Pré-formation AST - C / Algorithmique

ENSTA - 2ème année

François Pessaux

U2IS

2021-2022

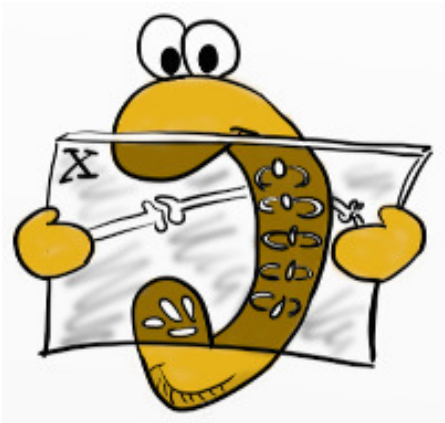
prenom.nom@ensta-paristech.fr

Le langage

Le langage C

- Déjà vu : Python.

- Programmation, algorithmique $\neq$ langage.
- Langage = support, mots pour le dire.



- Maintenant. . . C.
- Créé B. Kernighan, D. Ritchie et K. Thompson aux Bell Labs en 1972.
- Une mine d'infos : `http://www.lysator.liu.se/c/`

C pourquoi ?

[MODE publicité=ON]

- Très utilisé dans l'industrie malgré certains défauts/difficultés.
- « *Souple* » (programmation haut et bas niveau).
- Disponible quelle que soit (à 99.9999%) l'architecture.
- Seul moyen « *simple* » de piloter du matériel.
- Excellentes performances (sauf sur un algo pourri bien sûr).

[MODE publicité=OFF]

[MODE inconvénient=ON]

- Gestion de la mémoire explicite.
- Structures de données de base rudimentaires.
- Fait apparaître des détails de compilation du langage.

[MODE inconvénient=OFF]

Python vs C

```python
up_to_number = 10000
print("Prime numbers up to " + str(up_to_number) +
      ":")
for number in range(2, up_to_number + 1):
    isprime = True
    for divisor in range(2, number - 1):
        remainder = number
        while (remainder >= divisor):
            remainder = remainder - divisor
        if (remainder == 0):
            isprime = False
    if isprime:
        print(number)
```

```c
#include <stdio.h>
#include <stdbool.h>
#define UP_TO_NUMBER (10000)
int main () {
  unsigned int number ;
  printf ("Prime numbers up to %d:\n", UP_TO_NUMBER) ;
  for (number = 2; number <= UP_TO_NUMBER; number++) {
    unsigned int divisor ;
    bool isprime = true ;
    for (divisor = 2 ; divisor < number; divisor++) {
      unsigned int remainder ;
      remainder = number ;
      while (remainder >= divisor) {
        remainder = remainder - divisor ;
      }
      if (remainder == 0) isprime = false ;
    }
    if (isprime == true) printf ("%d\n", number) ;
  }
  return (0) ;
}
```

```
mymac:/tmp$ time python allprimes.py > /dev/null
real 0m42.976s
user 0m42.960s
sys 0m0.014s
mymac:/tmp$ time ./allprimes.x > /dev/null
real 0m0.887s
user 0m0.885s
sys 0m0.002s
```

L'incontournable premier programme en C

joe.c

```c
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
  printf ("Eat at Joe's\n") ;
  return (0) ;
}
```

- La plus simple manière d'afficher une simple chaîne de caractères.
- Équivalent au programme Python :

```
print "Eat at Joe's"
```

L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
  printf ("Eat at Joe's\n") ;
  return (0) ;
}
```

- `#include <>` : Demander accès aux fonctions entrée/sortie (`printf`).
- `/* ... */` : Commentaires.
  - Ignorés par le langage.
  - **Mais** pas par les lecteurs !
  - Servent à la documentation du code.
  - **Documentez** vos programmes !
- Contrairement à Python, espaces non significatifs...
- mais continuez à **indenter** votre code !

L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
  printf ("Eat at Joe's\n") ;
  return (0) ;
}
```

- Fonction `main` : **Le** point d'entrée du programme.
- Toujours automatiquement appelée en premier.
- Un programme en C doit toujours contenir **une** fonction `main`.
- Type `int` (entier) au début : type renvoyé par `main`.
- Pour le `main` : toujours un entier.

L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
  printf ("Eat at Joe's\n") ;
  return (0) ;
}
```

- Fonction `printf` : affiche un message dans le terminal.
- `"Eat at Joe's\n"` : chaîne de caractères (à afficher).
  - `\n` : représente le caractère *« retour à la ligne »*.
- `;` Point-virgule : fin d'instruction, marque la séquence.

L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
  printf ("Eat at Joe's\n") ;
  return (0) ;
}
```

- Retour de la fonction `main` :
  - Code de sortie du programme.
  - Existe pour toutes les commandes Unix
  - Permet de tester si le programme s'est exécuté normalement.
  - Convention : 0 ≡ « OK ».

Compilation et exécution d'un programme

- Édition du source :
  - `> emacs joe.c &`
  - `emacs` : Éditeur de texte, permet de taper le source.
  - « *Human-readable* ».
  - Vous pouvez utiliser n'importe quel éditeur (vi, vim, joe...)
- Compilation :
  - `gcc joe.c` ou `gcc joe.c -o first.x`
  - gcc (compilateur) transforme le source en exécutable.
  - « *Computer-readable* » : compilateur ≡ traducteur.
  - Par défaut, nom de l'exécutable = « a.out ».
  - Option `-o` : permet de spécifier un nom d'exécutable.
- Exécution :
  - `./a.out` ou `./first.x`
  - ... en fonction du nom de votre exécutable.
  - Exécute votre programme dans le répertoire courant.
  - « ./ » devant car le répertoire courant n'est pas dans le `PATH`.

Le compilateur gcc

- Compilateur : programme transformant un source en exécutable.
- Dispose de nombreuses options dont :
  - `-W` : Activer des warnings (`-Wall` : tous, plus que recommandé).
  - `-g` : Activer les infos de débug.
  - `-I` : Ajouter des répertoires où trouver les entêtes (`#include`).
  - `-c` : Compile sans « *linker* » (lorsque plusieurs fichiers sources).
  - `-L` : Transfère des options au « *linker* »
  - `-O0 -O1 -O2 -O3` : Optimise pas, peu, plus, beaucoup, (trop).
  - `-l` : Lier une bibliothèque (Ex : `-lm` pour la bibliothèque maths).
- Si tout s'est bien passé, le compilateur n'affiche rien.
- S'il affiche une erreur, l'exécutable n'est pas créé.
- S'il affiche un/des warning/s, l'exécutable est créé, mais risque de ne pas fonctionner correctement.

Les messages d'erreur et d'avertissement

- Erreurs et avertissements sont détaillés par gcc !
  - nature de l'erreur,
  - numéro de ligne du source incriminée.

$\Rightarrow$

Quelques constructions élémentaires

Expressions (de base) du langage

- À partir d'expressions de base, on combine les expressions.
- Expressions de base : les variables et les littéraux.
  - Variables : toute variable déclarée et de portée accessible.
  - Entiers : `4` `-5`
  - (~ Booléens : `true` `false`)
  - Caractères : `'U'` `'n'`
  - Flottants : `-4.6` `5e-12`
  - Chaînes : `"plop"` `"\tGlop\n"`

Composition d'expressions

- Arithmétique : entre entiers et/ou flottants.
  - +, -, /, * avec leur sens « habituel », % (modulo)
  - Ex : `4 - (5 * 7)`
- Relationnel (« tests ») : entre expressions de même type.
  - Rem : conversions implicites entre scalaires (entiers, flottants, caractères et booléens).
  - == (égalité), != (inégalité), <, >, <=, >=.
  - Ex : `y <= 5  (6 * 4)< x`
- Logique : entre booléens (donc implicitement, entiers).
  - && (et), || (ou), ! (négation).
  - Ex : `(y <= 5)&& ((6 * 4)< x)`
  - && et || sont *« paresseux »*.
- Binaire (« bit-à-bit ») : entre entiers / caractères.
  - Fonction logique opérant sur chacun des bits de la représentation.
  - ~ (négation), ^ (ou exclusif), & (et), | (ou).
  - Ex : `(~x ^ y)^ 0xFE`

La mémoire

- Avant d'être exécuté, le programme est chargé en mémoire.
- Un «*pointeur* » indique où en est l'exécution.
- À chaque instruction il avance d'une «*case* ».
- Il y a parfois des sauts (appels de fonctions, tests, boucles. . .).
- Pendant l'exécution, le programme peut aussi réserver de la mémoire :
  - Déclaration (création) de variables, réceptacles d'information.
  - À chaque variable correspond une «*case mémoire* ».

Nécessité de typage

- Mémoire : contient de l'information codée en binaire.
- Nous avons vu quelques formes de données.
- $\Rightarrow$ un octet en mémoire peut avoir différentes significations.
- L'octet `01010111` peut représenter :
  - l'entier 87 ($= 64 + 16 + 4 + 2 + 1$),
  - le flottant $4.85 \cdot 10^{-270}$,
  - (le caractère 'W'),
  - une instruction en langage machine,
  - une adresse mémoire, etc ...

- ⚠ En C, ce n'est pas comme en Python !
- Il faut associer un type à chaque case mémoire, chaque variable !
- C'est son type (lors de la déclaration) qui va définir sa signification réelle.

Types de base en C

- `char` : caractères (on y reviendra).
- `int` : entiers (on y reviendra)..
- `float` : flottants (simple précision).
- `double` : flottants (double précision).

- Chaînes de caractères pas « *natives* » (on y reviendra).
- Pas de « vrais » booléens : utilisation d'entiers.
  - $0 \equiv$ « faux ».
  - Autre $\neq 0 \equiv$ « vrai ».
  - Type `bool` avec les constantes `true` et `false` disponibles en utilisant le fichier d'entête `stdbool.h`.

Déclaration et initialisation de variable (1)

```
int main ()
{
  float c ;
  int i, j = 42 ;
  double b ;
  b = 0.00345 ;
  c = 3.14159 ;
  return (0) ;
}
```

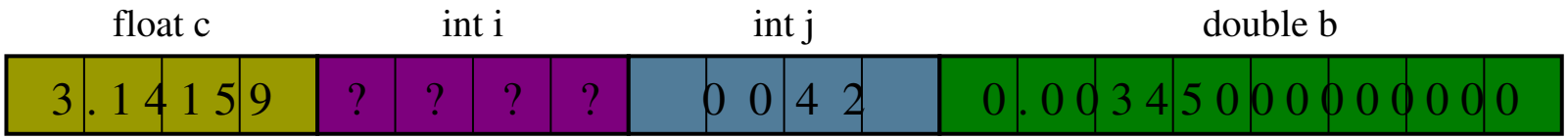
- On déclare une variable en donnant son type suivi de son nom.
- Cela réserve une case mémoire contenant *a priori* une valeur quelconque.
- L'initialisation fixe sa valeur.
- Cela peut se faire dès la déclaration, ou après.

- Le tout c'est de ne pas lire dans une variable avant son initialisation !
- Sa valeur est « ce qu'il y avait dans la mémoire ».
- Ça n'a aucun sens de l'utiliser, c'est une erreur.

Déclaration et initialisation de variable (2)

```
int main ()
{
  float c ;
  int i, j = 42 ;
  double b ;
  b = 0.00345 ;
  c = 3.14159 ;
  return (0) ;
}
```

- Avec ce code, la mémoire ressemble (+/−) à :



L'affectation

- Dénotée par le signe = (à ne pas confondre avec ==).
- Donne une valeur à une variable.
- Besoin de « compatibilité » entre type de la case mémoire et type de la valeur à stocker !
- Si les types sont les mêmes, aucun problème.
- Si les types sont «*compatibles* », transformation implicite :
  - Entier ↤ entier avec signe et/ou taille différents,
  - Flottant ↤ entier,
  - Entier ↤ caractère.
- . . . **mais** attention aux problèmes de précision et débordement !
- Et sinon, les types n'ont rien à voir (ex : chaîne à la place d'un entier) : erreur de typage par le compilateur.

Instructions d'affectation-arithmétique abrégées

- Il existe des versions courtes des opérations «*courantes*» :

```
int i = 0 ;

i++ ;                   // i = i + 1 ;
i-- ;                   // i = i - 1 ;
i += 3 ;                // i = i + 3 ;
i -= 3 ;                // i = i - 3 ;
i *= 3 ;                // i = i * 3 ;
i /= 3;                 // i = i / 3 ;
i %= 3 ;                // i = i % 3 ;
i |= 3 ;                // i = i | 3 ;
i &= 3 ;                // i = i & 3 ;
i ^= 3 ;                // i = i ^ 3 ;
```

Instruction conditionnelle (`if`)

- `if` (*expression*) { *instruction*(*s*) ; } :
  - *instruction*(*s*) exécutée(s) si *expression* renvoie $\neq 0$

  instruction(s) exécutée(s) si expression renvoie $\neq 0$.
- `if` (*expression*) { $instruction_1(s)$; } `else` { $instruction_2(s)$; } :
  - $instruction_1(s)$ exécutée(s) si *expression* renvoie $\neq 0$
  - sinon, $instruction_2(s)$ exécutée(s).
- Condition toujours entre parenthèses !
- Remarquez la construction { ... } : groupement de plusieurs instructions : « *bloc* ».
  - ⚠ Différent de Python où l'indentation jouait ce rôle !

« Tant que » . . . faire : la boucle `while`

- `while` (*expression*) { *instruction*(*s*) }
- Sémantique :
  1. Si *expression* s'évalue en « *vrai* »
     1.1 Exécuter *instruction*(*s*).
     1.2 Retourner en 1.
- Ex : $U_{n+1} = \begin{cases} \frac{U_n}{2} & \text{Si } U_n \text{ est pair} \\ 3 * U_n + 1 & \text{Sinon} \end{cases}$

syracuse.c

```
int main ()
{
  int x = 134560 ;
  while (x != 1) {    // Répéter tant que x est différent de 1.
    if (x % 2 == 0)   // Si x est pair.
      x = x / 2 ;     // On le divise par 2.
    else x = 3 * x + 1 ;
  }
  return (0) ;
}
```

Sortie à l'écran `printf` (1)

- Nécessite : `#include <stdio.h>`
- `printf ("Eat at Joe's\n");`
- `printf ("%d = %d + %d\n", x + y, x, y);`
- Format= chaîne de caractères contenant (ou pas) des séquences %♪.

`%d`	un `int`
`%ld`	un `long int` en décimal
`%u`	un `unsigned int` en décimal
`%x`	un `int` en hexadécimal
`%f`	un `float`
`%lf`	un `double`
`%e`	un `double` en notation scientifique
`%.7lf`	un `double` avec 7 chiffres après la virgule
`%07d`	un `int` en décimal sur 7 digits (remplissement frontal avec des 0)
`% 7d`	un `int` en décimal sur 7 digits (remplissement frontal avec des ' ')
`%c`	un `char` (comme caractère ASCII, pas comme entier)

Sortie à l'écran avec printf (2)

```
#include <stdio.h>

int main ()
{
  printf ("%d\n", 42) ;
  printf ("%ld\n", 42) ;
  printf ("%u\n", 42) ;
  printf ("%x\n", 42) ;
  printf ("%f\n", 42.0) ;
  printf ("%e\n", 42.0) ;
  printf ("%.7f\n", 42.0) ;
  printf ("%07d\n", 42) ;
  printf ("% 7d\n", 42) ;
  printf ("%c\n", 42) ;
  return (0) ;
}
```

```
42
42
42
2a
42.000000
4.200000e+01
42.0000000
0000042
     42
*
```

Approfondissement des types de base de C

Les entiers (naturels et relatifs)

- Plusieurs tailles.
- Signés ou non.
- Par défaut : un entier est signé.
- Ex : 0 -3 +0 15657

	Taille (bits)	Valeur	
		Min	Max
short	16	-32768	32767
unsigned short	16	0	65535
int	32	-2^{31}	$2^{31}-1$
unsigned int	32	0	$2^{32}-1$
long	32 ou 64	dépend de l'architecture	
unsigned long	32 ou 64	dépend de l'architecture	
long long	64	-2^{63}	$2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

Codage des entiers non signés

- Entiers codés en binaire : base 2.
  - $421 \mapsto 0000000110100101$ (sur 16 bits)
  - $= 2^8 + 2^7 + 2^5 + 2^2 + 2^0$
- Base 2 :
  - Pratique au niveau électronique mais encombrant !
  - $\Rightarrow$ On préfère la base 16 (hexadécimal).
  - 0000 0001 1010 0101

    0x 0 1 A 5

Codage des entiers signés

- ⚠ Le bit de poids le plus fort sert à représenter le signe.
  - 0 ⇒ positif.
  - 1 ⇒ négatif.
- Pour obtenir l'opposé d'un nombre : complément à 2.
  - Inverser les bits de l'écriture binaire de sa valeur absolue,
  - puis ajouter 1 au résultat.
  - Calcule $2^l - |x|$ où l est la longueur de représentation d'un entier.
  - ⇒ 1 seule représentation de 0.
  - ⇒ + et − identiques sur signés / non-signés, positifs / négatifs.
- 4 ⟶ -4 :

