Les bases de l'Informatique

Thierry Vaira

BTS SN

v1.0 - 5 juillet 2017

Les types composés

- Il est possible de regrouper des types scalaires différents dans des **structures de données**.
- Il est possible de regrouper des types scalaires identiques dans des **tableaux**.
- La **chaîne de caractères** peut être considérée comme un type composé.
- On verra ensuite des **collections** de données (arbre, liste, pile, file, ...).

Présentation

- Dans de très nombreuses situations, les types de base s'avèrent insuffisants pour permettre de traiter un problème : il peut être nécessaire, par exemple, de stocker un certain nombre de valeurs en mémoire afin que des traitements similaires leurs soient appliqués.
- Dans ce cas, il est impensable d'avoir recours à de simples variables car tout traitement itératif est inapplicable.
- D'autre part, il s'avère intéressant de pouvoir regrouper ensemble plusieurs variables afin de les manipuler comme un tout.
- Pour répondre à tous ces besoins, le langage C comporte la notion de **type agrégé** en utilisant : les **tableaux**, les **structures**, les **unions** et les **énumérations**.

Les tableaux

- Un tableau est un **ensemble d'éléments de même type désignés par un identificateur unique** (un nom).
- Chaque élément est repéré par une valeur entière appelée **indice** (ou index) indiquant sa position dans l'ensemble.
- Les tableaux sont toujours à **bornes statiques** et leur indiçage démarre toujours à partir de **0**.
- La forme générique de déclaration d'un tableau est la suivante :

  `[classe de mémorisation] type identificateur[`$dimension_1$`]...[`$dimension_n$`]`

Contrairement à beaucoup d'autres langages, il n'existe pas en C de véritable notion de tableaux multidimentionnels. De tels tableaux se définissent par composition de tableaux, c'est à dire que les éléments sont eux-mêmes des tableaux.

Exemple : les tableaux

```
#define MAX 20  // définit l'étiquette MAX egale à 20

int   t[10];     // tableau de 10 éléments entiers (int)

// tableau à 2 dimensions de 2 lignes et 5 colonnes :
int   m[2][5] = { 2,  6, -4,  8, 11,   // initialise avec des valeurs
                  3, -1,  0,  9,  2 };

int   x[5][12][7]; // tableau a 3 dimensions, rarement au-delà de cette dimension

float f[MAX]; // tableau de MAX éléments de type float

t[2] = 6; // accès en écriture au 3eme élément du tableau t
printf("%d", m[1][3]); // affiche la valeur 9
```

Particularités des tableaux

- L'identificateur du tableau désigne non pas le tableau dans son ensemble, mais plus précisément l'**adresse en mémoire du début du tableau**.
- Ceci implique qu'il est impossible d'affecter un tableau à un autre : `int a[10], b[10]; a = b; // cette affectation est interdite`
- L'identificateur d'un tableau sera donc "vu" comme un **pointeur constant**.

Danger : le plus grand danger dans la manipulation des tableaux est d'accéder en écriture en dehors du tableau. Cela provoque un accès mémoire interdit qui n'est pas contrôlé au moment de la compilation. Par contre, lors de l'exécution, cela provoquera une exception de violation mémoire (*segmentation fault*) qui se traduit généralement par une sortie prématurée du programme avec un message "Erreur de segmentation".

- La dimension d'un tableau peut être omise dans 2 cas :

Exemple : les tableaux

1. le compilateur peut en définir la valeur

```
int t[]={2, 7, 4}; // tableau de 3 éléments
char msg[]="Bonjour"; // chaîne de caractères
```

2. l'emplacement mémoire correspondant a été réservé

```
// la fonction fct admet en parametre
void fct(int t_i[]) // un tableau d'entiers qui existe déjà
```

Exemple : tableaux et pointeurs

```
int t[5] = {0, 2, 3, 6, 8}; // un tableau de 5 entiers
int *p1 = NULL; // le pointeur est initialisé à NULL (précaution obligatoire)
int *p2; // pointeur non initialisé : il pointe donc sur n'importe quoi (gros
    danger)

p1 = t;             // p1 pointe sur t c'est-a-dire la première case du tableau
// identique a : p1 = &t[0];

p2 = &t[1];         // p2 pointe sur le 2eme élément du tableau

*p1 = 4;             // la première case du tableau est modifiée
printf("%d ou %d\n", *p1, t[0]); // affiche 4 ou 4

printf("%d ou %d\n", *p2, t[1]); // affiche 2 ou 2
p2 += 2;             // p2 pointe sur le 4eme élément du tableau (indice 3)
printf("%d ou %d\n", *p2, t[3]); // affiche 6 ou 6

// on peut utiliser les [] sur un pointeur :
p1[1] = 8;           // identique à : *(p1+1) = 8; ou a : t1[1] = 8;
printf("%d\n", t[1]); // affiche 8
```

