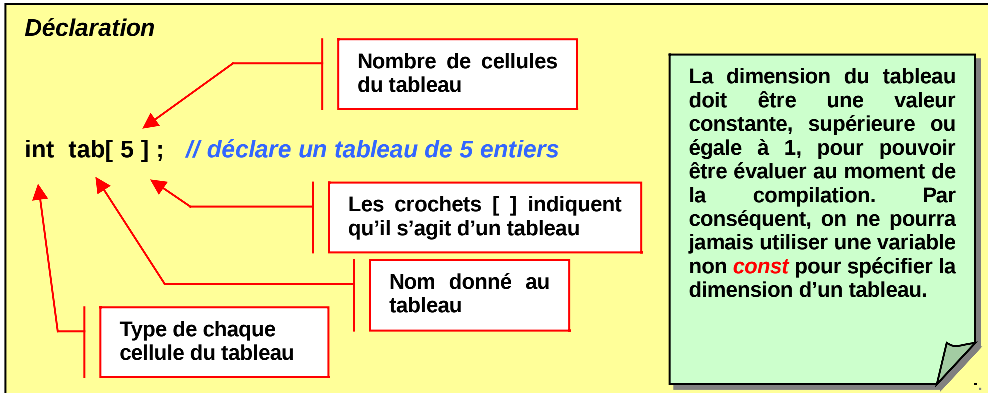
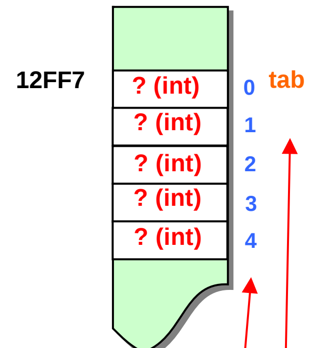
Les tableaux

Un tableau est une collection d'éléments d'un seul et même type. Les éléments individuels ne sont pas nommés ; on accède à chacun d'eux par leur position dans le tableau, repéré par un indice. La taille d'un tableau est le nombre des cellules représentant chacun des éléments. Cette taille doit être connue dès la déclaration car elle conditionne avec le type, la place mémoire allouée au tableau.



Allocation Mémoire



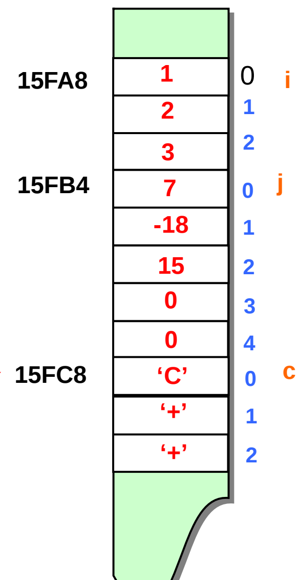
Une fois que la déclaration du tableau est réalisée, il est possible de récupérer ou de modifier chacune des cellules du tableau. Les cellules ne sont pas nommées, la seule possibilité de les repérer est d'utiliser leur position dans le tableau. Il suffit alors d'utiliser la syntaxe des crochets **[ ]** avec à l'intérieur l'indice correspondant à la cellule à atteindre.

Attention, la première cellule du tableau est référencée par l'indice **0** et la dernière par l'indice **n-1** pour un tableau de **n** cases.

Nom du tableau.
Ce nom correspond à l'adresse de la première case du tableau.

```
int main()
{
  const int nombre = 15;
  double resultats[nombre-2]; // tableau de 13 réels
  int i, it[10]; // i est un entier et it tableau de 10 entiers
  i = it[2];  // affecte à i la valeur stockée dans la cellule
              // d'indice 2, soit la 3eme case du tableau
  it[7] = i;  // cette fois-ci, c'est la case d'indice 7 du tableau
              // qui change de valeur et qui prend la valeur de i
  it[i+3] = it[i+2]; // il est possible d'avoir des indices variables

  return 0;
}
```

Allocation Mémoire



Initialisation d'un tableau :

