Séance 3 : Les Fonctions, les tableaux

Langage C : séance 3

Les tableaux statiques :

- Déclaration d'un tableau statique :

type nom_tableau[taille];

Si la variable est locale ➔ stocké dans la pile.
Si la variable est globale ➔ stocké dans le segment des données globales.
Toutes les données du tableau seront consécutives en mémoire.
L'expression « nom_tableau » correspond à l'adresse du premier élément.
L'expression « nom_tableau » est une adresse constante.

Exemple de déclaration d'un tableau de 10 entiers :

```
int montableau[10];
```

- Déclaration et initialisation d'un tableau statique :

type nom_tableau[taille]={ valeur1, valeur2, … };

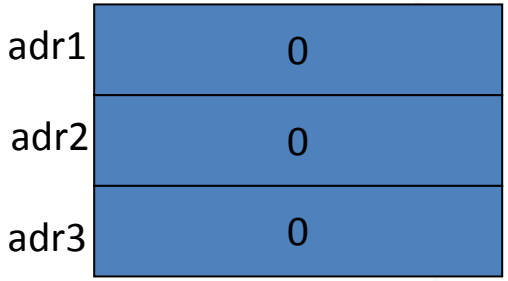
Le nombre de valeur doit être inférieure ou égal à taille.
on peut écrire : **type nom_tableau[]={ valeur1, valeur2, … };**

Langage C : séance 3

Les tableaux statiques :

- Déclaration d'un tableau statique :

int nom_tableau[3]={0,0,0};



Occupation mémoire de ce tableau :

3 (cases) x 4 (octets/int) = 12 octets consécutifs.

sizeof(nom_tableau) ➔ 12

Adresse mémoire de la première case :

« nom_tableau » ➔ adr1

printf(''%p'', nom_tableau); ➔ affiche la valeur de l'adresse en héxa (adr1)

printf(''%p'', nom_tableau+1); ➔ affiche la valeur de l'adresse en héxa (adr2)

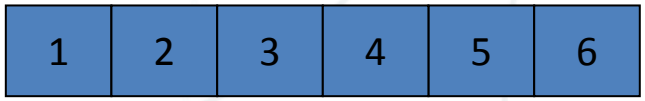
Langage C : séance 3

Les tableaux statiques de tableaux :

- Déclaration d'un tableau statique de tableaux :

type nom_tableau[taille1][taille2];

Toutes les données seront consécutives en mémoire.
int t[2][3]={1,2,3,4,5,6}; ou int t[2][3]={{1,2,3},{4,5,6}};



L'expression « nom_tableau » correspond à l'adresse du premier élément(&t[0][0]).
L'expression « nom_tableau » est une adresse constante.

Accès : t[0][1] ➔ 2
Aucune vérification de débordement :
On peut écrire : t[40][50]=8; ➔ compilation : ok, exécution : pb!!!

Langage C : séance 3

Le passage de tableaux statiques en paramètre de fonction :

- **Passage de tableau :**

Tout paramètre de fonction déclaré comme tableau d'éléments de type T est automatiquement converti en un pointeur vers T à la compilation.

On peut donc écrire :

Pour un tableau à une dimension,

void f(T t[]); ou void f(T * t);

Pour un tableau à deux dimensions , il faut toujours préciser la dernière dimension :

void g(T t[][10]); ou void g(T (*t)[10]);

Langage C : séance 3

Type de retour d'une fonction :

- **Pas de type de retour : void**

Exemple :

```
void messageFin() {
        printf(''bonne fin de journée'');
}
```

- **Avec un type non void :**

Exemple :

```
int maxEntier(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

Type de retour non void donc au moins un return dans la fonction

On sort de la fonction dès qu'un « return » est exécuté.

Appel de la fonction :

```
k=maxEntier( i,j );
ou  k=maxEntier(10,20);
```

Il faut récupérer la valeur de retour à l'appel, Sinon elle est perdue.

Langage C : séance 3

Passage par valeur de paramètres :

Exemple :

```
void afficherEntier(int j) {
        printf(" valeur = %d" , j);
}

int main() {
int i=5;
afficherEntier(i);
return 0;
}
```

Paramètre formel de la fonction

Variable locale (au main)

La variable i est un paramètre réel ou effectif lors de l'appel à la fonction

Langage C : séance 3

Passage par valeur de paramètres :

Exemple :

```
void afficherEntier(int j) {
        printf(" valeur = %d" , j);
}

int main() {
int i=5;
afficherEntier(i);
return 0;
}
```

Variable locale (au main)

Etape 1 de l'exécution :

création de la variable i dans la pile (zone des variables locales) et initialisation

Langage C : séance 3

Passage par valeur de paramètres :

Exemple :

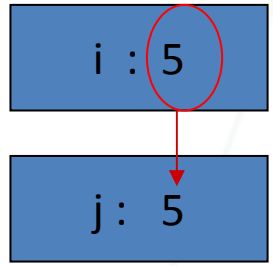
```
void afficherEntier(int j) {
        printf(" valeur = %d" , j);
}

int main() {
int i=5;
afficherEntier(i);
return 0;
}
```

Paramètre formel de la fonction

La variable i est un paramètre réel ou effectif lors de l'appel à la fonction

Recopie de la valeur du paramètre réel (i) pour initialiser le paramètre formel j



Etape 2 de l'exécution :
création de la variable j dans la pile,
et initialisation avec la valeur du paramètre réel.

Langage C : séance 3

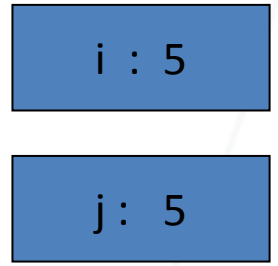
Passage par valeur de paramètres :

Exemple :

