Eléments d'Informatique

Cours 7 – Tableaux de variables et fonctions d'argument de type tableau

Catherine Recanati

UNIVERSITÉ PARIS 13 NORD

Plan général

- **Représentation des nombres. Notion de variable.**
- **Programme. Expressions.**
- **Architecture des ordinateurs: langage machine, langage assembleur, AMIL.**
- **Systèmes d'exploitation : fichiers, processus, compilation.**
- **Instructions de contrôle: boucles et branchements.**
- **Programme. Définition de fonction. Appel fonctionnel.**
- **Tableaux de variables et fonctions d'arguments de type tableau.**
- Sens d'un programme, pile d'exécution, compilation.
- Pointeurs et tableaux.
- Chaines de caractères, bibliothèque <string.h>.
- Allocation dynamique, liste chaînées.
- Révisions.

- Cours 7 –
Tableaux de variables et fonctions d'arg. de type tableau

- Rappels
- Tableaux de variables
- Déclarations et Initialisations
- Fonctions d'argument de type tableau
- Passage d'argument de type tableau
- Warning du compilateur
- Conversions de types

La mémoire

Au lancement d'un programme, le code machine du programme est chargé en mémoire de travail, puis exécuté.

La mémoire de travail (ou mémoire vive) est le dispositif électronique dans lequel sont stockées les données en cours de traitement. Les données y sont codées en binaire et regroupées en octets.

Du point de vue logiciel la mémoire se présente comme une suite de mots (ou cases) mémoire, généralement de 2 ou 4 octets, numérotés à partir de 0. Les numéros sont les adresses des cases mémoire.

Variable impérative

Une variable impérative possède :

- un nom : son identificateur.
- un type : représentant un ensemble abstrait de données.
- une adresse qui est un numéro de mémoire alloué dynamiquement à l'exécution d'une déclaration en fonction du type de la variable.
- une valeur qui est encodée par les bits qui se trouvent à l'adresse de la variable. On a besoin du type pour décoder une valeur à partir de son adresse.

On affecte une valeur à une variable en recopiant les bits de la valeur d'une expression à l'adresse de la variable. Le compilateur vérifie que le type de l'expression et celui de la variable sont compatibles.

Types primitifs

Le *type* d'une variable spécifie l'ensemble des valeurs abstraites qu'elle peut prendre. Il détermine les opérations qu'on peut faire avec la variable. Les types primitifs sont :

1. des types numériques (entiers ou de réels : `int`, `unsigned int`, `short int`, `float`, `double`, etc.).
2. un type caractère: le type `char`. Il recouvre l'ensemble des caractères ascii ou unicode, selon la plateforme.

- Les types primitifs ne sont que partiellement définis par le langage C: ils dépendent fortement de la plateforme.
- Un *type* primitif sur une plateforme donnée a une taille donnée: `sizeof` (*type*). C'est la place prise par l'encodage d'une valeur de ce type.

Types complexes

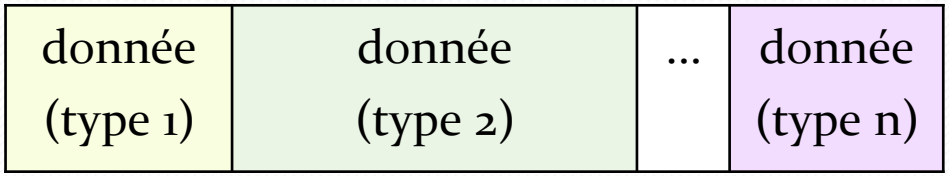
Le langage C propose deux types complexes permettant au programmeur d'utiliser d'autres types construits à partir des types primitifs :

- le type tableau qui peut contenir un certain nombre de données d'un même type
- le type structure qui peut contenir des données de types différents

Plan

- Un tableau est une variable composée de données de même type, stockées de manière contiguë en mémoire (les unes à la suite des autres).

- Une structure est une variable composée de données de types hétérogènes, stockées en mémoire les unes derrière les autres.



Tableaux

```
int tab [4] ;
```

On déclare ici un tableau de 4 variables de type `int`. Ce tableau est statique (sa taille est fixe et ne peut être modifiée).

