

SL121_LOGA

Programmation en C

Théorie

Chapitre 8

Pointeurs et types complexes

Christian HUBER (CHR)
Version 1.2 Janvier 2008
Version 1.3 Janvier 2010
Version 1.4 Décembre 2011
Version 1.5 Novembre 2014
Version 1.6 Janvier 2016

CONTENU DU CHAPITRE 8

8. Pointeurs et types complexes	8-1
8.1. Objectifs	8-1
8.2. Les pointeurs	8-1
8.2.1. Déclaration d'un pointeurs	8-1
8.2.2. Initialisation d'un pointeur	8-1
8.2.3. Accès indirect aux variables	8-2
8.2.4. Arithmétique des pointeurs	8-4
8.2.4.1. Soustraction de pointeurs	8-4
8.2.4.2. Incrément et addition	8-5
8.2.4.3. Décrément et soustraction	8-5
8.2.5. Imposer une adresse a un pointeur	8-6
8.2.6. Pointeurs, exercice	8-6
8.2.7. Pointeurs, solution exercice	8-7
8.3. Les tableaux	8-8
8.3.1. Déclaration d'un tableau unidimensionnel	8-8
8.3.2. Initialisation d'un tableau unidimensionnel	8-8
8.3.3. Accès aux éléments d'un tableau unidimensionnel	8-9
8.3.4. Tableaux unidimensionnel et pointeurs	8-10
8.3.5. Déclaration d'un tableau multidimensionnel	8-11
8.3.6. Initialisation d'un tableau multidimensionnel	8-11
8.3.7. Rangement en mémoire d'un tableau multidimensionnel	8-12
8.3.7.1. Programme pour observation du rangement en mémoire	8-12
8.3.7.2. Résultat de l'exécution du programme	8-13
8.3.8. Accès aux éléments d'un tableau multidimensionnel	8-14
8.3.9. Tableaux multidimensionnels et pointeurs	8-14
8.4. Les structures	8-15
8.4.1. Déclaration des structures	8-15
8.4.1.1. Déclaration globale	8-15
8.4.1.2. Déclaration locale	8-15
8.4.2. Déclaration des variables de type struct	8-16
8.4.2.1. Déclaration variable avec structure globale	8-16
8.4.2.2. Déclaration variable avec structure locale	8-16
8.4.3. Initialisation des variables de type struct	8-16
8.4.4. Accès aux champs des variables de type struct	8-17
8.4.4.1. exemple affectation valeurs aux champs d'une structure	8-17
8.4.5. Structure et pointeur	8-17
8.4.6. tableaux de structures	8-18
8.5. Les unions	8-19
8.5.1. Déclaration des unions	8-19
8.5.1.1. Déclaration globale	8-19
8.5.1.2. Déclaration locale	8-20
8.5.2. Déclaration des variables de type union	8-20
8.5.2.1. Déclaration variable avec union globale	8-20

8.5.2.2.	Déclaration variable avec union locale	8-20
8.5.3.	Initialisation des variables de type union	8-21
8.5.4.	Accès aux champs des variables de type union	8-21
8.5.4.1.	exemple affectation valeurs aux champs d'une union	8-21
8.5.5.	Organisation des valeurs en mémoire	8-22
8.5.6.	Listing du programme	8-23
8.6.	Les champs de bits (bit fields)	8-24
8.6.1.	Déclaration de champs de bits	8-24
8.6.2.	Utilisation de champs de bits	8-24
8.6.2.1.	Vérification du résultat	8-25
8.6.2.2.	Union utilisée	8-25
8.6.3.	Champs de bits conclusion	8-25
8.7.	Conclusion	8-26
8.8.	Historique des version	8-26
8.8.1.	Version 1.2 Janvier 2008	8-26
8.8.2.	Version 1.3 Janvier 2010	8-26
8.8.3.	Version 1.4 Décembre 2011	8-26
8.8.4.	Version 1.5 Novembre 2014	8-26
8.8.5.	Version 1.6 Janvier 2016	8-26

8. POINTEURS ET TYPES COMPLEXES

Ce chapitre a pour but d'introduire les pointeurs et les types complexes, afin de permettre l'utilisation de donnée plus complexes que de simples variables. Par types complexes, il faut comprendre les tableaux, les structures et les unions.

8.1. OBJECTIFS

A la fin de ce chapitre, l'étudiant sera capable de :

- Déclarer et utiliser des pointeurs sur des variables, des tableaux et des structures.
- De déclarer et d'utiliser des tableaux
- De déclarer et d'utiliser des structures
- De déclarer et d'utiliser des unions

8.2. LES POINTEURS

Un pointeur est un nouveau type, le type pointeur. Une variable du type pointeur contient une adresse, en général l'adresse d'une variable. Dans le passage de paramètre à une fonction, on parle de passage par valeur ou de passage par référence. Lors d'un passage par référence on fournit l'adresse de la variable. Dans le prototype de la fonction le paramètre est du type pointeur.

8.2.1. DECLARATION D'UN POINTEURS

La déclaration d'un pointeur est assez proche à celle d'une variable à l'exception du symbole * qui indique la notion d'indirection.

Syntaxe : `type *pNom;`

Le type peut être un des type de base que nous avons déjà vu, ou un type complexe que nous allons étudier dans ce chapitre. Le pNom représente le nom de la variable pointeur. Le préfix p est une recommandation pour éviter de confondre une variable avec un pointeur. Si il existe une variable X, pX est un pointeur sur la variable X.

Voici quelques exemples de déclarations de pointeurs.

```
int    *pValInt;        // pointeur sur un entier
char   *pValChar;       // pointeur sur un caractère
float  *pValInt;        // pointeur sur un float
```

8.2.2. INITIALISATION D'UN POINTEUR

L'initialisation d'un pointeur consiste à lui attribuer une adresse. Comme les adresses ne sont généralement connue que du compilateur, on utilise le symbole & signifiant ici "adresse de" et on doit donner le nom d'un élément existant comme une variable.

Par exemple

```
int    Val1;            // variable entière
int    *pVal1 = &Val1;  // pointeur sur la variable entière Val1
```

Nous allons reprendre l'exemple précédant en présentant la situation dans la mémoire et en séparant la déclaration et l'initialisation du pointeur.

<code>int Val1;</code>	Val1	<div style="border: 1px solid black; padding: 2px 20px; display: inline-block;">?</div>	12FED4
<code>int *pVal1;</code>	pVal1	<div style="border: 1px solid black; padding: 2px 20px; display: inline-block;">?</div>	12FEC8
<code>pVal1 = &Val1;</code>	pVal1	<div style="border: 1px solid black; padding: 2px 20px; display: inline-block;">12FED4</div>	12FEC8

Après l'exécution de `pVal1 = &Val1;` pVal1 contient l'adresse de la variable Val1.

La valeur des adresses des variables dépend de la plateforme (processeur et compilateur utilisé). Elle dépend aussi si la déclaration est globale ou locale à une fonction.