Les chaînes de caractères

- Une chaîne de caractères est un tableau de caractères dont le dernier caractère est le **caractère nul (valeur 0) qui marque ainsi la fin de la chaîne**. En C, un caractère est un code ASCII sur 8 bits (cf. `man ascii`).
- De nombreuses fonctions de la librairie standard (les fonctions commençant par str, comme `strlen`, `strcpy`, `strcat`, ...) reçoivent en paramètre des chaînes de caractères (et doivent donc posséder le fin de chaîne final).

Danger : omettre le fin de chaîne provoquera généralement une "Erreur de segmentation" car le traitement itératif des caractères se poursuivra en dehors du tableau. Il est donc recommandé de ne pas oublier de déclarer son tableau d'une taille suffisante pour y stocker la chaîne de caractères agrandie d'un caractère pour y placer le fin de chaîne.

Exemple : les chaînes de caractères

```c
#include <stdio.h>
#include <string.h> /* pour les fonctions str... */

int main()
{
   char tab[4] = { 'a', 'b', 'c', 'd' }; // un tableau de caractères (il n'y a
        pas de fin de chaînes)
   char msg[] = "Bonjour"; // chaîne de caractères (le fin de chaîne est ajouté
        automatiquement ici)

   printf("msg : %s contient %d caractères\n", msg, strlen(msg));

   printf("tab : %s contient %d caractères\n", tab, strlen(tab)); // risque !

   tab[3] = 0; // maintenant, on place le caractère nul de fin de chaîne
   printf("tab : %s contient %d caractères\n", tab, strlen(tab));

   return 0;
}
```

Traité tab comme une chaîne engendrera un bug à l'exécution :

```
$ ./a.out
msg : Bonjour contient 7 caractères

tab : abcdp\? contient 8 caractères <---- ici !

tab : abc contient 3 caractères
```

string en C++ (1/2)

Attention : Ce n'est pas un type de base (c'est une classe) qui permet de stocker (et de manipuler) une suite de lettres (une chaîne de caractères).

Utilisation de string en C++ :

```
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;
int main (int argc, char **argv) {
   string le0 = "Chuck";
   string le1 = "Norris";
   cout << "Il n'y a que deux éléments en informatique : le " << le0 << " et le
        " << le1 << '.' << endl;
   if(le0 != le1) cout << "Remarque: Les deux chaînes sont différentes" << endl;
   printf("Une chaîne en C : Si windows existe encore c'est parce que %s %s ne s
        'est jamais intéressé à l'informatique.\n", le0.c_str(), le1.c_str());
   return 0;
}
```

string en C++ (2/2)

La définition de variable de type `string` est très simple :

```
string s1; // s1 contient 0 caractère
string s2 = "New York"; // s2 contient 8 caractères
string s3(60,'*'); // s3 contient 60 étoiles (*)
string s4 = s3; // copie de s3 dans la nouvelle s4 (operator=)
string s5(s3); // idem mais avec le constructeur de copie
string s6(s2, 4, 2); // s6 contient "Yo"
```

Lecture et écriture très simple également :

```
string s;
cin >> s; // La taille de s s'adapte toute seule à la saisie.
cout << s;
```

La Salle Avignon

Compromis entre string et iostream

Le compromis entre un string et un iostream (istream ou ostream) est stringstream (istringstream ou ostringstream). Cela permet : de lire/écrire dedans avec » et « et d'obtenir un string comme résultat (en utilisant la méthode str()).

```
#include <iostream>
#include <sstream> // stringstream !
#include <string>
using namespace std;
int main() {
   string s("pi"); int n=2; float pi=3.14;
   ostringstream oss;
   oss << n << s << '=' <<2*pi; // on écrit dans oss, sans affichage
   cout << "resultat : " << oss.str() << endl; // affiche : resultat : 2pi=6.28
   return 0;
}
```

struct

- Une structure est un **objet agrégé comprenant un ou plusieurs champs (membres)** d'éventuellement différents types que l'on regroupe sous un seul nom afin d'en faciliter la manipulation et le traitement.
- Chacun des champs peut avoir n'importe quel type, y compris une structure, à l'exception de celle à laquelle il appartient.