4 →	0	1	0	0	$(*)$
not $(*)$ →	1	0	1	1	$(**)$
$(**) + 1$ →	1	1	0	0	

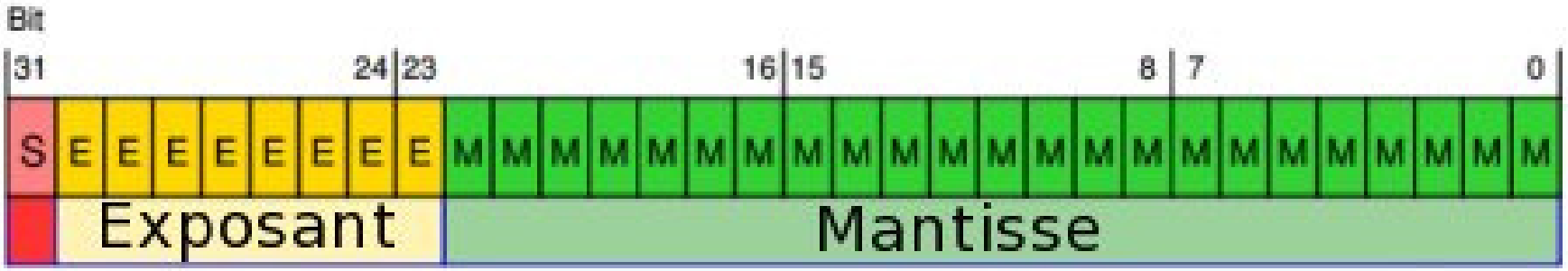
Débordement d'entiers

- Arithmétique entière et modulo la taille des mots machine.
- Pour les exemples suivants, taille = 4 bits.
- ⇒ Perte de plein de « *bonnes* » propriétés mathématiques ☹
  - Débordement de non signé : 9 + 7 = 16 ?
    $1001 + 0111 = 1|0000$ = sur 4 bits ... 0000 = 0
  - Débordement de signé : -8 - 1 = -9 ?
    $1000 - 0001 = 1|1111$ = sur 4 bits ... 1111 = -1
  - Conversion de signé → non signé : -4 → ?
    1100 = 12
  - Conversion de non signé → signé : 15 → ?
    1111 = -1
  - Associativité de $*$ et / : (1 / 4) * 4 = (1 * 4) / 4 ?
    1 / 4 = 0.25 En entiers ... 0. Donc, * 4 → 0
    1 * 4 = 4 En entiers ... 4. Donc, / 4, → 1

Les réels

- Couramment appelés « flottants » (« *floating point numbers* »).
- Toujours signés.
- 2 tailles ⇒ 2 précisions.
- Ex : 1.0 3.14159 -6.14 +1.6e-15

	Taille (bits)	Valeur Min	Valeur Max	Précision max
float	24 + 8	$1.1 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	10^{-6}
double	53 + 11	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	10^{-15}



La réalité des réels : des problèmes bien réels

- Nombres flottants pratiques pour approximer les réels mathématiques.
- Pourtant bien différents de la « beauté » mathématique :
  - Notion de précision :
    - ★ ⇒ Tous les réels ne sont pas représentables entre 2 réels.
    - ★ Il existe des « trous » entre 2 réels.
    - ★ Ex : 48431.1231 = 48431.1250 (c.f. slide suivant).
  - Comme pour les entiers : notion de réels min et max représentables.
  - Mauvais conditionnement :
    - ★ Opérations entre grand et petit flottants parfois incohérentes.
    - ★ Ex : 100.0 + 4e+15 = . . . 4e+15
  - Arrondis lors des calculs.
    - ★ Test d'égalité == proscrit.
    - ★ Test « modulo un ϵ ».
    - ★ Attention, ϵ n'est pas absolu et dépend du problème !
  - Conversion réel → entier : 0, débordement, troncature.

Des réels différents égaux...

thiskillsme.c

```
#include <stdio.h>

int main ()
{
  float fl1 = 48431.1231 ;
  float fl2 = 48431.1239 ;
  float fl3 = 48431.1250 ;

  printf ("fl1: %f  fl2: %f  fl3: %f\n", fl1, fl2, fl3) ;
  printf ("fl1 == fl2 ? %d\n", (fl1 == fl2)) ;
  printf ("fl2 == fl3 ? %d\n", (fl2 == fl3)) ;
  printf ("fl1 == fl3 ? %d\n", (fl1 == fl3)) ;
  fl1 = fl1 + 0.0001 ;
  printf ("fl1 + 0.0001: %f\n", fl1) ;
  return (0) ;
}
```

```
> ./thiskillsme
fl1: 48431.125000  fl2: 48431.125000  fl3: 48431.125000
fl1 == fl2 ? 1
fl2 == fl3 ? 1
fl1 == fl3 ? 1
fl1 + 0.0001: 48431.125000
```

. . . et des réels égaux différents

thiskillsmeagain.c

```c
#include <stdio.h>
#include <math.h>
#define EPSILON (1e-6)  // Tout petit epsilon...

int main ()
{
  double fl1 = 0.1 ;
  double fl2 = 0.2 ;
  double fl3 = 0.3 ;
  double fl4 = fl1 + fl2 ;

  printf ("fl1: %f  fl2: %f  fl3: %f  fl4: %f\n", fl1, fl2, fl3, fl4) ;
  printf ("fl3 = fl4 ? %d\n", (fl3 == fl4)) ;
  printf ("fl3 ~ fl4 ? %d\n", (fabs (fl3 - fl4) < EPSILON)) ;
  return (0) ;
}
```

```
> ./thiskillsmeagain
fl1: 0.100000  fl2: 0.200000  fl3: 0.300000  fl4: 0.300000
fl3 = fl4 ? 0
fl3 ~ fl4 ? 1
```

Les chaînes de caractères

- Caractère : type prédéfini `char` = entiers sur 8 bits.
  - Par ex : `'a'` = 97 (= 01100001 en binaire).

- Chaîne = tableau de caractères terminé par le caractère '\0'.
- Fichier d'en-tête requis : `#include <string.h>`
- Peuvent être copiées : fonction `strcpy` (*dest*, *source*).
- Longueur d'une chaîne : fonction `strlen` (*str*).
  - ⚠ Ne compte pas le '\0' final !
  - `strlen ("Damned")` $\longrightarrow$ 6.
  - Pourtant :

D	*a*	*m*	*n*	*e*	*d*	\0

Quelques formes particulières de caractères

- Utilisables en tant que caractères individuels ou dans des chaînes.
  - `'\n'` : Retour à la ligne.
  - `'\\'` : Le caractère \.
  - `'\''` : Le caractère '.
  - `'\"'` : Le caractère ".
  - `'\t'` : La tabulation.
  - `'\x'` •• : Le caractère dont le code en hexadécimal suit.
  - `'\'` ••• : Le caractère dont le code en octal suit.
  - ...et quelques autres que l'on passe sous silence.
- Ex : `printf ("foo\n\"\'bar\\\x65\046");` ⟶

```
foo
"'bar\e&
```

Plus de constructions

Faire *n* fois : la boucle `for`

- `for` (*instruction*$_1$; *expression* ; *instruction*$_2$) { *instruction*$_3(s)$ }
- ⚠ Des points-virgules ! Pas des virgules !
- Sémantique :
  1. Initialisation : exécuter *instruction*$_1$.
  2. Si *expression* s'évalue en `true`
     - 2.1 Exécuter *instruction*$_3(s)$.
     - 2.2 Exécuter *instruction*$_2$ (« post-traitement »).
     - 2.3 Retourner en 2.

```
#include <stdio.h>
int main ()
{
  int i ;
  for (i = 0; i < 10; i++) {
    printf ("%d ", i) ;
  }
  printf ("\n") ;
  return (0) ;
}
```

```
mymac:/tmp$ gcc enumerate.c -o enumerate
mymac:/tmp$ ./enumerate
0 1 2 3 4 5 6 7 8 9
```

La boucle `for` dégénérée

- Possibilité d'omettre des parties de la construction.
- Initialisation, condition, post-traitement, voire corps !
- Poussé à l'extrême : formes étranges voire illisibles ☹. À éviter !

```
i = 0 ; j = 0 ;
for (/**/; i < 10; i++) {
  j += i ;
}

j = 0 ;
for (i = 0; i < 10;/**/) {
  j += i ;
  i++ ;
}
```

```
i = 0 ; j = 0 ;
for (/**/;i < 10;/**/) {
  j += i ;
  i++ ;
}

i = 0 ; j = 0 ;
for (/**/;/**/;/**/) {
  j += i ;
  if (i == 9) break ;
  i++ ;
}
```

Équivalence entre boucles `for` et `while`

- Remplacement de `for` en `while` :

```
for (init; test; postlude) {
  instruction(s) ;
}
```

```
init ;
while (test) {
  instruction(s) ;
  postlude ;
}
```

- Remplacement de `while` en `for` :

```
while (test) {
  instruction(s) ;
}
```

```
for ( ; test; ) {
  instruction(s) ;
}
```

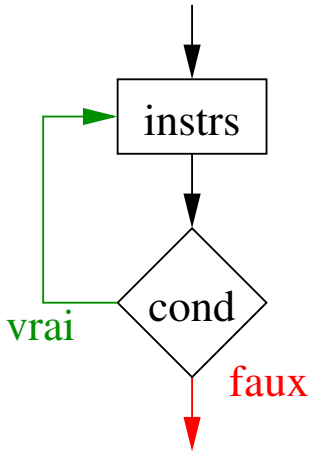
Faire « tant que » : la boucle `do-while`

- Boucle `while` avec le test à la fin du traitement.
- ⇒ Traitement toujours exécuté au moins 1 fois.
- do { *instruction*(*s*) } `while` (*expression*) ;
- ⚠ Syntaxe : n'oubliez pas le " ; " final !

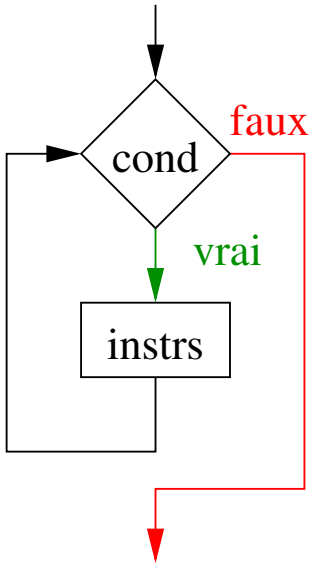
```
#include <stdio.h>

int main ()
{
  char c ;
  do {
    printf ("Entrez un chiffe: ") ;     // Lecture...
    scanf ("%c", &c) ;
  } while ((c < '0') ||  (c > '9')) ; // Jusqu'à succès.

  return (0) ;
}
```

En résumé des boucles

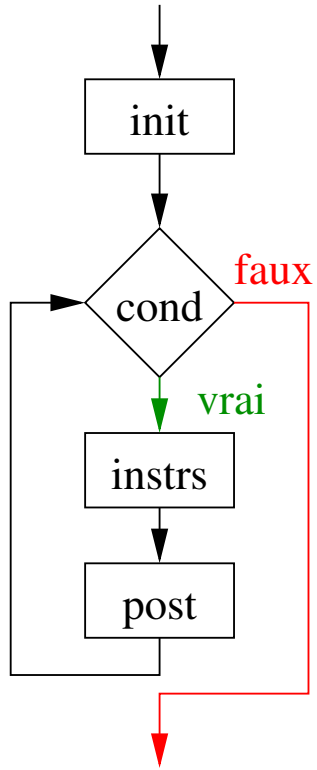
do { instrs } while (cond) ;



while (cond) { instrs }



for (init; cond; post) { instrs }



Construction conditionnelle « généralisée » (1)

- Besoin de tester une valeur contre plusieurs cas mutuellement exclusifs.
- Ex : $x == 0$ ou $x == 1$ ou $x == 4$ ou ... ou « sinon ».
- On peut cascader des `if-else`

```
if (x == 0) printf ("x: 0\n") ;
else {
  if (x == 1) printf ("x: 1\n") ;
  else {
    if (x == 4) printf ("x: 4\n") ;
    else {
      if ... else ... if ... else ...
                                ...       printf ("Autre cas\n") ;
      } } ... }
    }
  }
}
```

- Ni pratique ni lisible ☹

Construction conditionnelle « généralisée » (2)

- Discrimination par cas (sur constantes se « réduisant » à des entiers).
- ⇒ Cas doivent être mutuellement exclusifs.
- Construction `switch` (*expression*) { ... }
- Suite de `case` *constante* : *instructions*
- Les *instructions* exécutées : celles correspondant au cas où *constante* est égal à la valeur de *expression*.
- Fin de chaque cas par `break ;`
- Cas « *autre* », par défaut : `default`.

```
...
switch (x) {
  case 0: printf ("x: 0\n") ; break ;
  case 1: printf ("x: 1\n") ; break ;
  case 4: printf ("x: 4\n") ; break ;
  default: printf ("Autre cas\n") ; break ;
}
```

Quelques utilitaires (autour) de C

Constantes, macros et préprocesseur (1)

- Définir des constantes littérales dans le source : pas maintenable !
- Nécessité de changer partout la valeur.
- ⇒ Utilisation de macros.

```
#define SIZE (42)
...
int i = SIZE ;
if (j < SIZE) ...
```

- Remplacement textuel avant compilation par le préprocesseur C (cpp).
- Mieux vaut parenthéser :

```
#define SIZE 42 + 1
...
int i = SIZE * 10 ;
```

  - → i = 52 et non 430 !

Constantes, macros et préprocesseur (2)

- Possibilité de macros « paramétrées » :
  `#define TWICE(x)((x)* (x))`
- Remplacement textuel en substituant `x` par le texte argument lors de l'utilisation.
- `Bla TWICE(ah bon ?)` ⟶
  `Bla ((ah bon ?) * (ah bon ?))`
- ⚠ Syntaxe : pas d'espace entre le nom de la macro et la parenthèse ouvrante (définition et utilisation).
- Possibilité d'arguments multiples : `#define MUL(a,b)((a)* (b))`
- Bli MUL(((15)) , autre truc) ⟶
  `Bli ((((15 ))) * ( autre truc))`

Alias de types

- Vous en avez assez de taper `unsigned long int` ?
- Vous devez changer partout `int` en "unsigned int" ?
- ⇒ Définissez un alias.
- `typedef unsigned long int uli_t ;`
  - `uli_t` : nom, abréviation de `unsigned long int`.
  - Utilisable ensuite comme nom de type : `uli_t i = 42 ;`
- Ne définit pas un nouveau type !
- De manière générale : `typedef` *<type existant> <nom>*
- Fonctionnera avec les types que nous verrons plus tard.

Contrainte de type (transtypage, cast)

- Parfois besoin de changer le type selon lequel une donnée est « *vue* ».
- `float` → `int`, `unsigned int` → `long int`, ...

⇒ Utilisation d'un cast : (nom-type) expression.

```
float v = 3.14159 ;
int i ;
unsigned long j = 3000000 ;

i = (int) v ;
j = 15 * ((unsigned long) (i + 1)) + j ;
```

- ⚠ Possibilité de perte / corruption d'information ! (c.f. sides 5 et 7).
- ⚠ « *Conversion* » entier ↔ flottant : change la représentation interne.
- Autres « *conversions* » : ne changent pas la représentation, juste la façon de considérer cette représentation.

Les fonctions

- Une fonction
  - Prend une ou des valeurs en argument.
  - Retourne une valeur.
- D'où, une fonction est définie par :
  - le type de la valeur qu'elle retourne,
  - son nom,
  - les types des paramètres qu'elle attend,
  - et son corps : la description du calcul qu'elle effectue, spécification exécutable.

Exemples

```
int twice (int i)
{
  return (i * 2) ;    // Corps: retourne bien un int.
}

float square_minus (float x, float y)
{
  float t = (x - y) ;  /* Variable locale t. */
  return (t * t) ;
}
```

- ⚠ Des virgules entre les paramètres ! Pas des points-virgules !

Rappels / remarques importants

- Écrire une fonction ne l'exécute pas ! Il faut l'appeler !
- Appel : nom + arguments entre parenthèses séparés par des virgules. :

```
...
v = twice (2) ;
v = square_minus (v, -v) + 42 ;
...
```

- Corps de la fonction uniquement exécuté lors de l'appel à la fonction.
- Paramètre formel : nom donné dans la définition de la fonction : `i` pour `twice`, `x` et `y` pour `square_minus`.
- Paramètre effectif : expression donnée à un paramètre formel lors de l'appel.
- Appel :
  1. Évaluation (calcul) des expressions arguments effectifs $\rightsquigarrow$ valeurs.
  2. « Association » de chaque paramètre formel avec sa valeur.
  3. Calcul du corps de la fonction sous ces hypothèses d'association.
  4. Transmission à l'appelant de la valeur retournée par ce corps.