Il est possible, comme pour les variables simples, d'initialiser explicitement le tableau dès sa déclaration. Il suffit alors de préciser la liste des valeurs entre accolades séparées par des virgules. Normalement, le nombre de valeurs initiales dans votre liste doit correspondre à la dimension de votre tableau. Toutefois, si ce nombre est inférieur, les éléments non explicitement initialisés prendront la valeur 0. Attention, la liste des valeurs ne doit pas dépasser la dimension du tableau. Pour éviter de se tromper, il est possible de ne pas spécifier explicitement la dimension du tableau. Le compilateur déterminera alors sa taille grâce au nombre d'éléments listés.

```
int main()
{
  int i[3] = {1, 2, 3};        // tableau de 3 entiers
  int j[5] = {7, -18, 15};     // tableau de 5 entiers
  char c[] = {'C', '+', '+'};  // tableau de 3 caractères
  return 0;
}
```

Affectation d'un tableau avec un autre :

Le nom du tableau correspond uniquement à l'adresse de l'emplacement mémoire de la première case. Cette adresse est fixe pendant tout le déroulement du programme. Il n'est donc pas permis d'affecter directement un tableau à l'aide d'un autre, d'une part, parce que un tableau n'est pas représenté par la totalité de ses cases, et d'autre part, parce que justement cette adresse est définitivement figée. Si nous désirons, malgré tout, avoir une copie du premier tableau, nous sommes obligés de remplir le nouveau tableau, cellule par cellule, à l'aide d'une itérative.

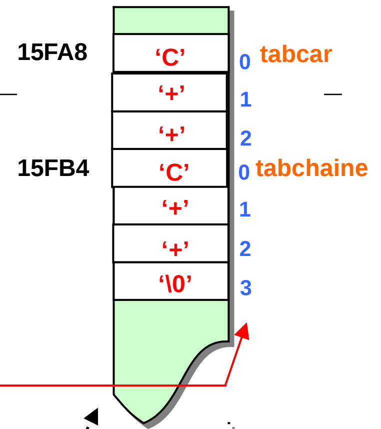
```
int main()
{
  int i[3] = {1, 2, 3}; // tableau de 3 entiers
  int j[3] = i;         // écriture interdite
  j = i;     Non toléré // écriture interdite
  int j[3];
  for (int indice=0; indice<3; indice++)
    j[indice] = i[indice];
  return 0;
}
```

Dépassement des bornes :

Lorsque vous effectuez une copie de tableau, il est judicieux de travailler avec des tableaux de même dimension. Toutefois, le langage n'offre aucun moyen de vérifier si l'indice que vous proposez, est situé à l'intérieur de l'intervalle du tableau, que ce soit à la compilation ou à l'exécution. Rien n'empêche un programmeur de franchir les bornes. Il n'est pas rare qu'un programme, compilé et exécuté, soit malgré tout encore totalement faux.

```
int main()
{
  char buffer[10];   // tableau de 10 caractères
  buffer[30] = 0;    // Attention, accès en dehors
  return 0;          // du tableau
}
```

Allocation Mémoire



Les chaînes de caractères

Une chaîne de caractères est une collection de caractères. Elle correspond donc à la définition du tableau. C'est en fait, un cas particulier d'un tableau de caractères, qui comporte un caractère supplémentaire de fin de chaîne qui s'appelle le caractère nul terminal **'\0'**. C'est un caractère de contrôle.

Un tableau de caractères peut-être initialisé soit avec une liste de caractères littéraux séparés par des virgules, soit avec une constante littérale chaîne. Remarquons toutefois que les deux formes ne sont pas équivalentes. La différence se situe au niveau du caractère nul terminal.

Constante littérale chaîne :

Il est possible d'exprimer simplement le message correspondant à une chaîne de caractères. Il suffit de placer votre texte entre guillemet. On peut alors initialiser un tableau, utiliser cette constante comme paramètre d'une fonction, ou proposer un affichage.

