Université Claude Bernard Lyon 1

LIF5 - Algorithmique et programmation procédurale

Carole Knibbe

Samir Akkouche

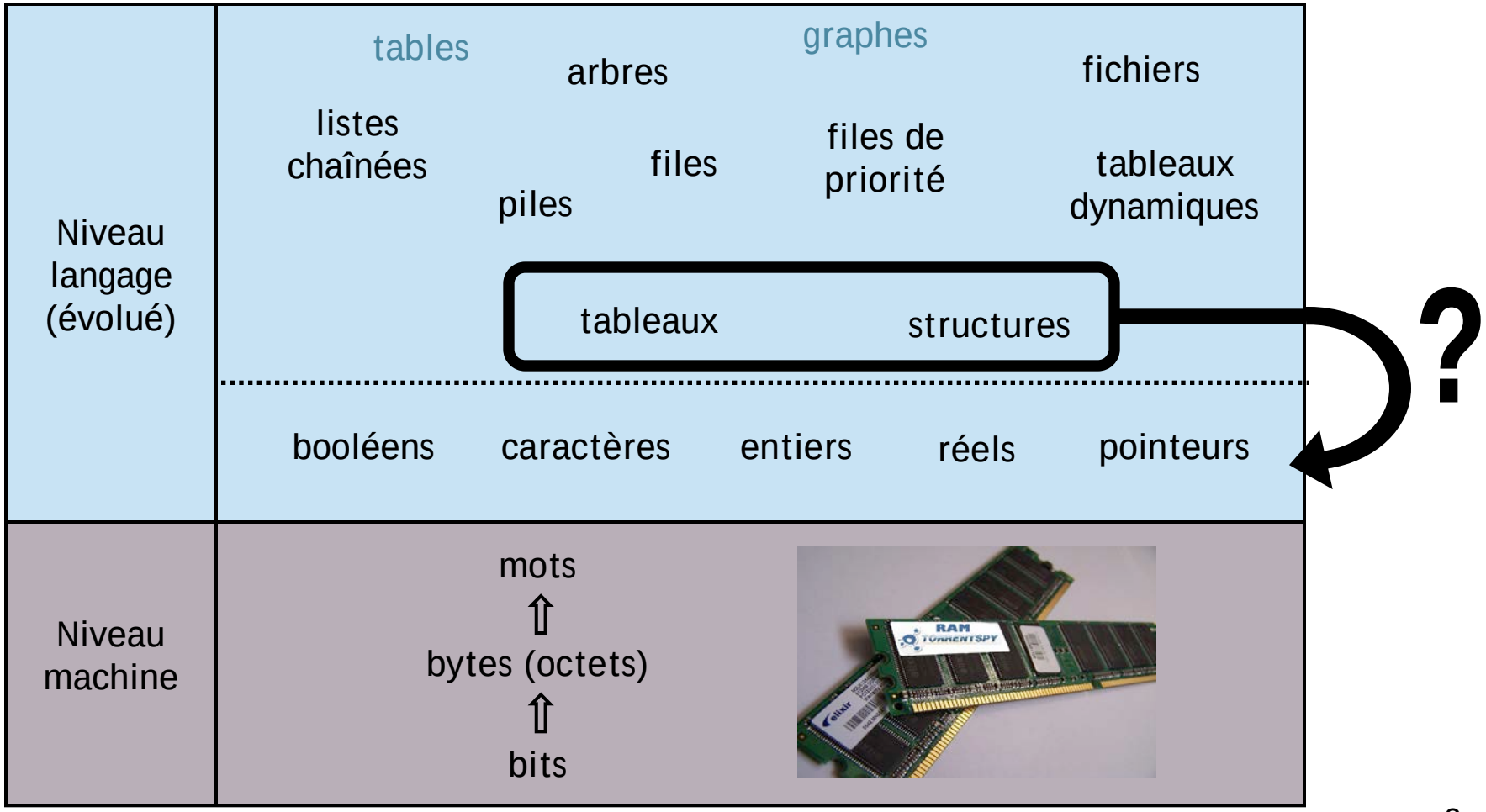
LIRIS

Chapitre 4
Types agrégés : tableaux et structures

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."

Stan Kelly-Bootle

Comment construire des structures de données plus complexes à partir des types de base ?



Sommaire

1. Tableaux
   - Mise en œuvre en C
   - Stockage en mémoire, lien pointeur / tableau en C
   - Comment passer un tableau en paramètre ?
   - Comment retourner un tableau ?
   - **Algorithmes pouvant s'appuyer sur une structuration en tableau**

2. Chaînes de caractères
   - Mise en œuvre en C
   - Stockage en mémoire, lien tableau / chaîne de caractères en C

3. Structures
   - Mise en œuvre en C
   - Structures à tableaux et tableaux de structures
   - Structures à pointeurs et pointeurs sur structures

1. Tableaux

Notion de tableau

- Utilité : éviter d'avoir à définir 10 variables différentes pour stocker 10 notes, par exemple
  - ~~n1, n2, n3, n4, n5, n6, n7, n8, n9, n10 : réels~~
  - notes : tableau [1..10] de réels
- Tableau = ensemble ordonné d'éléments contigus de même type
- Chaque élément est repéré par son indice = sa position dans le tableau

1	2	3	4	5	6	7	8	9	10
15,5	12,5	10,0	8,5	17,0	14,5	16,0	13,0	12,5	10,5

- Accès à la valeur d'un élément par l'opérateur []
  - notes[1] ← 16.0
  - afficher(notes[3])

Tableau : mise en œuvre en C

- Déclaration d'un tableau statique : double notes[10];
- Indices : de 0 à N-1 attention !