Cette déclaration permet d'accéder à quatre cases mémoires comme s'il s'agissait de variables, à l'aide d'identificateurs composés d'un indice, comme le sont les composantes d'un vecteur mathématique :

`tab[0],tab[1],tab[2]` et `tab[3]`

Nombre d'éléments

Le nombre d'éléments d'un tableau est calculable à partir de la taille du tableau, et de celle de son 1er élément (ou du type de son 1er élément) :

nb d'éléments = `sizeof tab / sizeof tab[0]`

Note : l'opérateur `sizeof` renvoie la taille en bytes d'une variable ou d'un type. La taille renvoyée a le type `unsigned long`, donc le nombre d'éléments ainsi calculé aussi.

Nombre d'éléments

```
#define nbElem(t) sizeof t/sizeof t[0]
```

Cette définition s'adresse au préprocesseur.

Elle a pour effet de remplacer (dans le texte source du programme) toutes les apparitions (ou occurrences) de `nbElem(<qqc>)`

par

`sizeof <qqc>/sizeof <qqc>[0]`

Attention: à ne pas terminer une ligne commençant par `#define` par ";".

Tableaux

!!!! W A R N I N G !!!!

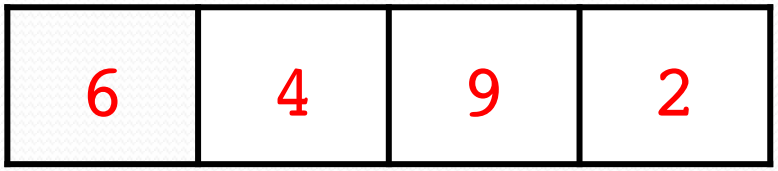
Si on tente d'accéder à `t[j]` avec un indice `j` négatif ou supérieur ou égal au nombre d'éléments du tableau, l'exécution du programme risque d'être interrompue, ou le programme pourrait terminer en retournant un résultat faux, car la variable `t[j]` n'est en effet pas définie.

Dans ce cas, un avertissement (*warning*) sera émis par le compilateur.

Déclaration + Initialisation

- Avec `int tab[4] = {6,4,9,2};`

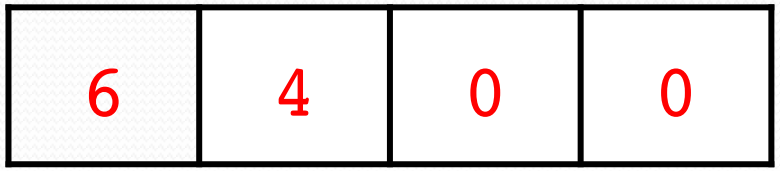
Les valeurs des variables `tab[0]`, ..., `tab[3]` seront définies par les constantes figurant entre accolades. Il doit y avoir au moins une constante, et pas plus que le nombre d'éléments déclaré.



Déclaration + initialisation

- Avec `int tab[4] = {6,4};`

Les deux premières valeurs seront définies par 6 et 4, et les suivantes par 0.



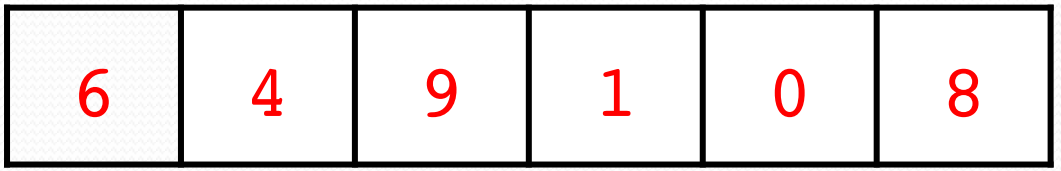
Ainsi, pour initialiser le tableau par des valeurs nulles, on pourra écrire

```
int tab[4] = {0};
```

Déclaration + initialisation

- Avec `int tab[] = {6,4,9,1,0,8};`

Les valeurs des variables `tab[0]`, ... et `tab[5]` seront définies par ces valeurs constantes, et le nombre d'éléments du tableau sera égal au nombre de constantes figurant entre les accolades, soit ici six.



- Avec `int tab[4];`

L'espace mémoire est réservé mais les valeurs ne sont pas initialisées.