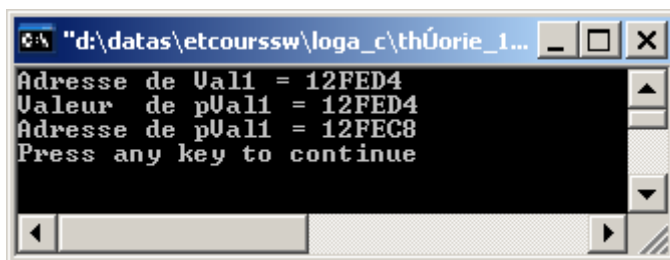
En écrivant le programme suivant avec le Visual C++

```
#include <stdio.h>

int main (void)
{
    int Val1;
    int *pVal1;
    pVal1 = &Val1;
    printf ("Adresse de Val1 = %X \n", &Val1);
    printf ("Valeur de pVal1 = %X \n", pVal1);
    printf ("Adresse de pVal1 = %X \n", &pVal1);

    return(0);
}
```

On obtient le résultat suivant :



On constate que l'on obtient la même valeur pour l'adresse de Val1 que pour la valeur de pVal1, puisque pVal1 contient l'adresse de Val1.

8.2.3. ACCES INDIRECT AUX VARIABLES

En utilisant un pointeur il est possible d'accéder en lecture ou en écriture à une variable. Voici un exemple avec deux variables et deux pointeurs.

<code>int Val1;</code>	Val1	<div style="border: 1px solid black; padding: 2px 20px; display: inline-block;">?</div>	12FED4
<code>int Val2;</code>	Val2	<div style="border: 1px solid black; padding: 2px 20px; display: inline-block;">?</div>	12FEC8

<code>int *pVal1 = &Val1;</code>	<code>pVal1</code>	<table border="1"><tr><td>12FED4</td></tr></table>	12FED4	12FEB0
12FED4				
<code>int *pVal2 = &Val2;</code>	<code>pVal2</code>	<table border="1"><tr><td>12FEC8</td></tr></table>	12FEC8	12FEB0
12FEC8				
<code>*pVal1 = 25;</code>	<code>Val1</code>	<table border="1"><tr><td>25</td></tr></table>	25	12FED4
25				
<code>*pVal2 = *pVal1;</code>	<code>Val2</code>	<table border="1"><tr><td>25</td></tr></table>	25	12FEC8
25				

Au niveau des deux dernières lignes, on utilise l'opérateur * dans le rôle d'indicateur d'indirection. L'action est qualifiée de déréférencement.

Avec `*pVal1 = 25;` il y a un accès indirecte en écriture à la variable Val1 en passant par le pointeur pVal1 qui contient l'adresse de Val1.

Avec `*pVal2 = *pVal1;` il y a un accès indirecte en lecture à la variable Val1 en passant par le pointeur pVal1 qui contient l'adresse de Val1, suivit d'un accès en écriture à Val2 via pVal2.

Voici le programme complet :

```
#include <stdio.h>

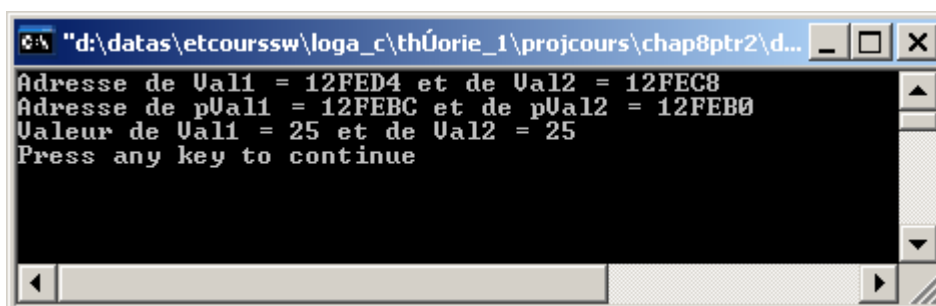
int main (void)
{
    int Val1;
    int Val2;
    int *pVal1 = &Val1;
    int *pVal2 = &Val2;

    *pVal1 = 25;
    *pVal2 = *pVal1;

    printf ("Adresse de Val1 = %X et de Val2 = %X \n",
            &Val1, &Val2);
    printf ("Adresse de pVal1 = %X et de pVal2 = %X \n",
            &pVal1, &pVal2);
    printf ("Valeur de Val1 = %d et de Val2 = %d \n",
            Val1, Val2);

    return(0);
}
```

Voici le résultat lors de l'exécution du programme :



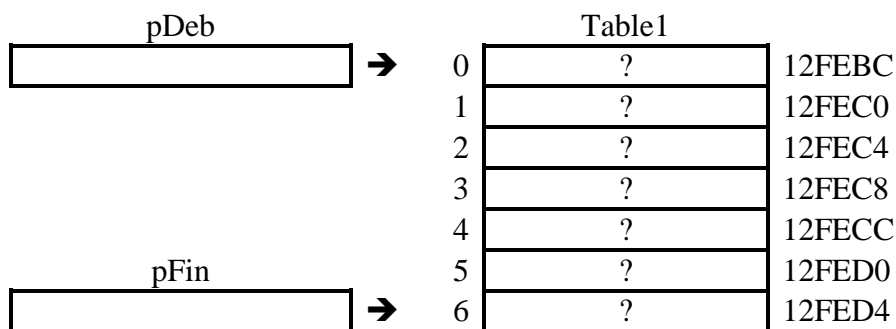
```
C:\d:\datas\etcourssw\loga_c\théorie_1\projcours\chap8ptr2\d...
Adresse de Val1 = 12FED4 et de Val2 = 12FEC8
Adresse de pVal1 = 12FEB0 et de pVal2 = 12FEB0
Valeur de Val1 = 25 et de Val2 = 25
Press any key to continue
```

8.2.4. ARITHMETIQUE DES POINTEURS

Parmi les opérations les plus courantes que l'on applique aux pointeurs, on trouve l'incrément et la décrémentation, ainsi que l'addition et la soustraction. Il est possible d'additionner ou de soustraire une valeur à un pointeur. Il est assez courant de réaliser la différence entre deux pointeurs (soustraction). Effectuer la différence entre deux pointeurs prend un sens si les deux pointeurs sont utilisés pour gérer un bloc de mémoire.

Dans l'exemple suivant, nous allons utiliser un tableau d'entiers (32 bits) pour illustrer l'arithmétique des pointeurs.

Après initialisation des pointeurs nous sommes dans la situation suivante :

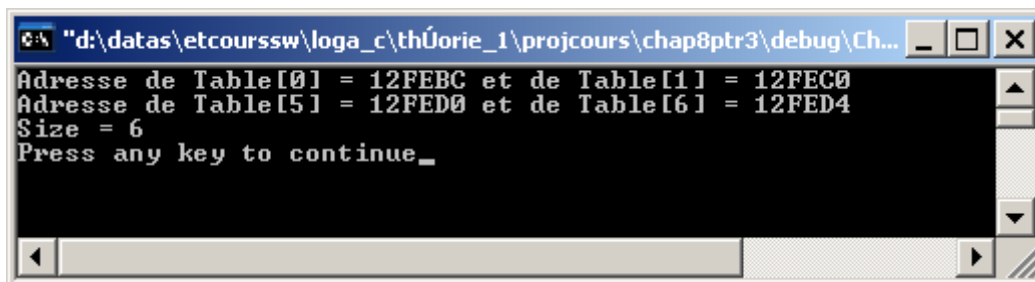


8.2.4.1. SOUSTRACTION DE POINTEURS

Si nous effectuons l'opération suivante :

Size = pFin - pDeb;

Nous obtenons le nombre d'éléments, non compris le dernier, soit 6, malgré le fait que les adresses aillent par pas de 4, car les int avec le Visual C++ sont de 32 bits.