0	1	2	3	4	5	6	7	8	**9**
15,5	12,5	10,0	8,5	17,0	14,5	16,0	13,0	12,5	10,5

- Accès à la valeur d'un élément par l'opérateur []
  - notes[0] = 16.0;
  - printf("%f ", notes[2]);
- Aucune opération globale possible sur les tableaux
  - tab1 = tab2 ne fonctionne pas !
  - il faut faire la copie élément par élément, avec une boucle

Tableau : mise en œuvre en C

c'est-à-dire connue à la compilation

- En C90, la taille d'un tableau statique doit être une constante entière
- Dans une expression constante, tous les opérateurs arithmétiques sont autorisés, ainsi que l'opérateur sizeof

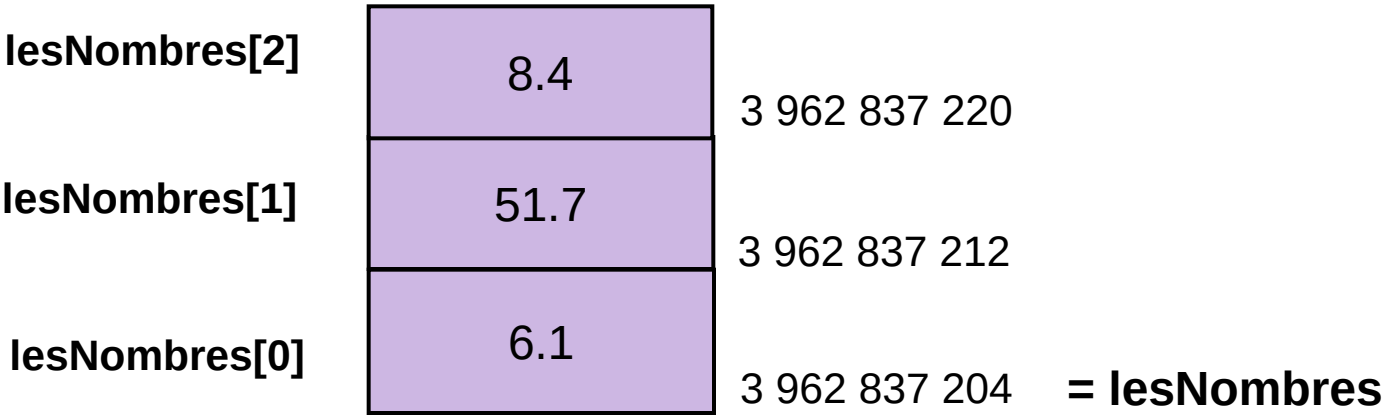
int tab[4];	ok
int tab[3*8 + 2]	ok
#define LIMITE 8 int tab[LIMITE + 3]	ok
int tab[8*sizeof(float)]	ok
const n = 8; int tab[n];	interdit en C90 ok en C++98
int n = maFonction(); int tab[n];	interdit

- La taille peut être omise lorsque le compilateur peut en déduire sa valeur
  - unsigned int mesEntiersPositifs[] = {3, 5, 1, 12} ;

Lien entre pointeur et tableau en C

- « tab » seul ne désigne PAS le tableau entier, mais un pointeur constant contenant l'adresse de tab[0]

```
double lesNombres[] = {6.1, 51.7, 8.4};
```



- On accède en fait aux autres cases en ajoutant la bonne quantité à l'adresse de base…

Arithmétique des pointeurs en C

- Si p est une variable de type pointeur contenant l'adresse 3 925 632 140, que vaut p + 1 ?
- Cela dépend du type pointé par p !
  - int * p p + 1 == 3 925 632 14**4** **(p + 1int)**
  - double * p p + 1 == 3 925 632 14**8** **(p + 1double)**
  - char * p p + 1 == 3 925 632 14**1** **(p + 1char)**
- p + i désigne l'adresse située **i*sizeof(*type*)** octets plus loin
- Conséquence : tab + i = adresse de tab[i] = &tab[i]¤
  tab[i] = valeur dans la case i = *(tab + i)

A retenir

En C, la mise en œuvre de l'opérateur [ ] repose sur l'arithmétique des pointeurs

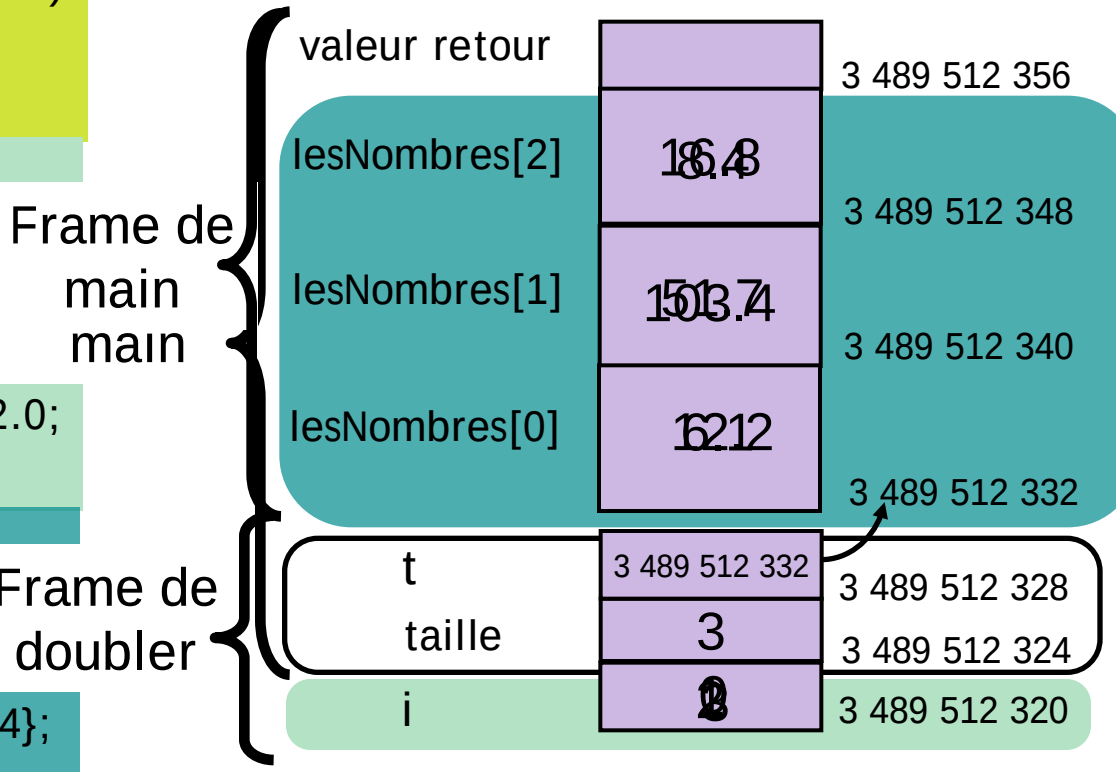
Passer un tableau en paramètre en C

```
void proc(int tab[4])

void proc(int tab[])

void proc(int * tab)
```