Les fonctions ne retournant « *rien* »

- Si pas d'argument : absence dans la définition, absence dans l'appel.
- Fonction retournant « *rien* » :
  - « *Retourne* » le type `void`.
  - Aucune expression après l'instruction `return`.
  - Dernière instruction `return` optionnelle.
  - `return` peut être nécessaire si plusieurs points de terminaison de la fonction.

```
int const_fun ()
{
  return (42) ;
}

void just_print (int i)
{
  printf ("I just print %d and %d\n", i, const_fun ()) ;
  return ;
}
```

Variables locales, variables globales

- On peut déclarer des variables locales dans une fonction.
- Plus généralement : déclaration de variables locales dans des blocs ({ ... }).
- Corps de la fonction ≡ bloc.
- ⇒ Vivantes seulement jusqu'à la fin de la fonction / du bloc.
- Variables globales (définies hors de fonctions) restent accessibles dans la fonction.

```
int glob = 5 ;          /* Variable globale. */

int f (int x)
{
  int i = x * 2 ;       /* Variable locale à la fonction f. */
  {
    int j = i + 5 ;     /* Variable locale au bloc interne à f. */
    i = i + j ;
  }                     /* Fin du bloc: j n'est plus accessible. */
  return (i + 3 + glob) ;
}
```

Déclaration / prototype (1)

- Utilisation d'une fonction en dehors de son fichier hôte ?
- Utilisation d'une fonction avant sa définition ?

⇒ Son type doit être connu.

- Donner juste le type : déclaration ou « *prototype* » (et non définition).
- Rend visible par toute personne ayant accès à cette déclaration.
- Règle valable pour les autres types de variables.
- Déclaration d'une entité : son nom + son type.
- Déclaration :
  - Faite au « *toplevel* » d'un fichier d'en-tête (`.h`).
  - Utilisée via les directives `#include`.

Déclaration / prototype (2)

- Pour une fonction :
  - son type de retour,
  - son nom,
  - les types (et les noms optionnellement) de ses arguments.
- Pour une variable :
  - le mot clef `extern` (sinon ça serait une définition de variable !),
  - son type,
  - son nom.

```
/* Déclaration d'une variable (globale) entière. */
extern int errno ;
/* Déclaration de la fonction arc-tangente de y/x. */
float atan2 (float y, float x) ;
```

- Indique « comment » utiliser la fonction.
- . . . Mais pas ce qu'elle fait (et surtout pas comment).

Quelques fonctions déjà vues

- `printf` : affichage à l'écran.
  - Prototype un peu bizarre car nombre et types des arguments variables.
  - `void printf (const char *restrict format, ...);` ☹
- `main` : **La** fonction point d'entrée de tout programme.
  - `int main ();`
    - ★ Retourne un entier : le code de retour du programme.
    - ★ Ne prend pas d'argument.
  - `int main (int argc, char *argv[]);`
    - ★ Forme alternative : arguments en provenance de la ligne de commande.
    - ★ `argc` : nombre d'entrées dans `argv`.
    - ★ `argv` : tableau de chaînes de caractères.
    - ★ 1er élément : nom + chemin de l'excutable.
    - ★ Éléments suivants : arguments effectifs.
    - ★ Ex : `> ./myprog 1 hop 4`

$\Rightarrow$ `argv` =

0	1	2	3
"*myprog*"	"1"	"*hop*"	"4"

Une autre fonction : entrée au clavier avec `scanf` (1)

- Nécessite : `#include <stdio.h>`
- Lecture au clavier selon le format spécifié (séquences %♪).
- Si saisie pas en accord avec ce qu'attend le format : imprévisible.
- Format avec uniquement des % (pas de *« message »*).

input.c

```
#include <stdio.h>

int main ()
{
  int i, j ;
  scanf ("%d %d", &i, &j) ;
  printf ("i = %d, j = %d\n", i, j);
  return (0) ;
}
```

```
mymac:/tmp$ gcc input.c -o input
mymac:/tmp$ ./input
45 67
i = 45, j = 67
mymac:/tmp$ ./input
5
FHG
i = 5, j = 0
mymac:/tmp$ ./input
DFG 6
i = 0, j = 0
```

Entrée au clavier avec `scanf` (2)

- Pas de :
  `scanf ("Entrez l'âge du capitaine : %d\n", &age);`
- `scanf ("%d", &i) ;` 
- Pour le moment, `&` reste un peu « magique ».
- Sera approfondi ultérieurement.
- Idée : « à mettre devant une variable pour permettre sa modification par la fonction appelée (ici, `scanf`) ».
- Rem : la lecture de chaînes de caractères (%s) nécessite quelques explications ultérieures. On vivra sans pour le moment.

Appel(s) de fonction(s)

Déroulement d'un appel de fonction

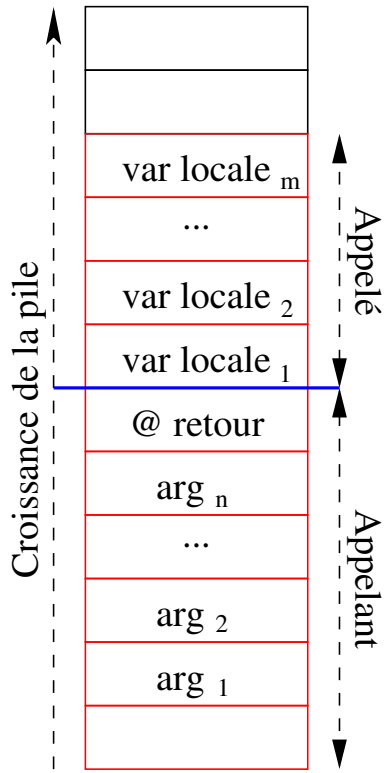
- Illustration sur le calcul de la racine carrée d'un réel a.
- Méthode itérative de Newton.
- $\sqrt{a}$ donnée par $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$
- n : Nombre d'itérations (à fixer arbitrairement).
- $x_0 > 0$: Point de départ (à fixer arbitrairement).

Protocole d'appel

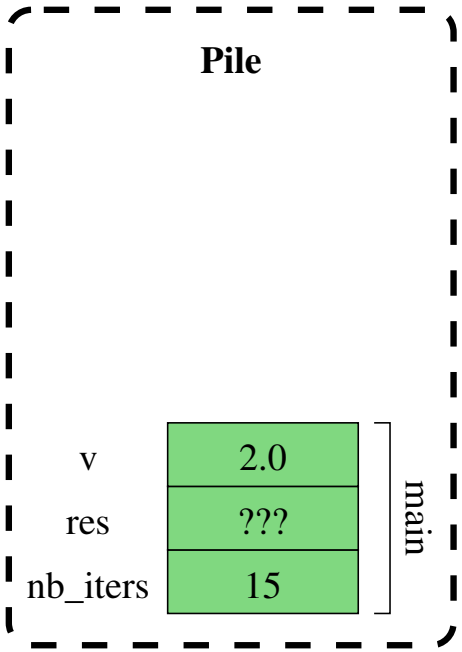
- Appeler une fonction c'est :
  - stocker la valeur de ses paramètres,
  - aller exécuter son code,
  - et en revenir avec le résultat.
- Différentes conventions d'appel selon les langages, les architectures.
- Principe restant majoritairement le même : utilisation de la pile.
- Différences dans « qui (appelant/appelé) fait quoi ».
- Différences dans l'agencement mémoire d'informations (registres vs pile).

Protocole d'appel de manière générique

- Côté appelant :
  - Évaluation des arguments à passer → mis sur la pile.
  - Adresse de retour mise sur la pile.
- Côté appelé :
  - Réservation des variables locales sur la pile.
  - Exécution du code.
  - Stockage de la valeur à retourner (en registre ou pile).
  - Aller . . . à l'adresse de retour.
- Côté appelant :
  - Récupérer la valeur retournée par l'appelé.
  - Supprimer de la pile les arguments passés lors de l'appel.



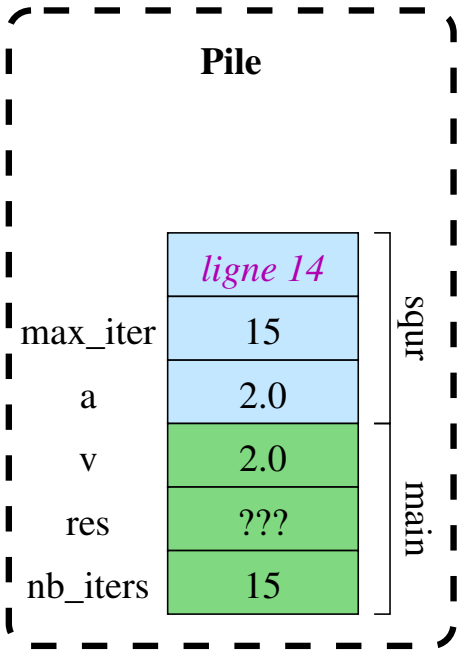
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15 ► double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Définition et initialisation des variables locales de `main`

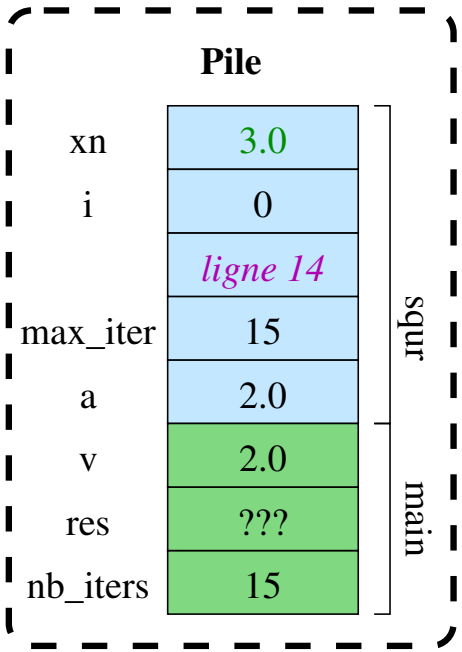
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16 ► res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Appel de la fonction squr

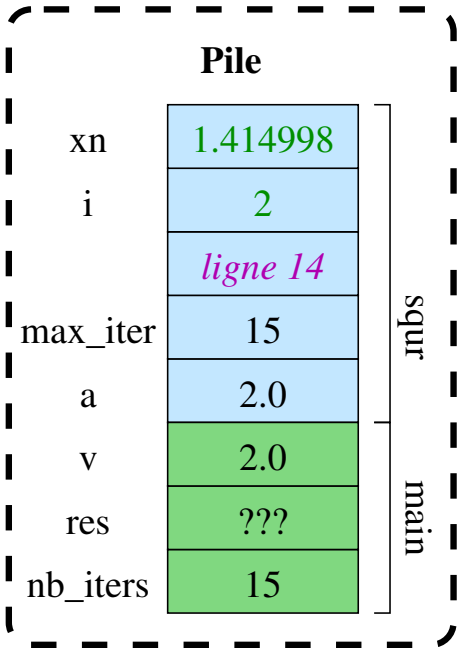
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5  ► double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Définition et initialisation des variables locales de `squr`

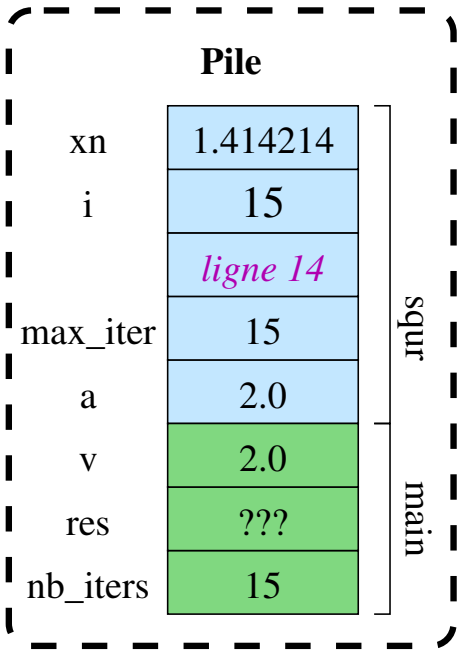
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8  ►   i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Itérations de la boucle (ici $i = 2$, $xn = 1.41499$)

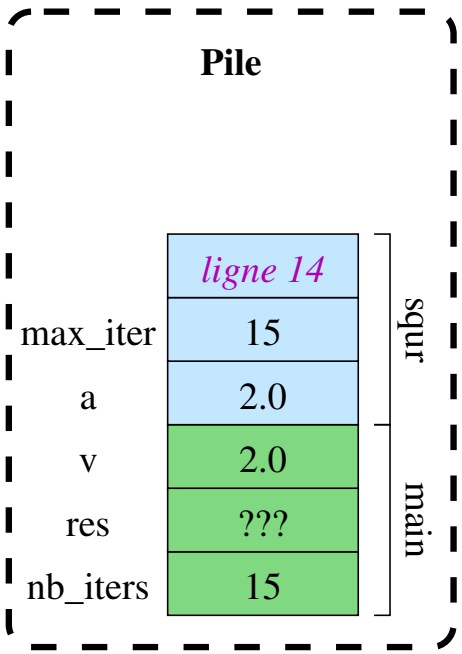
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10 ► return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Sortie de la boucle, prêt à retourner de la fonction squr

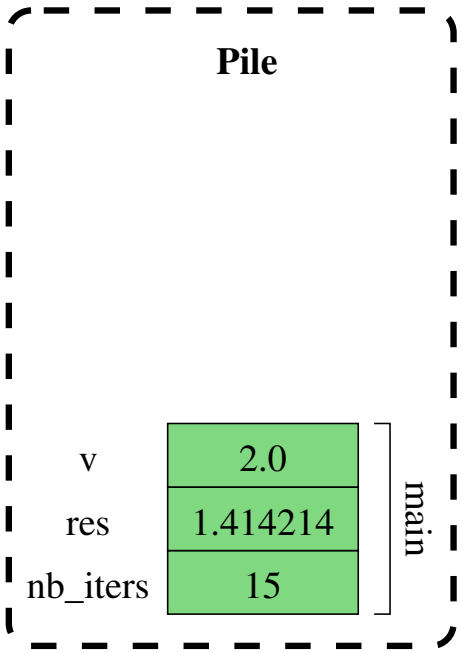
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 ►}
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Retour de la fonction : suppression des variables locales, retourner à l'adresse spécifiée sur la pile

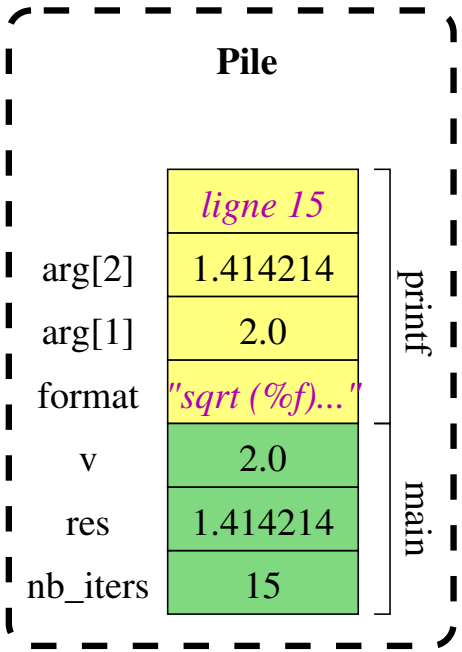
Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16 ►  res = squr (v, nb_iters) ;
17   printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Affectation de la valeur de retour de la fonction squr

Déroulement d'un appel de fonction




```
1  double X0 = 3.0 ; // Germe quelconque > 0
2
3  double squr (double a, int max_iter) {
4    int i = 0 ;
5    double xn = X0 ;
6    while (i < max_iter) {
7      xn = 0.5 * (xn + (a / xn)) ;
8      i++ ;
9    }
10   return (xn) ;
11 }
12
13 int main () {
14   int nb_iters = 15 ;
15   double res, v = 2.0 ;
16   res = squr (v, nb_iters) ;
17 ► printf ("sqrt (%f) = %f\n", v, res) ;
18   return (0) ;
19 }
```

Appel à la fonction `printf` ... etc ...

La récursivité

Fonctions (mutuellement) récursives

- En C, (comme dans beaucoup de langages) : toutes les fonctions sont implicitement mutuellement récursives (si besoin).
- Mais la première définie doit *« connaître »* la (les) suivante(s).

⇒ Utilisation de déclarations.

```
#include <stdbool.h>
bool natural_even (unsigned int n) ;   /* Décl. de natural_even. */
bool natural_odd (unsigned int n) ;    /* Décl. de natural_odd. */

bool natural_even (unsigned int n)     /* Déf. de natural_even. */
{
  if (n == 0) return (true) ;
  else return (natural_odd (n - 1)) ;
}

bool natural_odd (unsigned int n)      /* Déf. de natural_odd. */
{
  if (n == 0) return (false) ;
  else return (natural_even (n - 1)) ;
}
```

« Récursion, boucles, ça se termine quand ? »

- Pour obtenir un résultat, un calcul doit terminer.
- Boucle avec condition d'arrêt toujours fausse ⇒ boucle « infinie ».
- Récursion avec cas de base (≡ condition d'arrêt) jamais atteint ⇒ récursion « infinie ».
- Dans ces cas . . . on peut attendre longtemps.
- La terminaison est une propriété très importante.

Comment s'assurer de la terminaison ?