Déclaration d'une structure

```
[classe de memorisation] struct [etiquette]
{
   type champ_1;
   ...
   type champ_n;
} [identificateur];
```

Déclaration d'une structure

- Le mot clé `struct` indique la déclaration d'une structure qui correspond à une liste de déclarations entre accolades.
- Il est possible de faire suivre le mot `struct` d'un nom baptisé etiquette de la structure.
- Cette etiquette désigne cette sorte de structure et, par la suite, peut servir pour éviter d'écrire entièrement toute la déclaration.

Déclaration d'une structure date :

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

struct date date_naissance;
```

Définition de variable structurée

- Il suffit pour cela de faire suivre la liste entre accolade d'identificateurs de variables.

Définition de variables de type struct :

```
struct
{
   int   jour,
         mois,
         annee;
} naissancePierre, naissanceMarie;
```

Déclaration et Définition de variable structurée

- Il est aussi possible de combiner les deux possibilités : `date` est le nom de la structure (le modèle), tandis que `naissancePierre`, `naissanceMarie` et `mortColuche` sont des variables.

Définition de variables de type struct date :

```
struct date
{
   int   jour,
         mois,
         année;
) naissancePierre,
  naissanceMarie = {7, 11, 1867}; // initialisation de la structure
       naissanceMarie

struct date mortColuche;
```

Particularités des structures

- La taille d'une structure est la somme des tailles de tous les objets qui la compose (cf. `sizeof()`). Dans notre exemple, la structure aura une taille de 3*4 (`int`) soit 12 octets.
- Pour accéder aux champs dune structure, il faut distinguer 2 cas :
  1. la structure est délivrée par une variable : cet accès se fait à l'aide de l'opérateur `.` (point)

     Exemple : `naissanceMarie.mois` désigne le champ `mois` de la variable `naissanceMarie` (soit 11 dans notre exemple).
  2. la structure est délivrée par un pointeur `p` : cet accès se fait à l'aide de l'opérateur `->` (indirection)

     Exemple : `p->jour` désigne le champ `jour` de la variable `p`

     Ou : `(*p).jour` désigne le champ `jour` de la variable `p`

Définition de synonyme de type

- Utilisation courante : le programmeur a l'habitude de définir un **nouveau type** lorsqu'il déclare une nouvelle structure. La convention habituelle est de mettre ce nom en majuscules.

Utilisation de typedef :

```
// directement :
typedef struct
{
   int   jour,
         mois,
         annee;
} DATE; // le type DATE

// ou apres :
typedef struct date DATE_NAISSANCE; // le type DATE_NAISSANCE
```

Utilisation des structures :

```
int main()
{
   struct date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie
   struct date *p_naissanceMarie = &naissanceMarie;
   DATE_NAISSANCE naissancePierre = {15, 5, 1859}; // initialisation de la structure naissancePierre
   DATE mortColuche = {19, 6, 1986};

   printf("Marie Curie est née le %02d/%02d/%4d\n", naissanceMarie.jour, naissanceMarie.mois,
         naissanceMarie.annee);

   printf("Marie Curie est née le %02d/%02d/%4d\n", p_naissanceMarie->jour, p_naissanceMarie->mois,
         p_naissanceMarie->annee);

   printf("Pierre Curie est né le %02d/%02d/%4d\n", naissancePierre.jour, naissancePierre.mois,
         naissancePierre.annee);

   printf("Coluche est mort le %02d/%02d/%4d\n\n", mortColuche.jour, mortColuche.mois, mortColuche.annee);

   printf("La structure struct date occupe une taille de %d octets\n", sizeof(struct date));

   return 0;
}
```

L'exécution suivante illustre différentes utilisations des structures :

```
Marie Curie est nee le 07/11/1867
Marie Curie est nee le 07/11/1867

Pierre Curie est ne le 15/05/1859

Coluche est mort le 19/06/1986

La structure struct date occupe une taille de 12 octets
```

union

- Une union est conceptuellement identique à une structure mais peut, à tout moment, contenir n'importe lequel des différents champs.
- Rappel : Une structure définit une suite de champs, portant chacun un unique nom et tous présents simultanément. Ils sont rangés dans des emplacements mémoire consécutifs.
- Une union, d'un autre côté, définit **plusieurs manières de regarder le même emplacement mémoire**. A l'exception de ceci, la façon dont sont déclarés et référencées les structures et les unions est identique.

Déclaration d'une union

