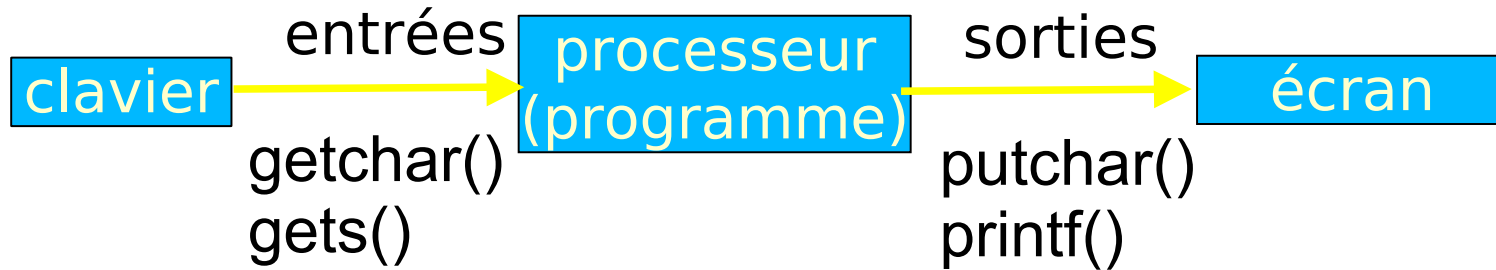
#define MINI      0      /* en passant : definitions */
#define MAXI     300    /* de constantes */
#define INTER    20     /* symboliques */

main ()
{
    int fahr;

    for (fahr=MINI; fahr<=MAXI; fahr=fahr+INTER)
        printf("%3d %6.1lf\n", fahr, 5.0*(fahr-32)/9.0);
}
```



Entrées / Sorties de caractères



```
#include <stdio.h>
main ()
{
    char car;

    car = getchar();
```

```
while (car != EOF) {
    putchar(car);
    /* printf("%c", car) */
    car = getchar();
}
}
```



Entrées / Sorties de caractères

```
#include <stdio.h>
main ()
{
    char car;

    while ((car = getchar()) != EOF)
        putchar(car);
}
```

Une **affectation** est une expression :
sa valeur est celle de son membre **droit**



Entrées / Sorties de caractères

- gets(**buffer**)

```
char chaine[100];  
gets(chaine);
```

lecture jusqu'à EOF ou fin de ligne.

la fin de ligne remplacée par '\0'

- fgets(**buffer**, **taille**, **entrée**)

```
int i = 300;  
char chaine[301];  
fgets(chaine, i, stdin);
```

```
#include<stdio.h>  
FILE *stdin;
```

lecture jusqu'à EOF ou fin de ligne et ajout de⁴⁶\0'



Affichage formaté

`printf(format, ...);`

type	taille (octets)	constantes (exemple)	affichage (format)
char	1	'a' '\n'	%c
int	4	1 - 242	%d
float	4	4.58 - 12.0	%f
double	8	4.58 - 12.0	%lf
char *	4	"coucou"	%s

```
printf("%c%d %d\n", car, fahr, taille);  
printf(s); // si s contient un % ?  
printf("%s", s);
```



Affichage formaté

- `sprintf(buffer, format, ...);`

```
char chaine[100];  
int i = 3;  
sprintf(chaine, "i vaut %d", i);  
printf(chaine);
```

- `fprintf(sortie, format, ...);`

```
int i = 3;  
fprintf(stderr, "i vaut %d", i);
```

```
#include<stdio.h>  
FILE *stderr;  
FILE *stdout;
```




Les tableaux

- Stocker plusieurs valeurs de même **type** dans une seule **variable**.

int fahr ; une seule valeur dans **fahr**

int fahrs[10] ;

10 valeurs distinctes dans **fahrs**

- Taille des variables en mémoire ?

float b[20] ; **char toto[N+1]** ; **int * ptrs[5]** ;



Les tableaux : accès

- Par l'**index** de l'élément dans la variable

```
tableau[index] ;
```

- *Attention* : les index débutent à **0** .