On pourra initialiser ces valeurs dans la fonction `main`, en faisant des affectations sur les variables tab[*i*].

```
int main() {
      int tab[4];

      tab[0] = 6;  tab[1] = 4;
      tab[2] = 9;  tab[3] = 2;
      return EXIT_SUCCESS;
}
```

Variante avec une boucle et la macro qui renvoie le nombre d'éléments du tableau. On initialise ici toutes les variables du tableau avec l'entier 9.

```
#define nbElem(t) sizeof t/sizeof t[0]
…
int main() {
    int tab[4];

    for (int i=0; i<nbElem(tab) ;i++)
        tab[i]= 9;
}
```

On peut aussi initialiser le tableau dans la fonction `main` en appelant une fonction prenant en paramètre ce tableau.

Par exemple on peut appeler la procédure `init9` qui affecte toutes les valeurs du tableau passer en argument avec l'entier 9. Sa déclaration est

```
void init9(int tableau[],int max);
```

Remarque : Le premier argument a le type tableau de variables de type `int`, sans précision de taille. C'est l'argument `max` de qui permettra de préciser le nb d'éléments.

```
/* déclaration de init9 */
void init9(int tableau[],int max);

int main() {
      int tab[4];

      init9(tab,4);
      return EXIT_SUCCESS;
}
/* définition de init9 */
void init9(int tableau[],int max){
      for (int i=0 ; i<max ; i++)
            tableau[i]= 9;
      return;
}
```

Exemples (à faire en exercice)

Dans toutes les déclarations qui suivent, on suppose que `nb` *est le nombre d'éléments du tableau* `tab` *passé en argument.*

- `void imprime(int tab[], int nb)`

Cette procédure affiche les valeurs du tableau entre crochets en les séparant avec des virgules

- `void init(int tab[],int nb, int val)`

cette procédure initialise un tableau d'entiers en affectant la valeur val à tous ses éléments.

Exemples (à faire en exercice)

- `void imprimeChar(char tab[], int nb)`

cette procédure imprime un tableau de caractères en affichant successivement les valeurs de tous ses éléments.

- `void traduire(short tab[], int nb, char ascii[])`

cette procédure effectue la traduction d'un tableau d'entiers courts en un tableau de caractères (de même taille) où pour chaque i < nb, ascii[i] sera le caractère de code ASCII tab[i].

```
void imprime(int tab[], int nb)
{ /* on suppose que nb est le nb d'elements de tab */
  if (nb == 0) {
        - afficher: [ ]
        - quitter la fonction imprime
   }
   sinon: on sait ici que l'on a nb >=1
   – afficher d'abord [tab[0]
   – faire une boucle à partir de i= 1 qui
            - affiche  ,tab[i]
            - incrémente i
      tant que i<= nb-1 (condition d'entrée)
    ici, on sait que i == nb  car la condition est fausse
    et on a déjà affiché [val0, ..., valn-1
    – afficher  ]
    – retourner void
}
```

```
void imprime(int tab[], int nb)
{/* on suppose que nb est le nb d'elements de tab */
  int i;
  if (nb == 0) {
     printf("[]");/* affiche: []   */
     return; /* retourne void */
  }
  printf("[%d",tab[0]);/* affiche [tab[0] */
  i = 1;
  while (i < nb) {
     printf(",%d", tab[i]);
     i = i+1;
  }
  /* on a déjà affiché [val0, ..., valn-1 */
  printf("]");    /* affiche ] */
  return;
}
```

Pour en savoir plus sur le passage d'argument de type tableau

```
/* déclaration de init9 */
void init9(int tableau[],int max);

int main() {
      int tab[4];

      init9(tab,4);
      return EXIT_SUCCESS;
}
/* définition de init9 */
void init9(int tableau[],int max){
      for (int i=0 ; i<max ; i++)
            tableau[i]= 9;
      return;
}
```

Question: Quelle est la valeur de la variable `tab` de type tableau qui figure dans l'appel `init9(tab,4)` de la fonction `main`?

Réponse:

`tab` est une variable dont la valeur est l'adresse de son premier élément: `&tab[0]`.

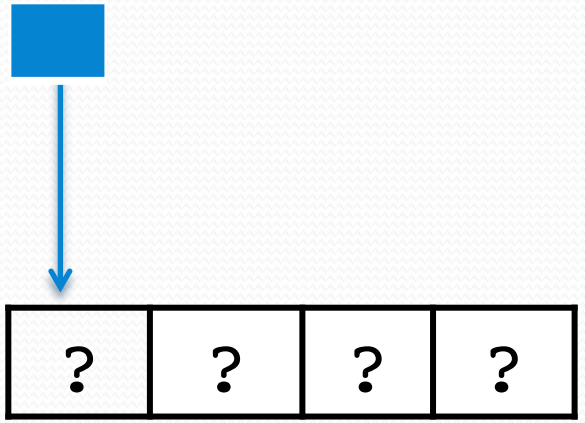
Un tableau `tab` est en effet, par définition en C, une variable dont la valeur est une adresse - celle du début de stockage des variables du tableau, c-à-d. celle de la première variable `tab[0]`.