Voici le programme utilisé pour produire ce résultat.

```

#include <stdio.h>

int main (void)
{
    int Table1[7];           // tableau de 7 entiers
    int *pDeb = &Table1[0]; // pointe sur 1er élément
    int *pFin = &Table1[6];  // pointe sur dernier élément
    int Size;
    Size = pFin - pDeb;
  
```



```
printf ("Adresse de Table[0] = %X et de Table[1] = %X\n", &Table1[0], &Table1[1]);  
printf ("Adresse de Table[5] = %X et de Table[6] = %X\n", &Table1[5], &Table1[6]);  
printf ("Size = %d\n", Size);  
  
return(0);  
}
```

8.2.4.2. INCREMENT ET ADDITION

Lorsque on incrémente un pointeur, comme par exemple `pDeb++`; le compilateur n'ajoute pas 1 au pointeur, mais 1x la taille de l'élément pointé soit 4 dans notre exemple.

En ajoutant les lignes suivantes au programme :

```
printf ("Val de pDeb = %X\n", pDeb );  
pDeb++;  
printf ("Val de pDeb = %X\n", pDeb );
```

On obtient:

Val de pDeb = 12FEBC

Val de pDeb = 12FEC0

Donc la valeur du pointeur a été augmentée de 4, ce qui le fait pointer maintenant sur Table[1].

Pour illustrer l'addition, ajoutons les lignes suivantes au programme :

```
printf ("Val de pDeb = %X\n", pDeb );  
pDeb += 2;  
printf ("Val de pDeb = %X\n", pDeb );
```

On obtient:

Val de pDeb = 12FEC0

Val de pDeb = 12FEC8

Donc la valeur du pointeur a été augmentée de 2*4, ce qui le fait pointer maintenant sur Table[3].

8.2.4.3. DECREMENT ET SOUSTRACTION

Le même principe s'applique si on décrémente ou soustrait une valeur à un pointeur.

Pour illustrer la soustraction, ajoutons les lignes suivantes au programme :

```
printf ("Val de pDeb = %X\n", pDeb );  
pDeb -= 3;  
printf ("Val de pDeb = %X\n", pDeb );
```

On obtient:

Val de pDeb = 12FEC8

Val de pDeb = 12FEBC

Donc la valeur du pointeur a été diminuée de 3*4, ce qui le fait pointer maintenant sur Table[0].

8.2.5. IMPOSER UNE ADRESSE A UN POINTEUR

Si en reprenant notre exemple avec la table de 7 éléments et les deux pointeurs et que l'on cherche à imposer une valeur à pFin, comme ci-dessous:

```
pFin = 0x12FED0;
```

On obtient : *error C2440: '=' : impossible de convertir de 'int' en 'int *'*

Car on a un conflit de type. Il est nécessaire d'indiquer que cette valeur correspond à un pointeur.

```
pFin = (int *)0x12FED0;
```

Cette méthode est assez dangereuse, on ne l'utilisera que dans les cas particulier où on connaît les adresses du hardware.

8.2.6. POINTEURS, EXERCICE

Dans le programme principal ci-dessous on souhaite utiliser la fonction permute qui permet de croiser le contenu de deux variables que l'on passe par référence à la fonction.

Complétez le programme, (définition de la fonction, corps de la fonction et appelle de la fonction).

```
#include <stdio.h>

void permute (short int ....., short int .....)
{
    .....
    .....
    .....
    .....
    .....
    .....
}

int main (void)
{
    short int Val1 = 12;
    short int Val2 = 59;

    printf ("Val1 = %d Val2 = %d \n", Val1, Val2 );
    permute(....., .....);
    printf ("Val1 = %d Val2 = %d \n", Val1, Val2 );

    return(0);
}
```

8.2.7. POINTEURS, SOLUTION EXERCICE

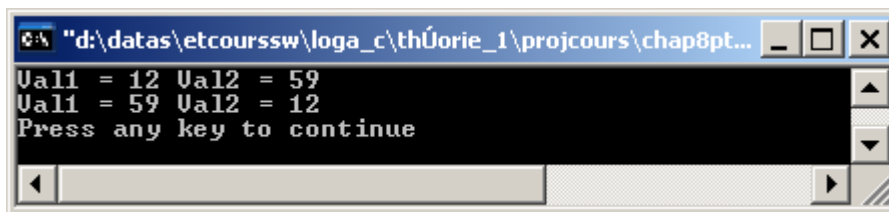
Voici la fonction permute :

```
void permute (short int *p1, short int *p2)
{
    short int tmp;
    tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

Appel de permute :

```
permute(&Val1, &Val2);
```

Résultat :



8.3. LES TABLEAUX

Un tableau est une variable qui se compose d'un certain nombre de données élémentaires de même type, rangées en mémoire les unes à la suite des autres. Chaque donnée élémentaire représente elle-même une variable. Le type des éléments du tableau peut être n'importe lequel du langage C

- types élémentaires : char, short, int, long, float, double
- pointeurs
- tableau
- structure

Il est possible de créer des tableaux à une dimension ou à plusieurs dimension.

8.3.1. DECLARATION D'UN TABLEAU UNIDIMENSIONNEL

La déclaration d'un tableau unidimensionnel est la suivante :

<Type> <Nom du tableau> [Nombres d'éléments] ;

Voici quelques exemples :

```
short int TableVal1 [10];    // tableau de 10 short int
float TableVal2 [5];        // tableau de 5 float
char Buffer1 [20];          // tableau de 20 char
```

8.3.2. INITIALISATION D'UN TABLEAU UNIDIMENSIONNEL

Il est possible d'initialiser un tableau au niveau de sa déclaration.

Cas1 : il manque des valeurs initiales

```
float TableVal2 [5] = { 1.25, 1.76, 0.35 };
```

Cas2 : omission de la dimension du tableau

```
float TableVal3 [] = { 0.0, 0.25, 0.5, 0.75 };
```

Les valeurs fournies doivent être des constantes. Autrement, le compilateur affichera un message d'erreur. C'est le cas également lorsque le nombre de valeurs initiales dépasse le nombre d'éléments, c'est-à-dire s'il est supérieur à <Nombre d'éléments>.

Si le nombre des valeurs initiales est inférieur au nombre d'éléments du tableau, les éléments complémentaires sont remplis avec des 0.

Signalons qu'on peut renoncer à la spécification du nombre des éléments d'un tableau initialisé. En effet, dans ce cas, il aura autant d'éléments qu'il y a de valeurs initiales déclarées.

Les valeurs initiales sont affectées aux éléments du tableau depuis la gauche vers la droite le premier élément reçoit la première valeur, le second reçoit la seconde valeur, etc.

8.3.3. ACCES AUX ELEMENTS D'UN TABLEAU UNIDIMENSIONNEL

L'accès à un élément du tableau s'effectue en indiquant:

<Nom du tableau> [<index>] ;

<index> doit être une expression positive ou nul.

Le premier élément du tableau à pour index 0

Le dernier élément du tableau à pour index nombre éléments - 1

Donc pour **n** élément l'indice varie de **0 à n-1**

Exemple :

```
int k = 3;
TableVal1[2] = 3;
TableVal2[k] = 3,67;
TableVal2[k+1] = 78.92;
```

Pour mieux illustrer l'accès et vérifier les valeurs obtenue à l'initialisations, voici un programme qui initialise les deux tableaux de float et affiche les valeurs obtenues de deux manières.