```
{ float tab[6];  
  int i;  
  for (i=0; i<6; i++) tab[i] = 0 ;  
  for (i=0; i<5; i++)  
    tab[i] = tab[i] + tab[i+1] + 3.0*i ;  
}
```

écriture

lecture



Les tableaux : remarques

- Il n'y a pas de contrôle lors de l'accès

```
{  
    float tab[6];  
    tab[4] = 12.7 ;  
    tab[9] = 8.2 * tab[6] ;  
    tab[4] = 't' ;  
}
```



Quelles instructions
sont correctes ?

- C'est au programmeur de savoir si le $i^{\text{ème}}$ élément de son tableau existe



Les tableaux : initialisation

```
{
```

```
int a[3] = { 1, 2, 3 } ;
```

```
int a[3] = { 1, 2 /*, 0 */ } ;
```

```
int a[] = { 1, 2, 3 } ;
```

```
}
```



Les tableaux

Cas des chaînes de caractères

- Une chaîne de caractères est un tableau de caractères

```
char chaîne[16];
```

- Initialisation

```
char chaîne[] = "Une chaîne";
```

```
/* chaîne[3] ? */
```

- Le caractère NULL (ASCII 0) termine :

U	n	e		c	h	a	i	n	e	\0
---	---	---	--	---	---	---	---	---	---	----



Chaines de caractères quelques fonctions

- concaténation
`strcat(char ch1[], const char ch2[]);`
- copie
`strcpy(char ch1[], const char ch2[]);`
- recherche
`strstr(const char ch1[], const char ch2[]);`
- comparaison
`strcmp(const char ch1[], const char ch2[]);`



Chaines de caractères fonction typique

- Toutes basées sur le parcours de chaîne jusqu'au caractère '\0'
- Exemple :

```
strchar(char car, const char chaine[])  
{  
    int idx = 0, trouve = 0; char cour = chaine[0];  
    while (cour != car && cour != '\0') {  
        idx = idx + 1;  
        cour = chaine[idx]; }  
    if (cour) trouve = 1;  
}
```



Les tableaux multi-dimensionnels

■ Définition

```
{  
    float tab[6][8];  
    char chaines[46][18][45];  
    int a[2][3] = { {1, 5, -4}, {-8, 10, 45} } ;  
    /* ... */  
}
```




Les tableaux multi-dimensionnels exemple

```
main()
{
    int i, j, N=5, M=8;
    int tab[N][M] ;

    /* initialisation */
    ....
```

```
/* affichage */
for (i=0; i<N; i++) {
    for (j=0; j<M; j++)
        printf("%d", tab[i][j]);
    printf("\n");
}
}
```



Les fonctions

- Sous programme
- Boîte noire vue de l'extérieur
- Décomposition
- Réutilisation



Les fonctions exemple

```
int puiss(int m, int n) ;
```

```
main()
```

```
{  
    int i;  
  
    for (i=0; i<10; i++)  
        printf("%d %d\n",  
            i, puiss(i, 3) );  
}
```

```
int puiss(int base, int n)
```

```
{  
    int i, p;  
  
    p = 1;  
    for (i=1; i<= n; i++)  
        p = p * base;  
    return p;  
}
```



Les fonctions

Déclaration



```
int puiss(int m, int n);
```

```
main()
```

```
{
```

```
    int i;
```

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

```
        printf("%d %d\n",
```

```
            i, puiss(i, 3);
```

```
}
```

Appel



```
int puiss(int base, int n)
```

```
{
```

```
    int i, p;
```

```
    p = 1;
```

```
    for (i=1; i<= n; i++)
```

```
        p = p * base;
```

```
    return p;
```

```
}
```

Définition





Les fonctions

- Syntaxe de la définition :

type retourné **fonction**(paramètres formels)

{

définition de variables locales

instructions : *variables locales et paramètres sont utilisables*

}

```
int puiss(int base, int n) { ... }
```



Les fonctions paramètres

- Syntaxe des paramètres :
type nom , type nom, ...

```
int puiss(int base, int n) { ... }
```

- Paramètres : remplacés par des valeurs (arguments) à chaque appel :

```
puiss(i, 3);
```



Les fonctions : appels (suite)

```
puiss(i+1, 3);
```

appel

```
int puiss(int base, int n) { ... }
```

définition

- Lors d'un appel, chaque argument (expression) est évaluée. Une copie du résultat est placée dans une variable locale, comme si :

```
int puiss() {  
    int base = 7; int n = 3;  
    ...  
}
```



Exercices

- Déclarer une fonction utilisant un entier et un caractère et retournant un float
- Définir une fonction utilisant un tableau de int et un int et ne retournant rien
- Donner des exemples d'utilisation des deux fonctions



Les fonctions : noms locaux

```
main()
```

```
{  
    int nb ;  
  
    for (nb=0; nb<10; nb++)  
        printf("%d %d\n",  
            nb, puiss(nb, 3) );  
}
```