- Les 3 écritures sont strictement équivalentes
- Elles reviennent à passer à proc l'adresse du 1er élément du tableau
- La taille du tableau n'est pas transmise
- Pas de copie locale du tableau = accès direct à l'original = passage en mode résultat ou donnée-resultat
- Pour un passage en mode donnée : void proc(**const** int * tab)
  Toujours pas de copie locale, mais on s'engage à ne pas modifier le contenu du tableau original

Passer un tableau en paramètre en C

```
void afficher (const double * t, int taille)
{  ...
}

void doubler (double * t, int taille)
{
  int i;
  for(i = 0; i < taille; i++)   t[i] = t[i] *2.0;
}

int main()
{
  double lesNombres[] = {6.1, 51.7, 8.4};
  doubler(lesNombres, 3);
  afficher(lesNombres, 3);
  return 0;
}
```



Retourner un tableau en C

```
void afficher (const double * t, int taille) {...}

double * doubler (const double * t)
{
 /* Précondition : t tableau de taille 3 */
  double res[3];
  int i;
  for(i = 0; i < 3; i++)   res[i] = t[i] *2.0;
  return res;
}

int main()
{
  double lesNombres[] = {6.1, 51.7, 8.4};
  double * lesAutres ;
  lesAutres = doubler(lesNombres);
  afficher(lesAutres, 3);
  return 0;
}
```

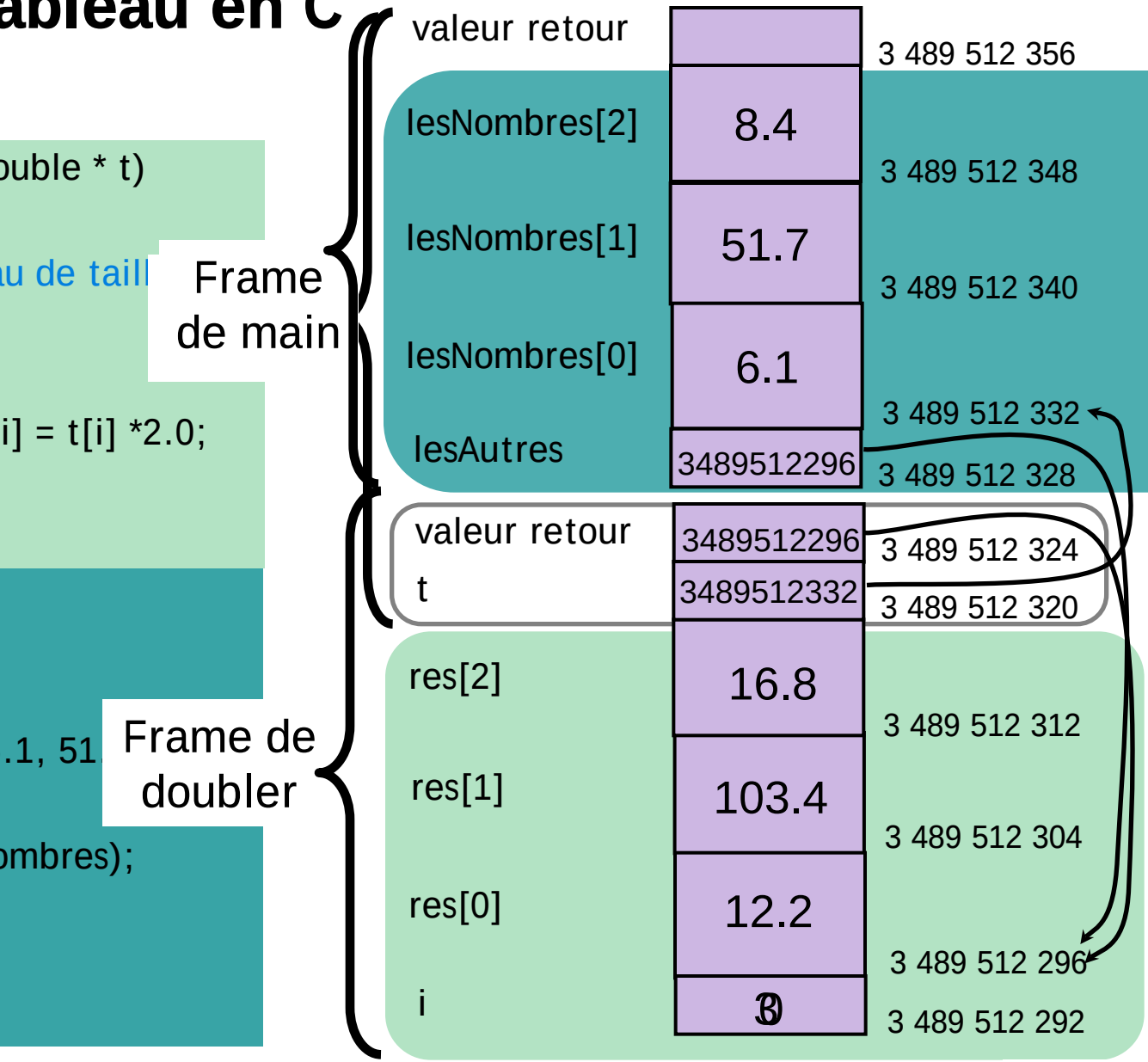
Que pensez-vous de ce programme ?