Revenons à l'exécution de la fonction principale, qui commence par la déclaration du tableau `tab`, suivie de l'appel de la fonction `init9` avec la variable `tab` et la constante `4`.

```
int main() {
      int tab[4];


      init9(tab,4);
      return EXIT_SUCCESS;
}
```

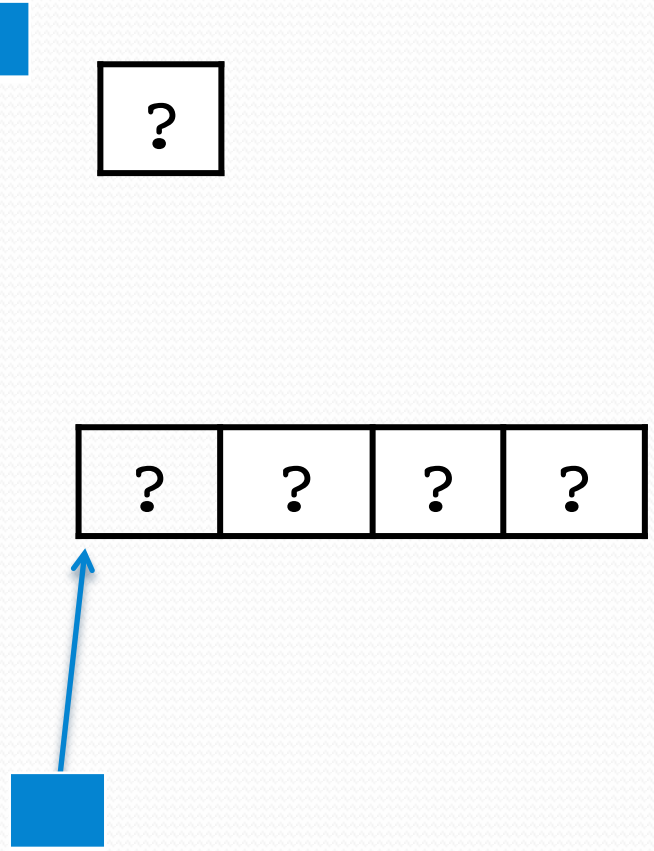
Graphiquement, on peut représenter la déclaration de `tab` par un schéma illustrant des allocations de cases mémoire (effectuées dynamiquement par le code produit par le compilateur) :

```
int main() {
      int tab[4];

      init9(tab,4);
      return EXIT_SUCCESS;
}
```