```
int puiss(int base, int n)
```

```
{  
    int nb, p;  
  
    p = 1;  
    for (nb=1; nb<= n; nb++)  
        p = p * base;  
    return p;  
}
```

nb & **nb** complètement indépendants



Les fonctions : copies & noms locaux

```
void truc();
```

```
main()
```

```
{
```

```
    int i = 6;
```

```
    truc(i);
```

```
    printf("%d\n", i);
```

```
    /* affichage ? */
```

```
}
```

```
void truc(int j)
```

```
{
```

```
    j = 1;
```

```
}
```



Les fonctions : copies & noms locaux (2)

```
void inc(int i)
```

```
{
```

```
    i = i+1;
```

```
}
```

```
main()
```

```
{
```

```
    int i = 6;
```

```
    inc(i);
```

```
    // Que vaut i ?
```

```
}
```



Exercices

- Montrer ce qui se passe en mémoire lorsque on exécute le code suivant

```
int fct(char car, int sens) {  
    if (sens > 0)  
        car = car + 'A' - 'a';  
    else  
        car = car + 'a' - 'A';  
    return car;  
}
```

```
main() {  
    int u = fct('g', 1);  
    int vv = fct('V' - 2, -1);  
    int uu = fct(u, vv);  
    char car;  
    // u, vv, uu, car ???  
}
```



Les fonctions : valeur de retour

- Que vaut la fonction ?

```
float sqrt(float number) ;
```

```
float v = sqrt(5.3);
```

- Comment spécifier le résultat d'un appel à la fonction ?



Les fonctions : valeur de retour (suite)

- Si la fonction ne renvoie rien :

```
void toto() ;
```

- Sinon : **return**

```
int puiss(int base, int n) {  
    ...  
  
    for (i=1; i<= n; i++)  
        p = p * base;  
    return p;  
}
```



Les fonctions : valeur de retour (suite)

- La valeur de retour est typée

```
int puiss();  
void toto();
```

- L'instruction `return expression ;` :
 - expression doit être du type prévu
 - la fonction se termine



Les fonctions

exemple de retour

```
int fonc1()
{
    int a = 7;

    if (a<12)
        return 4;
    a = a * 3;
    return a + 8;
}
```

```
int fonc2()
{
    int c = 4, toto;

    toto = fonc1() + c*fonc1();
    return toto % fonc1();
}
```




Utilisation du retour strchar()

```
int strchar(char car, const char chaine[])
{
    int idx = 0;
    while (chaine[idx] != '\0') {
        if ( chaine[idx] == car) return 1;
        idx = idx + 1;
    }
    return 0;
}
```



Les fonctions retours multiples

```
??? heure()  
{  
    ....  
    /* return heures,  
       minutes, secondes;  
    */  
}
```

- Comment renvoyer plusieurs valeurs ?
- Structures
- Passage par *adresse*

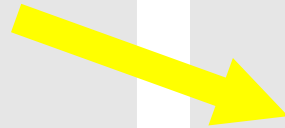
& *



Les fonctions passage par adresse