```
#include <stdio.h>

int main (void)
{
    int i;
    // Cas1 : il manque des valeurs initiales
    float TableVal2 [5] = { 1.25, 1.76, 0.35 };

    // Cas2 : omission de la dimension du tableau
    float TableVal3 [] = { 0.0, 0.25, 0.5, 0.75 };

    printf ("TableVal2[0] = %f \n", TableVal2[0]);
    printf ("TableVal2[1] = %f \n", TableVal2[1]);
    printf ("TableVal2[2] = %f \n", TableVal2[2]);
    printf ("TableVal2[3] = %f \n", TableVal2[3]);
    printf ("TableVal2[4] = %f \n", TableVal2[4]);
    printf ("\n");
    for (i = 0; i < 4 ; i++) {
        printf ("TableVal3[%d] = %f \n", i, TableVal3[i]);
    }
    return(0);
}
```

Résultat obtenu :

```

C:\ "d:\datas\etcourssw\loga_c\thUorie_1\projcours\chap8tab1...
TableVal2[0] = 1.250000
TableVal2[1] = 1.760000
TableVal2[2] = 0.350000
TableVal2[3] = 0.000000
TableVal2[4] = 0.000000

TableVal3[0] = 0.000000
TableVal3[1] = 0.250000
TableVal3[2] = 0.500000
TableVal3[3] = 0.750000
Press any key to continue.
```

8.3.4. TABLEAUX UNIDIMENSIONNEL ET POINTEURS

Le nom d'un tableau correspond à un pointeur sur le premier élément du tableau. Ce qui permet des manipulations en utilisant le nom du tableau comme un pointeur.

Le programme ci-dessous présente quelques aspects :

```
int main (void)
{
    short int Table7 [5] = {1, 2, 3, 4, 5 };
    short int *pItemT7;

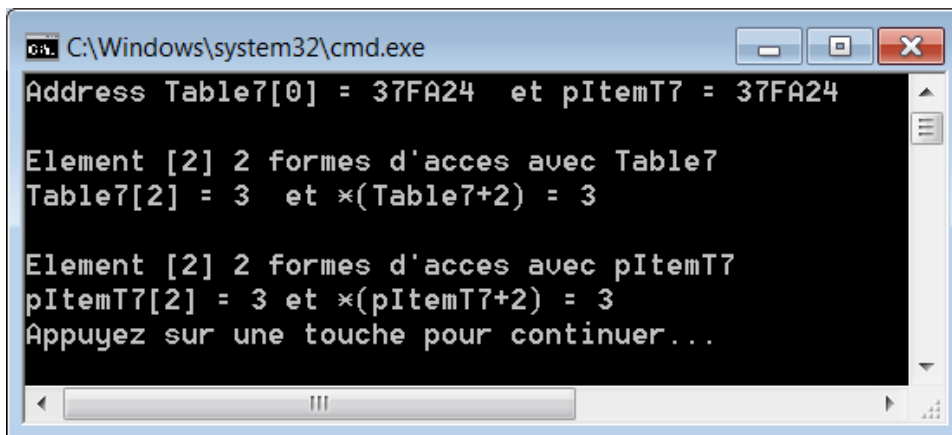
    // Init du pointeur
    pItemT7 = Table7;    // identique à &Table7[0]
    printf ("Address Table7[0] = %X et pItemT7 = %X \n\n",
            &Table7[0], pItemT7);

    printf("Element [2] 2 formes d'accès avec Table7 \n");
    printf ("Table7[2] = %d et *(Table7+2) = %d \n\n",
            Table7[2], *(Table7+2));

    printf("Element [2] 2 formes d'accès avec pItemT7 \n");
    printf ("pItemT7[2] = %d et *(pItemT7+2) = %d \n",
            pItemT7[2], *(pItemT7+2));
    return(0);
}
```

Aussi possible !

Résultat lors de l'exécution :



```
C:\Windows\system32\cmd.exe
Address Table7[0] = 37FA24 et pItemT7 = 37FA24

Element [2] 2 formes d'accès avec Table7
Table7[2] = 3 et *(Table7+2) = 3

Element [2] 2 formes d'accès avec pItemT7
pItemT7[2] = 3 et *(pItemT7+2) = 3
Appuyez sur une touche pour continuer...
```

Il est important de remarquer que la forme `*(Table7+2)` se comporte identiquement à la forme `Table7[2]`. En fait le compilateur utilise l'index comme offset en nombre d'éléments par rapport au début du tableau.

~~Actions non acceptées :~~

~~Table7++;~~

~~Table7 +=2;~~

Le compilateur n'accepte pas de modifier la valeur de Table7, ce qui signifie que Table7 se comporte pour certains aspects comme un pointeur, mais n'est pas un pointeur

8.3.5. DECLARATION D'UN TABLEAU MULTIDIMENSIONNEL

La déclaration d'un tableau multidimensionnel est la suivante :

<Type> <Nom du tableau> [Nb. Elem 1] [Nb. Elem 2] [Nb. Elem n] ;

Le nombre de dimensions d'un tableau n'est pas limité. Il est fixé par le nombre des valeurs entre crochets. Ces valeurs sont des entiers positifs qui indiquent le nombre d'éléments dans chaque dimension du tableau. Leur produit fournit le nombre des éléments du tableau.

La définition `int k [3][4];` crée, en conséquence, un tableau bidimensionnel nommé `k` et possédant $3 * 4 = 12$ éléments de type `int`.

Nous pouvons nous les représenter ainsi, disposés en lignes et colonnes

	Colonne 0	Colonne 1	Colonne 2	Colonne 3
Ligne 0	k[0] [0]	k[0] [1]	k[0] [2]	k[0] [3]
Ligne 1	k[1] [0]	k[1] [1]	k[1] [2]	k[1] [3]
Ligne 2	k[2] [0]	k[2] [1]	k[2] [2]	k[2] [3]

La définition `int m3d [2][3][4];` crée un tableau tridimensionnel comportant $2*3*4 = 24$ éléments. On peut le représenter comme 2 plans comportant chacun une matrice à 2 dimensions.

Plan 0				
	Colonne 0	Colonne 1	Colonne 2	Colonne 3
Ligne 0	m3d[0][0][0]	m3d[0][0][1]	m3d[0][0][2]	m3d[0][0][3]
Ligne 1	m3d[0][1][0]	m3d[0][1][1]	m3d[0][1][2]	m3d[0][1][3]
Ligne 2	m3d[0][2][0]	m3d[0][2][1]	m3d[0][2][2]	m3d[0][2][3]
Plan 1				
	Colonne 0	Colonne 1	Colonne 2	Colonne 3
Ligne 0	m3d[1][0][0]	m3d[1][0][1]	m3d[1][0][2]	m3d[1][0][3]
Ligne 1	m3d[1][1][0]	m3d[1][1][1]	m3d[1][1][2]	m3d[1][1][3]
Ligne 2	m3d[1][2][0]	m3d[1][2][1]	m3d[1][2][2]	m3d[1][2][3]

8.3.6. INITIALISATION D'UN TABLEAU MULTIDIMENSIONNEL

De même que pour les tableaux unidimensionnels, Il est possible d'initialiser un tableau multidimensionnel au niveau de sa déclaration. Il y a cependant une différence, c'est qu'il y a la possibilité d'initialiser comme un tableau unidimensionnel ou en séparant les dimensions par des paires d'accolades `{ }`.

Cas1 : initialisation linéaire