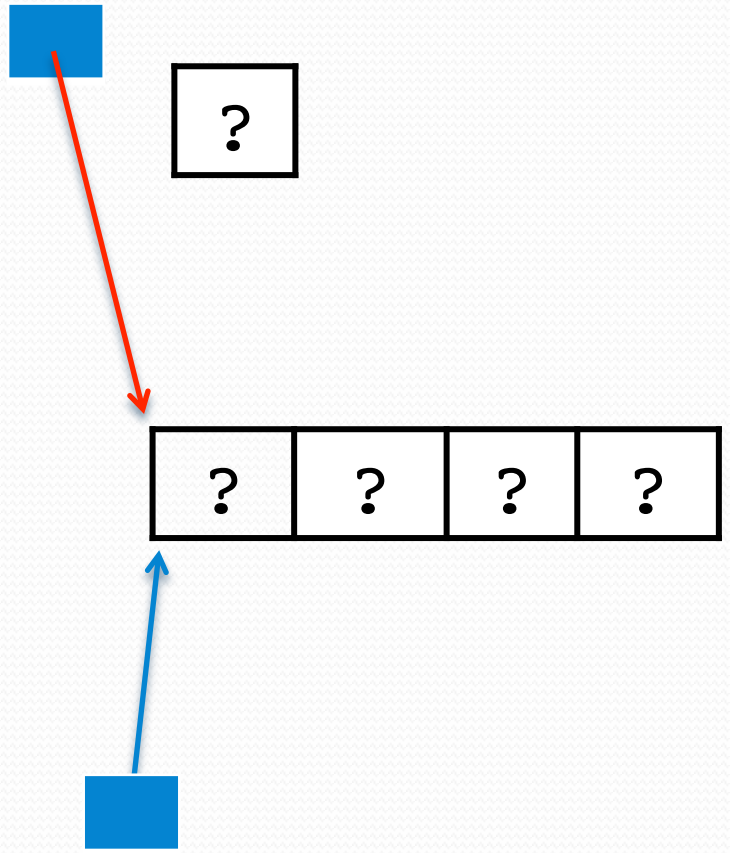


Poursuivons maintenant cette figuration de l'exécution avec l'appel à `init9(tab,4)`.

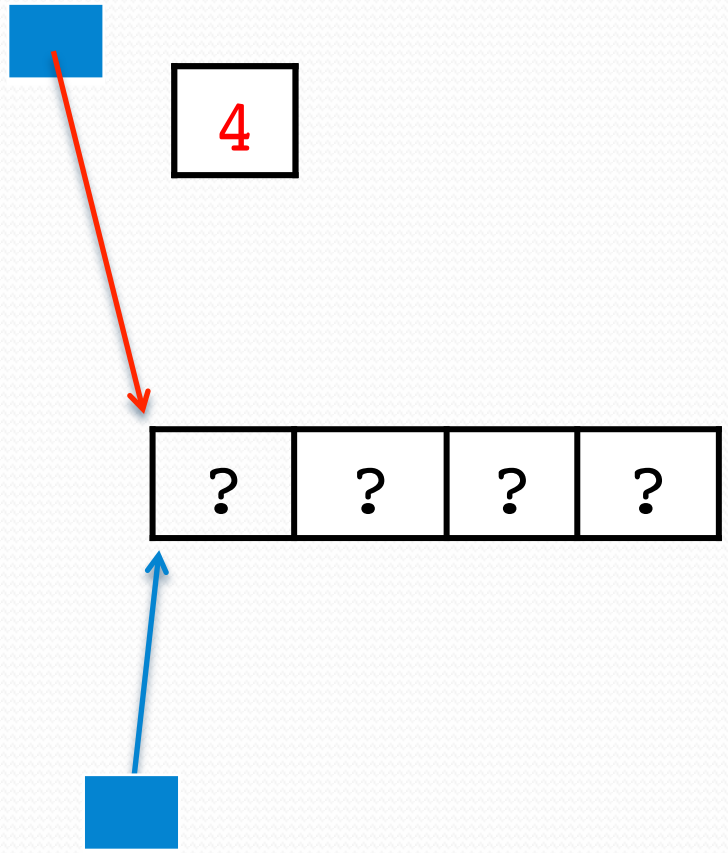
```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{

    ...
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



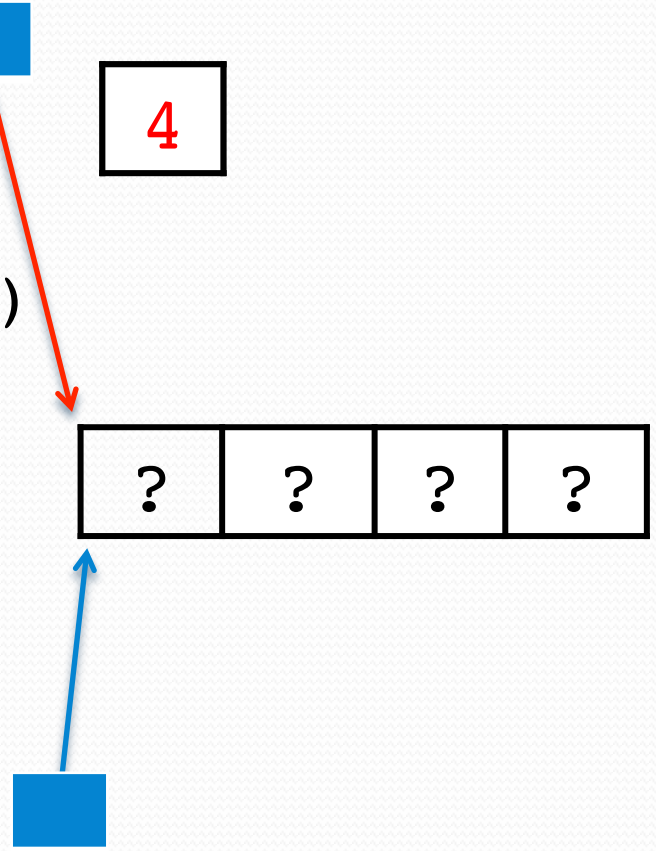
```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{