```
void heure(int h, int m, int s)
```

```
{  
    ....  
    h = ... ;  
    m = ... ;  
    s = ....;  
}
```



```
void heure(int *h, int *m, int *s)
```

```
{  
    ....  
    *h = ... ;  
    *m = ... ;  
    *s = ....;  
}
```

**Passage par
adresse**

Ne marche pas !



Pointeurs == variables

- Rappel: variable == nom symbolique d'une zone mémoire

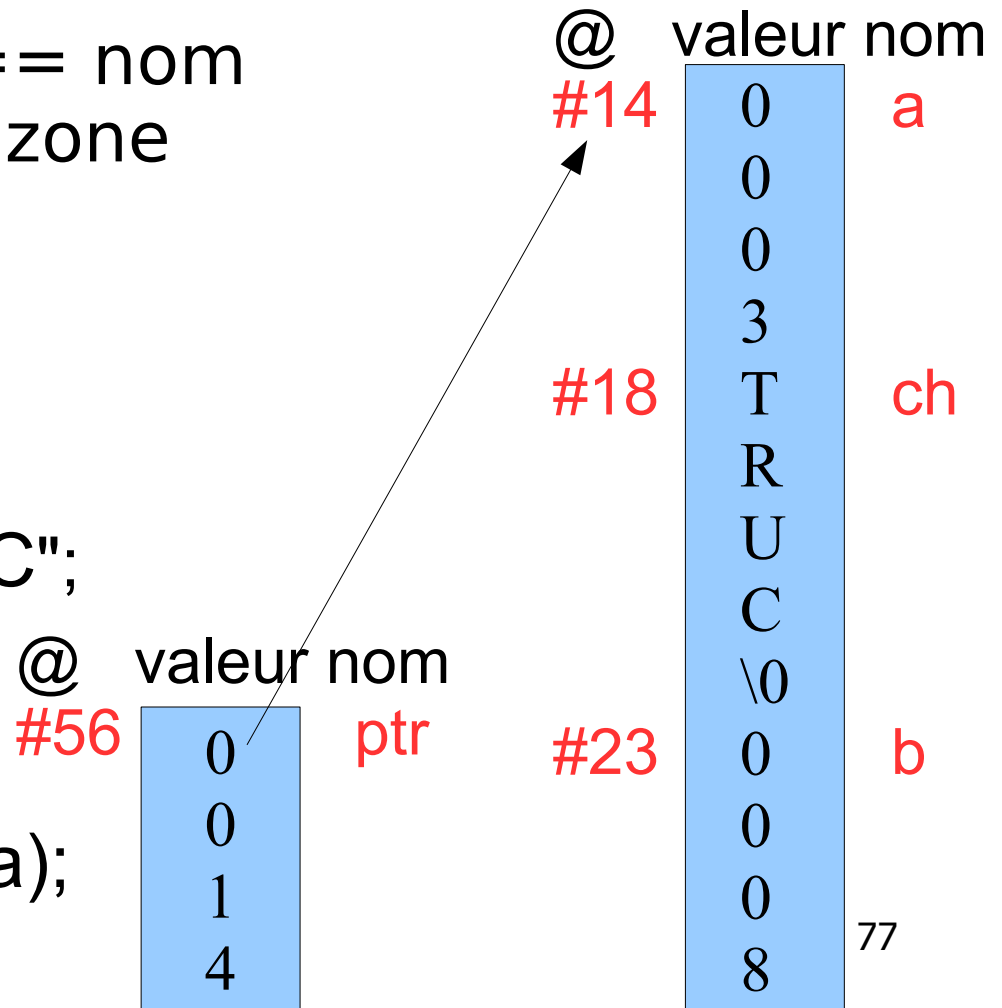
```
{  
    int a = 3;  
    char ch[5] = "TRUC";  
    int b = 8;  
}
```

@	valeur	nom
#14	0	a
	0	
	0	
	3	
#18	T	ch
	R	
	U	
	C	
	\0	
#23	0	b
	0	
	0	
	8	
		76

Pointeurs == variables

- Rappel: variable == nom symbolique d'une zone mémoire

```
{  
    int a = 3;  
    char ch[5] = "TRUC";  
    int b = 8;  
  
    adresse ptr = adr(a);  
}
```





Pointeurs - usage

- Copie rapide de variables complexes/ de grande taille
 - `int tab[100000] ; // tab est une adresse !`
 - `Pointeur tab2 = adresse(tab1)`
- Partage (ex. Entre fonctions)
 - `tab2[5] = 8; // tab1[5] == 8`
 -
- Allocation dynamique à l'exécution si taille inconnue à la compilation (alloc statique)
 - `Pointeur tab; lire(taille);`
 - `tab = allocMem(taille);`
 -



Pointeurs notations

```
void heure(int *h, int *m, int *s)
```

- h, m, et s sont des **pointeurs** d'entiers (notation étoile)
- On peut modifier leur *contenu*

```
*h = 10 ;
```
- et y accéder :

```
calcul = (*m + 40 + *s * 100) % 60 ;
```
- On obtient un pointeur avec **&** :

```
int uneh, unem, unes;  
heure(&uneh, &unem, &unes);
```



Pointeurs typage

```
int *ph; char *pm;
```

TYPE * variablePointeur;

```
char *chaines[2] = { "toto", "titi" };  
char **ptr = &chaine[0];
```

Pointeurs et tableaux `int tab[5]`
`*(tab + 5) == *(5 + tab) == 5[tab];`
`// 5 x sizeof(int)`

```
int tab[3];  
tab[1] = 8;  
printf("%d %d\n", 1[tab], tab[1]);  
// 8 8
```




Pointeurs et mémoire