```
[classe de memorisation] union [etiquette] {
   type champ_1;
   ...
   type champ_n;
} [identificateur];
```

Définition d'une union

- On va déclarer une `union` pour conserver une valeur d'une mesure issue d'un capteur générique, qui peut par exemple fournir une mesure sous forme d'un `char` (-128 à +127), d'un `int` (-2147483648 à 2147483647) ou d'un `float`.
- Une telle définition définit une variable de nom `valeurCapteur` et de type `union` qui peut, à tout moment, contenir SOIT un entier, SOIT un réel, SOIT un caractère.
- La taille mémoire de la variable `valeurCapteur` est égale à la taille mémoire du plus grand type qu'elle contient (ici c'est `float`).

Définition d'une `union` :

```
union mesureCapteur {
   int   iVal;
   float fVal;
   char  cVal;
} valeurCapteur;
```

Utilisation d'une union :

```c
#include <stdio.h>
typedef union mesureCapteur {
   int   iVal;
   float fVal;
   char  cVal;
} CAPTEUR;

int main() {
   CAPTEUR vitesseVent, temperatureMoteur, pressionAtmospherique;
   pressionAtmospherique.iVal = 1013; /* un int */
   temperatureMoteur.fVal = 50.5; /* un float */
   vitesseVent.cVal = 2; /* un char */
   printf("La pression atmosphérique est de %d hPa\n", pressionAtmospherique.iVal);
   printf("La température du moteur est de %.1f °C\n", temperatureMoteur.fVal);
   printf("La vitesse du vent est de %d km/h\n", vitesseVent.cVal);
   printf("Le type CAPTEUR occupe une taille de %d octets\n", sizeof(CAPTEUR));
   return 0;
}
```

L'exécution du programme d'essai permet de vérifier cela :

```
La pression atmosphérique est de 1013 hPa
La température du moteur est de 50.5 °C
La vitesse du vent est de 2 km/h
Le type CAPTEUR occupe une taille de 4 octets
```

Champs de bits

- Les champs de bits ("Drapeaux" ou "*Flags*"), qui ont leur principale application en informatique industrielle, sont des **structures** qui ont la possibilité de regrouper (au plus juste) plusieurs valeurs.
- La taille d'un champ de bits **ne doit pas excéder celle d'un entier**. Pour aller au-delà, on créera un deuxième champ de bits.
- On utilisera le mot clé `struct` et on donnera le type des groupes de bits, leurs noms, et enfin leurs tailles :

Déclaration d'un champ de bits

```
[classe de memorisation] struct [etiquette] {
{
   type champ_1 : nombre_de_bits;
   type champ_2 : nombre_de_bits;
   [...]
   type champ_n : nombre_de_bits;
} [identificateur];
```

Champ de bits : exemple

- Si on reprend le type structuré `date`, on peut maintenant décomposer ce type en trois groupes de bits (`jour`, `mois` et `annee`) avec le nombre de bits suffisants pour coder chaque champ.
- Les différents groupes de bits seront tous accessibles comme des variables classiques d'une structure ou d'une union.
- La taille mémoire d'une variable de ce type sera égale à 2 octets ($5 + 4 + 7 = 16$ bits).

Le champ de bits `date` :

```
struct date
{
  unsigned short jour : 5; // 2^5 = 0-32 soit de 1 à 31
  unsigned short mois : 4; // 2^4 = 0-16 soit de 1 à 12
  unsigned short annee : 7; // 2^7 = 0-128 soit de 0 à 99 (sans les siècles)
};
```

Utilisation d'un champ de bits :

```
typedef struct date DATE_NAISSANCE;
int main (int argc, char **argv) {
   struct date naissanceRitchie = {9, 9, 41};
   struct date *p_naissanceRitchie = &naissanceRitchie;
   DATE_NAISSANCE naissanceThompson = {4, 2, 43};
   struct date mortRitchie = {12, 10, 11};
   printf("Dennis Ritchie est né le %02d/%02d/%2d\n", naissanceRitchie.jour, naissanceRitchie.mois,
         naissanceRitchie.annee);
   printf("Dennis Ritchie est né le %02d/%02d/%2d\n", p_naissanceRitchie->jour, p_naissanceRitchie->mois,
         p_naissanceRitchie->annee);
   printf("Dennis Ritchie est mort le %02d/%02d/%2d\n\n", mortRitchie.jour, mortRitchie.mois, mortRitchie.
         annee);
   printf("Ken Thompson est né le %02d/%02d/%2d\n", naissanceThompson.jour, naissanceThompson.mois,
         naissanceThompson.annee);
   printf("La structure champs de bits date occupe une taille de %d octets\n", sizeof(struct date));
   return 0;
}
```

L'exécution du programme d'essai permet de vérifier cela :