Attention, tabchaine est composée de 4 caractères

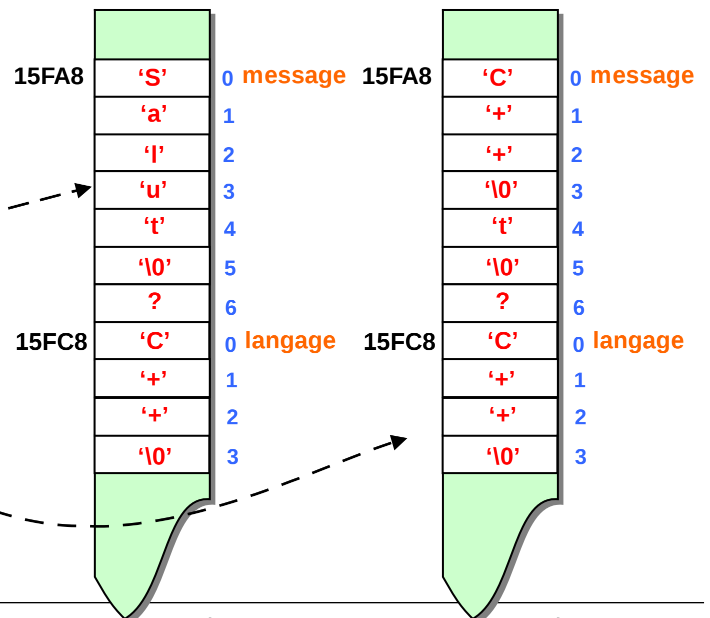
```
int main()
{
  char tabcar[] = {'C','+','+'}; // tableau de caractères
  char tabchaine[] = "C++";      // chaîne de caractères
  return 0;
}
```

Attention :

Une chaîne de caractères est un tableau comme un autre, il n'est pas possible d'affecter une chaîne de caractères avec une constante littérale chaîne. En effet, une affectation n'est pas une initialisation. La constante littérale chaîne est en fait un tableau de caractères qui ne porte pas de nom (*anonyme*) et qui se trouve quelque part en mémoire. Nous sommes donc en présence de deux tableaux, et nous savons déjà que l'affectation entre deux tableaux est interdite.

```
int main()
{
  char message[7] = "Salut";
  char langage[] = "C++";

  message = "C++";      // écriture interdite
  message = langage;    // écriture interdite

  for (int i=0; message[i]=langage[i]; i++);
  return 0;
}
```

Allocation Mémoire *Allocation Mémoire*



Les tableaux multidimensionnels

Les tableaux multidimensionnels peuvent également être définis. Chaque dimension est spécifiée avec sa propre paire de crochets, par exemple :

Déclaration

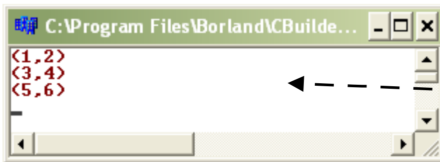
Nombre de lignes

int a [3] [2]; **// tableau à deux dimensions**

Nombre de colonnes

a est un tableau à 2 dimensions de 3 lignes ayant chacun 2 éléments. Les tableaux multidimensionnels peuvent également être initialisés.

int a[3][2] = { {1,2}, {3,4}, {5,6} } ;




```
#include <iostream.h>
//---------------------------------------------------------------
int main()
{
  int tab[3][2];
  for (int ligne=0; ligne<3; ligne++)
    for (int colonne=0; colonne<2; colonne++)
      tab[ligne][colonne] = ligne*2 + colonne + 1;
  for (int i=0; i<3; i++)
    cout<<'('<<tab[i][0]<<','<<tab[i][1]<<')'<<endl;
  cin.get();
  return 0;
}
```

Grâce à cette technique il est possible d'implémenter des tableaux de chaînes de caractères. Dans l'exemple qui suit, nous demandons à avoir un tableau de 3 chaînes de caractères. Chacune d'entre elle dispose de 10 cases :

char personnages[3][10] = { « Tintin », « Milou », « Tournesol »} ;

T	i	n	t	i	n	\0			
M	i	l	o	u	\0				
T	o	u	r	n	e	s	o	l	\0

Dans ce genre de situation, il faut être très attentif. Nous devons faire en sorte de disposer de suffisamment de place pour que la chaîne puisse être introduite en entier. Attention au caractère terminal nul. Du coup, certaines cases mémoires ne seront peut-être jamais utilisées. Tant pis, il faut faire un choix, et il est préférable d'avoir toutes les chaînes utilisables.