```
char c;
```

```
c = 'z'
```

```
char *ptr;
```

```
ptr = c;
```

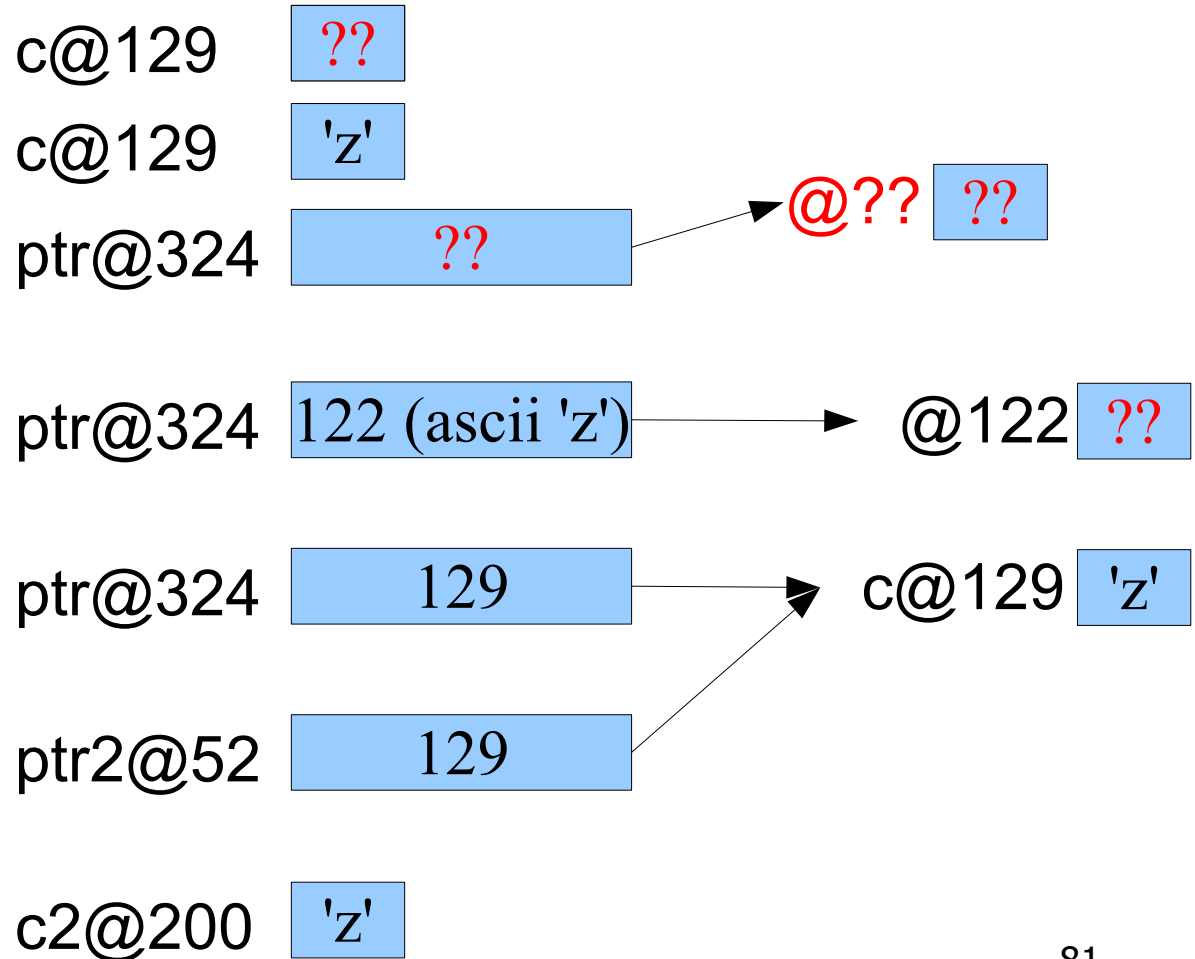
```
ptr = &c;
```

```
char *ptr2;
```

```
ptr2 = ptr;
```

```
char c2;
```

```
c2 = *ptr;
```





Pointeurs utilisation (copie)

```
int tab1[1000000], tab2[1000000];  
int *data = &(tab1[0]);  
int *result = &(tab2[0]);  
for(i=0; i< generations; i++) {  
    traitement(data, result);  
    tmp = data;  
    data = result;  
    result = tmp;  
}
```



Pointeurs utilisation (allocation dynamique)

```
int *tab;  
int taille;  
scanf("%d", &taille);  
tab = (int *)malloc(taille * sizeof(int));  
for(i=0; i< taille; i++)  
    tab[i] = i;
```



Pointeurs utilisation (partage)

```
int scanf(format, arg1, ...)
```

- Format décrit les entrées attendues sur l'entrée standard – renvoie le nb de succès en lecture
- arg1, ... sont des **pointeurs** du type des entrées attendues
- scanf modifie des valeurs pointées