Indication : le compilateur râle...

warning: address of local variable 'res' returned

Retourner un tableau en C

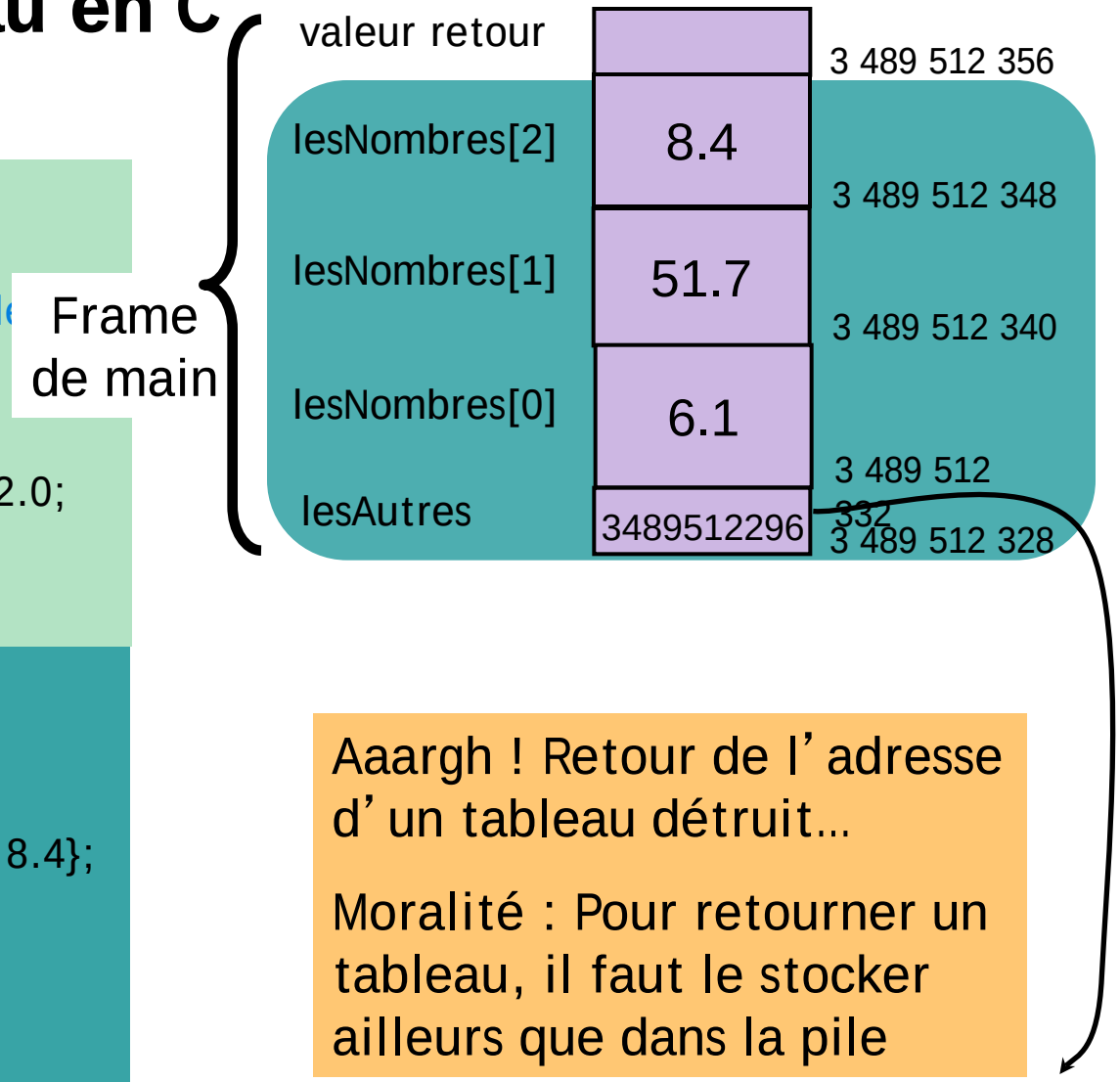
```
double * doubler (const double * t)
{
 /* Précondition : t tableau de taill
  double res[3];
  int i;
  for(i = 0; i < 3; i++)   res[i] = t[i] *2.0;
  return res;
}

int main()
{
  double lesNombres[] = {6.1, 51
  double * lesAutres ;
  lesAutres = doubler(lesNombres);
  afficher(lesAutres, 3);
  return 0;
}
```



Retourner un tableau en C

```
double * doubler (const double * t)
{
 /* Précondition : t tableau de taille
  double res[3];
  int i;
  for(i = 0; i < 3; i++)   res[i] = t[i] *2.0;
  return res;
}
```

```
int main()
{
  double lesNombres[] = {6.1, 51.7, 8.4};
  double * lesAutres ;
  lesAutres = doubler(lesNombres);
  afficher(lesAutres, 3);
  return 0;
}
```