```
int m3d [2][3][4] =
{100, 101, 102, 103, 110, 111, 112, 113, 120, 121, 122, 123,
 200, 201, 202, 203, 210, 211, 212, 213, 220, 221, 222, 223};
```

Il faut cependant respecter l'organisation des 3 dimensions. (Les valeurs d'initialisation correspondent à plan, ligne, colonne, avec 1 et 2 pour le plan à la place de 0 et 1.

Cas2 : initialisation en séparant les dimensions

```
int m3d [2][3][4] = {
    { {100, 101, 102, 103},          // plan 0 ligne 0
      {110, 111, 112, 113},          // plan 0 ligne 1
      {120, 121, 122, 123} },        // plan 0 ligne 2
    { {200, 201, 202, 203},          // plan 1 ligne 0
      {210, 211, 212, 213},          // plan 1 ligne 1
      {220, 221, 222, 223} }        // plan 1 ligne 2
};
```

Les accolades aident à respecter l'organisation des dimensions.

8.3.7. RANGEMENT EN MEMOIRE D'UN TABLEAU MULTIDIMENSIONNEL

La mémoire n'étant qu'à une dimension, il est intéressant de vérifier dans quel ordre les valeurs sont rangées en mémoire.

Le programme ci-dessous propose par l'usage d'un pointeur une lecture croissante de la mémoire pour vérifier son contenu. En comparaison, le contenu est lu au moyen de boucle imbriquée.

8.3.7.1. PROGRAMME POUR OBSERVATION DU RANGEMENT EN MEMOIRE

```
#include <stdio.h>

int main (void)
{
    int m3d [2][3][4] = {
        { {100, 101, 102, 103},          // plan 0 ligne 0
          {110, 111, 112, 113},          // plan 0 ligne 1
          {120, 121, 122, 123} },        // plan 0 ligne 2
        { {200, 201, 202, 203},          // plan 1 ligne 0
          {210, 211, 212, 213},          // plan 1 ligne 1
          {220, 221, 222, 223} }        // plan 1 ligne 2
    };
    // pointeur pour lecture linéaire
    int *pItem = &m3d[0][0][0];
    short int iP, iL, iC;                // index des 3 dimensions
    short int iLin;                       // index linéaire
    short int maxItem = 2*3*4;            // nb. max. d'éléments
    int valItem;

    // Lecture par un pointeur de la zone mémoire
    // correspondant à m3d
    for (iLin = 0; iLin < maxItem; iLin++) {
        valItem = *pItem;
        printf ("adresse = %X Val = %d \n",
                pItem, valItem);
        pItem++;
    } // end for
```



```
// Lecture par boucles imbriquées
for (iP = 0; iP < 2; iP++) {
    for (iL = 0; iL < 3; iL++) {
        for (iC = 0; iC < 4; iC++) {
            printf ("m3d [%d][%d][%d] = %d \n",
                    iP, iL, iC, m3d[iP][iL][iC]);
        }
    }
}
return(0);
}
```

8.3.7.2. RESULTAT DE L'EXECUTION DU PROGRAMME

Voici, placé dans un tableau pour comparaison le résultat des deux boucles de lecture

Lecture par pointeur	Lecture par boucles imbriquées
adresse = 12FE74 Val = 100	m3d [0][0][0] = 100
adresse = 12FE78 Val = 101	m3d [0][0][1] = 101
adresse = 12FE7C Val = 102	m3d [0][0][2] = 102
adresse = 12FE80 Val = 103	m3d [0][0][3] = 103
adresse = 12FE84 Val = 110	m3d [0][1][0] = 110
adresse = 12FE88 Val = 111	m3d [0][1][1] = 111
adresse = 12FE8C Val = 112	m3d [0][1][2] = 112
adresse = 12FE90 Val = 113	m3d [0][1][3] = 113
adresse = 12FE94 Val = 120	m3d [0][2][0] = 120
adresse = 12FE98 Val = 121	m3d [0][2][1] = 121
adresse = 12FE9C Val = 122	m3d [0][2][2] = 122
adresse = 12FEA0 Val = 123	m3d [0][2][3] = 123
adresse = 12FEA4 Val = 200	m3d [1][0][0] = 200
adresse = 12FEA8 Val = 201	m3d [1][0][1] = 201
adresse = 12FEAC Val = 202	m3d [1][0][2] = 202
adresse = 12FEB0 Val = 203	m3d [1][0][3] = 203
adresse = 12FEB4 Val = 210	m3d [1][1][0] = 210
adresse = 12FEB8 Val = 211	m3d [1][1][1] = 211
adresse = 12FEB8 Val = 212	m3d [1][1][2] = 212
adresse = 12FEC0 Val = 213	m3d [1][1][3] = 213
adresse = 12FEC4 Val = 220	m3d [1][2][0] = 220
adresse = 12FEC8 Val = 221	m3d [1][2][1] = 221
adresse = 12FECC Val = 222	m3d [1][2][2] = 222
adresse = 12FED0 Val = 223	m3d [1][2][3] = 223

On peut en conclure que les valeurs sont bien placées en mémoire conformément à l'organisation prévue et décrite à la fois dans la déclaration et l'initialisation.

8.3.8. ACCES AUX ELEMENTS D'UN TABLEAU MULTIDIMENSIONNEL

L'accès à un élément du tableau s'effectue en indiquant:

<Nom du tableau> [<index1>] [<index2>] ... [<indexN>];

Les **<index>** doivent être des expressions positives ou nulles.

La première valeur de l'index est 0

La dernière valeur de l'index correspond à nombre éléments - 1

Comme exemple d'accès nous reprenons la lecture du tableau tridimensionnel m3d par des boucles imbriquées.