```
int num;  
char rue[MAXNOM];  
  
scanf("%d rue %s", &num, &rue[0]);
```



Les structures

- Tableaux : plusieurs valeurs **de même type** dans une **variable**.

accès aux composantes de la variable **par index** (la $i^{\text{ème}}$ valeur dans la variable)

- Structures : plusieurs valeurs **de types différents** dans une même **variable**.

accès aux composantes de la variable **par nom** (le champ, la partie appelée X dans la variable)

- Problème : définir un nouveau type



Les structures : définition de type

- Syntaxe :

```
struct nomType {  
    type1 champ1;  
    type2 champ2;  
    ...  
};
```

- Exemples :

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

```
struct personne {  
    char nom[10];  
    int age;  
};
```



Les structures : définition de variables

- Syntaxe : `struct nomType variable;`

```
// en dehors des blocs
struct point {
    int x, y;
    double norme;
};
struct personne {
    char nom[10];
    int age;
};
```

```
main() {
    int i;
    struct point unPoint;
    struct personne l, m;
    struct personne toto =
        { "Mr toto", 10 };
    ...
}
```



Les structures : accès aux champs

- Une variable X est de type structuré, avec un champ y : la valeur du champ y est obtenue par X.y (notation point)

```
struct personne {  
    char nom[10];  
    int age;  
};
```

```
main() {  
    int i;  
    struct personne toto =  
        { "Mr toto", 10 };  
  
        i = toto.age ;  
        strcpy(toto.nom, "Mr Titi") ;  
}
```




Structures exemple

```
struct hr {  
    int heure;  
    int minute;  
    int secondes;  
};
```

```
struct hr heure()  
{  
    struct hr now;  
    now.heure = 10;  
    now.minute = 20;  
    now.secondes = 30;  
  
    return now;  
}
```



Structures

exemple plus réel - temps

```
#include <sys/time.h>
```

```
long msectime()  
{
```

```
    struct timeval tv;
```

```
    gettimeofday(&tv, (struct timezone *)0);
```

```
    return (1000L*tv.tv_sec) + (((long) tv.tv_usec)/1000L);
```

```
}
```

```
// time.h
```

```
...
```

```
struct timeval {  
    time_t tv_sec;  
    suseconds_t tv_usec;  
};
```

```
...
```



Exercices

- Définissez une structure point
- Définissez 2 points (initialisez les) et un pointeur de point qui pointe vers le premier point.
- Définissez un tableau de points, et stockez y les deux points. Modifiez le deuxième champ du 2ème point du tableau.
- Faites de même en utilisant un tableau de pointeurs de points.



Classes mémoire

- Visibilité et durée de vie des variables
 - globales/locales
 - dans la pile ou le tas

	globale	locale
pile		register/auto
tas	static	static



Classes mémoire variables globales

```
int a ;
```

définition d'une variable globale

```
extern int b ;
```

```
fct() {
```

```
    int b;
```

```
    .....
```

```
}
```

déclaration d'une variable globale qui est définie dans un autre fichier (indépendante de la variable b de fct() qui n'est PAS globale)

```
int v ;
```

```
int verboseLevel ;
```



Classes mémoire variables globales

- A ne pas utiliser car effet de bord. Sauf
 - Valeur **unique** utilisée dans de nombreuses fonctions, susceptible d'évoluer
 - (Limitable à un fichier (globale static))

~~int age ;~~

int ageDoyenDesFrancais;



Classes mémoire variables locales

```
fct(int y, int z) {  
    int b;
```

```
    register int c;
```

```
    volatile int d;
```

```
    static int e;
```

```
    .....
```

```
}
```

y, z: les paramètres sont des vars locales

b: définition d'une variable locale

c: Indication au compilateur : garder si possible c en priorité dans un registre

d: Indication au compilateur : attention, d peut être modifiée en mémoire par un autre processus. La recharger en registre systématiquement

e: locale implantée dans le tas. Conserve sa valeur d'un appel à l'autre de fct()



Exemples d'utilisation de locales statiques

```
int firstHasBonus(int points) {  
    static int first = 1;  
    if (first) {  
        first = 0;  
        return points*5 + 1000;  
    }  
    else  
        return points*5;  
}
```