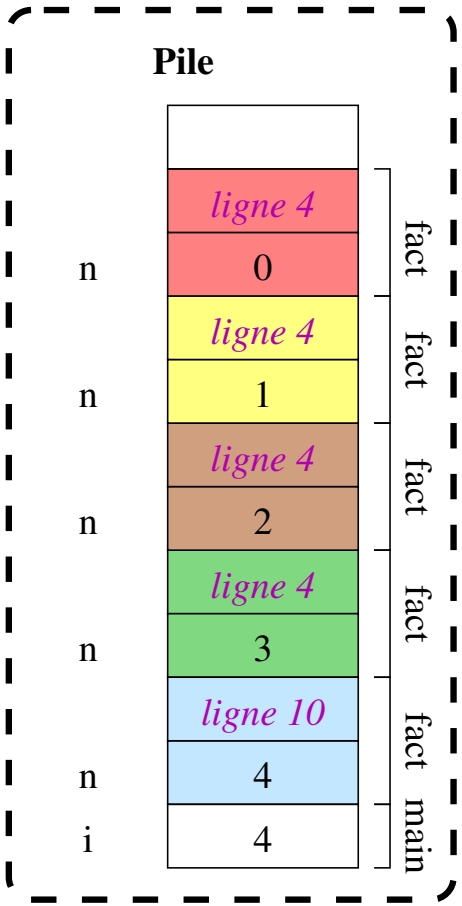
- Pas de méthode automatique.
- Nécessite une preuve de terminaison.
- $\Rightarrow$ Problème de raisonnement, pas de calcul.
- Intuition : démontrer que chaque appel récursif s'effectue sur un argument « plus petit »
  - $\Rightarrow$ Nécessité d'un ordre $<$ sur le domaine des arguments.
  - Ordre bien fondé : $\nexists$ suite décroissante infinie.
- $\forall n \in \mathbb{N}$, `fact (n) : si n == 0 alors 1 sinon n * fact (n - 1)`
  Facile : appel récursif sur $n-1$ qui est bien $< n$.
- $$Ack(m,n) = \begin{cases} n+1 & \text{Si m=0} \\ Ack(m-1,1) & \text{Si } m>0 \text{ et } n=0 \\ Ack(m-1,Ack(m,n-1)) & \text{Si } m>0 \text{ et } n>0 \end{cases}$$
  Nettement moins évident ! Mais elle termine (ordre lexicographique).

Appels de fonctions récursives

- Principe d'appel identique aux autres fonctions.
  - À chaque appel on empile les informations nécessaires.
  - Lorsque l'on sort d'un appel, on dépile.
- $\Rightarrow$ La pile augmente avec la profondeur de récursion.
- En cas de récursion trop profonde, possibilité de dépassement de pile (« *stack overflow* »).
- Pour autant la récursivité n'est pas à bannir !
  - Permet d'exprimer simplement et élégamment certains algorithmes.
  - Incontournable pour certains problèmes (sinon, obligation de se gérer manuellement une pile en fait).
  - Les compilateurs savent dé-récursiver la récursion dite terminale.

Appels de fonction(s) récursive(s)

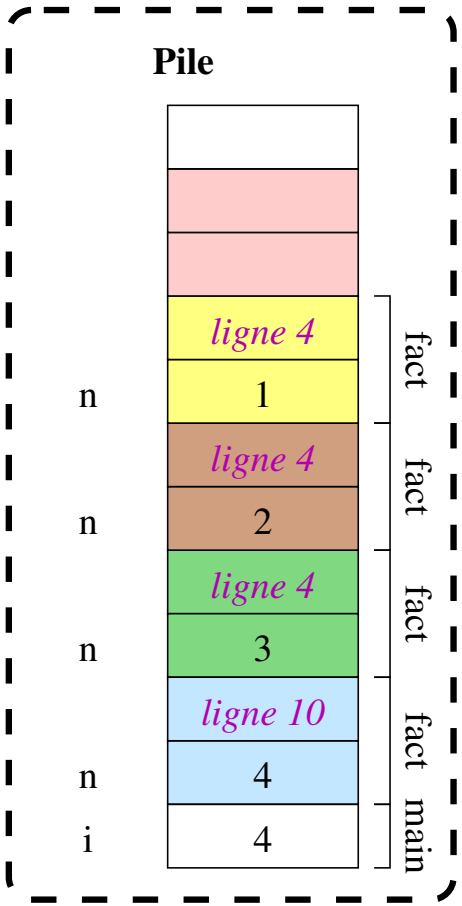
Déroulement d'appels de fonction récursive




```
1  ▶unsigned int fact (unsigned int n)
2  {
3    if (n == 0) return (1) ;
4    else return (n * fact (n - 1)) ;
5  }
6
7  int main ()
8  {
9    unsigned int i = 4 ;
10   printf ("Fact(%d)=%d\n", i, fact (i)) ;
11   return (0) ;
12 }
```

État de la pile à l'entrée de l'appel à `fact` avec l'argument 0.

Déroulement d'appels de fonction récursive




```
1  unsigned int fact (unsigned int n)
2  {
3    if (n == 0) return (1) ;
4  ► else return (n * fact (n - 1)) ;
5  }
6
7  int main ()
8  {
9    unsigned int i = 4 ;
10   printf ("Fact(%d)=%d\n", i, fact (i)) ;
11   return (0) ;
12 }
```

État de la pile au retour de l'appel à `fact` avec l'argument 0.

Les tableaux

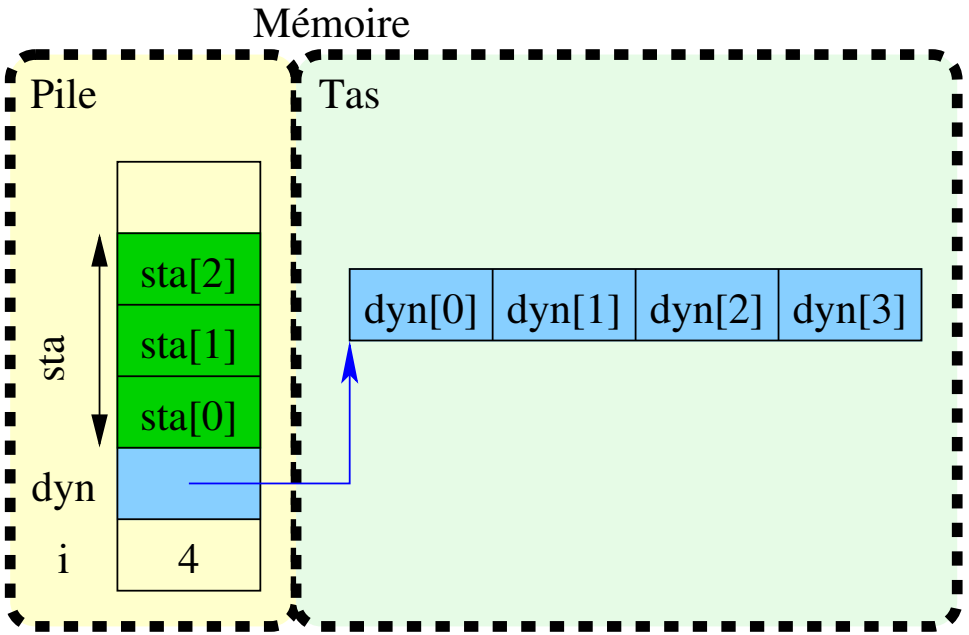
« *Rappels, remarques* »

- Tableau =
  - ensemble de « *cases* » mémoire consécutives,
  - du même type,
  - dénoté par un seul nom de variable.
- Accès à 1 case particulière (« élément ») par indexation : `t[3]`.
- ⚠ Contrairement à Python, les indices sont uniquement des entiers.
- Le type d'un élément détermine la taille d'une « case ».
- ⇒ Si on connaît où se trouve la $1^{\text{ère}}$ « *case* », (« base ») on trouve facilement l'endroit où se trouve la $i^{\text{ème}}$ « *case* » en mémoire.
- Numérotation des cases (« indices ») commence à 0 !
- Élément$_i$ à l'octet : « base » + i × taille d'un élément.

⇒ Accès aux éléments rapide (tps constant).

« Tableaux et mémoire »

- Deux sortes de tableaux :
  - Statiques : taille connue à la compilation.
    → gérés à la compilation comme des variables « *standard* ».
  - Dynamiques : taille déterminée à l'exécution.
    ⇒ Nécessite une fonction d'allocation de mémoire.
    ⇒ Cours ultérieur : uniquement des tableaux statiques pour commencer.




```
int i ;
/* Blabla ... i ← 4 */
/* Dynamique. */
int dyn[] = «allouer i cases»
/* Statique. */
int sta[3] ;
```

« Déclaration de tableaux statiques en C »

- *type nom* [*taille*] ;
- La *taille* doit être connue à la compilation : constante entière littérale.
- `int t[5] ;` *t* est un tableau de 5 entiers.
- On préfère nommer la taille à mettre un littéral : plus facile pour changer la taille et les traitement qui en dépendent.
  - ▶ Utilisation d'un `#define`.
  - → On change juste la valeur d'initialisation du define.

```
#define SIZE (42)
int tab[SIZE] ;
for (i = 0; i < SIZE; i++) blabla (tab[i]) ;
```

Je ne veux PAS voir...

```
int f (...)
{
  int size ;
  ...
  size =
    truc + machin * fact (bidule + chose) ;
  int tab[size] ;
  for (...) tab[i] = blabla ;
  ...
}
```

NON !

Initialisation de tableaux

- Lorsqu'on déclare un tableau, il n'est pas initialisé.
- Il contient *« ce qu'il y avait dans la mémoire »*.
- Comme pour les autres variables, il faut l'initialiser :
  $\Rightarrow$ donner une valeur à chaque case.
- Initialisation case par case :

```
float t[3] ;
t[0] = 1.0 ;
t[1] = 1.5 ;
t[2] = 2.0 ;
```

- Tableaux statiques : on peut initialiser d'un coup à la définition :

```
float t[3] = { 1.0, 1.5, 2.0 } ;
```

- `t = { 2.0, 4.7 } ;` ne marche pas !

Parcours de tableaux

- Un tableau étant une zone contiguë indexée, il est naturel d'utiliser une boucle pour le parcourir.
- Boucle sur les indices de 0 à *« taille du tableau −1 »*.

```
int t[SIZE] ;
int i = 0 ;
while (i < SIZE) {
  do_something (t[i]) ;
  i++ ;
}
```

- Exemple : initialisation

```
int t[SIZE] ;
int i ;
for (i = 0; i < SIZE; i++) t[i] = i * i ; /* t[i] = i^2. */
```

- Boucle `for` très pratique !

Petit retour sur les chaînes de caractères `string`

- En C : chaîne = un tableau de caractères. . .
- . . . terminé par le caractère '\0'.

str.c

```
#include <stdio.h>
int main ()
{
  int i ;
  char foo[] = "stupefix" ;
  for (i = 0; foo[i] != '\0'; i++)
    printf ("%c ", foo[i]) ;
  printf ("\n") ;

  return (0) ;
}
```

```
mymac:/tmp$ gcc str.c -o str
mymac:/tmp$ ./str
s t u p e f i x
```

Tableaux à plusieurs dimensions

- Une image est une partie de plan : coordonnées x et y.
- Espace à 2 dimensions $\Rightarrow$ structure 2 D $\Rightarrow$ tableau à 2 dimensions.
- *type nom* `[` $taille_1$ `][` $taille_2$ `]` `;`
- Accès par indexation sur les 2 dimensions : `t[x][y] = ... ;`
- Généralisable à n dimensions : `int t[3][2][6][7][4] ;`
- Tailles de tableaux statiques : connues à la compilation.
- Tableaux dynamiques : subtilités d'initialisation, on verra plus tard.

Tableaux statiques à plusieurs dimensions

st_array.c

```
#include <stdio.h>
#define X (4)
#define Y (3)

int main ()
{
  int x, y ;
  int c[X][Y] ; /* Statique/ */

  for (x = 0; x < X; x++) {
    for (y = 0; y < Y; y++)
      c[x][y] = x + y ;
  }

  for (x = 0; x < X; x++) {
    for (y = 0; y < Y; y++)
      printf ("%d ", c[x][y]) ;
    printf ("\n") ;
  }
  return (0) ;
}
```

```
mymac:/tmp$ gcc st_array.c -o st_array
mymac:/tmp$ ./st_array
c:
0 1 2
1 2 3
2 3 4
3 4 5
```

Un tableau bien utile : argv du main

- Rappel du prototype : `int main (int argc, char *argv[])`
- `argv` : Tableau contenant les chaînes de caractères passées sur la ligne de commande lors du lancement du programme.
- Permet de dialoguer avec l'entrée du programme dès son lancement.
- Contient toujours au moins 1 chaîne : `argv[0]` = nom du programme.

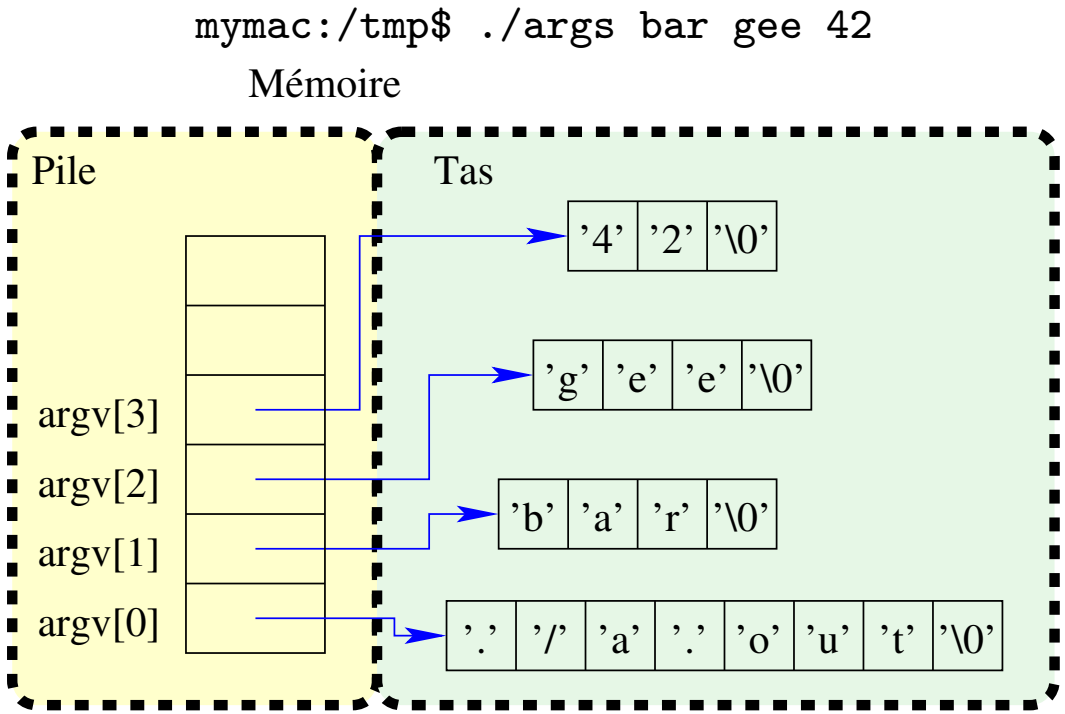
args.c

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i ;
  for (i = 0; i < argc; i++)
    printf ("i=%d  %s\n", i, argv[i]) ;
  return (0) ;
}
```

```
mymac:/tmp$ dmd args.d
mymac:/tmp$ ./args bar gee 42
i=0  ./args
i=1  bar
i=2  gee
i=3  42
```

Quelques remarques sur argv (1)

- Déterminer le nombre d'éléments pour savoir jusqu'où aller dans argv.
- → Utilisation de la valeur de argc.



Quelques remarques sur `argv` (2)

- `argv` «*contient* » uniquement des chaînes de caractères.
- On peut parfois recevoir des nombres en arguments.
- ⇒ Nécessité de conversion chaîne → nombre.
- Nécessite une transformation algorithmique, et non un changement «simple » de regarder la zone mémoire !
- Fonctions à disposition (requièrent `#include <stdlib.h>`) :
  - `atoi` string → int
  - `atol` : string → long int
  - `atoll` : string → long long int
  - `atof` : string → float

Quelques remarques sur argv (3)

string-to-nums.c

```c
#include <stdio.h>
#include <stdlib.h>  /* Pour accéder à atoXXX */
int main (int argc, char *argv[])
{
  int i = atoi (argv[1]) ;
  long l = atol (argv[2]) ;
  long long ll = atoll (argv[3]) ;
  float f = atof (argv[4]) ;
  printf ("i: %d, l: %ld, ll: %lld, f: %f\n", i, l, ll, f) ;
  return (0) ;
}
```

```
mymac:/tmp$ gcc -Wall string-to-nums.c
mymac:/tmp$ ./a.out 1 2 4559924234322424 4.5
i: 1, l: 2, ll: 4559924234322424, f: 4.500000
```

Structures de données élémentaires

Le besoin

- Vus jusqu'à présent :
  - Scalaires (entiers, flottants, caractères),
  - Chaînes de caractères,
  - Tableaux.
- Comment modéliser une localisation GPS : 48°42'39.1"N 2°13'09.4"E ?
  - Orientation de latitude (Nord ou Sud),
  - degrés et minutes de latitude (entiers),
  - secondes et fractions de latitude (flottant),
  - orientation de longitude (Est ou Ouest),
  - etc comme pour la latitude...
- $\Rightarrow$ Besoin de 2 nouveaux types de données.

Types énumérés (1)

- Une orientation de latitude c'est « *Nord* » ou « *Sud* »
- . . . et c'est tout !
- Besoin de valeurs « *atomiques* » disjointes (somme disjointe).
- Encoder par des entiers ?
  - `int Nord = 30, Sud = 17 ;`

  ou bien `#define NORD (30)`... `#define SUD (17)`.

- La valeur entière importe peu ⇒ pourquoi devoir la spécifier ?
- Spécifier les valeurs : source d'erreur si on ré-attribue la même valeur :
  - `int Nord = 1, Sud = 01`

Types énumérés (2)

- Déclaration : `enum nom-type { val1 , val2 ... };`
- Déclaration d'une variable : `enum` nom-type nom-variable ;
- Utilisation des valeurs : leur nom. I.e : `val1`, `val2` ...