```
// Lecture par boucles imbriquées
for (iP = 0; iP < 2; iP++) {
    for (iL = 0; iL < 3; iL++) {
        for (iC = 0; iC < 4; iC++) {
            printf ("m3d [%d][%d][%d] = %d \n",
                    iP, iL, iC, m3d[iP][iL][iC]);
        }
    }
}
```

Au cœur des 3 boucles, nous trouvons une instruction printf qui affiche la valeur des indexes de chaque dimension, ainsi que la valeur d'un élément de la table, obtenu par l'expression suivante :

m3d[iP][iL][iC]

iP est l'index de la première dimension, le plan, dont la valeur est fourni par la boucle la plus extérieure.

iL est l'index de la deuxième dimension, la ligne, dont la valeur est fourni par la boucle médiane.

iC est l'index de la troisième dimension, la colonne, dont la valeur est fourni par la boucle la plus intérieure.

L'évolution des indexes nous est donnée par la colonne de gauche du tableau du paragraphe 8.3.7.2

8.3.9. TABLEAUX MULTIDIMENSIONNELS ET POINTEURS

Contrairement à un tableau unidimensionnel, le compilateur n'accepte pas le nom du tableau comme adresse du premier élément.

L'écriture suivante n'est pas acceptée:

```
int *pItem = m3d;
```



Il est nécessaire d'écrire :

```
int *pItem = &m3d[0][0][0];
```

8.4. LES STRUCTURES

Le langage C permet la création de types complexes, composé de différents types de bases ou d'autre type complexe. Cela permet de regrouper des données de natures différentes.

8.4.1. DECLARATION DES STRUCTURES

La déclaration d'une structure varie légèrement selon qu'elle est réalisée de manière globale (utilisable par toutes les fonctions et le programme principal) ou de manière locale (utilisable uniquement par la fonction dans laquelle elle est déclarée).

8.4.1.1. DECLARATION GLOBALE

```
typedef struct
{
    <Type_Champ1>    <Nom_Champ1> ;
    <Type_Champ2>    <Nom_Champ2> ;

    <Type_ChampN>    <Nom_ChampN> ;
} <Nom_TypeStructure> ;
```

8.4.1.1.1. Déclaration globale, exemple

```
typedef struct
{
    char    nom[20];
    unsigned long numero;
    double credit;
} compteBancaire;
```

Le type défini est : compteBancaire

8.4.1.2. DECLARATION LOCALE

```
struct <Nom_TagTypeStructure>
{
    <Type_Champ1>    <Nom_Champ1> ;
    <Type_Champ2>    <Nom_Champ2> ;

    <Type_ChampN>    <Nom_ChampN> ;
};
```

8.4.1.2.1. Déclaration locale, exemple

```
struct comptePostal
{
    char    nom[20];
    char    numero[20];
    double credit;
} ;
```

Le type struct défini est: **struct comptePostal**. Le "TagType" définit est utilisé avec le mot clef **struct** pour former le type .

8.4.2. DECLARATION DES VARIABLES DE TYPE STRUCT

La déclaration est légèrement différente en fonction de la déclaration de la structure.

8.4.2.1. DECLARATION VARIABLE AVEC STRUCTURE GLOBALE

Avec le `typedef` un nouveau type a été défini, d'où une déclaration comme avec un type de base.

```
<Nom_TypeStructure> <Nom_VariableStructure> ;
```

Exemple :

```
compteBancaire compteBancairePaul;
```

8.4.2.2. DECLARATION VARIABLE AVEC STRUCTURE LOCALE

Avec la déclaration locale, le mot clef **struct** est nécessaire. Il est aussi possible de déclarer les variables à la suite de la déclaration de la structure.

```
struct <Nom_TagTypeStructure> <Nom_VariableStructure> ;
```

```
struct <Nom_TagTypeStructure>
{
    <Type_Champ1>    <Nom_Champ1> ;
    <Type_Champ2>    <Nom_Champ2> ;

    <Type_ChampN>    <Nom_ChampN> ;
} <Nom_Variable1>, <Nom_Variable2>, ..., <Nom_VariableN>;
```

Exemples :

```
// Déclaration de la structure et de 2 variables
struct comptePostal
{
    char  nom[20];
    char  numero[20];
    double credit;
} ComptePostalPierre, ComptePostalJean;
// Déclaration d'une troisième variable
struct comptePostal ComptePostalLuc;
```

Remarque : pour les 3 variables le type utilisé est **struct** comptePostal

8.4.3. INITIALISATION DES VARIABLES DE TYPE STRUCT

Il est possible d'attribuer une valeur aux champs de la structure lors de la déclaration de la variable, en fournissant une liste de valeurs.

```
struct <Nom_TagTypeStructure> <NomVariableStructure> = {
    <Valeur_champ1>,
    <Valeur_champ2>,
    ...
    <Valeur_champN>
};
```

Exemple:

```
struct comptePostal ComptePostalLuc =
    { "Luc LaMainFroide",
      "18-10207-5",
      153.25
    };
```

8.4.4. ACCES AUX CHAMPS DES VARIABLES DE TYPE STRUCT

On utilise l'opérateur de champ qui est le point (•), avec le principe suivant:

<Nom_Variable>.<Nom_Champ1>

<Nom_Variable>.<Nom_Champ2>

...

<Nom_Variable>.<Nom_ChampN>

L'expression **<NomVariable>.<Nom_Champ>** s'utilise comme une variable d'un type de base.

8.4.4.1. EXEMPLE AFFECTATION VALEURS AUX CHAMPS D'UNE STRUCTURE

```
strcpy( compteBancairePaul.nom, "Paul blanc" );
compteBancairePaul.numero = 123478956;
compteBancairePaul.credit = 2345.75;
```

Attention : il faut noter l'usage de la fonction **strcpy** pour initialiser le champ **nom** qui est un tableau de caractères.

L'affectation ci-dessous provoque une erreur de compilation :

```
compteBancairePaul.nom = "Paul blanc" ;
error C2440: '=' : impossible de convertir de 'char [11]' en 'char [20]'
```

8.4.5. STRUCTURE ET POINTEUR

Lorsque l'on utilise un pointeur sur une structure, on utilise l'opérateur de champ qui est la flèche (->), avec le principe suivant:

<Nom_Pointeur>-><Nom_Champ>

Voici un exemple d'utilisation:

```
// Déclaration pointeur sur la structure compte bancaire
compteBancaire *pCompteBancaire;

// Utilisation d'un pointeur sur une structure
pCompteBancaire = &compteBancairePaul;

// Accès aux champs via le pointeur
strcpy(pCompteBancaire->nom, "Paul blanc & Fils");
if ( pCompteBancaire->numero = 123478956 ) {
    pCompteBancaire->credit = pCompteBancaire->credit - 200;
}
```

8.4.6. TABLEAUX DE STRUCTURES

Les structures étant un type il est tout à fait possible de créer des tableaux de structures.

Voici un exemple montrant la déclaration et l'initialisation d'un tableau de structures, ainsi que l'accès aux champs.