L'énumération

Souvent lors d'une programmation, il est nécessaire de définir un ensemble d'attributs alternatifs à associer à un objet. Un fichier, par exemple, peut être ouvert dans l'un de ces trois états : lecture, écriture, ajout. Généralement, on garde une trace de ces valeurs et attributs d'état en associant un nombre constant unique à chacun, comme suit :

```
const int lecture = 0 ;
const int ecriture = 1 ;
const int ajout = 2 ;
```

Cette approche fonctionne très bien, mais elle présente un certain nombre de faiblesses. La principale est qu'il n'existe aucun moyen de contraindre une variable (donc de type entier) à rester sur l'une de ces trois valeurs uniquement, puisque, par définition, cette variable peut prendre beaucoup plus de valeurs dans le monde des entiers.

Les énumérations sont un moyen commode de regrouper des constantes, lorsqu'elles ont une signification voisine, ou reliée.

```
enum  ModeOuverture  { lecture, ecriture, ajout } ;
```

Les énumérations permettent donc de définir, mais également de grouper des ensembles de constantes entières. Bien plus, grâce à elle, nous venons de créer de toute pièce un nouveau type : 'ModeOuverture' (type défini par l'utilisateur). Comme c'est un type, il est possible de déclarer des variables associées et même de l'initialiser.

```
ModeOuverture fichier = lecture ;
```

Nous indiquons ainsi que fichier ne prendra que l'une des trois valeurs alternatives prévues par l'énumération.

Déclaration

Une énumération est définie avec le mot clé **enum**, un nom d'énumération optionnel (Type de l'énumération) suivi d'une liste d'énumérateurs séparés par des virgules et entourés d'accolades et enfin, éventuellement, des variables associées à cette énumération. Par défaut, la valeur 0 est affectée au premier énumérateur. Chaque énumérateur suivant se voit affecter une valeur augmentée de 1 par rapport à l'énumérateur le précédant immédiatement. Il sera alors possible par la suite de faire un changement de type d'une énumération vers un entier.

```
enum Type { énumérateur1, énumérateur2, ... } variable1, variable2, … ;
```

énumérateur1 = 0
énumérateur2 = 1
….

Une valeur peut être explicitement affectée à un énumérateur. Par ailleurs, cette valeur n'est pas impérativement unique dans la liste proposée.

```
enum Points { point2d=2, point2w, point3d=3, point3w };
Points point = point2w ;
…
point = point2d ;
```

point2d = 2
point2w = 3
point3d = 3
point3w = 4

Déclaration de variable et initialisation (Points point = point2w ;)

affectation (point = point2d ;)

L'énumération présente l'énorme avantage de travailler avec des mots qui évoquent quelque chose plutôt que de manipuler des nombres qui en tant que tel ne veulent rien dire. Par exemple, si nous décidons de faire un logiciel sur un jeu de cartes, il est judicieux d'évoquer par exemple la couleur de ces cartes :

```
enum CouleurCarte  { Pique, Cœur, Carreau, Trefle } ;
```

La page suivante vous donne quelques exemples et montre l'intérêt de l'énumération.

```
enum Test {Faux, Vrai}; // Nouveau type Test qui peut prendre deux valeurs faux et vrai
Test test = Faux;       // test est de Type Test est prend comme valeur initiale faux
test = Vrai;            // changement de valeur en cours de programme
if (test==Faux) ;       // ce genre d'écriture est plus lisible
switch (test) {         // même remarque
  case Faux : ;
  case Vrai : ;
}

// autres écritures
enum {Faux, Vrai} test = Faux; // cette énumération sera associée uniquement à test
enum Test {Faux, Vrai} test;   // déclaration classique sans initialisation
enum {Faux, Vrai};             // énumération anonyme
int i = Faux;                 // qui peur être utilisée comme un ensemble de constantes

enum Clavier {Home=71, Up, PgUp, Left=75, Right=77, End=79, Down, PgDown, Ins, Annul};

enum JourSemaine {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};
for (JourSemaine jour=Lundi; jour<=Dimanche; jour++)
  switch (jour) {
    case Lundi :
    case Mardi :
    case Mercredi :
    case Jeudi :
    case Vendredi : cout << "Bon travail"; break;
    case Samedi :
    case Dimanche : cout << "Bon repos"; break;
  }
```