     ...
     return;
}

int main() {
int tab[4];

      init9(tab,4);
}
```



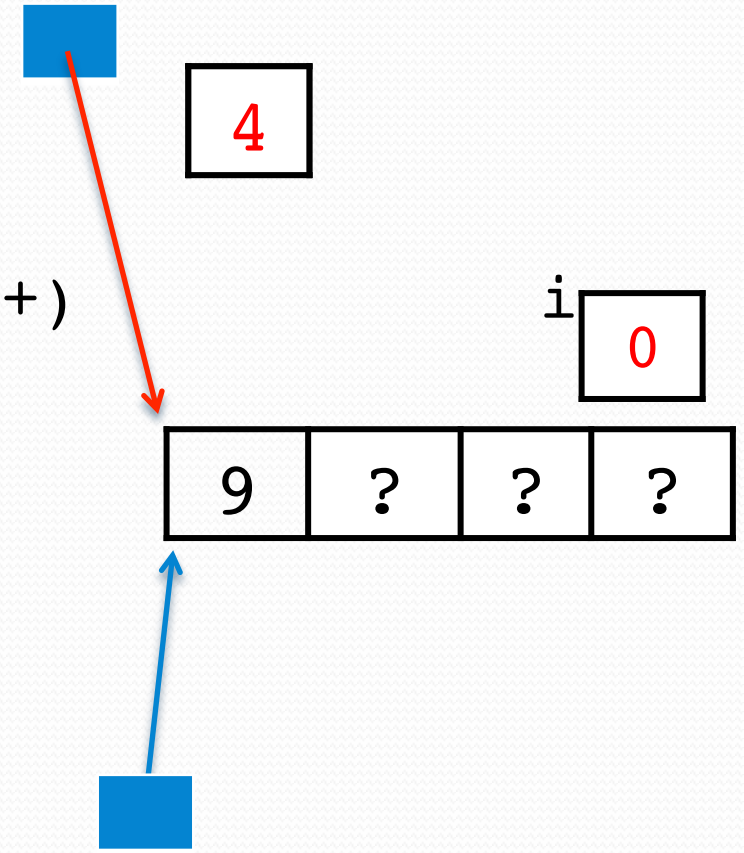
```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{

    ...
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



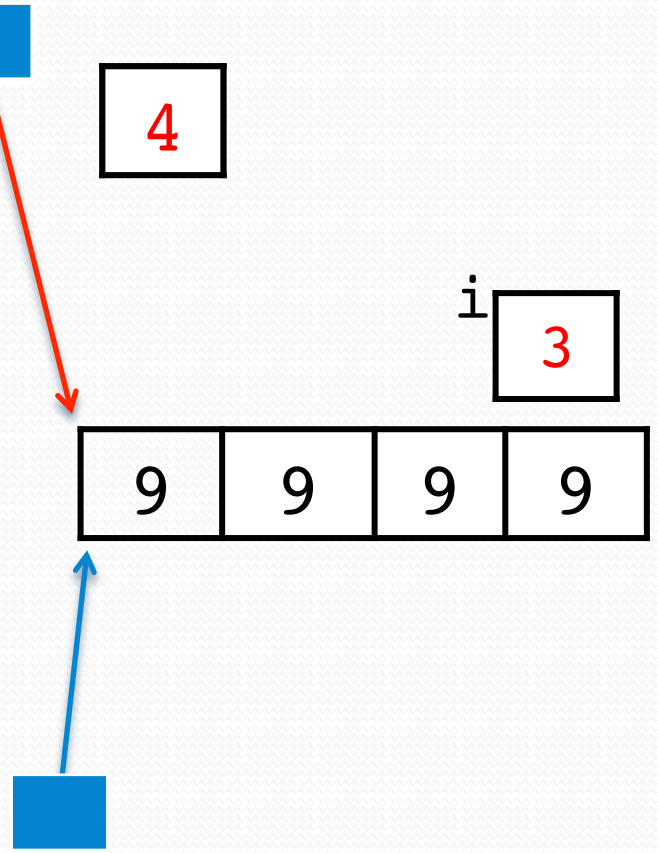
```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{
    for (int i=0; i<max; i++)
         tableau[i]= 9;
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



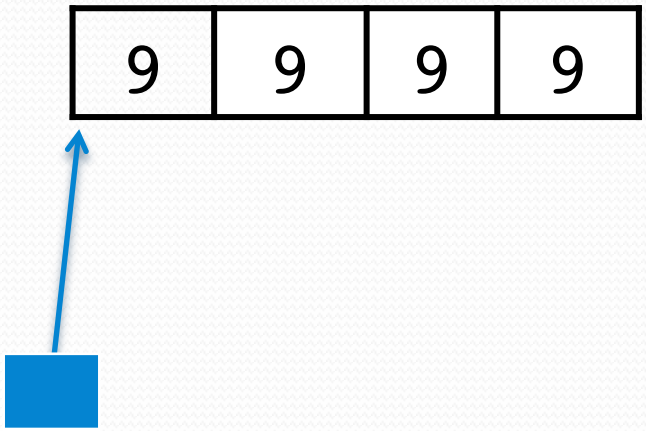
```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{
    for (int i=0; i<max;i++)
         tableau[i]= 9;
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{
    for (int i=0; i<max;i++)
         tableau[i]= 9;
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



```
void init9(
    int tableau[],
    int max /* nb de cases */ )
{
    for (int i=0; i<max;i++)
         tableau[i]= 9;
    return;
}

int main() {
int tab[4];

     init9(tab,4);
}
```