```
#include <stdio.h>

typedef struct
{
    char  nom[20];
    unsigned long  numero;
    double credit;
} compteBancaire;

int main (void)
{
    // Déclaration et initialisation d'un tableau
    // de compteBancaires
    compteBancaire ListeCompte[3] = {
        { "Paul Blanc",    123478956,    2345.75 },
        { "Bruno Kramer",  123423331,    359.20 },
        { "David Mouron",  123408959,   -137.15 }
    };
    int i;
    int nbNegatif = 0;

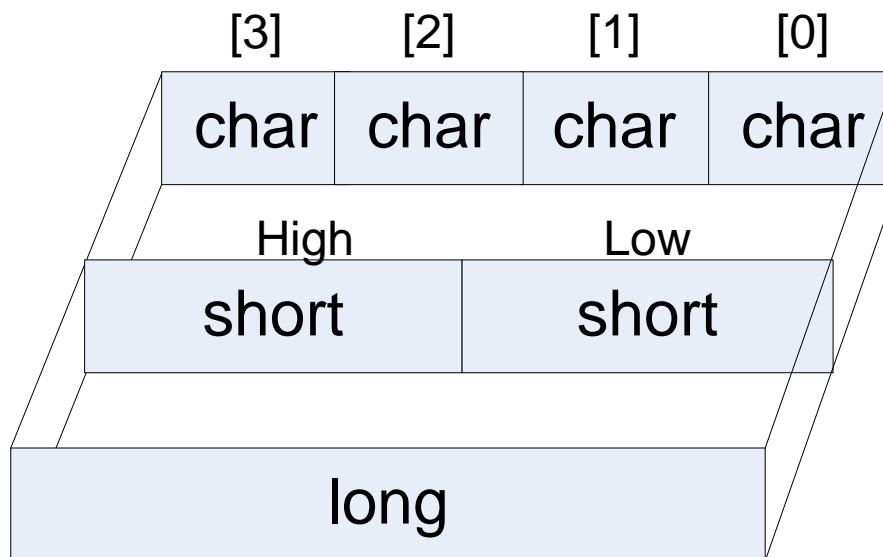
    // Accès aux champs
    for (i=0; i<3 ; i++) {
        if (ListeCompte[i].credit < 0.0) {
            nbNegatif++;
        }
    }
    return(0);
}
```

On remarquera que le (.) est placé après les []

8.5. LES UNIONS

Les unions permettent de définir un type de donnée permettant d'obtenir plusieurs descriptions ou comportements d'une même zone mémoire. Contrairement à une structure les champs se superposent.

Les unions sont utiles pour des opérations de conversion et pour supporter des variantes dans la représentation des données.



La figure ci-dessus nous montre une zone mémoire de 4 octets que l'on peut considérer comme un long int ou 2 short (high et low), ainsi que 4 octets.

8.5.1. DECLARATION DES UNIONS

La déclaration d'une union varie légèrement selon qu'elle est réalisée de manière globale (utilisable par toutes les fonctions et le programme principal) ou de manière locale (utilisable uniquement par la fonction dans laquelle elle est déclarée).

8.5.1.1. DECLARATION GLOBALE

typedef union

```
{
    <Type_Champ1>    <Nom_Champ1> ;           // variante 1
    <Type_Champ2>    <Nom_Champ2> ;           // variante 2
    <Type_ChampN>    <Nom_ChampN> ;           // variante N
} <Nom_TypeUnion> ;
```

8.5.1.1.1. Déclaration globale, exemple

```
typedef union
{
    long ValLong;
    struct { short low; short high; } Val2short;
} uLong2views;
```

Le type union défini est: **uLong2views**

Une variante d'une union peut être une structure. Il n'est pas nécessaire que la taille de chaque variante soit la même, le compilateur réserve de la place pour la plus grande des variantes.

8.5.1.2. DECLARATION LOCALE

```
union <Nom_TagTypeUnion>
{
    <Type_Champ1>    <Nom_Champ1>;           // variante 1
    <Type_Champ2>    <Nom_Champ2>;           // variante 2

    <Type_ChampN>    <Nom_ChampN>;           // variante N
};
```

8.5.1.2.1. Déclaration locale, exemple

```
union uLong3views
{
    long ValLong;
    struct {
        unsigned short low;
        unsigned short high;
    } Val2short ;
    unsigned char Val4Bytes[4];
} ;
```

Le type union défini est: **union uLong3views**. Le "TagType" défini est utilisé avec le mot clef **union** pour former le type.

8.5.2. DECLARATION DES VARIABLES DE TYPE UNION

La déclaration est légèrement différente en fonction de la déclaration de l'union.

8.5.2.1. DECLARATION VARIABLE AVEC UNION GLOBALE

Avec le **typedef** un nouveau type a été défini, d'où une déclaration comme avec un type de base.

```
<Nom_TagTypeUnion> <Nom_VariableUnion>;
```

Exemple :

```
uLong2views ValLong2views;
```

8.5.2.2. DECLARATION VARIABLE AVEC UNION LOCALE

Avec la déclaration locale, le mot clef **union** est nécessaire. Il est aussi possible de déclarer les variables à la suite de la déclaration de l'union.

```
union <Nom_TagTypeUnion> <Nom_VariableUnion>;
```

```
union <Nom_TagTypeUnion>
{
    <Type_Champ1>    <Nom_Champ1>;
    <Type_Champ2>    <Nom_Champ2>;

    <Type_ChampN>    <Nom_ChampN>;
} <Nom_VariableUnion1>, <Nom_VariableUnion2>, ..., <Nom_VariableN>;
```


Exemples :

```
// Déclaration de l'union et de 2 variables
union uLong3views
{
    long ValLong;
    struct    {
        unsigned short low;
        unsigned short high;
    } Val2short ;
    unsigned char Val4Bytes[4];
} Val1Long3views, Val2Long3views;
// Déclaration d'une troisième variable
union uLong3views convLong;
```

8.5.3. INITIALISATION DES VARIABLES DE TYPE UNION

L'initialisation est possible pour une union, la valeur à fournir est celle qui correspond à la première variante de l'union.

union <Nom_Union> <NomVariableUnion> = {<Valeur_1^{ère} variante>;

Exemple:

```
union uLong3views convLong = {315616};
```

8.5.4. ACCES AUX CHAMPS DES VARIABLES DE TYPE UNION

On utilise l'opérateur de champ qui est le point (•), avec le principe suivant:

<Nom_VariableUnion>.<Nom_Champ1>

<Nom_VariableUnion>.<Nom_Champ2>

...

<Nom_VariableUnion>.<Nom_ChampN>

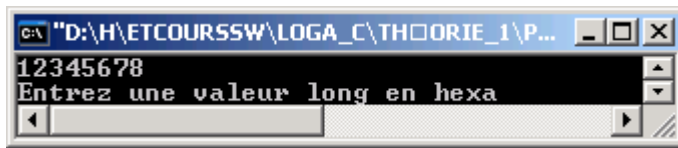
L'expression <NomVariableUnion>.<Nom_Champ> s'utilise comme une variable d'un type de base.

8.5.4.1. EXEMPLE AFFECTATION VALEURS AUX CHAMPS D'UNE UNION

```
// Accès aux variantes
convLong.ValLong = 0x12345678;
convLong.Val2ushort.low = 0x5678;
convLong.Val2ushort.high = 0x1234;
convLong.Val4Bytes[0] = 0x78;
convLong.Val4Bytes[1] = 0x56;
convLong.Val4Bytes[2] = 0x34;
convLong.Val4Bytes[3] = 0x12;
```