Nota :
Le type bool est une énumération mise en œuvre pour le C++ dont la déclaration interne est la suivante :
enum bool {false, true} ;

Les structures

Les données d'un programme sont rarement dispersées. Elles peuvent en général être pensées sous la forme de groupes plus ou moins important, ayant une cohérence significative. Par exemple, dans la gestion de personnel, on utilisera des fiches contenant le nom, le prénom, l'âge, l'adresse, etc., de chaque employé. Il serait peu logique de placer chacun des éléments de ces fiches dans des tableaux différents, car cela compliquerait la recherche de l'ensemble des caractéristiques d'un employé donné.

Déclaration

Une structure est définie avec le mot clé **struct**, un nom de structure optionnel (Type de la structure) suivi de la liste des champs séparés par des points virgules et entourés d'accolades et enfin, éventuellement, des variables associées à cette structure. Chaque champ doit être déclaré. La syntaxe reste classique, il faut préciser d'abord le type du champ (type primitif, tableau, énumération, structure) suivi de son nom d'identification. Si plusieurs champs comportent le même type, la déclaration peut se faire sur la même ligne en utilisant l'opérateur virgule qui sépare chacun des identificateurs.

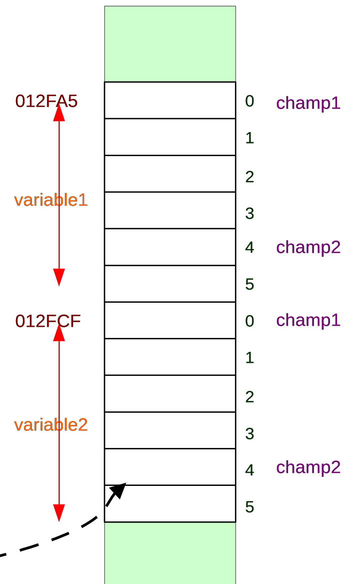
```
{
    type1  champ1 ;
    type2  champ2, champ3 ;
    …
    typeN   champN ;
}  variable1, variable2, … ;
```

Nom donné à la structure
Nouveau type défini par l'utilisateur

Liste des champs qui peuvent être de types différents. Chaque champ peut être un type primitif, un type composé ou défini par l'utilisateur. Par exemple, nous pouvons avoir, une structure à l'intérieur d'une autre structure.

Variables associées à la structure. Chacune des variables dispose d'un espace mémoire dont la capacité correspond à la taille de la structure

Allocation mémoire



Alors qu'un tableau recueille des éléments de même nature, une structure associe des éléments d'entités différentes. A l'inverse du tableau, chaque champ possède son propre nom, ce qui offre plus de souplesse d'utilisation.

Généralement, nous fabriquons d'abord la structure à laquelle nous associons un nom de type, et plus tard, nous déclarons toutes les variables associées à cette structure.
Exceptionnellement, nous pouvons directement déclarer une variable structurée sans lui associée de nom de type ; la structure est alors anonyme.

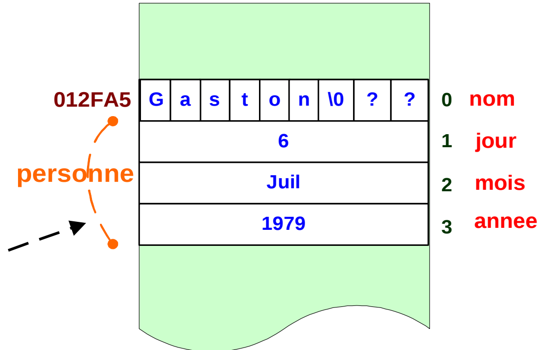
```
struct Personne {              // structure Personne
  char nom[12];
  int age;
};

Personne personnes[50];  // tableau de 50 fiches de personnes (nom, âge)

enum Mois {Jan, Fev, Mar, Avr, Mai, Juin, Juil, Aou, Sep, Oct, Nov, Dec};

struct Date {                  // structure Date
  unsigned short jour;
  Mois mois;
  int annee;
};

struct                         // structure anonyme
{                              // elle n'est utile que
  char nom[12];                // pour la variable
  Date naissance;              // personne
} personne;
```