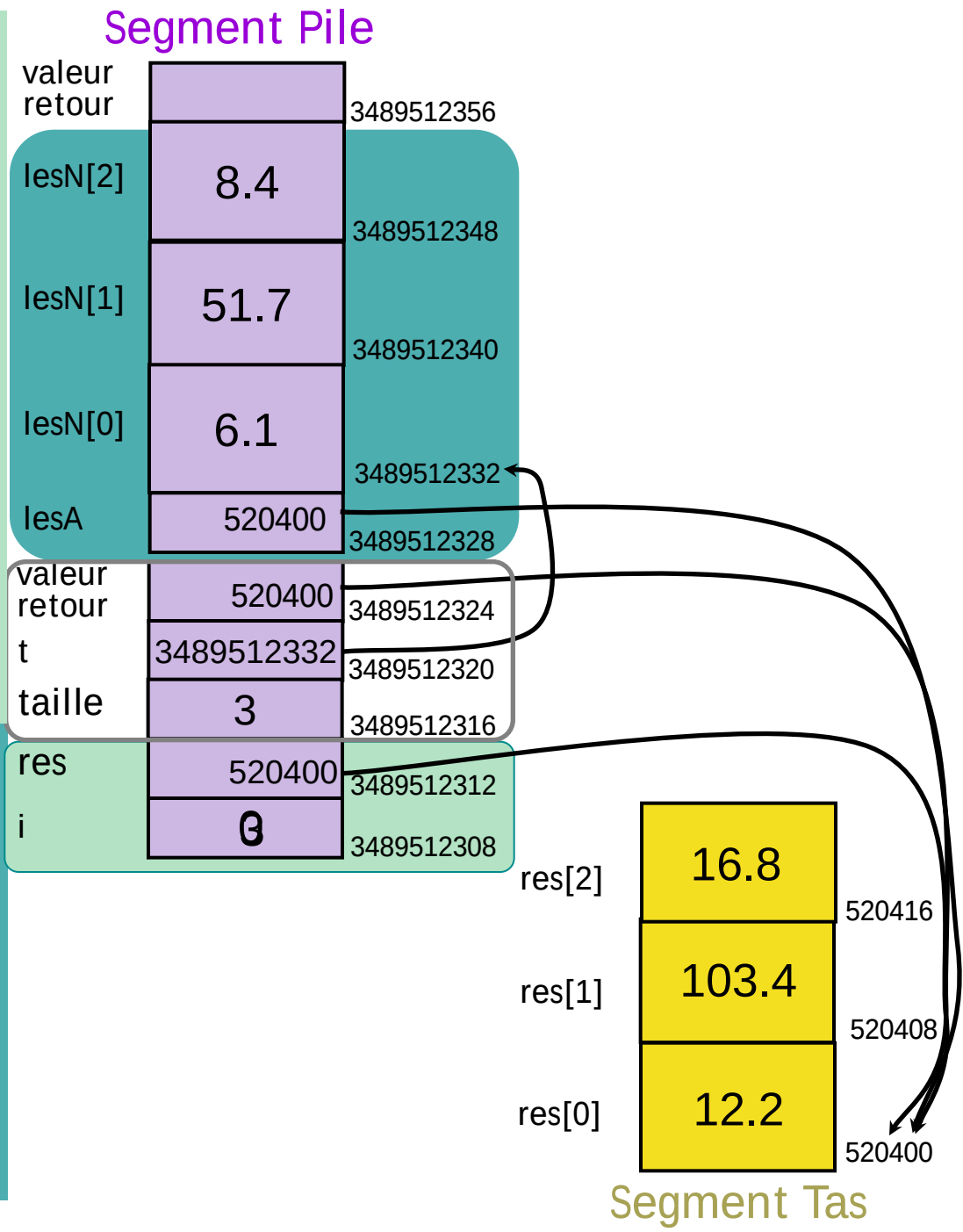


Aaargh ! Retour de l'adresse d'un tableau détruit…

Moralité : Pour retourner un tableau, il faut le stocker ailleurs que dans la pile

```
double * doubler (const double * t, int taille)
 {
 /* Précondition : t adresse d'un tableau
      contenant 'taille' élément de type
      double
      Résultat : adresse d'un tableau de
      'taille' doubles, situé dans le tas.
      Charge à l'appelant de stocker cette
      adresse dans une variable et d'invoquer
      free() sur cette variable quand il n'aura
      plus besoin du tableau... */
  double * res = (double*)
     malloc(taille*sizeof(double));
  int i;
  for(i = 0; i < taille; i++)   res[i] = t[i] *2.0;
  return res;
}
```

```
int main()
{
  double lesN[] = {6.1, 51.7, 8.4};
  double * lesA ;
  lesA = doubler(lesN, 3);
  afficher(lesA, 3);
  free(lesA);
  return 0;
}
```



Retourner un tableau en C : bilan

- On ne peut pas stocker le tableau dans la pile si on veut le renvoyer

- Il faut le stocker dans le tas : allocation dynamique par malloc

- Risque de « fuite de mémoire » si on ne pense pas à libérer le tableau

Algorithmes pouvant s'appuyer sur une structuration en tableau simple

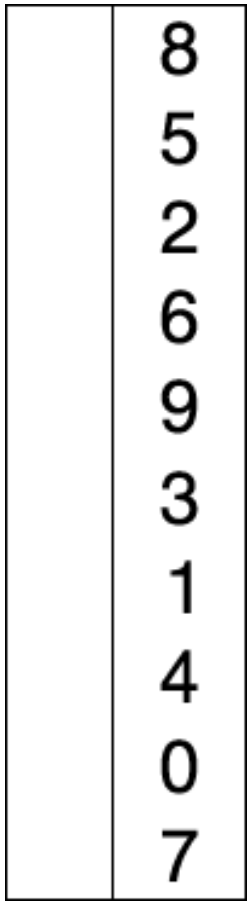
- Tri par sélection, aussi appelé tri du minimum
- Tri par insertion
- Et bien d'autres…

Tri par sélection : principe

- Pour chaque i de 1 à n-1, échanger a[i] avec l'élément minimum de a[i], …, a[n]

- Les éléments à gauche de i sont à leur place finale (et ne seront plus modifiés), donc le tableau est complètement trié lorsque i arrive à l'extrémité droite

Tri par sélection : principe



```
Procédure tri_selection(tab : tableau[1..n] de Element)
Précondition : tab[1], tab[2], .. , tab[n] initialisés
Postcondition : tab[1] ≤ tab[2] ≤ .. ≤ tab[n]
Paramètre en mode donnée-résultat : tab
Variables locales :
    i, j, indmin : entiers
    min : Element

Début
    Pour i de 1 à n-1 par pas de 1 Faire
        indmin <- i
        Pour j de i+1 à n Faire
            Si tab[j] < tab[indmin] alors
                indmin <- j
            FinSi
        FinPour
        min <- tab[indmin]
        tab[indmin] <- tab[i]
        tab[i] <- min
    FinPour
Fin tri_selection
```

réel, entier, caractère…

on recherche l'index du minimum de la partie non triée

on permute le minimum avec l'élément courant

Tri par sélection : invariant de boucle

- Invariant de boucle = propriété des données telle que…
  - Initialisation : La propriété est vraie avant la 1e itération
  - Conservation : Si la propriété est vraie avant l'itération i, alors elle reste vraie avant l'itération i+1
  - Terminaison : Une fois la boucle terminée, la propriété est utile pour montrer la validité de l'algorithme

Question

Dans le cas du tri par sélection, quel est l'invariant de boucle ?