```
void afficherEntier(int j) {
        printf(" valeur = %d" , j);
}

int main() {
int i=5;
afficherEntier(i);
return 0;
}
```

Paramètre formel de la fonction



Etape 3 de l'exécution :

Affichage de la valeur du paramètre formel j.

Langage C : séance 3

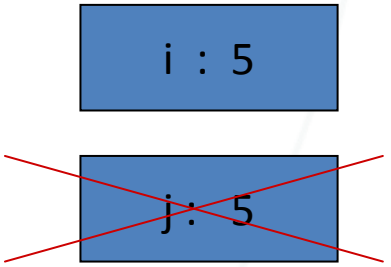
Passage par valeur de paramètres :

Exemple :

```
void afficherEntier(int j) {
        printf(" valeur = %d" , j);
}

int main() {
int i=5;
afficherEntier(i);
return 0;
}
```

Etape 4 de l'exécution :
Retour dans le main : destruction des variables de la fonction afficheEntier.



Langage C : séance 3

Passage par valeur de paramètres :

Résumé du passage par valeur :

- **Le passage par valeur ne modifie pas le paramètre réel**.

```
void f(T t1);
T t=val1;
f(t);
// t a toujours la même valeur (val1)
```

- **Lors d'un passage par valeur, la valeur du paramètre réel peut être une constante**.

```
void f(int t1);
f( 10);        // autorisé
```

Langage C : séance 3

Principe de la récursivité :

La récursivité permet de décrire un processus en faisant appel à ce même processus.

Exemple : calcul de la factorielle.

fact(0)=1

fact(n) =n x fact(n-1)

En langage C :

- toute fonction en C peut-être récursive.
- une fonction est récursive si elle s'appelle elle-même.
- la récursivité est plus coûteuse en mémoire pour le système qu'une simple boucle.

Langage C : séance 3

Fonction récursive :

Certaines fonctions qui s'écrivent avec une boucle peuvent également s'écrire en utilisant le principe de récursivité.

Exemple :

Calcul de la somme des n premiers entiers positifs :

```
/* version itérative : */
long somme(long n)
{
    long i,s=0;
    for (i=1 ; i<=n ; i++)
        /* accumulation dans une variable */
        s+=i;
    return s;
}
```

```
/* version récursive : */
long somme(long n)
{
    long s=0;
    if  (n==1) /* condition d'arrêt */
        s=1;
    else
        s=n+somme(n-1);
    return s;
}
```

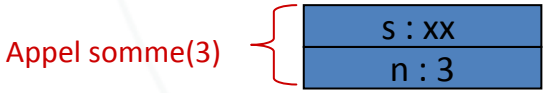
Langage C : séance 3

Fonction récursive :

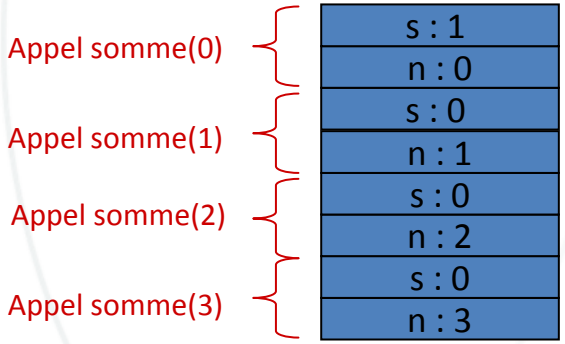
Calcul de l'occupation mémoire maximale (en pile) de l'exécution des fonctions précédentes :

➔Appel : somme(3)

/ version itérative : */*



/ version récursive : */*



Langage C : séance 3

Fonction récursive : quel intérêt ?

Pour écrire certains algorithmes, il faudrait écrire un nombre important de boucles imbriqués (donc algorithme très compliqué et lourd à écrire, à corriger et à faire évoluer).
➔En utilisant la récursivité, certains de ces algorithmes peuvent s'écrire en quelques lignes !

➔ Quelques critères pour une utilisation possible et justifiée de la récursivité :
- La complexité du problème se réduit bien en utilisant la récursivité.
- La méthode itérative ne peut pas se mettre en place, est trop lourde ou peu adaptée.
- Plusieurs solutions à votre problème sont envisageables.
- La complexité du problème est imprévisible, et pas forcément linéaire (explorer des tableaux contenant des tableaux, par exemple).

Les algorithmes de construction de structures de données complexes de types arbres sont plus faciles à implémenter en utilisation la récursivité.