```
enum lat_orient_t { La_North, La_South } ;
enum long_orient_t { Lo_West, Lo_East } ;

enum long_orient_t invert (enum long_orient_t o)
{
  enum lat_orient_t res  ;
  switch (o) {
  case Lo_West: res = Lo_East ; break ;
  case Lo_East: res = Lo_West ; break ;
  default: break ;      // Ne devrait jamais se produire.
  }
  return (res) ;
}
```

Les structures (1)

- Besoin d'agréger des données de types différents.
- Tableaux ? → **NON** car agrègent uniquement des données de même type.

⇒ Structure : groupement de données par champs nommés.

- Un peu comme les classes en Python, mais sans méthodes/fonctions.

```
enum lat_orient_t { La_North, La_South } ;

struct gps_loc_t {
  enum lat_orient_t lao ;
  unsigned int la_deg, la_min ;
  float la_sec ;
  enum long_orient_t loo ;
  unsigned int lo_deg, lo_min ;
  float lo_sec ;
};
```

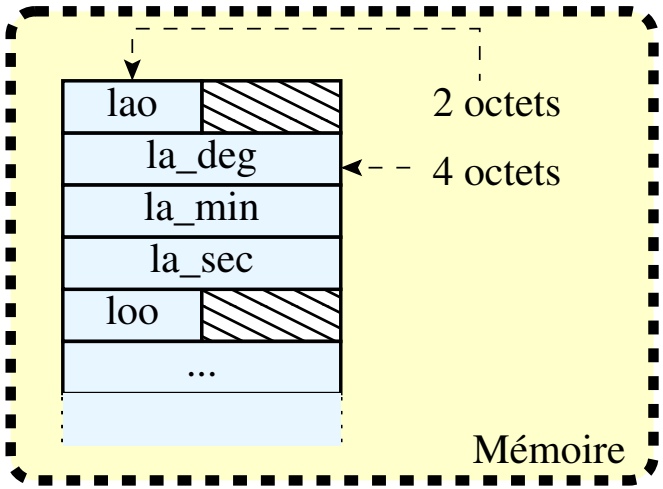
Les structures (2)

- Déclaration de variable : `struct` nom-type nom-variable ;
  - `struct gps_loc_t somewhere ;`
- Initialisation à la déclaration : énumération dans l'ordre entre accolades et séparées par des virgules :
  - ```
    struct gps_loc_t here =
      { North, 48, 42, 39.1, East, 2, 13, 9.4 } ;
    ```
- Accès à une donnée par nom de champ :
  - Notation pointée si : `struct gps_loc_t there` :
    ```
    printf ("%d, %d\n", there.la_deg, there.la_min);
    ```
  - Notation fléchée si : `struct gps_loc_t` *there :
    ```
    printf ("%d, %d\n", there->la_deg, there->la_min);
    ```
- Nommage des champs :
  - ⇒ Facilité d'accès aux différentes parties de la structure.
  - ⇒ Indépendance par rapport à l'organisation en mémoire.
    ```

Les structures (3)

- Champs stockés de manière (presque) contiguë en mémoire.
- Attention : il peut y avoir des « *trous* » entre les champs
  - ▶ Contraintes d'architecture matérielle, de performance, etc.

  ⇒ Ne pas s'appuyer sur l'agencement effectif mémoire.



- Les structures sont gérées comme les autres types :
  - Peuvent être des variables locales (détruites en fin de portée).
  - Recopiées lorsque passées en arguments de fonction.
  - ⇒ Pour de grosses structures, coût de recopie et coût de pile !
    - ▶ Les passer comme argument par adresse (cours ultérieur).

Structuration, **modularité** et abstraction

- Besoin de rendre le logiciel robuste
  - Aux changements de structure / implémentation interne.
  - Aux modifications sauvages mettant en péril des invariants.
- Différentes formes de structures de données vues. . .
  - ▶ Utiliser la plus adaptée au modèle à implémenter !
  - ▶ Ex : `struct` à 4 champs `nord`, `sud`, `est`, `ouest` versus tableau de taille fixe = 4.
- Modularité : répartir dans des fichiers différents les traitements sur des concepts différents.
  - ▶ Projet réaliste = plusieurs $10(0)^{\text{aines}}$ de milliers de lignes
  - ▶ ⇒ Tout n'est pas dans le même fichier.
  - ▶ ⇒ Répartition en fonction des sous-problèmes.
- Abstraction : ne montrer *« à l'extérieur »* que le minimum.
  - ▶ ⇒ Ne mettre dans les `.h` que ce qui est nécessaire.

Organisation mémoire et pointeurs

Organisation de la mémoire

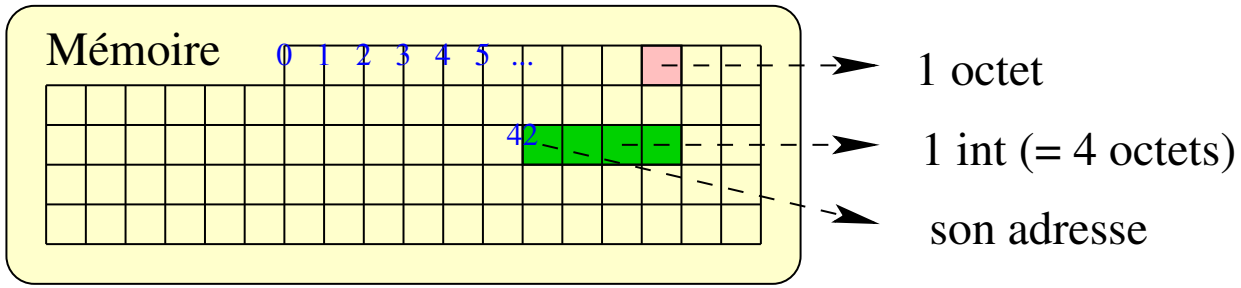
- Pour le processeur : mémoire = grand tableau fini d'octets :
  - de taille 2^{32} sur machine 32 bits,
  - de taille 2^{64} sur machine 64 bits.
- Tout le « *tableau* » n'est pas utilisable :
  - Une partie de la mémoire est utilisée par les autres processus.
  - Certaines zones sont réservées au système.
  - Certaines zones peuvent être restreintes en accès (lecture/écriture/exécution).
  - Certaines zones peuvent être « *non connectées* » (pas de mémoire physique présente).

Écologie et politesse en mémoire

- Ressource en quantité finie ⇒ gestion de ressource :
  - En demander au besoin : allocation.
  - La restituer : libération (« *désallocation* »).
- Arbitrage et autorisation d'accès gérés par le système d'exploitation (« *OS* »).
- Allocation et libération via des appels systèmes.
- Accès à une zone non attribuée au processus → « *segmentation fault* ».
- ⇒ Toujours avoir demandé la mémoire que l'on veut utiliser.

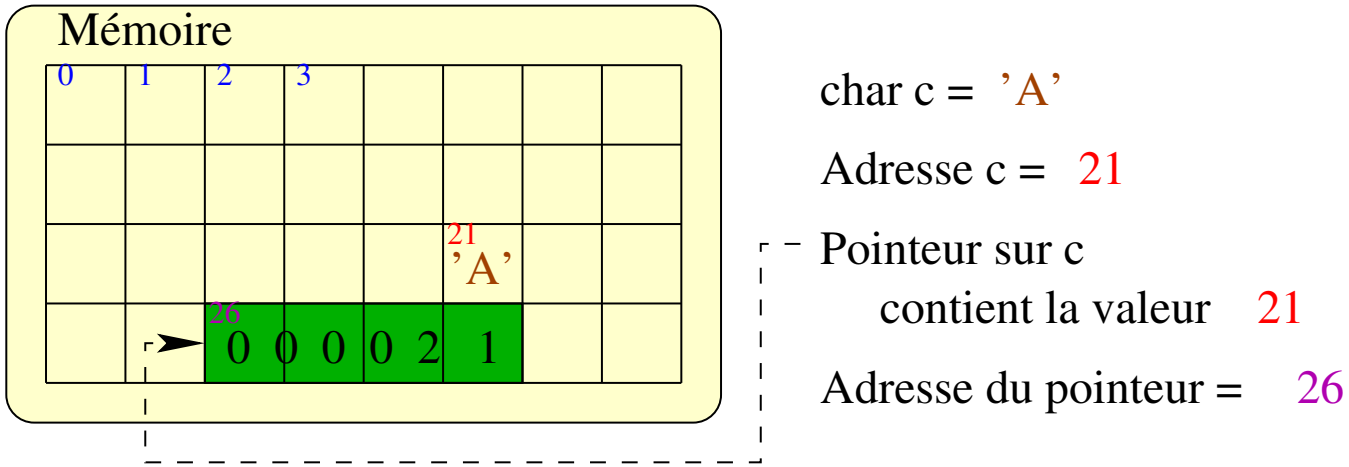
Adresse mémoire

- Les informations, octets, `int`'s, chaînes sont stockées en mémoire.
- ⇒ Sont logées à un « *certain endroit* » de la mémoire : adresse.
- Adresse ≈ index de l'info dans le grand « *tableau* » de la mémoire.



Adresses et pointeurs (1)

- Une adresse est une information comme une autre.
- ⇒ Besoin de la mémoriser, manipuler et d'accéder à son contenu.
- Une valeur « *adresse* » est un pointeur.
- Une variable contenant une adresse est de type pointeur.



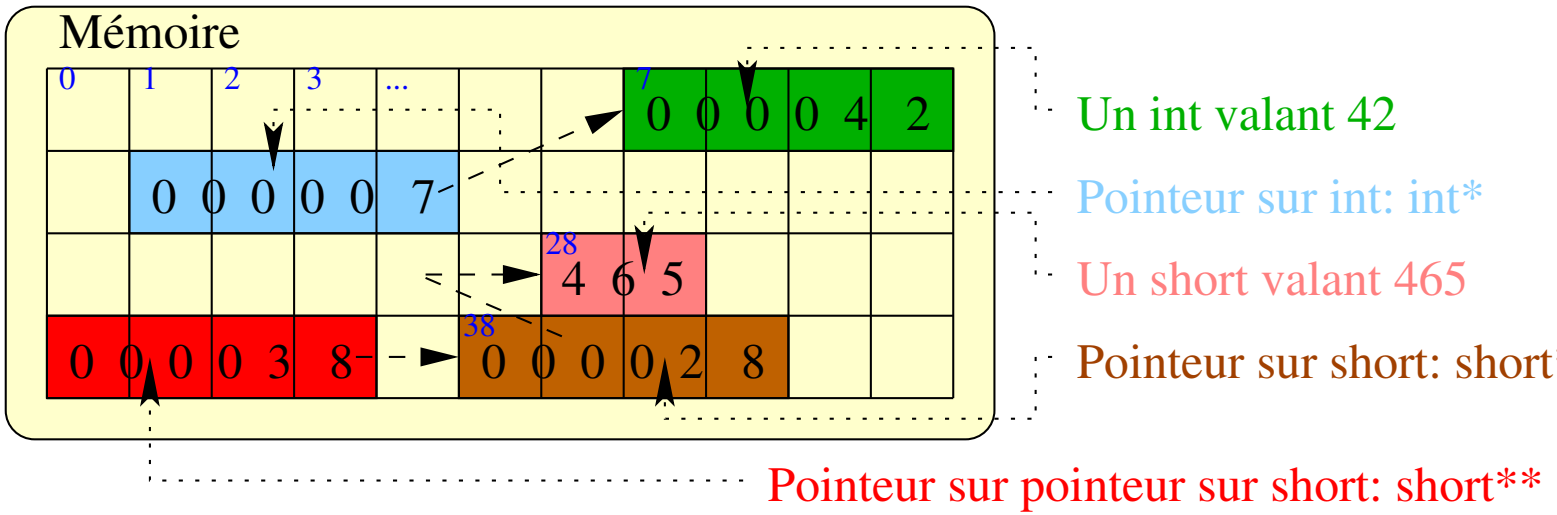
- À une adresse se trouve une donnée d'un certain type.
  ▶ ⇒ Un pointeur est une adresse avec un type associé : le type des données stockées à cette adresse.

Adresses et pointeurs (2)

- Un pointeur désigne un point dans la mémoire :
  - emplacement d'une donnée,
  - ou emplacement du début d'une zone contiguë de données.

- Déclaration de pointeurs en C :

```
int *a ;          // a est un pointeur vers un/des int.
short *b ;        // b est un pointeur vers un/des short (int).
short **c ;       // c est un pointeur vers un/des short*.
```



Adresses et pointeurs (3)

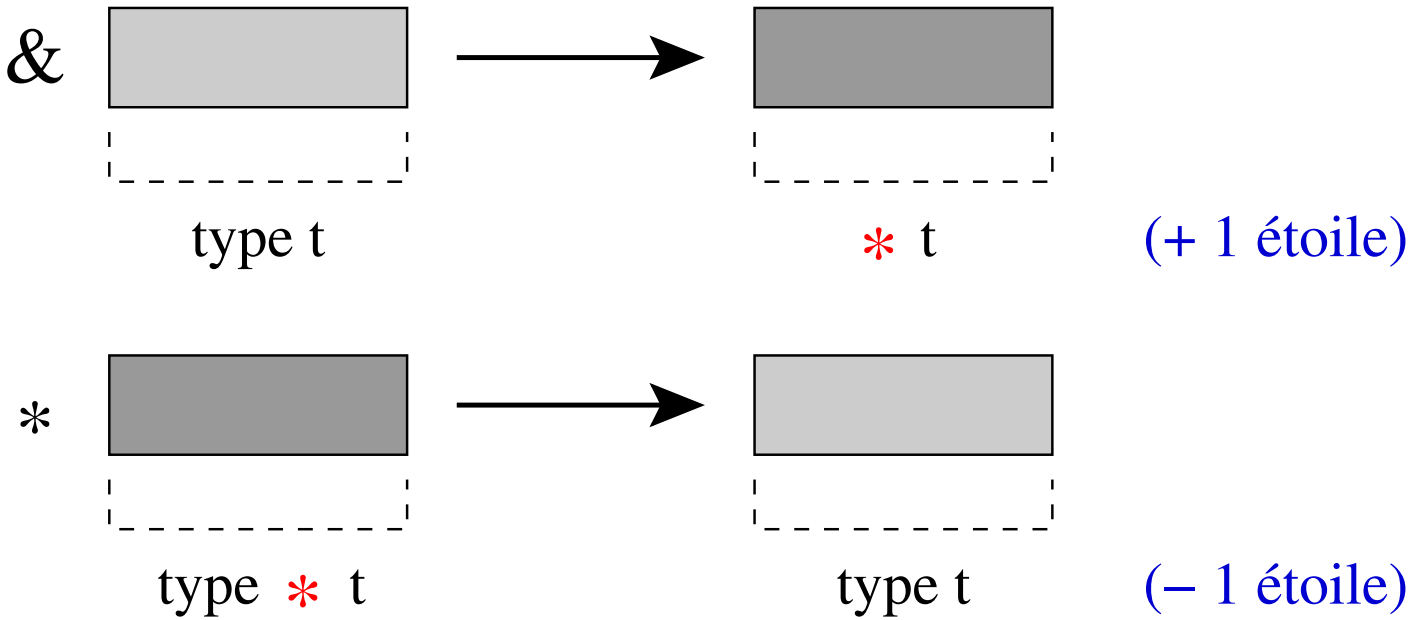
- Chaque variable possède une adresse.
- Obtenir l'adresse d'une variable : opérateur préfixe &.
- Accéder au contenu d'une adresse : opérateur préfixe*.

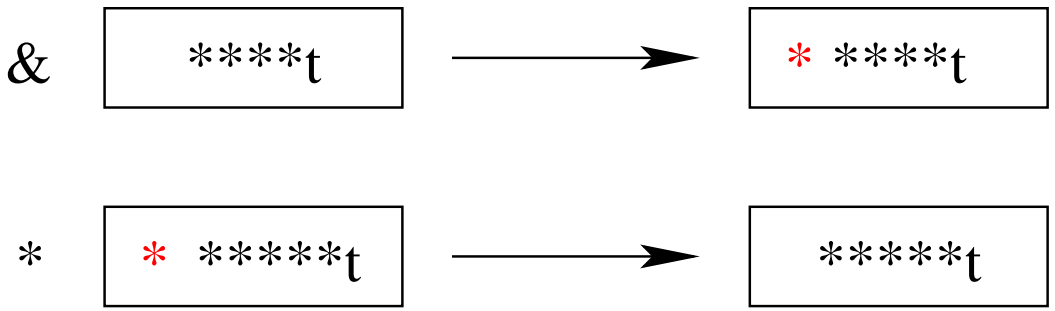
```
int a ;          // a est un int.
int *b ;         // b est un pointeur vers un int.
b = &a ;         // b contient l'adresse de a -> Pointeur vers a.
*b = 5 ;         // On stocke 5 dans *b. -> Dans a.
a = 1 + *b ;     // On stocke dans a 1 + la valeur contenue à
                 // l'adresse b. -> a = a + 1.
```

- & et * ont des rôles symétriques : *(&a) est la même chose que a.