```
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est mort le 12/10/11
Ken Thompson est né le 04/02/43
La structure champs de bits date occupe une taille de 2 octets
```

Les conteneurs en C++ (1/2)

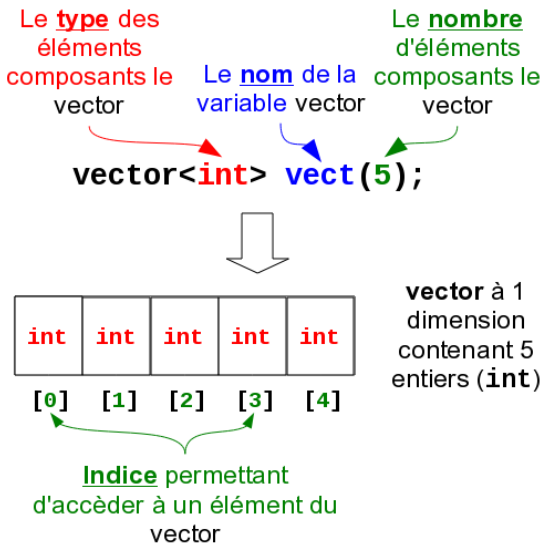
- Le C++ possède une bibliothèque standard (SL pour *Standard Library*) qui est composée, entre autre, d'une bibliothèque de flux, de la bibliothèque standard du C, de la gestion des exceptions, ..., et de la **STL** (*Standard Template Library* : bibliothèque de modèles standard). En fait, STL est une appellation historique communément acceptée et comprise. Dans la norme, on ne parle que de SL.
- Un **conteneur** (*container*) est un **objet qui contient d'autres objets**.
- Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets qui dans le cas de la STL consiste très souvent en un **itérateur**.

Les conteneurs en C++ (2/2)

- Les itérateurs permettent de **parcourir une collection d'objets** sans avoir à se préoccuper de la manière dont ils sont stockés. Ceci permet aussi d'avoir une interface de manipulation commune, et c'est ainsi que la STL fournit des algorithmes génériques qui s'appliquent à la majorité de ses conteneurs (tri, recherche, ...).
- Parmi les conteneurs disponibles dans la STL on trouve les **tableaux** (***vector***), les **listes** (*list*), les **ensembles** (*set*), les **piles** (*stack*), et beaucoup d'autres.
- Nous allons nous intéresser à la notion de **vector**.

vector : les tableaux en C++

- Un *vector* est un **tableau dynamique** où il est particulièrement aisé d'accéder directement aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin.
- A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.
- La forme générique de déclaration d'un tableau est la suivante : `vector<type> identificateur(taille);`



Exemple 1 : les vector

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v1; // un vecteur d'entier vide

  v1.push_back( 10 ); // ajoute l'entier 10 à la fin
  v1.push_back( 9 ); // ajoute l'entier 9 à la fin
  v1.push_back( 8 ); // ajoute l'entier 8 à la fin
  // enleve le dernier élément
  v1.pop_back(); // supprime l'entier 8

  // utilisation d'un indice pour parcourir le vecteur v1
  cout << "Le vecteur v1 contient " << v1.size() << " entiers : \n";
  for(int i=0;i<v1.size();i++)
      cout << "v1[" << i << "] = " << v1[i] << '\n';
  cout << '\n';

   return 0;
}
```

Exemple 2 : les vector

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v2(4, 100); // un vecteur de 4 entiers initialisés avec la valeur
       100

  // utilisation d'un itérateur pour parcourir le vecteur v2
  cout << "Le vecteur v2 contient " << v2.size() << " entiers : ";
  for (vector<int>::iterator it = v2.begin(); it != v2.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

   return 0;
}
```

Ces deux exemples donnent :

```
Le vecteur v1 contient 2 entiers :
v1[0] = 10
v1[1] = 9

Le vecteur v2 contient 4 entiers : 100 100 100 100
```