Initialisation d'une structure :

Comme pour les tableaux, Il est possible d'initialiser explicitement la structure dès sa déclaration. Il suffit alors de préciser la liste des valeurs entre accolades séparées par des virgules. S'il y a moins de valeurs d'initialisation que de champs dans la structure, le procédé est analogue à ce qui se passe dans les tableaux. Les champs pour lesquels il n'y a pas de valeur sont initialisés à 0. Ceci dit, il est quand même préférable de bien initialiser explicitement.

```
struct Personne {
  char nom[12];
  int age;
};

Personne personne = {"Gaston", 25};

enum Mois {Jan,Fev,Mar,Avr,Mai,Juin,Juil,Aou,Sep,Oct,Nov,Dec};

struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
} date = {18, Jan, 2003};

struct {
  char nom[12];
  Date naissance;
} personne = {"Gaston", {6, Juil, 1979}};
```

Allocation mémoire



Peut aussi s'écrire :
personne = {« Gaston », 6, Juil, 1979} ;

Lorsque vous avez des structures à l'intérieur d'autres structures, au moment de l'initialisation, vous vous retrouvez avec des accolades imbriquées. Vous avez la possibilité de supprimer les accolades intérieures. Toutefois, pensez à la clarté de votre code.

Accès aux champs d'une structure

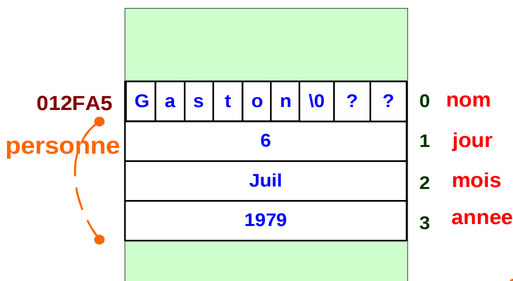
Les différents éléments d'une structure sont appelés des champs, ou des données membres. Lorsqu'on désire accéder à l'un de ces champs, il suffit d'indiquer le nom de la variable représentant la structure suivie du nom du champ. Pour séparer ces deux entités, on utilise l'opérateur de champ qui est tout simplement le point '**.**'.

```
struct  Type
{
    type1  champ1 ;
    type2  champ2, champ3 ;
    …
    typeN   champN ;

}  variable1, variable2, … ;
…
variable1.champ1
variable1.champ2
…
variable2.champ1
```

Le point est un opérateur de séparation entre le nom de la variable représentant la structure et le nom d'un des champs.

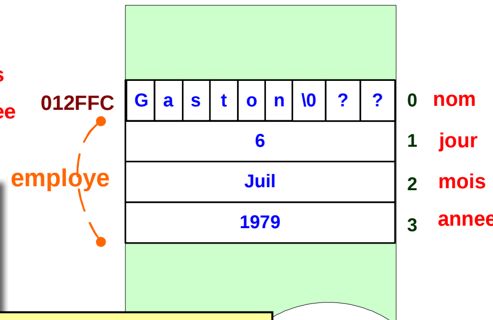
```
struct Personne {
  char nom[12];
  int age;
} personne = {"Gaston", 25};

enum Mois {Jan, Fev, Mar, Avr, Mai, Juin, Juil, Aout, Sep, Oct, Nov, Dec};

struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
} date = {18, Jan, 2003};

struct Employe {
  char nom[12];
  Date naissance;
} employe = {"Gaston", {6, Juil, 1979}};

personne.nom = "lagafe";  // change le nom de personne
personne.age = 33;        // et ensuite son âge
int age = personne.age;   // On récupère l'age de la personne et on la
  // place dans une nouvelle variable. Remarquez la compatibilité de type.
date.mois = Fev;          // compatibilité de type

date.mois = employe.naissance.mois;  // compatibilité de type

// structure dans une structure, il faut deux points de séparation.
```