Tri par sélection : invariant de boucle

Invariant de boucle :

> **« Si i >= 2, alors tab est trié entre les indices 1 et i-1, et tous les éléments restants sont > ou égaux à tab[i-1]** (donc les éléments 1 à i-1 sont à leur place finale) »

- **Initialisation** : pour i=2 (on vient de faire l'affectation i=2 mais on n'a pas exécuté le corps de la boucle pour i=2). A-t-on la propriété suivante : « tab est trié entre les indices 1 et 1, et tous les éléments restants sont sup. ou égaux à tab[1] » ?

  Entre les indices 1 et 1, il n'y a qu'un seul élément donc ce morceau de tableau est forcément trié. Par ailleurs, l'élément qui se trouve en position 1 est le min de tout le tableau, donc les éléments restants lui sont donc bien sup. ou égaux.

Tri par sélection : invariant de boucle

Invariant de boucle :

« Si i >= 2, alors tab est trié entre les indices 1 et i-1, et tous les éléments restants sont > ou égaux à tab[i-1] (donc les éléments 1 à i-1 sont à leur place finale) »

- **Conservation** : Supposons que la propriété est vraie avant l'itération i, soit : « tab est trié entre les indices 1 et i-1, et tous les éléments restants sont > ou égaux à tab[i-1] ». On doit montrer que la propriété restera vraie pour i+1, soit : « tab est trié entre les indices 1 et i, et tous les éléments restants sont > ou égaux à tab[i] ».

  Lors de l'itération i, on va prendre un élément dans la partie non triée pour le mettre à la place i, et on ne touche pas aux éléments 1 à i-1. Cet élément est > ou égal à tab[i-1] donc le début du tableau restera bien trié, jusqu'à i inclus maintenant.

  Comme par ailleurs cet élément est le minimum de la partie qui était non triée, les éléments restants sont bien >= à tab[i].

Tri par sélection : invariant de boucle

Invariant de boucle :

> **« Si i >= 2, alors tab est trié entre les indices 1 et i-1, et tous les éléments restants sont > ou égaux à tab[i-1]** (donc les éléments 1 à i-1 sont à leur place finale) »

- **Terminaison** : Pour i=n (dernière valeur prise par i, qui va causer la sortie de la boucle), l'invariant de boucle s'écrit « tab est trié entre les indices 1 et n-1, et tous les éléments restants sont > ou égaux à tab[n-1] ».

  Donc on a un tableau trié jusqu'à n-1, suivi d'une valeur supérieure ou égale à toutes les autres. Le tableau est donc en fait trié de 1 à n, ce qui prouve que l'algorithme est correct.

Tri par sélection : complexité en temps

- On ne compte que les affectations et les comparaisons d' Element
  - on néglige les affectations et les comparaisons d' indices
- n-1 passages dans la boucle « i », avec à chaque passage :
  - 4 affectations (=> 4*(n-1))
  - 0 comparaison
  - n-i passages dans la boucle « j », avec à chaque passage :
    - 1 affectation => $\sum_1^{n-1}(n-i)$ affectations
    - 1 comparaison => $\sum_1^{n-1}(n-i)$ comparaisons

$$\sum_1^{n-1}(n-i) = \sum_1^{n-1} n - \sum_1^{n-1} i = n(n-1) - n(n-1)/2$$

- Soit au final :
  - 4*(n-1) + n*(n-1)/2 affectations
  - n*(n-1)/2 comparaisons

Tri par sélection : complexité en espace

- Tri sur place = réorganisation du tableau original, sans en faire de copie

- N'utilise que très peu de mémoire supplémentaire :
  - 2 indices
  - 1 Element

- Complexité en espace = constante indépendante de *n* = $O(1)$

Tri par insertion

- Principe :
  - Insérer le premier élément du sous-tableau restant à trier, dans la partie déjà triée du tableau
  - Itérer la procédure jusqu'au tri du tableau complet

- Algorithme, invariant de boucle et complexité : cf TD !

Tableaux à 2 dimensions : norme algorithmique

	1	2	3	4	5	6	7	8	9	10
1	15,5	12,5	10,0	8,5	17,0	14,5	16,0	13,0	12,5	10,5
2	6,0	18,5	19,0	7,5	13,5	13,0	11,0	13,0	15,5	9,5
3	10,5	10,0	12,0	15,5	11,5	9,0	8,5	14,0	18,5	17,0