ATTENTION : avoir un pointeur sous la main **ne** signifie **pas** avoir le droit de manipuler la mémoire à l'adresse contenue par ce pointeur !

Adresses et pointeurs (4)





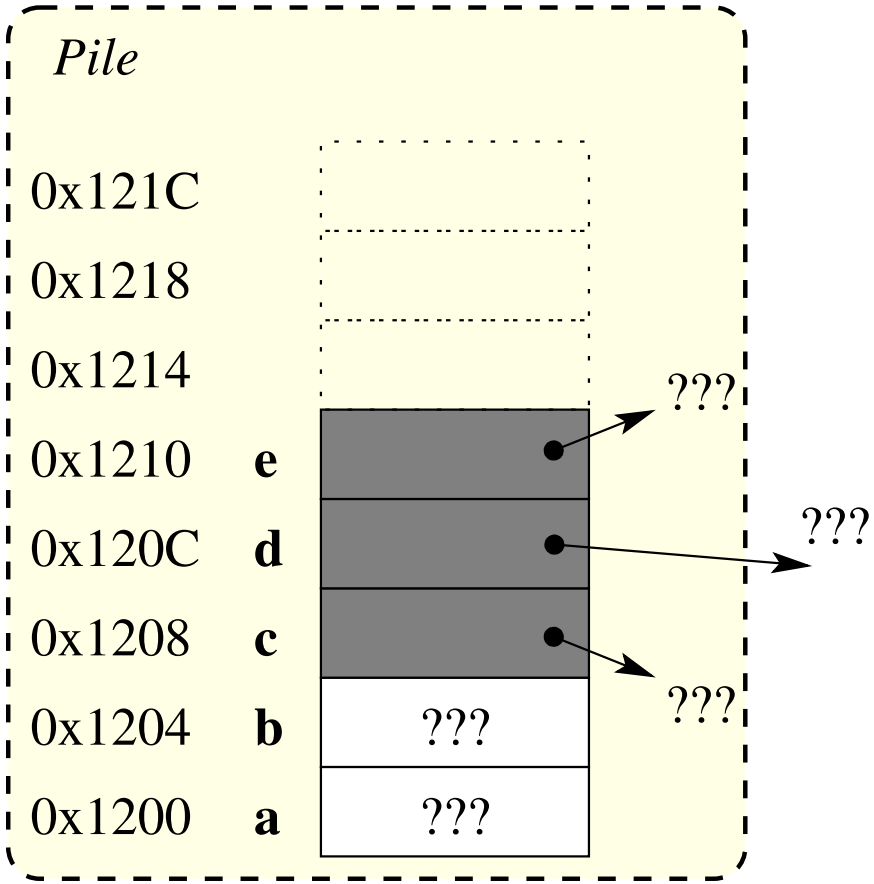
Pourquoi un pointeur doit avoir un type (1)

- Une adresse est juste un entier positif.
- En mémoire on stocke des infos d'un certain type :
  - ⇒ donc d'une certaine forme.
- L'accès à une info dépend de la forme de cette info.
  - ⇒ L'accès à une info par un pointeur dépend de la forme de l'info contenue à cette adresse.
- ⇒ Un pointeur doit spécifier l'adresse de « *quoi* » il est.
- Les pointeurs sont donc typés
  - `char*` : pointeur sur `char`.
  - `int**` : pointeur sur `int*`.

Pourquoi un pointeur doit avoir un type (2)

- Quand on lit la valeur de `*v` :
  - Si `a` est un `char*`, on ne lit qu'un octet,
  - Si `a` est un `int*`, on lit 4 octets,
  - Si `a` est un `double*`, on lit 8 octets.
- Les tableaux sont désignés par l'adresse de leur $1^{\text{ère}}$ case.
  - `t[i]` n'est pas le $i^{\text{ème}}$ octet, mais le $i^{\text{ème}}$ élément.
  - $\Rightarrow$ Il faut savoir de combien d'octets avancer ($i \times 4$, $i \times 8$ ...)
  - $\Rightarrow$ Nécessaire de connaître la taille d'un élément $\rightarrow$ donc son type.
- Rappel (cours 4) : accès au champ d'un pointeur de type `struct` :
  - `struct foo_t { int ch ; };`
  - `struct foo_t my_foo ;`
  - Accès au champ `ch` par notation pointée : `my_foo.ch`
  - `struct foo_t *my_pfoo ;`
  - Accès au champ `ch` par notation fléchée : `my_pfoo->ch`
  - Équivalence : `my_pfoo->ch` = `(*my_pfoo).ch`

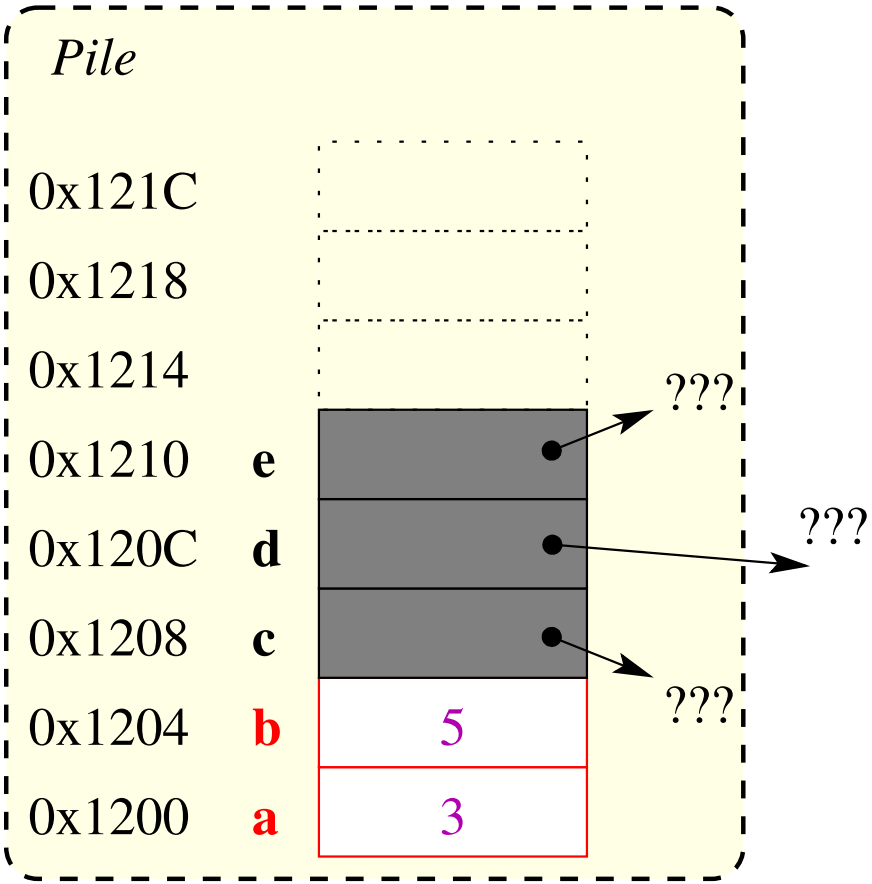
Exemple de manipulations de pointeurs (1)




```
int main ()
{
► int a, b, *c, *d, **e ;
  a = 3 ;    b = 5 ;
  c = &b ;   d = &a ;
  *c = 4 ;
  *d = (*c) + 3 ;
  e = &c ;
  **e = *d ;
  *e = &a ;
  ...
}
```

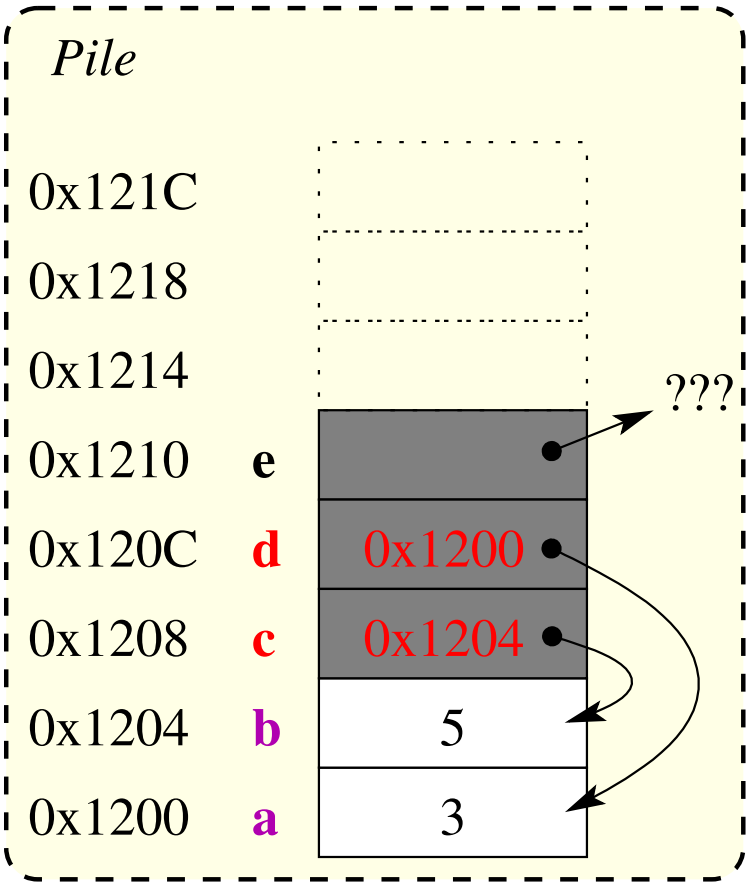
- Chaque case mémoire a une adresse.
- Par habitude et simplicité, adresses écrites en hexadécimal.

Exemple de manipulations de pointeurs (2)




```
int main ()
{
   int a, b, *c, *d, **e ;
►  a = 3 ;  b = 5 ;
   c = &b ;  d = &a ;
   *c = 4 ;
   *d = (*c) + 3 ;
   e = &c ;
   **e = *d ;
   *e = &a ;
   ...
}
```

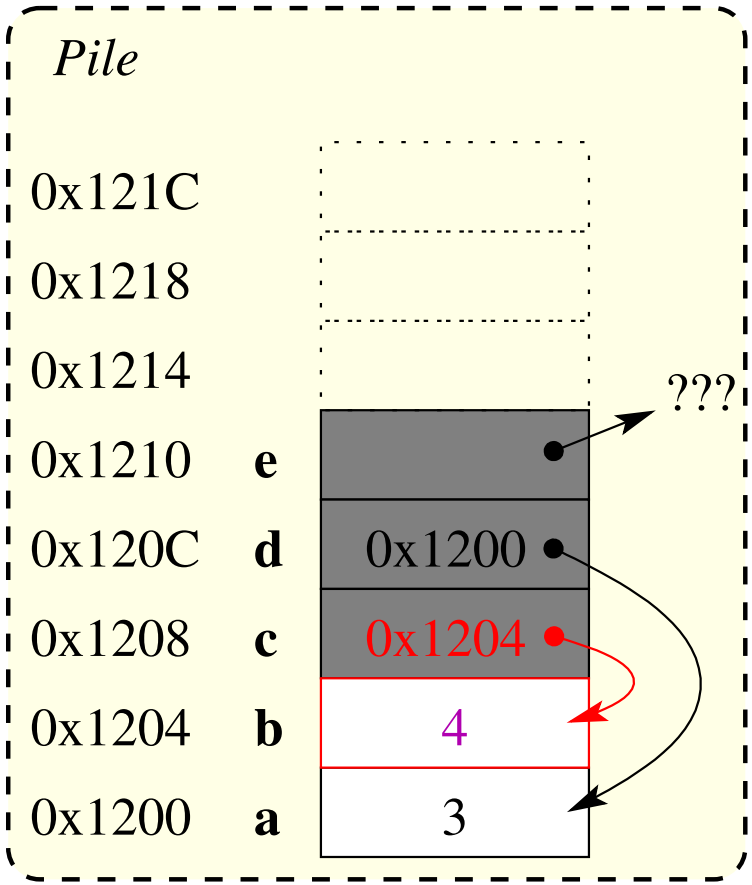
Exemple de manipulations de pointeurs (3)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
►  c = &b ; d = &a ;
   *c = 4 ;
   *d = (*c) + 3 ;
   e = &c ;
   **e = *d ;
   *e = &a ;
   ...
}
```

- c et d contiennent les adresses de b et a.
  - Ils « pointent » vers b et a.

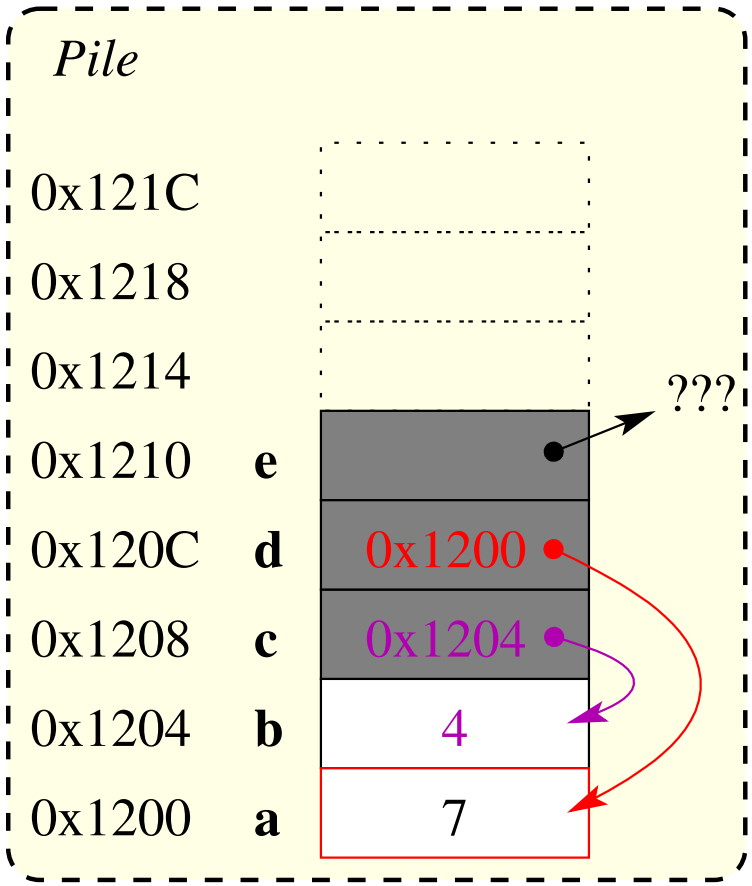
Exemple de manipulations de pointeurs (4)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
   c = &b ;   d = &a ;
►  *c = 4 ;
   *d = (*c) + 3 ;
   e = &c ;
   **e = *d ;
   *e = &a ;
   ...
}
```

- On suit le pointeur dans c et on stocke 4 dans la case mémoire correspondante → dans b.