```
int uniqueID() {  
    static int id = 0;  
  
    id++;  
  
    return id;  
}
```




Classes mémoire variables non modifiables

```
fct(int y, const int z) {
```

```
    int b;
```

```
    const int c = 7;
```

```
    ...
```

```
}
```

```
const int g = 8;
```

```
...
```

z: sa valeur ne peut être modifiée dans la fonction. Erreur à la compilation si affectation sur z dans le code

c et g: leur valeur ne peut être modifiée après leur définition (affectation obligatoire lors de la définition)



Le préprocesseur #include

- Appliqué avant la compilation
- Substitutions de texte
 - #include pour les prototypes de fonctions et les déclarations de variables et de types

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



Le préprocesseur #define

- Constantes symboliques

```
#define __STDC  
#define MMSG "un message"  
#define MAXI 100
```

- Fonctions en ligne

```
#define MAX(x, y) ((x)<(y) ? (y) : (x))  
#define FOR(borne) for(i=0; i<borne; i++)
```



Rappel : La chaine de compilation

- pré-processeur cpp (cc -E fic.c → fic.i)
manipulations textuelles

```
// toto.c
```

```
#include "m.h"
```

```
printf ("%d\n", 3*MAX);
```

```
// m.h
```

```
#define MAX 2 + 1
```

- **toto.i**

```
# 1 "toto.c"
```

```
# 1 "<interne>"
```

```
# 1 "<ligne de commande>"
```

```
# 1 "toto.c"
```

```
# 1 "m.h" 1
```

```
# 3 "toto.c" 2
```

```
printf ("%d\n", 3*2 + 1);
```



Le préprocesseur

#ifdef

- Instructions conditionnelles

```
#ifdef VAR
```

```
...
```

```
#endif
```

```
#else
```

```
#ifndef VAR
```

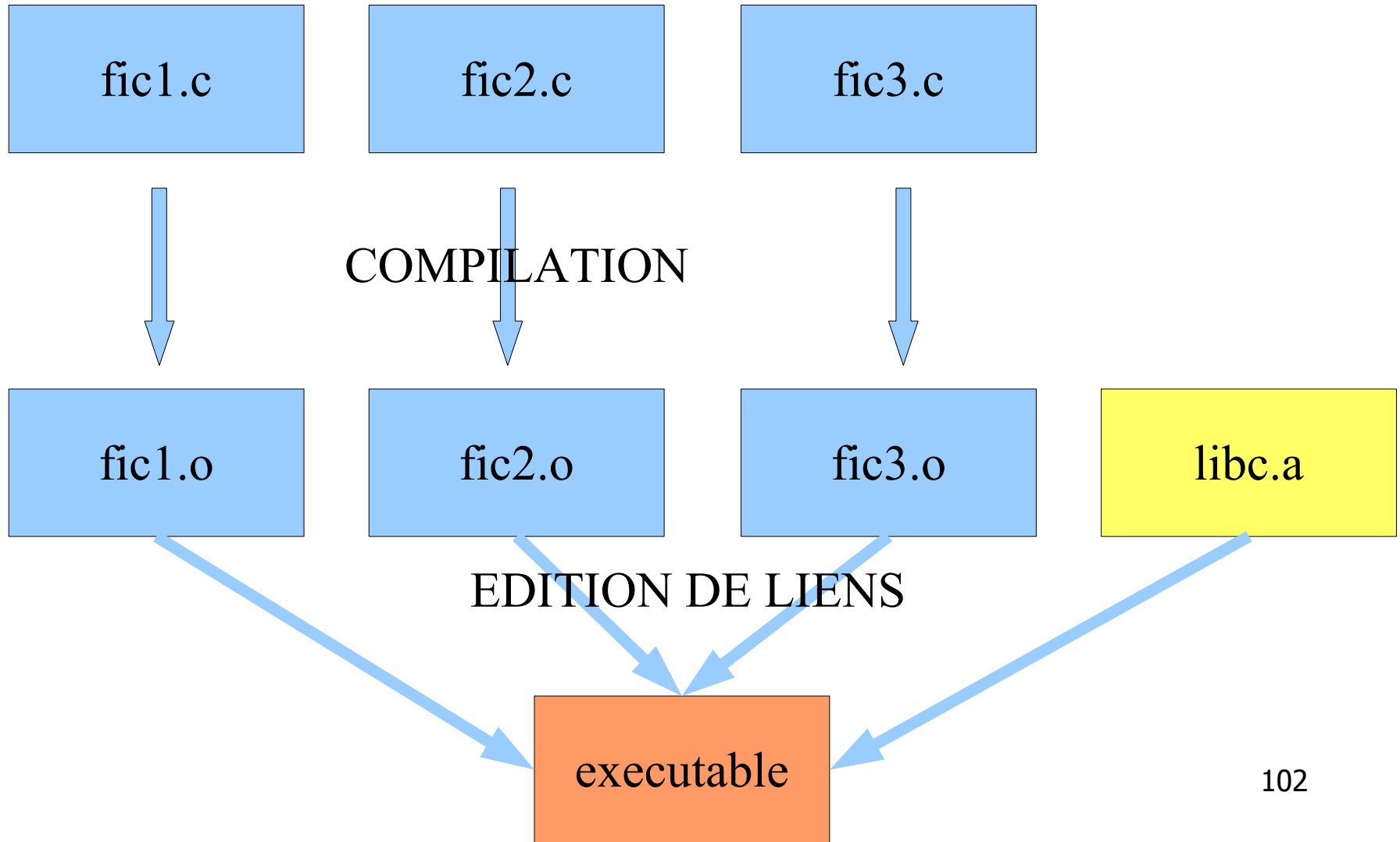
```
#ifdef _DEBUG
```