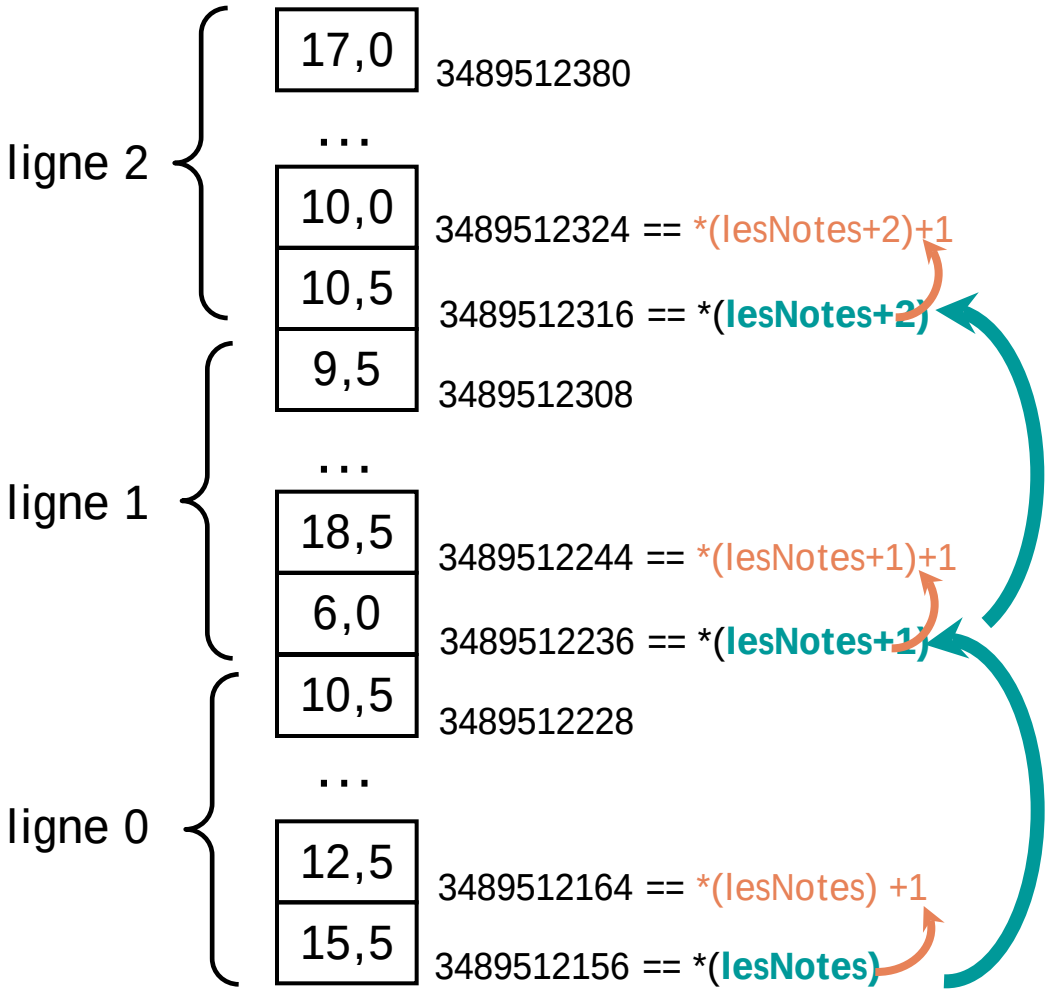
- Déclaration : *nom* : tableau[1..L][1..C] de *type*
  - exemple : notes : tableau[1..3][1..10] de réels
- Lignes numérotées de 1 à L, colonnes de 1 à C
- Accès à un élément : notes[2][9] ← 16.0
- Parcours de toute la matrice à l'aide de 2 boucles imbriquées

Tableau à 2 dimensions : mise en œuvre en C

	0	1	2	3	4	5	6	7	8	9
0	15,5	12,5	10,0	8,5	17,0	14,5	16,0	13,0	12,5	10,5
1	6,0	18,5	19,0	7,5	13,5	13,0	11,0	13,0	15,5	9,5
2	10,5	10,0	12,0	15,5	11,5	9,0	8,5	14,0	18,5	17,0

- Déclaration d'un tableau 2D statique : `type nom[L][C];`
  - L et C doivent être des constantes entières
  - exemple : `double notes[3][10];`
- Lignes numérotées de 0 à L-1, colonnes de 0 à C-1
- Accès à un élément : `notes[1][8] = 16.0;`
- Parcours de toute la matrice à l'aide de 2 boucles imbriquées

Tableau à 2 dimensions : stockage en mémoire



- double lesNotes[3][10];
- lesNotes est un pointeur sur tableau de 10 doubles, donc un « pointeur sur pointeur sur double » (double **)
- lesNotes+i : pointeur sur pointeur sur double, ajout de i*10*sizeof(double)
- *(lesNotes+i) + j : pointeur sur double, ajout de j*sizeof(double)
- lesNotes[i][j] == *(*(lesNotes+i) + j)

2. Chaînes de caractères

Tableaux et chaînes de caractères

- En C, les tableaux 1D de caractères servent également à manipuler des chaînes de caractères (de taille inférieure à celle du tableau)

- Possibilité d'utiliser des fonctions de la librairie standard si '\0' en fin de la chaîne (convention) ➔ prévoir une case en plus pour le '\0'
  - strlen, strcmp, strcpy… (#include <string.h>)
  - fonctions d'entrée-sortie (#include <stdio.h>)

```
printf("%s", maChaine);
scanf("%10s", maChaine);
```

Fonctions de manipulation globale des chaînes

- Exemple : tableau de taille 5 (au moins) pour stocker la chaîne « LIF5 »

0	1	2	3	4
'L'	'I'	'F'	'5'	'\0'

Initialisation d'une chaîne de caractères lors de sa définition

- Comme les autres tableaux :

```
char ch1[]={ 'b' , ' i' , ' c' , ' h' , ' e' , ' \n' , ' \0' };
char ch2[8]={ 'o' , ' u' , ' i' , 0};
```

- Avec une chaîne littérale :

```
char ch3[]= ' 'loup' ' ;   /* ajout automatique du '\0' */
```

- Ne pas confondre a, 'a' et "a"

Questions

Quelles sont les tailles des tableaux ch1, ch2, ch3 ?

ch2 respecte-t-elle la convention du '\0' ?

Initialisation de chaînes de caractères avec cin, strcpy, scanf, gets… : attention danger !!!

```
void loginProc()
{
      char username[10];
       scanf(« %s », username)
      /* ou gets(username) */
      /* ou cin >> username;  (C++) */
      …
}
```

Ne JAMAIS écrire ce type de code !