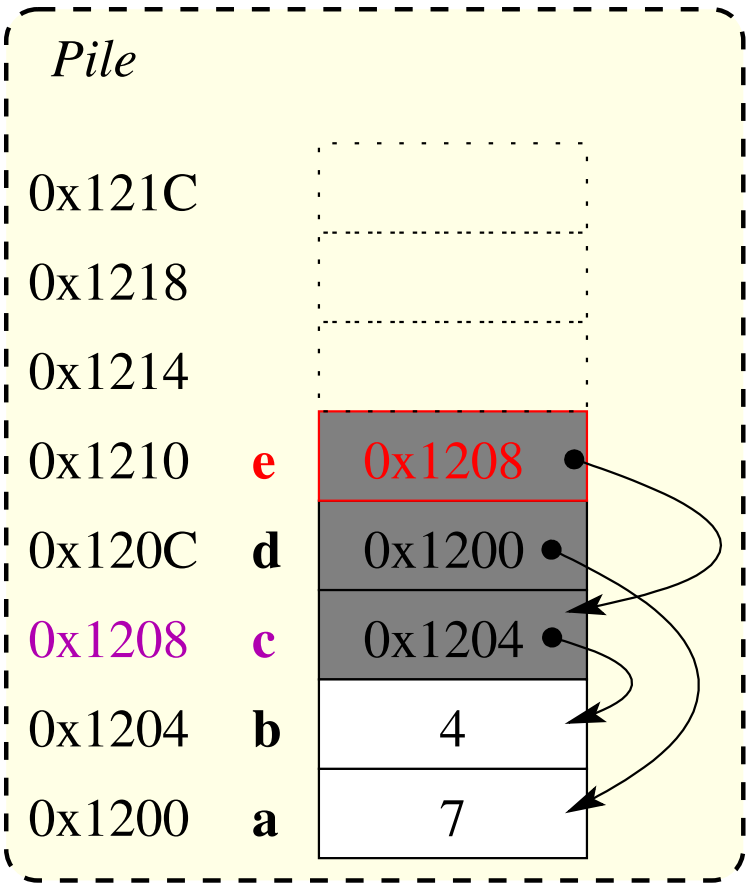
Exemple de manipulations de pointeurs (5)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
   c = &b ;   d = &a ;
   *c = 4 ;
▶  *d = (*c) + 3 ;
   e = &c ;
   **e = *d ;
   *e = &a ;
   ...
}
```

- À gauche du “=” : case mémoire où ira le résultat.
- À droite du “=” : on évalue, i.e. on suit le pointeur c et on regarde ce qui est dans la case.

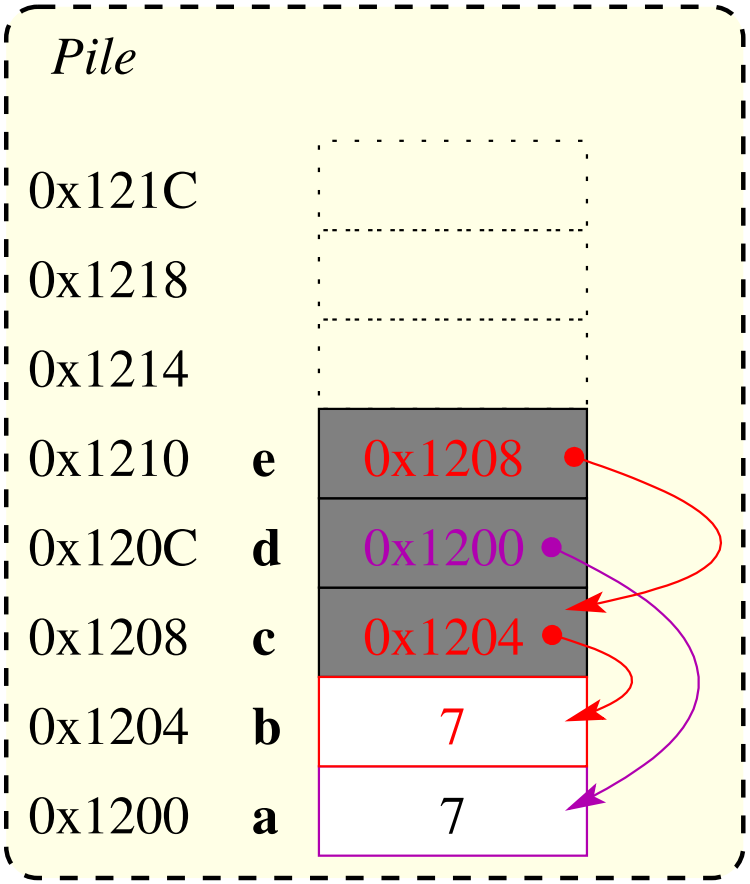
Exemple de manipulations de pointeurs (6)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
   c = &b ;   d = &a ;
   *c = 4 ;
   *d = (*c) + 3 ;
►  e = &c ;
   **e = *d ;
   *e = &a ;
   ...
}
```

- e pointe vers c.

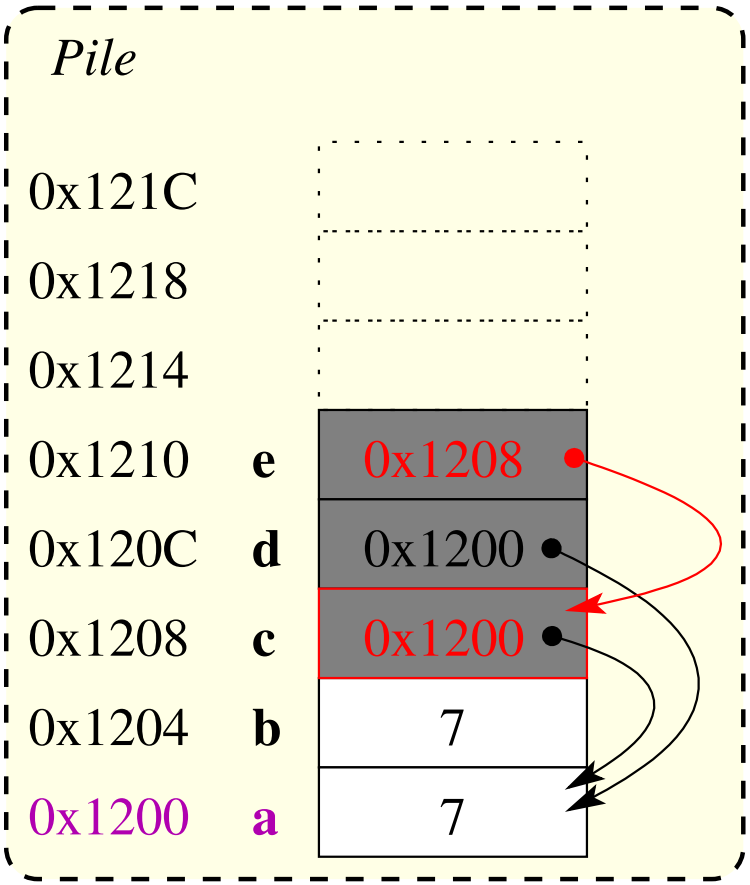
Exemple de manipulations de pointeurs (7)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
   c = &b ;   d = &a ;
   *c = 4 ;
   *d = (*c) + 3 ;
   e = &c ;
▶  **e = *d ;
   *e = &a ;
   ...
}
```

- Double étoile : on suit deux pointeurs à la suite
- I.e. pointeur sur un pointeur.
- I.e. variable qui contient (l'adresse d'une variable qui contient (l'adresse d'une variable)).

Exemple de manipulations de pointeurs (8)




```
int main ()
{
   int a, b, *c, *d, **e ;
   a = 3 ;    b = 5 ;
   c = &b ;   d = &a ;
   *c = 4 ;
   *d = (*c) + 3 ;
   e = &c ;
   **e = *d ;
►  *e = &a ;
   ...
}
```

- Deux pointeurs peuvent contenir la même adresse (« *alias* »).
  - On peut faire un test d'égalité dessus (`c == d`).
  - On peut aussi comparer les contenus (`*c == *d`).

Allocation et libération de mémoire

Statique versus dynamique (1)

- Allocation statique : effectuée à la compilation.
- Ex : variables entières déclarées, tableaux de taille fixe :
  - `int v = 42 ;`
  - `char s[10] ;`
  - Dans la pile (variable locale) ou dans le tas/section-data (variable globale).
- Allocation statique locale : automatiquement libérée en fin de fonction.
  - Comment faire si on veut « *faire durer* » la donnée plus longtemps ?

Statique versus dynamique (2)

- Parfois, mémoire nécessaire connue seulement lors de l'exécution :
  - Lecture du contenu d'un fichier : taille des fichiers variable.
  - Résolution d'un système d'équations : nombre d'inconnues à la demande de l'utilisateur.
- Changer le programme manuellement ⇒ recompiler ?
- ⇒ Inefficace, non-maintenable, inacceptable.
- On demande de la mémoire selon les besoins à l'exécution.
  - Zone allouée retournée désignée par son adresse de début.

Allocation dynamique

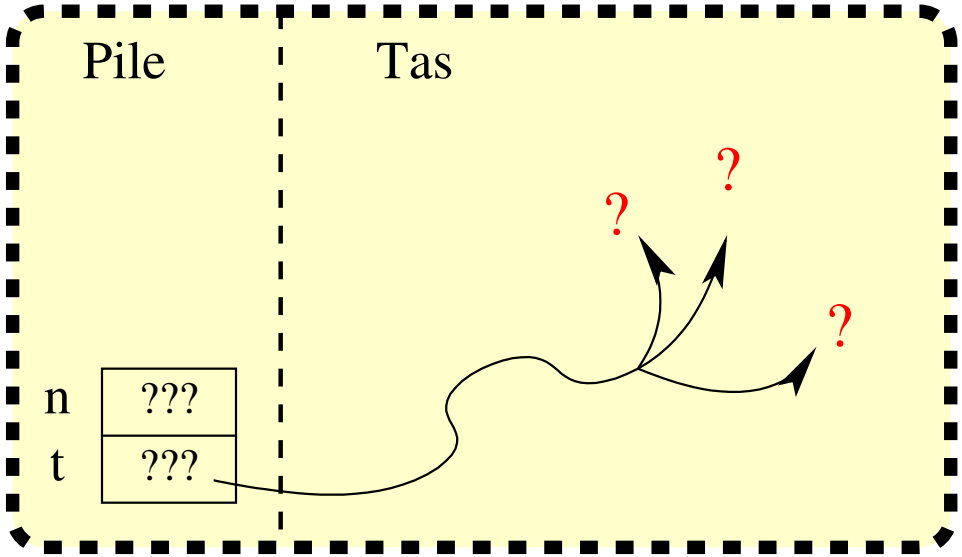
- Allocation de n'importe quoi : entier, entier**s** (tableau), `struct`, etc.
- Tableau dynamique C ≡ type pointeur dont la valeur est l'adresse de début du tableau.
  - `t[0]` ≡ `*t`.
  - `&(t[0])` ≡ `t`.
- On déclare un tableau comme un pointeur.
  - Le pointeur est alors non initialisé (pointe *« n'importe où »*).
  - On alloue de la mémoire (fonction `malloc`) ⇝ une adresse . . .
  - . . . et on enregistre cette adresse dans le pointeur.

Allocation dynamique : `malloc`

- `malloc` : `int` → `void*`
- Argument : nombre d'octets demandés.
- Résultat : adresse du début de la zone octroyée.
- `void*` : pointeur sans type
  - Implicitement compatible avec autres types de pointeurs.
  - ⇒ On ne le « *cast* » pas !
- Si mémoire non disponible : résultat = pointeur `NULL`.
- ⇒ Toujours tester la réussite de l'allocation !
  - `if (ptr == NULL)` *gérer l'erreur*
  - Sinon écriture dans une zone non allouée et « *segmentation fault* » à la clef.

Pas à pas (1)

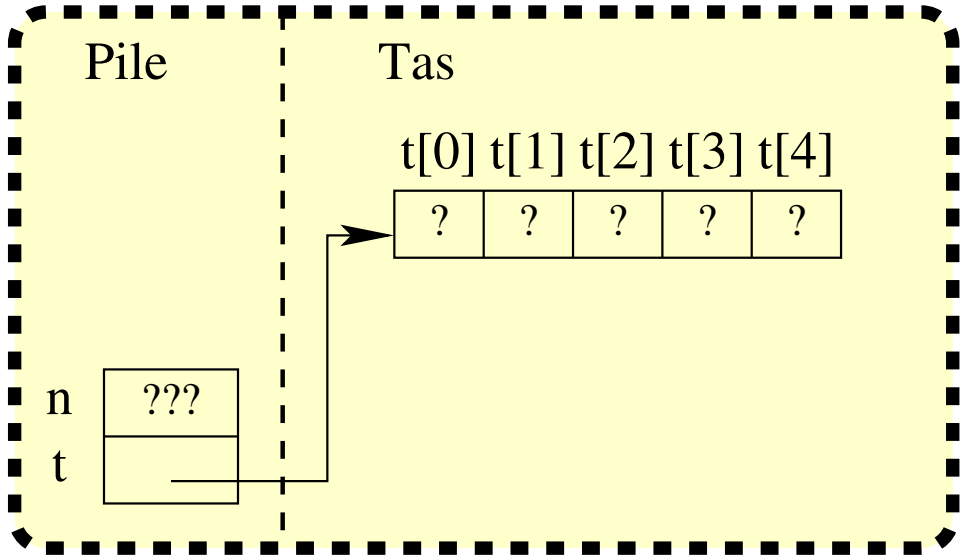
```c
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
► int n ;
  t = malloc (5 * sizeof (float)) ;
  if (t == NULL) exit (-1) ;
  t[2] = 3.14159 ;
  printf ("%f\n", t[2]) ;
  free (t) ;
  return (0) ;
}
```



- Variables locales `t` et `n` allouées sur la pile...
- ...mais pas initialisées.
- ⇒ Pointeur `t` désigne une adresse quelconque...
  - ...et vraisemblablement invalide.

Pas à pas (2)

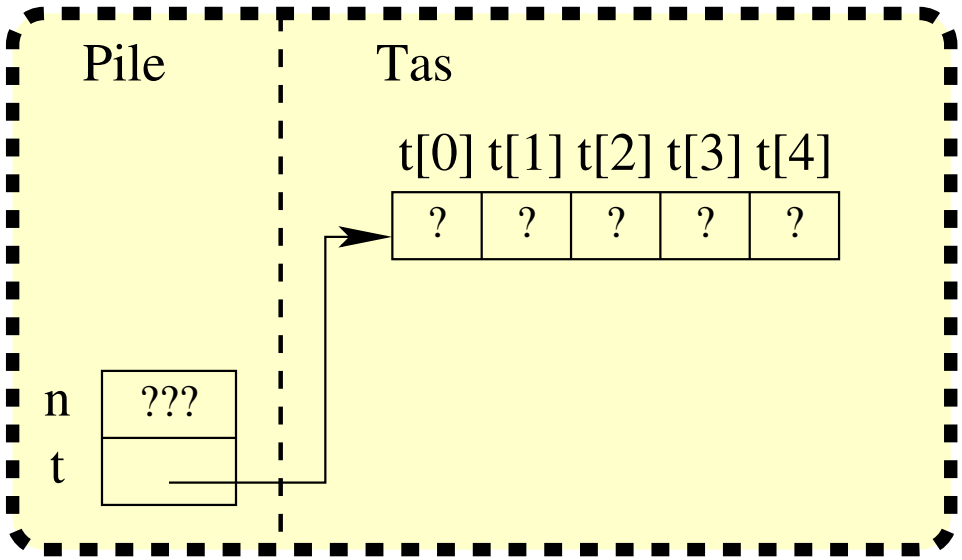
```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
  int n ;
► t = malloc (5 * sizeof (float)) ;
  if (t == NULL) exit (-1) ;
  t[2] = 3.14159 ;
  printf ("%f\n", t[2]) ;
  free (t) ;
  return (0) ;
}
```



- `sizeof` : construction du compilateur, → taille d'une donnée en octets.
  - N'est pas une fonction.
  - Taille élémentaire : dépend de plein de choses (architecture, modèle de compilation, etc.).
- Appel à la fonction d'allocation de mémoire `malloc`.
- Assignation du résultat au pointeur `t`.
- ⇒ `t` désormais initialisé...
- ...Mais à quoi ?

Pas à pas (3)

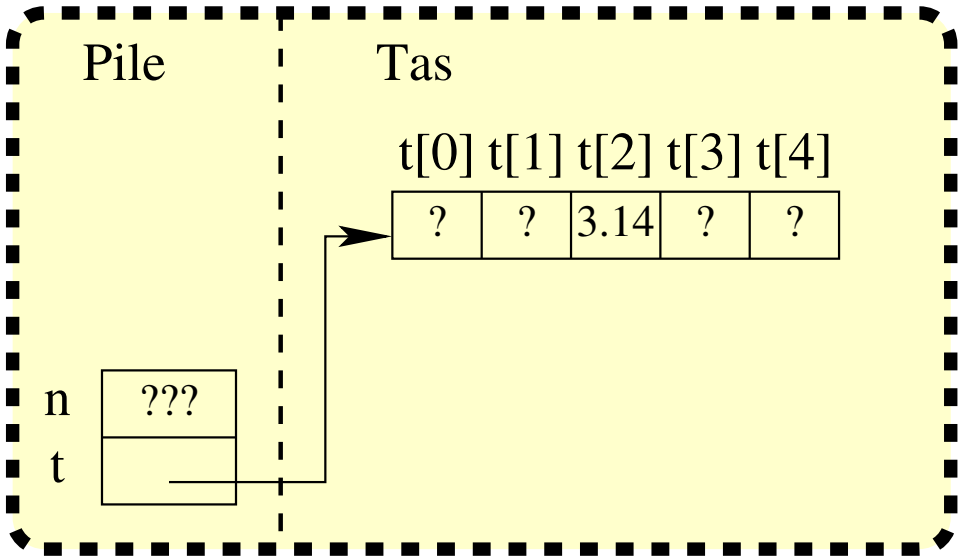
```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
  int n ;
  t = malloc (5 * sizeof (float)) ;
► if (t == NULL) exit (-1) ;
  t[2] = 3.14159 ;
  printf ("%f\n", t[2]) ;
  free (t) ;
  return (0) ;
}
```



- Si pas de mémoire octroyée, pointeur retourné alors NULL.
- Alors pas possible de continuer (ou gérer l'erreur *« comme il se doit »*).
- NULL désigne une adresse interdite en lecture/écriture
  - ⇒ *« Segmentation Fault »*.

Pas à pas (4)

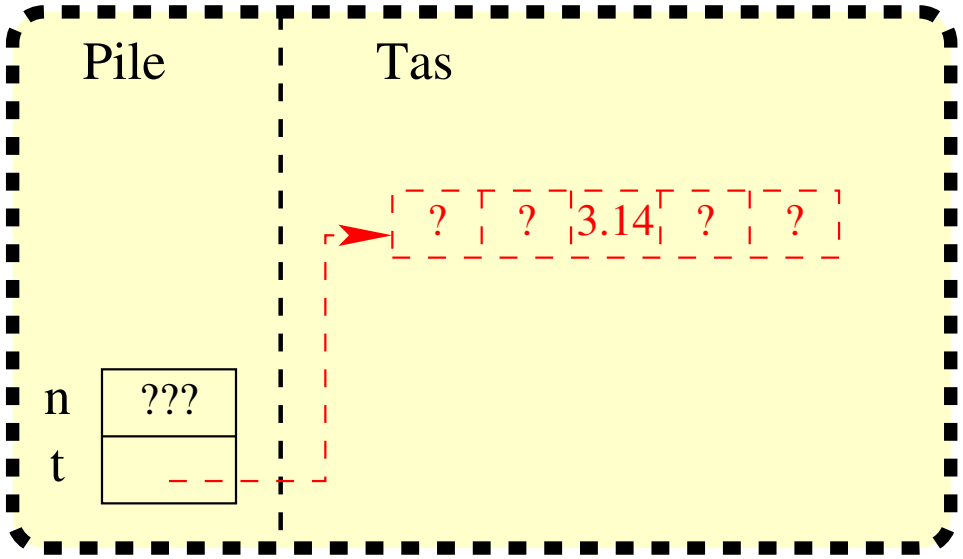
```c
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
  int n ;
  t = malloc (5 * sizeof (float)) ;
  if (t == NULL) exit (-1) ;
► t[2] = 3.14159 ;
  printf ("%f\n", t[2]) ;
  free (t) ;
  return (0) ;
}
```



- Écriture dans une partie de la mémoire allouée.
- Autorisé puisque la mémoire «*nous* » appartient.

Pas à pas (5)

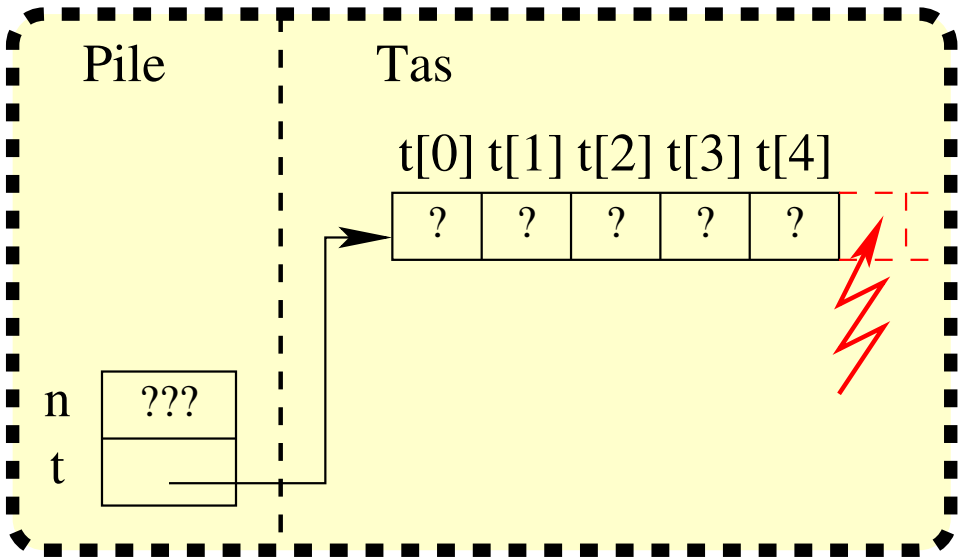
```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
  int n ;
  t = malloc (5 * sizeof (float)) ;
  if (t == NULL) exit (-1) ;
  t[2] = 3.14159 ;
  printf ("%f\n", t[2]) ;
► free (t) ;
  return (0) ;
}
```



- Rendre la mémoire plus utilsée au système → éviter la pénurie.
- N'efface pas la mémoire.
- N'efface pas le pointeur.
- Révoque simplement l'autorisation d'accès.
- ⇒ On a encore les moyens « *techniques* » d'y accéder...
- ...mais « *Segmentation Fault* » guette.

Pas à pas ($5\frac{1}{2}$)