Conclusion

- Le passage d'un argument de type tableau à une fonction permet de modifier les valeurs des variables de ce tableau.

Ces modifications subsistent en effet après le retour de l'appel de la fonction.

- La valeur de la variable de type tableau elle n'a pas été modifiée (c'est toujours la même adresse en mémoire) mais les variables du tableau ont été modifiées.

En particulier ici, cela à permis de les initialiser à certaines valeurs.

Pour en savoir plus

sur les erreurs sémantiques

... qui peuvent exister malgré une compilation syntaxique réussie !

Tableaux

!!! W A R N I N G !!!

Si on tente d'accéder à `t[j]` avec un indice `j` négatif ou supérieur ou égal au nombre d'éléments du tableau, cette variable n'étant pas définie, le programme risque d'être brutalement interrompu durant l'exécution, ou risque de retourner un résultat faux.

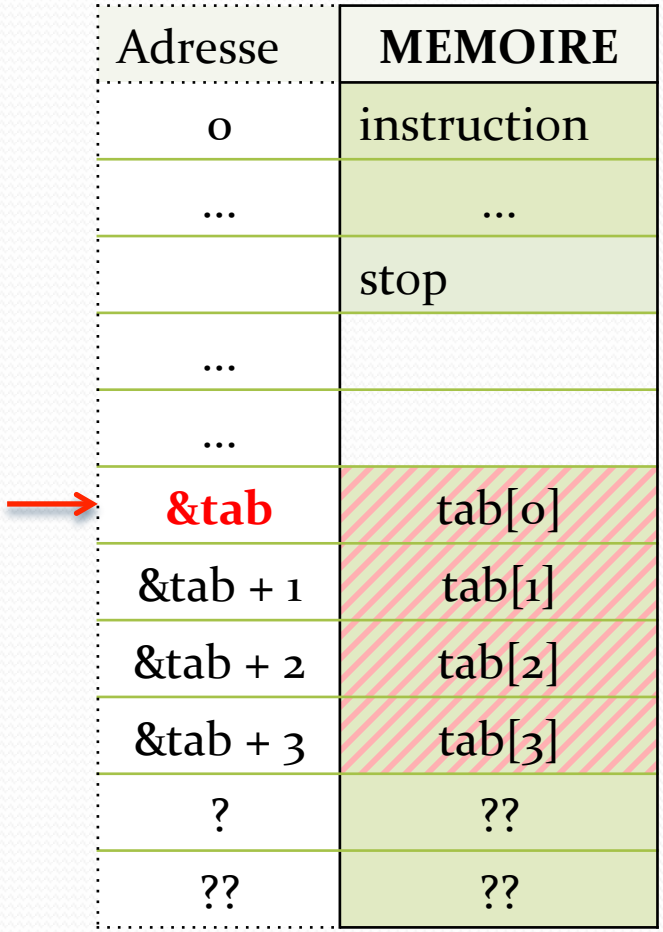
Dans ce cas, un avertissement (*warning*) sera émis par le compilateur.