```
// Avec l'instruction printf ci-dessous
printf("%08X \n", convLong.ValLong);
```

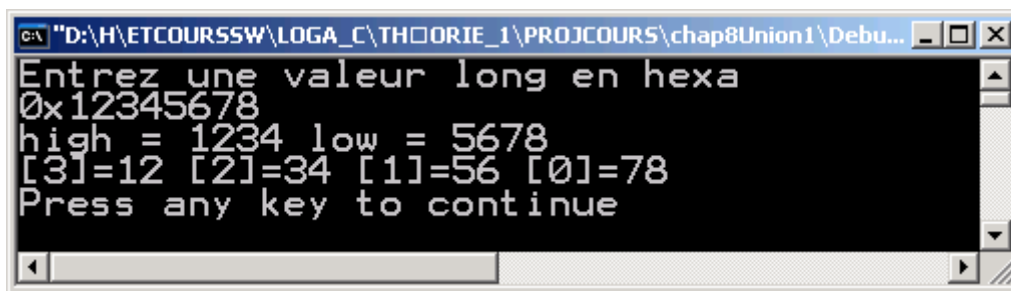
On obtient :



8.5.5. ORGANISATION DES VALEURS EN MEMOIRE

Le programme suivant utilise l'union uLong3views pour nous montrer comment sont placées les valeurs en mémoire.

Variante long	0x12345678			
	high		low	
Variante low, high	0x1234		0x5678	
	[3]	[2]	[1]	[0]
Variante 4 bytes	0x12	0x34	0x56	0x78



8.5.6. LISTING DU PROGRAMME

```
#include <stdio.h>

int main (void)
{
    short i;                                // variables de boucle
    long valIn;
    // définition locale de l'union
    union uLong3views
    {
        long ValLong;
        struct {
            unsigned short low;
            unsigned short high; } Val2ushort;
        unsigned char Val4Bytes[4];
    } ;
    // Déclaration de la variable convLong
    union uLong3views convLong;

    printf("Entrez une valeur long en hexa \n");
    scanf("%LX", &valIn);
    convLong.ValLong = valIn;

    // Affichage de la variante 2 unsigned short
    printf("high = %04X low = %04X \n",
        convLong.Val2ushort.high, convLong.Val2ushort.low);

    // Affichage de la variante 4 octets
    for (i=3;i>=0;i--) {
        printf("[%d]=%02X ", i, convLong.Val4Bytes[i]);
    }
    printf("\n");                          // ligne suivante
    return (0);
}
```

8.6. LES CHAMPS DE BITS (BIT FIELDS)

L'octet est le plus petit espace mémoire adressable pour une donnée de type fondamental. Parfois, certaines données n'occupent en fait que quelques bits. Pour ne pas perdre la place inoccupée, le type **structure** permet de travailler sur des groupes de bits d'un mot machine grâce à la définition de champs de bits.

8.6.1. DECLARATION DE CHAMPS DE BITS

Dans une structure, chaque membre est défini comme un champ en précisant à la fin de sa déclaration le nombre de bits qu'il occupe.

Un champ doit être d'un type intégral (int, unsigned int, char ou unsigned char) ou d'un type énumération.

Le nombre de bits doit être une valeur entre 1 et 16 ou 32, selon la taille des mots mémoire du système; des champs sans nom sont autorisés; en général, mais cela dépend de l'environnement.

Les champs sont stockés dans l'ordre du bit de poids le plus faible au bit de poids le plus fort.

Voici un exemple de déclaration d'une structure avec des champs de bit :

```
typedef struct {
    unsigned char dataBus :4;
    unsigned char enableBit :1;
    unsigned char writeBit :1;
    unsigned char encoded :2;
} S_bitField;

S_bitField portImage;
```

Poids faible

↓

Poids fort

Voici une représentation interne possible de la variable 8 bits portImage:

7	6	5	4	3	2	1	0
encoded	write	enable	DataBus				

8.6.2. UTILISATION DE CHAMPS DE BITS

Une variable champ de bits est utilisée comme une structure. L'accès aux différents champs se fait de la même manière que l'accès aux membres, sauf qu'il n'est pas possible de prendre l'adresse d'un champ.

Les opérations courantes sur les entiers ne sont pas applicables à une variable champ de bits. Lorsque la valeur que l'on veut stocker dans un champ est trop grande par rapport au nombre de bits réservés, la valeur est tronquée, les bits de poids fort sont perdus.

Voici un exemple d'utilisation, notez l'usage d'une union pour arriver à copier la structure complète sur un port en respectant les types puisque le port est vu comme un unsigned char.

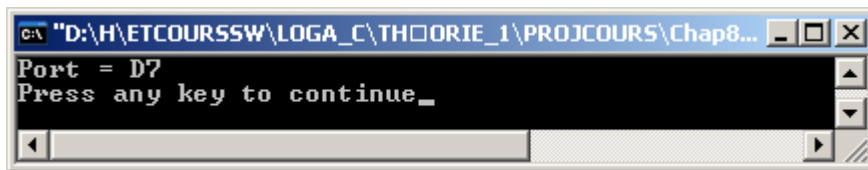
```
int main (void)
{
    unsigned char Port; // simulation d'un port E/S 8 bits

    union {
        S_bitField bitFieldsVal;
        unsigned char byteVal;
    } tmp;

    // Préparation de la valeur (Champs par champs)
    tmp.bitFieldsVal.dataBus = 7;
    tmp.bitFieldsVal.enableBit = 1;
    tmp.bitFieldsVal.writeBit = 0;
    tmp.bitFieldsVal.encoded = 3;

    // Simulation écriture sur le port
    Port = tmp.byteVal; // Port = 0xD7
    printf ( "Port = %02X \n", Port);
    return(0);
}
```

Résultat :



8.6.2.1. VERIFICATION DU RESULTAT

7	6	5	4	3	2	1	0
encoded		write	enable	DataBus			
3		0	1	7			
1	1	0	1	0	1	1	1
D				7			

8.6.2.2. UNION UTILISEE

L'union permet de voir un octet sous forme d'une structure composée de champs de bits ou d'un seul unsigned char.

7	6	5	4	3	2	1	0
encoded	write	enable	DataBus				

S_bitField
unsigned char

8.6.3. CHAMPS DE BITS CONCLUSION

Les structures avec champs de bits, souvent combinées avec des unions, permettent de gérer des variables de taille inférieure à l'octet. Cela permet de confier au compilateur les opérations de combinaisons des groupes de bits dans un mot, plutôt que de le faire manuellement en utilisant les opérateurs logique bit à bit et les décalages.

8.7. CONCLUSION

Ce chapitre a présenté les pointeurs et les types complexes. Ces éléments sont importants pour décrire les structures de données et les manipuler.

La combinaison des différents types complexes offre de grande possibilité dans la représentation et la manipulation des données. Par exemple un tableau de structure est très proche d'une table dans une base de données.

8.8. HISTORIQUE DES VERSION

8.8.1. VERSION 1.2 JANVIER 2008

Version aboutie.

8.8.2. VERSION 1.3 JANVIER 2010

Adaptation à office 2007 et ajout historique des versions.

8.8.3. VERSION 1.4 DECEMBRE 2011

Changement de LOGO et retouches orthographiques.

8.8.4. VERSION 1.5 NOVEMBRE 2014

Ajout no de module et suppression du théorie I dans les titres. Quelques retouches orthographiques.

8.8.5. VERSION 1.6 JANVIER 2016

Correction exemple du paragraphe 8.3.4