```
printf("on en est la");
```

```
#endif
```



Compilation séparée





Compilation séparée

- Compilation + édition de liens

```
cc fic1.c fic2.c fic3.c -o executable
```

- Compilation puis édition de liens

```
cc -c fic1.c
```

```
cc -c fic2.c
```

```
cc -c fic3.c
```



Fichiers objet

```
cc fic1.o fic2.o fic3.o -o executable
```



Un exemple

```
// francs.c
```

```
float fr2euros(int euros)  
{ return euros * 6.57; }
```

```
// main.c
```

```
main()  
{ printf("%f \n", fr2euro(10)); }
```

```
cc -c francs.c
```

```
cc -c main.c
```

```
cc -o francs2euros main.o francs.o
```

que va t il se passer ?

Compilation séparée

```
// f1.c
#include "point.h"
...

main() {
    point_t p = newP(3,4);
    ...
}
```

```
// point.c
#include "point.h"
point_t newP(int a, int b) {
    point_t n;
    n.x = a;
    n.y = b;
    return n;
}
```

```
// point.h
#ifndef __POINTH
#define __POINTH
struct point {
    int x, y;
};
typedef struct point point_t;

point_t newP(int, int);
#endif
```



Un exemple de TAD Pile

```
// pile.h

// donnees

#define MAXPILE 128

typedef struct st_pile {
    int nb;
    — Int data[MAXPILE];
} pile_t;
```

```
// main.c

#include <stdio.h>
#include "pile.h"

main() {
    pile_t p;
}
```

Structure de données
1er test

Une pile statique, ajout, retrait, affichage

cc main.c -o tadpile



Un exemple de TAD Pile v1

```
// pile.h
```

```
...
```

```
pile_t pile_nouv();
```

```
// main.c
```

```
#include "pile.h"
```

```
main() {  
    pile_t p = pile_nouv();  
}
```

```
// pile.c
```

```
#include "pile.h"
```

```
pile_t pile_nouv() {  
    pile_t p;  
    p.nb = 0;  
    return p;  
}
```

1ere fonction
2eme test



Un exemple de TAD Pile v2 (@)

```
// pile.h
```

```
...
```

```
void pile_init(pile_t *p);
```

```
// main.c
```

```
...
```

```
main() {  
    pile_t p;  
    pile_init(&p);  
}
```

```
// pile.c
```

```
#include "pile.h"
```

```
void pile_init(pile_t *p) {  
    p->nb = 0;  
}
```

1ere fonction
2eme test

108

cc main.c pile.c -o tadpile



Un exemple de TAD Pile

```
// pile.h
```

```
...
```

```
int pile_aff(pile_t *p);
```

```
// main.c
```

```
...
```

```
main() {  
    pile_t p;  
    pile_init(&p);  
    pile_aff(&p);  
}
```

```
// pile.c
```

```
#include "pile.h"
```

```
...
```

```
void pile_aff(pile_t *p) {  
    int i ;  
    for (i=0; i<p->nb; i++)  
        printf("%d ", p->data[i]);  
    — printf("\n");  
}
```

1ere fonction
3eme test



Un exemple de TAD Pile

```
// pile.h

// donnees
#define MAXPILE 128
typedef struct st_pile {
    int nb;
    int data[MAXPILE];
} pile_t;
// fonctions
void pile_init(pile_t *p);
void pile_aff(pile_t *p);
int pile_vide(pile_t *p);
int pile_pleine(pile_t *p);
int pile_ajout(int v, pile_t *p);
int pile_retire(int *v, pile_t *p);
```

Fonctions
à faire et à tester ...
Notation des noms uniforme
Style des paramètres uniforme
Tests au fur et à mesure