Il contient une faille de sécurité de type « buffer overflow »

3. Structures

Notion de structure

- Nouveau type de données
- Agrégat d'informations (de types éventuellement différents) relatives à une entité
- Les différentes informations s'appellent les champs
- Champ : type primitif, ou tableau, ou même une autre structure
- Accès à un champ par l'opérateur .

```
Structure date
        jour : entier
        mois : entier
        année : entier
Fin structure

Structure etudiant
        numero : entier
        nom : tableau[1..60] de caractères
        prenom : tableau[1..60] de caractères
        datenaiss : date
        notes : tableau[1..10] de réels
Fin structure


Variables
        monEtudiant : etudiant
Debut
        monEtudiant.numero ← 164987
        saisir(monEtudiant.nom)
        saisir(monEtudiant.prenom)
        saisir(monEtudiant.datenaiss.jour)
        ...
Fin
```

Notion de structure

- Une fonction peut renvoyer une structure

```
Fonction saisir(num : entier) : etudiant

Param. en mode donnée : num
Variables locales :  etu : etudiant

Début
        etu.numero ← num
        afficher(« Saisir le nom : »)
        saisir(etu.nom)
        afficher(« Saisir le prenom : »)
        saisir(etu.prenom)
        afficher(« Saisir le jour de naissance : »)
        saisir(etu.datenaiss.jour)
        ...
        Retourner etu

Fin saisirEtudiant
```

Structures et tableaux

- Une structure peut contenir des champs de type tableau
- On peut faire des tableaux de structures

```
Variables
    lesEtudiants : tableau[1..1000] d'etudiants
    i : entier
Debut
    Pour i allant de 1 à 1000 par pas de 1 faire
        lesEtudiants[i] ← saisir(10264 + i)
    Fin Pour
    ...
Fin
```

1	2	3	4
10265	10266	10267	10268
‘’ Dupon d’’	‘’ Dupont ’’	‘’ Michel’ ’’	‘’ Arman d’’
‘’ Ernes t’’	‘’ Arnol d’’	‘’ Aurélie ’’	‘’ Corali e’’
5	28	14	21
12	10	1	9
1990	1990	1990	1990
…	…	…	…

etc

Question : Que vaut

lesEtudiants[3].prenom ?

lesEtudiants[3].prenom[2] ?

Tableaux de structures: comment les trier ?

- Exemple : tri d'un tableau d'étudiants par numéro d'étudiant croissant
- Réutilisation de l'algorithme de tri par sélection ou par insertion… mais on ne peut pas écrire directement « Si tab[i] < tab[j] »
- Deux possibilités :
  - écrire « Si tab[i].numero < tab[j].numero »
  - écrire une fonction estInferieur(etudiant e1, etudiant e2) qui renvoie un booléen

Question

Quels sont les avantages et les inconvénients des deux possibilités ?

Pointeurs et structures

- On peut définir un pointeur sur une structure

```
pe : pointeur sur etudiant
etu : etudiant

pe ← &etu
pe↑ ← saisir(10264)
```

- Une structure peut contenir des champs de type pointeur

- Cas particulier :
  - une structure A ne peut PAS avoir de champ de type A
  - une structure A peut, par contre, avoir un champ de type A*

Structures : mise en œuvre en C

- Déclaration et définition :
  - mot-clé `struct`
  - ne pas oublier le `;` à la fin

- Accès aux champs : opérateur `.`

```
struct date  {
        int jour;
        int mois;
        int annee;
};

struct etudiant {
        int numero;
        char nom[60];
        char prenom[60];
        struct date datenaiss;
        double notes[10];
};

struct etudiant saisir(int num)
{
        struct etudiant etu;
        etu.numero = num;
        printf("Saisir le nom : \n »);
        scanf("%59s", etu.nom);
        /* ... */
        return etu;
}
```

Structures : mise en œuvre en C

- Déclaration et définition :
  - mot-clé struct
  - ne pas oublier le **;** à la fin

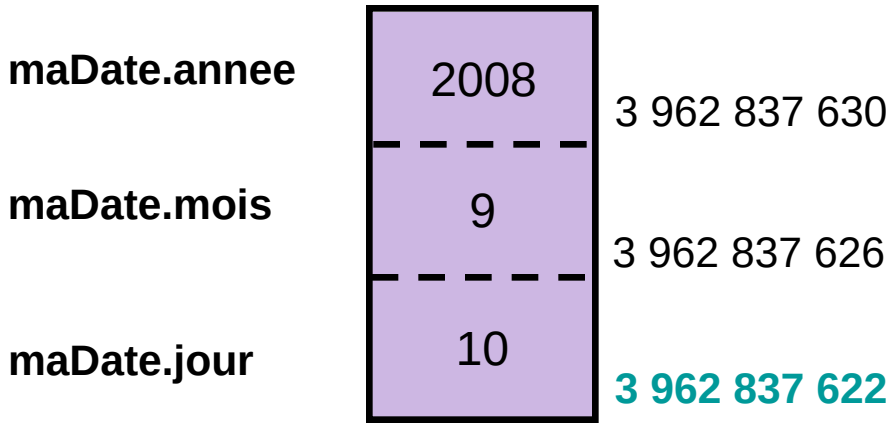
- Accès aux champs : opérateur **.**

- Possibilité d'utiliser typedef pour éviter l'emploi du mot-clé struct dans les déclarations ultérieures

```
struct date  {
        int jour;
        int mois;
        int annee;
};
typedef struct date s_date;

struct etudiant {
        int numero;
        char nom[60];
        char prenom[60];
        s_date datenaiss;
        double notes[10];
};
typedef struct etudiant s_etudiant;

s_etudiant saisir(int num)
{
        s_etudiant etu;
        etu.numero = num;
        printf("Saisir le nom : \n »);
        scanf("%59s", etu.nom);
        /* ... */
        return etu;
}
```

Structures : stockage en mémoire

- Adresse d'une structure = adresse de son premier champ




```
struct date  {
        int jour;
        int mois;
        int annee;
};
typedef struct date s_date;


int main()
{
        s_date maDate;
        maDate.jour = 10;
        maDate.mois = 9;
        maDate.annee = 2008;
        ...
}
```

&maDate == 3962837622 (type s_date *)
&maDate.jour == 3962837622 (type int *)

sizeof(s_date) == 3 * sizeof(int)

Pointeurs sur structures en C

```
s_date * pdate;
s_date maDate;



pdate = &maDate;
(*pdate).jour = 8;       /* ou:  pdate->jour = 8; */

(*pdate).mois = 12;      /* ou:  pdate->mois = 12; */

(*pdate).annee = 1999;   /* ou:  pdate->annee = 1999; */
```