Allocation mémoire




```
struct Employe {
  char nom[12];
  Date naissance;
};

Employe personne = {"Gaston", {6, Juil, 1979}};
Employe employe = personne;
    ou
employe = personne;
```

Allocation mémoire



Copie membre à membre

Utilisation de la structure :

Une fois que la déclaration est faite, nous pouvons utiliser la structure soit dans son ensemble, soit avec un des champs uniquement, comme nous venons de le découvrir.
Ce qui est très intéressant, c'est que nous pouvons affecter directement une structure vers une autre. Les champs sont alors copiés un à un, c'est ce que nous appelons une copie membre à

membre. Dès lors, la deuxième structure devient un clone de la première. Cette technique peut être appliquée également au moment de l’initialisation.
Par contre, si nous désirons changer de valeurs sur la totalité d’une structure, il n’est pas possible d’utiliser la syntaxe des accolades comme lors d’une initialisation. Nous sommes obligés d’affecter chacun des champs séparément.

```
struct Date {
  unsigned short jour;
  Mois mois;
  int annee;
};

struct Employe {
  char nom[12];
  Date naissance;
};

Employe personne = {"Gaston", {6, Juil, 1979}};

personne = {"Lagafe", {7, Aou, 1965}};

personne.nom = "Lagafe";
personne.naissance.jour = 7;
personne.naissance.mois = Aou;
personne.naissance.annee = 1965;
```

Interdit

Obliger d'initialiser membre après membre

Tableau de structure :
Il est extrêmement fréquent d’utiliser des tableaux de structures pour représenter par exemple l’ensemble des fiches des employés d’une entreprise. Chaque fiche est alors accessible par le numéro d’indice du tableau.

```
struct Personne {
  char nom[12];
  int age;
};

Personne personne[3];                      // Tableau de trois Personnes
personne[0].nom = "Gaston";                // affecte le nom de la première personne
int quelAge = personne[1].age;             // récupère l'âge de la deuxième personne
personne[2].age = personne[0].age;         // copie de l'age
personne[0] = personne[1];                 // copie complète d'une personne
Personne p[] = {"Gaston", 25,              // tableau de 2 personnes
                "Lagafe", 33};           // les accolades internes sont facultatives
```

Récupérer successivement toutes les valeurs d’un tableau (plage de valeurs C++11)

Nous venons de le découvrir, les tableaux servent de collection d’éléments de même nature. Il est très souvent utile de pouvoir consulter ultérieurement l’ensemble des éléments les uns à la suite des autres. La seule solution possible est de lire chacune des cases successivement et de passer dons par une itérative de type ***Pour***. A titre d’exemple, je vous propose le code ci-dessous qui initialise un tableau de 10 entiers avec la progression des multiples de 2, suivi d’un affichage de l’ensemble du tableau. Je vous propose deux solutions, une avec la boucle ***for(; ;)*** classique, l’autre avec la boucle ***for(:)*** ensembliste :

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    int multiples[10];
7
8    // innitialisation du tableau
9    for(int i=0, b=1; i<10; i++, b*=2) multiples[i]=b;
10
11   // affichage du tableau
12   cout << "[";
13   for (int i=0; i<10; i++) cout << multiples[i] << ' ';
14   cout << "\b]" << endl;
15
16   return 0;
17 }
```

main.cpp*

```
[1 2 4 8 16 32 64 128 256 512]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    int multiples[10], b=1;
7
8    // initialisation du tableau (plage de valeurs C++11)
9    for(int &multiple : multiples) { multiple=b; b*=2; }
10
11   // affichage du tableau (plage de valeurs C++11)
12   cout << "[";
13   for (int multiple : multiples) cout << multiple << ' ';
14   cout << "\b]" << endl;
15
16   return 0;
17 }
18
```