```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
int main ()
{
  float *t ;
  int n ;
  t = malloc (5 * sizeof (float)) ;
  if (t == NULL) exit (-1) ;
► t[5] = 3.14159 ;
  printf ("%f\n", t[2]) ;
  free (t) ;
  return (0) ;
}
```



- Tentative d'accès à la case 5 ($6^{\text{ème}}$ «*case* »)...
- Accès en dehors de la zone allouée.
- Peut «*appartenir* » à un autre (système, processus ...)
- ⇒ «*Segmentation Fault* » à la clef !
- **Mais** ... c'est pas forcément immédiat ! ☹
  - Bugs difficiles à trouver et rendre reproductibles.

Libération de mémoire

- Fonction `free` : `void*` → `void`
  - Argument : adresse de début de la zone à libérer.
  - Résultat : néant.
- À chaque `malloc` doit correspondre 1 et 1 seul `free`.
- ⇒ La libération de mémoire est explicite.
- Taille de la zone connue par le système : pas besoin de la spécifier.
- Libérer au plus tôt : inutile de consommer de la mémoire inutilement.
  ▶ → Pourra être ré-attribuée à quelqu'un d'autre.
- Ne pas libérer « *trop* » tôt.
  ▶ Tant qu'une zone mémoire accessible y fait référence, la zone allouée ne doit pas être libérée.

Allocation dynamique et erreurs courantes

- Ne pas initialiser un pointeur (i.e. ne pas allouer via `malloc`).
- Accéder en dehors d'une zone allouée (débordement).
- Faire `free` dans une boucle (désallocations multiples de la zone).
- Faire `free` avec une adresse non allouée dynamiquement.
- Faire `free` avec une adresse qui n'est pas le début de la zone.
- « *Perdre* » l'adresse d'une zone allouée :

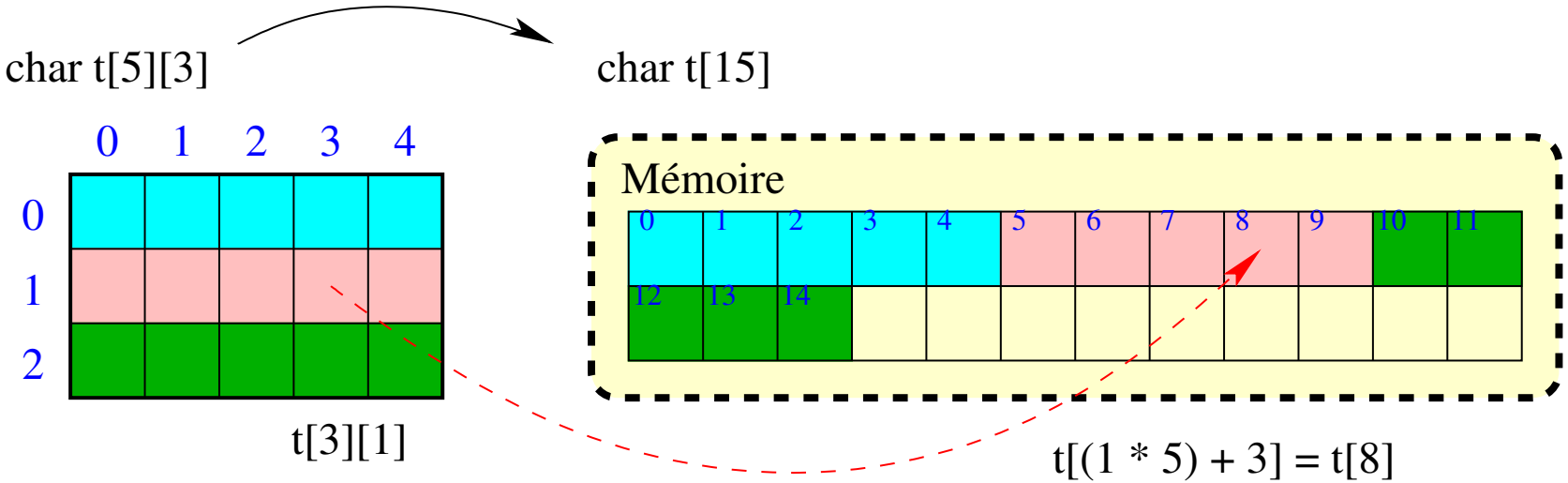
```
int *p = malloc (...) ;
...   // On ne mémorise pas la valeur de p ailleurs.
return (..) ;
```

  - → Impossible de désallouer.
  - ⇒ Fuites mémoire et dégradation de performances avec le temps.

Tableaux dynamiques à plusieurs dimensions

Les tableaux dynamiques à plusieurs dimensions (1)

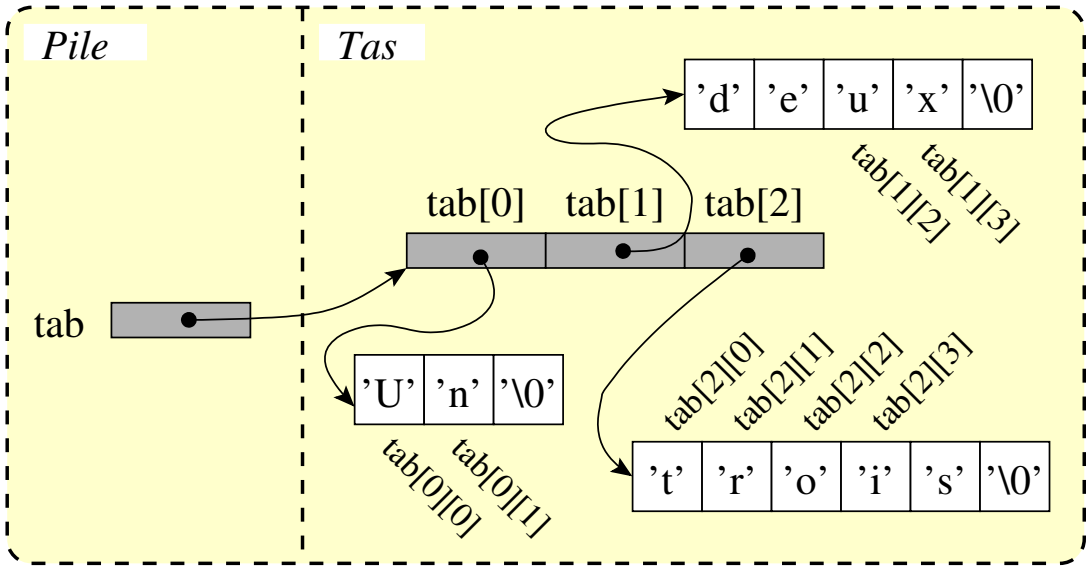
- Vus précédemment : `char t[5][3] ;`
- Comment faire si dimensions non fixes ? → Allocation dynamique.
- Solution 1 :
  - Un grand tableau de taille $DIM_1 * DIM_2 = 5 * 3$.
  - Linéarisation de l'adressage : $t[x][y] \longmapsto t[y * DIM_1 + x]$.



  - Exactement le cas de la mémoire graphique.

Les tableaux dynamiques à plusieurs dimensions (2)

- Comment faire si le tableau n'est pas rectangulaire ?
- ⇒ Tableau de pointeurs.
- Chaque pointeur pointe vers un tableau de taille propre.
- Exemple l'argument `char *argv[]` du `main`.
  - Chaque chaîne peut avoir une longueur différente.



Les tableaux dynamiques à plusieurs dimensions (3)

- Tableau d'`int`s à 2 dimensions, n, m : tableau de pointeurs sur `int`.
- → `int **t ;`
- On alloue n pointeurs (→ 1 allocation) . . .
- On alloue n fois m `int`s (→ n allocations).
- Même schéma de libération de mémoire.

```
int i ;
int **t ;

t = malloc (n * sizeof (int*)) ;
for (i = 0; i < n; i++)
  t[i] = malloc (m * sizeof (int)) ;
...
for (i = 0; i < n; i++)
  free (t[i]) ;
free (t) ;
```

(PS : C'est mal : on n'a pas testé le succès des allocations !)

Passage par adresse

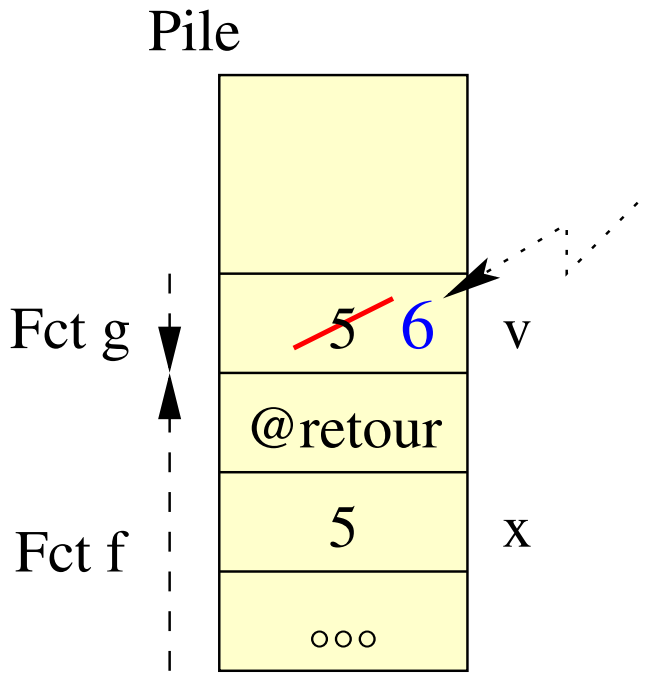
Retourner plusieurs valeurs ?

- Parfois une fonction « devrait » fournir plusieurs résultats en sortie.
- Ou un résultat « *composite* ».
- Ex : filtrage d'un point par convolution.
  - Retourne 3 composantes : rouge, vert, bleu.
- En C, `return` ne permet de retourner qu'une valeur.
- Utiliser des variables globales est peu satisfaisant.
- Déclarer une `struct` par fonction est pénible et pas maintenable :
  - Une `struct` pour retourner 2 entiers,
  - Une `struct` pour retourner 3 entiers,
  - Une `struct` pour retourner 2 entiers et 1 flottant ...
  - ☹

Passage par valeur

- En C, paramètres passés par valeur :
  1. Ils sont évalués
  2. Ils sont copiés sur la pile avant l'appel.
- ⇒ Impossible de modifier une variable de l'appelant depuis l'appelé.

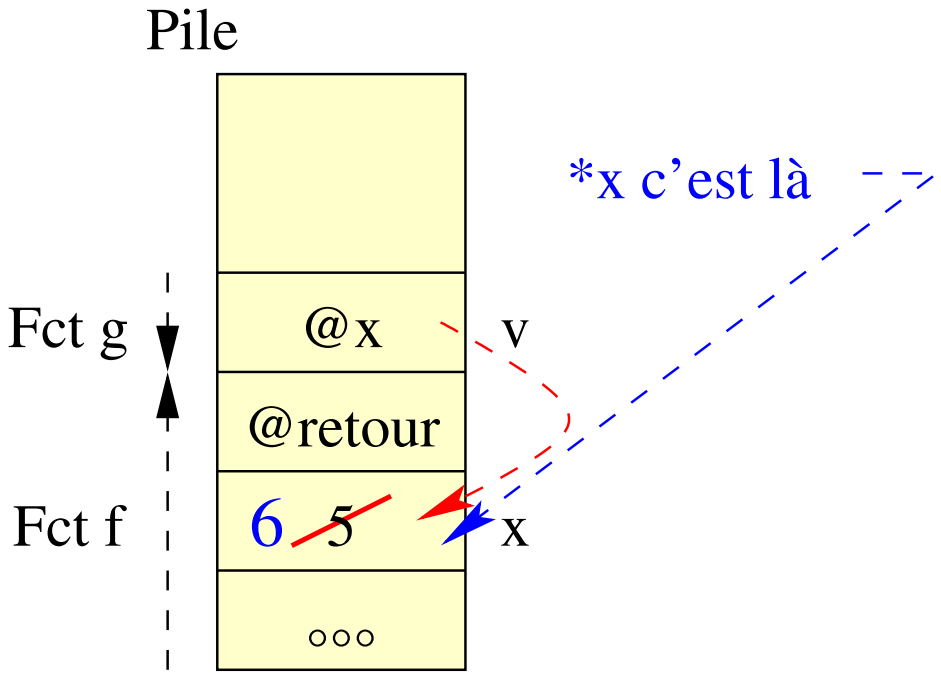
```
void g (int v)
{
► v++ ;
}

void f ()
{
  int x = 5 ;
  g (x) ;
}
```



Passage par adresse

- Et si à la place on passait l'adresse de la variable x ?
- C'est l'adresse qui serait copiée.
- Dans g on a donc un pointeur vers x de l'appelant.
- Il suffit de modifier la valeur pointée : `(*v)++`

```
void g (int *v)
{
► (*v)++ ;
}

void f ()
{
  int x = 5 ;
  g (&x) ;
}
```



Exploitation du passage par adresse

- Pour « *retourner* » plusieurs résultats, une fonction n'a qu'à prendre l'adresse de variables où stocker ses résultats.
- Mode de passage de paramètres appelé « *out* » (c.f. `scanf`).

```
int* random_array (int *size) {
  *size = 1 + random () ;
  return (malloc (*size * sizeof (int))) ;
}
```

- La fonction peut aussi utiliser les valeurs initiales de ces arguments avant de les écraser.
- Mode de passage de paramètres appelé « *in/out* ».

```
void scale_point (int *x, int *y, int scale) {
  *x = *x * scale ;
  *y = *y * scale ;
}
```

- Rem : un tableau étant un pointeur sur la 1ère case, il est forcément passé par adresse !