Pourquoi ?

```
int tab[4];
```

Cette déclaration conduit le compilateur à allouer de la mémoire pour la variable `tab`.

La place allouée est égale à 4 fois la taille d'un `int` et permet de ranger les 4 valeurs de tab[0], tab[1], tab[2] et tab[3].



Adresse	MEMOIRE
0	instruction
…	…
	stop
…	
…	
&tab	tab[0]
&tab + 1	tab[1]
&tab + 2	tab[2]
&tab + 3	tab[3]
?	??
??	??

Mais *que se passe-t-il si le code compilé cherche à accéder à* `tab[4]` (saut à `&tab + 4?`) ?

On distingue alors deux cas, selon que cette adresse appartient à un fragment accessible, ou non, de la mémoire.

Adresse	MEMOIRE
0	instruction
…	…
	stop
…	
…	
&tab	tab[0]
&tab + 1	tab[1]
&tab + 2	tab[2]
&tab + 3	tab[3]

1. Si &tab + 4 n'est pas accessible, l'évaluation de tab[4] provoquera une erreur d'exécution. Dans ce cas, on verra l'affichage du message d'erreur *segmentation fault* ou *memory fault* qui signifie qu'on a tenté d'accéder à une adresse non autorisée.

Le message supplémentaire *core dumped* indique que la mémoire a été recopiée dans un fichier qui pourra être exploité par un debugger (eg. gdb).

2. Si &tab + 4 est accessible, il n'y aura pas d'interruption, mais le programme est incorrect car la valeur de &tab + 4 est indéterminée.

Une autre erreur d'accès risque de se produire plus loin, et, même si le programme termine sans aucune interruption, son résultat peut être faux.

Adresse	MEMOIRE
0	instruction
…	…
	stop
…	
…	
&tab	tab[0]
&tab + 1	tab[1]
&tab + 2	tab[2]
&tab + 3	tab[3]
?	??
??	??

Dans la pratique…

La compilation avec la commande

```
gcc –Wall monfichier.c –o exec
```

affiche un *Warning* en cas d'attribution d'une valeur d'un type donné à une variable de type différent.

ex: int x = 4.5; provoque l'affichage

```
warning: implicit conversion from
'double' to 'int' changes value from
4.5 to 4
```

Conversions implicites

Le compilateur effectue en effet (dans le but de produire un code exécutable) des conversions implicites de types dans les opérations sur les variables.

Ici, il a fait la conversion `double -> int`.

Exemples:

1.
```
double x;
int y = 3;

x = y + 4.5;
```

L'addition entre deux objets de types `int` et `double` conduit à calculer une valeur `double` (ici `7.5`) qui sera affectée à `x`.

Conversions implicites

```
2. y = y + 4.5;
```

Cette fois, la valeur `double` (`7.5`) sera convertie en le type de `y`, ici `int`.

Il y aura suppression de bits et `y` se verra attribuer la valeur entière `7`.

Conversions implicites

Les règles du C concernant les opérateurs (+,-,*,/) entre valeurs de différents types sont de convertir l'opérande du type le plus « étroit » en celui du type le plus « large », et de retourner une valeur du type le plus large.

Les inclusions de types, en allant du plus étroit vers le plus large, vont dans cet ordre :

char short int long float double long double

Le C autorise aussi des inclusions de types `signed` ou `unsigned`, mais les résultats peuvent être parfois étranges et sont dépendants de la plateforme.

Conversions forcées: cast

Pour supprimer ces *warning*, on peut utiliser une conversion explicite (appelée aussi conversion forcée, *cast* ou *casting* en anglais) :

```
int  x =  (int) 3.8;
```

Ici, c'est le programmeur qui force la conversion en indiquant au compilateur la conversion de type qu'il souhaite forcer (c'est le type entre parenthèses devant l'expression).

Sans cette conversion forcée, le compilateur aurait émis un *warning*.

On retiendra que pour supprimer les *warning* sur les types du compilateur, on peut corriger le programme en rajoutant les opérations de cast souhaitées par le compilateur.

L'intérêt des *warning* est d'attirer l'attention du programmeur sur les incohérences dans l'usage de ses types de données.

Plan

Merci pour votre attention !

Des questions ?